FREE eBook

LEARNING multithreading

Free unaffiliated eBook created from
Stack Overflow contributors.

#multithreading

ing

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: multithreading

It is an unofficial and free multithreading ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official multithreading.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with multithreading

Remarks

Multithreading is a programming technique which consists of dividing a task into separate threads of execution. These threads run concurrently, either by being assigned to different processing cores, or by time-slicing.

When designing a multithreaded program, the threads should be made as independent of each other as possible, to achieve the greatest speed-up.
In practice the threads are rarely fully independent, which makes synchronisation necessary.
The maximum theoretical speed-up can be calculated using Amdahl's law.

Advantages

- Speed up execution time by using the available processing resources efficiently
- Allow a process to remain responsive without the need to split lengthy calculations or expensive I/O operations
- Easily prioritize certain operations over others

Disadvantages

- Without careful design, hard-to-find bugs may be introduced
- Creating threads involves some overhead

Examples

Purpose

Threads are the low level parts of a computing system which command processing occurs. It is supported/provided by CPU/MCU hardware. There are also software methods. The purpose of multi-threading is doing calculations in parallel to each other if possible. Thus the desired result can be obtained in a smaller time slice.

Deadlocks

A deadlock occurs when every member of some group of two or more threads must wait for one of the other members to do something (e.g., to release a lock) before it can proceed. Without intervention, the threads will wait forever.

A pseudocode example of a deadlock-prone design is:

```
thread_1 {
    acquire(A)
```

```
    ...
    acquire(B)
    ...
    release(A, B)
}

thread_2 {
    acquire(B)
    ...
    acquire(A)
    ...
    release(A, B)
}
```

A deadlock can occur when `thread_1` has acquired `A`, but not yet `B`, and `thread_2` has acquired `B`, but not `A`. As shown in the following diagram, both threads will wait forever.

How to Avoid Deadlocks

As a general rule of thumb, minimize the use of locks, and minimize code between lock and unlock.

Acquire Locks in Same Order

A redesign of `thread_2` solves the problem:

```
thread_2 {
    acquire(A)
    ...
    acquire(B)
    ...
    release(A, B)
}
```

Both threads acquire the resources in the same order, thus avoiding deadlocks.

This solution is known as the "Resource hierarchy solution". It was proposed by Dijkstra as a solution to the "Dining philosophers problem".

Sometimes even if you specify strict order for lock acquisition, such static lock acquisition order can be made dynamic at runtime.

Consider following code:

```
void doCriticalTask(Object A, Object B){
     acquire(A){
        acquire(B){

        }
    }
}
```

Here even if the lock acquisition order looks safe, it can cause a deadlock when thread_1 accesses this method with, say, Object_1 as parameter A and Object_2 as parameter B and thread_2 does in opposite order i.e. Object_2 as parameter A and Object_1 as parameter B.

In such situation it is better to have some unique condition derived using both Object_1 and Object_2 with some kind of calculation, e.g. using hashcode of both objects, so whenever different thread enters in that method in whatever parametric order, everytime that unique condition will derive the lock acquisition order.

e.g. Say Object has some unique key, e.g. accountNumber in case of Account object.

```
void doCriticalTask(Object A, Object B){
    int uniqueA = A.getAccntNumber();
    int uniqueB = B.getAccntNumber();
    if(uniqueA > uniqueB){
         acquire(B){
            acquire(A){

            }
        }
    }else {
         acquire(A){
            acquire(B){

            }
        }
    }
}
```

Race conditions

A data race or race condition is a problem that can occur when a multithreaded program is not properly synchronized. If two or more threads access the same memory without synchronization, and at least one of the accesses is a 'write' operation, a data race occurs. This leads to platform dependent, possibly inconsistent behavior of the program. For example, the result of a calculation could depend on the thread scheduling.

Readers-Writers Problem:

```
writer_thread {
    write_to(buffer)
}

reader_thread {
    read_from(buffer)
}
```

A simple solution:

```
writer_thread {
    lock(buffer)
    write_to(buffer)
    unlock(buffer)
```

```
}

reader_thread {
    lock(buffer)
    read_from(buffer)
    unlock(buffer)
}
```

This simple solution works well if there is only one reader thread, but if there is more than one, it slows down the execution unnecessarily, because the reader threads could read simultaneously.

A solution that avoids this problem could be:

```
writer_thread {
    lock(reader_count)
    if(reader_count == 0) {
        write_to(buffer)
    }
    unlock(reader_count)
}

reader_thread {
    lock(reader_count)
    reader_count = reader_count + 1
    unlock(reader_count)

    read_from(buffer)

    lock(reader_count)
    reader_count = reader_count - 1
    unlock(reader_count)
}
```

Note that `reader_count` is locked throughout the whole writing operation, such that no reader can begin reading while the writing has not finished.

Now many readers can read simultaneously, but a new problem may arise: The `reader_count` may never reach `0`, such that the writer thread can never write to the buffer. This is called starvation, there are different solutions to avoid it.

Even programs that may seem correct can be problematic:

```
boolean_variable = false

writer_thread {
    boolean_variable = true
}

reader_thread {
    while_not(boolean_variable)
    {
       do_something()
    }
}
```

The example program might never terminate, since the reader thread might never see the update from the writer thread. If for example the hardware uses CPU caches, the values might be cached. And since a write or read to a normal field, does not lead to a refresh of the cache, the changed value might never be seen by the reading thread.

C++ and Java defines in the so called memory model, what properly synchronized means: C++ Memory Model, Java Memory Model.

In Java a solution would be to declare the field as volatile:

```
volatile boolean boolean_field;
```

In C++ a solution would be to declare the field as atomic:

```
std::atomic<bool> data_ready(false)
```

A data race is a kind of race condition. But not all race conditions are data races. The following called by more than one thread leads to a race condition but not to a data race:

```
class Counter {
    private volatile int count = 0;

    public void addOne() {
     i++;
    }
}
```

It is correctly synchronized according to the Java Memory Model specification, therefore it is not data race. But still it leads to a race conditions, e.g. the result depends on the interleaving of the threads.

Not all data races are bugs. An example of an so called benign race condition is the sun.reflect.NativeMethodAccessorImpl:

```
class  NativeMethodAccessorImpl extends MethodAccessorImpl {
    private Method method;
    private DelegatingMethodAccessorImpl parent;
    private int numInvocations;

    NativeMethodAccessorImpl(Method method) {
        this.method = method;
    }

    public Object invoke(Object obj, Object[] args)
        throws IllegalArgumentException, InvocationTargetException
    {
        if (++numInvocations > ReflectionFactory.inflationThreshold()) {
              MethodAccessorImpl acc = (MethodAccessorImpl)
            new MethodAccessorGenerator().
            generateMethod(method.getDeclaringClass(),
                           method.getName(),
                           method.getParameterTypes(),
                           method.getReturnType(),
```

```
                                method.getExceptionTypes(),
                                method.getModifiers());
                                parent.setDelegate(acc);
            }
            return invoke0(method, obj, args);
    }
    ...
}
```

Here the performance of the code is more important than the correctness of the count of numInvocation.

Hello Multithreading - Creating new threads

This simple example shows how to start multiple threads in Java. Note that the threads are not guaranteed to execute in order, and the execution ordering may vary for each run.

```
public class HelloMultithreading {

    public static void main(String[] args) {

        for (int i = 0; i < 10; i++) {
            Thread t = new Thread(new MyRunnable(i));
            t.start();
        }
    }

    public static class MyRunnable implements Runnable {

        private int mThreadId;

        public MyRunnable(int pThreadId) {
            super();
            mThreadId = pThreadId;
        }

        @Override
        public void run() {
            System.out.println("Hello multithreading: thread " + mThreadId);
        }

    }

}
```

Can the same thread run twice?

It was most frequent question that can a same thread can be run twice.

The answer for this is know one thread can run only once .

if you try to run the same thread twice it will execute for the first time but will give error for second time and the error will be IllegalThreadStateException .

example:

```
public class TestThreadTwice1 extends Thread{
 public void run(){
   System.out.println("running...");
 }
 public static void main(String args[]){
  TestThreadTwice1 t1=new TestThreadTwice1();
  t1.start();
  t1.start();
 }
}
```

output:

```
running
       Exception in thread "main" java.lang.IllegalThreadStateException
```

Read Getting started with multithreading online:
https://riptutorial.com/multithreading/topic/1229/getting-started-with-multithreading

Chapter 2: Executors

Syntax

- **ThreadPoolExecutor**

- `ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue)`
- `ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue, RejectedExecutionHandler handler)`
- `ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory)`
- `ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory, RejectedExecutionHandler handler)`
- `Executors.callable(PrivilegedAction<?> action)`
- `Executors.callable(PrivilegedExceptionAction<?> action)`
- `Executors.callable(Runnable task)`
- `Executors.callable(Runnable task, T result)`
- `Executors.defaultThreadFactory()`
- `Executors.newCachedThreadPool()`
- `Executors.newCachedThreadPool(ThreadFactory threadFactory)`
- `Executors.newFixedThreadPool(int nThreads)`
- `Executors.newFixedThreadPool(int nThreads, ThreadFactory threadFactory)`
- `Executors.newScheduledThreadPool(int corePoolSize)`
- `Executors.newScheduledThreadPool(int corePoolSize, ThreadFactory threadFactory)`
- `Executors.newSingleThreadExecutor()`
- `Executors.newSingleThreadExecutor(ThreadFactory threadFactory)`

Parameters

Parameter	Detail
corePoolSize	Minimum number of threads to keep in the pool.
maximumPoolSize	Maximum number of threads to allow in the pool.
keepAliveTime	When number of threads is greater than the core, the noncore threads

Parameter	Detail
	(excess idle threads) will wait for time defined by this parameter for new tasks before terminating.
unit	Time unit for `keepAliveTime`.
timeout	the maximum time to wait
workQueue	The type of queue that our Executor is going to use
threadFactory	The factory to use when creating new threads
nThreads	The number of threads in the pool
executor	The underlying implementation
task	the task to run
result	The result to return
action	The privileged action to run
callable	The underlying task

Remarks

The different types of Threadpools and Queues that are explained below, have been taken from the information and knowledge from [oracle documentation][1] and [Jakob Jenkov][2] blog where you can learn a lot about concurrency in java.

Different types of ThreadPools

SingleThreadExecutor: Executor that uses a single worker thread operating off an unbounded queue, and uses the provided ThreadFactory to create a new thread when needed. Unlike the otherwise equivalent newFixedThreadPool(1, threadFactory) the returned executor is guaranteed not to be reconfigurable to use additional threads.

FixedThreadPool: thread pool that reuses a fixed number of threads operating off a shared unbounded queue, using the provided ThreadFactory to create new threads when needed. At any point, at most nThreads threads will be active processing tasks. If additional tasks are submitted when all threads are active, they will wait in the queue until a thread is available. If any thread terminates due to a failure during execution prior to shutdown, a new one will take its place if needed to execute subsequent tasks. The threads in the pool will exist until it is explicitly shutdown.

CachedThreadPool: Thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available, and uses the provided ThreadFactory to create new

threads when needed.

SingleThreadScheduledExecutor: Single-threaded executor that can schedule commands to run after a given delay, or to execute periodically. (Note however that if this single thread terminates due to a failure during execution prior to shutdown, a new one will take its place if needed to execute subsequent tasks.) Tasks are guaranteed to execute sequentially, and no more than one task will be active at any given time. Unlike the otherwise equivalent newScheduledThreadPool(1, threadFactory) the returned executor is guaranteed not to be reconfigurable to use additional threads.

ScheduledThreadPool: Thread pool that can schedule commands to run after a given delay, or to execute periodically. Different types of Work Queues

Examples

Defining a new ThreadPool

A `ThreadPool` is an `ExecutorService` that executes each submitted task using one of possibly several pooled threads, normally configured using Executors factory methods.

Here is a basic code to initialize a new ThreadPool as a singleton to use in your app:

```
public final class ThreadPool {

    private static final String TAG = "ThreadPool";
    private static final int CORE_POOL_SIZE = 4;
    private static final int MAX_POOL_SIZE = 8;
    private static final int KEEP_ALIVE_TIME = 10; // 10 seconds
    private final Executor mExecutor;

    private static ThreadPool sThreadPoolInstance;

    private ThreadPool() {
        mExecutor = new ThreadPoolExecutor(
            CORE_POOL_SIZE, MAX_POOL_SIZE, KEEP_ALIVE_TIME,
            TimeUnit.SECONDS, new LinkedBlockingQueue<Runnable>());
    }

    public void execute(Runnable runnable) {
        mExecutor.execute(runnable);
    }

    public synchronized static ThreadPool getThreadPoolInstance() {
        if (sThreadPoolInstance == null) {
            Log.i(TAG, "[getThreadManagerInstance] New Instance");
            sThreadPoolInstance = new ThreadPool();
        }
        return sThreadPoolInstance;
    }
}
```

You have two ways to call your runnable method, use `execute()` or `submit()`. the difference between them is that `submit()` returns a `Future` object which allows you a way to programatically

cancel the running thread when the object T is returned from the `Callable` callback. You can read more about `Future` here

Future and Callables

One of the features that we can use with Threadpools is the `submit()` method which allow us to know when the thread as finish his work. We can do this thanks to the `Future` object, which return us an object from the Callable that we can use to our own objetives.

Here is an example about how to use the Callable instance:

```
public class CallablesExample{

//Create MyCustomCallable instance
List<Future<String>> mFutureList = new ArrayList<Future<String>>();

//Create a list to save the Futures from the Callable
Callable<String> mCallable = new MyCustomCallable();

public void main(String args[]){
    //Get ExecutorService from Executors utility class, Creating a 5 threads pool.
    ExecutorService executor = Executors.newFixedThreadPool(5);



    for (int i = 0; i < 100; i++) {
        //submit Callable tasks to be executed by thread pool
        Future<String> future = executor.submit(mCallable);
        //add Future to the list, we can get return value using Future
        mFutureList.add(future);
    }
    for (Future<String> fut : mFutureList) {
        try {
            //Print the return value of Future, Notice the output delay in console
            //because Future.get() stop the thread till the task have been completed
            System.out.println(new Date() + "::" + fut.get());
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        }
    }
    //Shut down the service
    executor.shutdown();
}

 class MyCustomCallable implements Callable<String> {

    @Override
    public String call() throws Exception {
        Thread.sleep(1000);
        //return the thread name executing this callable task
        return Thread.currentThread().getName();
    }
}
}
```

As you can see, we create a Threadpool with 5 Threads, this means that we can throw 5 callables parallel. When the threads finish, we will get and Future object from the callable, in this case the

name of the thread.

WARNING

In this example, we just use the Futures as a object inside the array to know how many threads we are executing and print that many times a log in console with the data that we want. But, if we want to use the method `Future.get()`, to return us the data that we saved before in the callable, we will block the thread till the task is completed. Be care with this kind of calls when you want perform this as fast as possible

Custom Runnables instead of Callables

Another good practice to check when our threads have finished without block the thread waiting to recover the Future object from our Callable is to create our own implemetation for Runnables, using it together with the `execute()` method.

In the next example, I show a custom class which implements Runnable with a internal callback, with allow us to know when the runnables are finished and use it later in our ThreadPool:

```
public class CallbackTask implements Runnable {
    private final Runnable mTask;
    private final RunnableCallback mCallback;

    public CallbackTask(Runnable task, RunnableCallback runnableCallback) {
        this.mTask = task;
        this.mCallback = runnableCallback;
    }

    public void run() {
        long startRunnable = System.currentTimeMillis();
        mTask.run();
        mCallback.onRunnableComplete(startRunnable);
    }

    public interface RunnableCallback {
        void onRunnableComplete(long runnableStartTime);
    }
}
```

And here is our ThreadExecutor Implementation:

```
public class ThreadExecutorExample implements ThreadExecutor {

    private static String TAG = "ThreadExecutorExample";
    public static final int THREADPOOL_SIZE = 4;
    private long mSubmittedTasks;
    private long mCompletedTasks;
    private long mNotCompletedTasks;

    private ThreadPoolExecutor mThreadPoolExecutor;

    public ThreadExecutorExample() {
        Log.i(TAG, "[ThreadExecutorImpl] Initializing ThreadExecutorImpl");
        Log.i(TAG, "[ThreadExecutorImpl] current cores: " +
Runtime.getRuntime().availableProcessors());
```

```
        this.mThreadPoolExecutor =
            (ThreadPoolExecutor) Executors.newFixedThreadPool(THREADPOOL_SIZE);

    }

    @Override
    public void execute(Runnable runnable) {
        try {
            if (runnable == null) {
                Log.e(TAG, "[execute] Runnable to execute cannot be null");
                return;
            }
            Log.i(TAG, "[execute] Executing new Thread");

            this.mThreadPoolExecutor.execute(new CallbackTask(runnable, new
CallbackTask.RunnableCallback() {

                @Override
                public void onRunnableComplete(long RunnableStartTime) {
                    mSubmittedTasks = mThreadPoolExecutor.getTaskCount();
                    mCompletedTasks = mThreadPoolExecutor.getCompletedTaskCount();
                    mNotCompletedTasks = mSubmittedTasks - mCompletedTasks; // approximate

                    Log.i(TAG, "[execute] [onRunnableComplete] Runnable complete in " +
(System.currentTimeMillis() - RunnableStartTime) + "ms");
                    Log.i(TAG, "[execute] [onRunnableComplete] Current threads working " +
mNotCompletedTasks);
                }
            }));
        }
        catch (Exception e) {
            e.printStackTrace();
            Log.e(TAG, "[execute] Error, shutDown the Executor");
            this.mThreadPoolExecutor.shutdown();
        }
    }
}


 /**
 * Executor thread abstraction created to change the execution context from any thread from
out ThreadExecutor.
 */
interface ThreadExecutor  extends Executor {

    void execute(Runnable runnable);

}
```

I did this example to check speed of my threads in milliseconds when they are executed, without use Future. You can take this example and add it to your app to control the concurrent task working, and the completed/finished ones. Checking in all moment, the time that you needed to execute that threads.

Adding ThreadFactory to Executor

We use ExecutorService to assign threads from the internal thread pool or create them on-demand to perform tasks. Each ExecutorService has an ThreadFactory, but The ExecutorService

will use always a default one if we don't set a custom one. Why we should do this?

- To set a more descriptive thread name. Default ThreadFactory gives thread names in the form of pool-m-thread-n, such as pool-1-thread-1, pool-2-thread-1, pool-3-thread-1, etc. If you are trying to debug or monitoring something, it's hard to know what are that threads doing
- Set a custom Daemon status, the default ThreadFactory produces non-daemon results.
- Set priority to our threads, the default ThreadFactory set a medium priority to all their threads.
- You can specify `UncaughtExceptionHandler` for our thread using `setUncaughtExceptionHandler()` on thread object. This gets called back when Thread's run method throws uncaught exception.

Here is a easy implementation of a ThreadFactory over a ThreadPool.

```
public class ThreadExecutorExample implements ThreadExecutor {
private static String TAG = "ThreadExecutorExample";
private static final int INITIAL_POOL_SIZE = 3;
private static final int MAX_POOL_SIZE = 5;

// Sets the amount of time an idle thread waits before terminating
private static final int KEEP_ALIVE_TIME = 10;

// Sets the Time Unit to seconds
private static final TimeUnit KEEP_ALIVE_TIME_UNIT = TimeUnit.SECONDS;

private final BlockingQueue<Runnable> workQueue;

private final ThreadPoolExecutor threadPoolExecutor;

private final ThreadFactory threadFactory;
private ThreadPoolExecutor mThreadPoolExecutor;

public ThreadExecutorExample() {
    this.workQueue = new LinkedBlockingQueue<>();
    this.threadFactory = new CustomThreadFactory();
    this.threadPoolExecutor = new ThreadPoolExecutor(INITIAL_POOL_SIZE, MAX_POOL_SIZE,
            KEEP_ALIVE_TIME, KEEP_ALIVE_TIME_UNIT, this.workQueue, this.threadFactory);
}

public void execute(Runnable runnable) {
    if (runnable == null) {
        return;
    }
    this.threadPoolExecutor.execute(runnable);
}

private static class CustomThreadFactory implements ThreadFactory {
    private static final String THREAD_NAME = "thread_";
    private int counter = 0;

    @Override public Thread newThread(Runnable runnable) {
        return new Thread(runnable, THREAD_NAME + counter++);
    }
```

```
}
}

/**
 * Executor thread abstraction created to change the execution context from any thread from
out ThreadExecutor.
 */
interface ThreadExecutor extends Executor {

    void execute(Runnable runnable);

}
```

This example just modify the name of the Thread with a counter, but we can modify it as long as we want.

Read Executors online: https://riptutorial.com/multithreading/topic/6710/executors

Chapter 3: Semaphores & Mutexes

Introduction

Semaphores & Mutexes are concurrency controls used to synchronize multiple thread access to shared resources.

Remarks

Semaphore

Here's a brilliant explanation from this Stackoverflow question:

> Think of semaphores as bouncers at a nightclub. There are a dedicated number of people that are allowed in the club at once. If the club is full no one is allowed to enter, but as soon as one person leaves another person might enter.
>
> It's simply a way to limit the number of consumers for a specific resource. For example, to limit the number of simultaneous calls to a database in an application.

Mutex

A mutex is a semaphore of 1 (i.e. only one thread at a time). Using the nightclub metaphor, think of a mutex in terms of a bathroom stall in the nightclub. Only one occupant allowed at a time.

Examples

Mutex in Java & C++

Although Java doesn't have a Mutex class, you can mimic a Mutex with the use of a Semaphore of 1. The following example executes two threads with and without locking. Without locking, the program spits out a somewhat random order of output characters ($ or #). With locking, the program spits out nice, orderly character sets of either ##### or $$$$$, but never a mix of # & $.

```
import java.util.concurrent.Semaphore;
import java.util.concurrent.ThreadLocalRandom;

public class MutexTest {
   static Semaphore semaphore = new Semaphore(1);

   static class MyThread extends Thread {
      boolean lock;
      char c = ' ';
```

```
      MyThread(boolean lock, char c) {
         this.lock = lock;
         this.c = c;
      }

      public void run() {
         try {
            // Generate a random number between 0 & 50
            // The random nbr is used to simulate the "unplanned"
            // execution of the concurrent code
            int randomNbr = ThreadLocalRandom.current().nextInt(0, 50 + 1);

            for (int j=0; j<10; ++j) {
               if(lock) semaphore.acquire();
               try {
                  for (int i=0; i<5; ++i) {
                     System.out.print(c);
                     Thread.sleep(randomNbr);
                  }
               } finally {
                  if(lock) semaphore.release();
               }
               System.out.print('|');
            }
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
   }

   public static void main(String[] args) throws Exception {
      System.out.println("Without Locking:");
      MyThread th1 = new MyThread(false, '$');
      th1.start();
      MyThread th2 = new MyThread(false, '#');
      th2.start();

      th1.join();
      th2.join();

      System.out.println('\n');

      System.out.println("With Locking:");
      MyThread th3 = new MyThread(true, '$');
      th3.start();
      MyThread th4 = new MyThread(true, '#');
      th4.start();

      th3.join();
      th4.join();

      System.out.println('\n');
   }
}
```

Run `javac MutexTest.java; java MutexTest`, and you will get something like this:

> Without Locking:
> #$$$$$|$$$$$|$$#$$$|$$$$$|$$$$#$|$$$$$|$$$$$|$#$$$$|$$$$$|$$$#$$||#####|#####|#####|###

With Locking:
$$$$$|#####|$$$$$|#####|$$$$$|#####|$$$$$|#####|$$$$$|#####|$$$$$|#####|$$$$$|#####|$$$

Here's the same example in C++:

```
#include <iostream>       // std::cout
#include <thread>         // std::thread
#include <mutex>          // std::mutex
#include <random>         // std::random_device

class MutextTest {
   private:
      static std::mutex mtx;  // mutex for critical section

   public:
      static void run(bool lock, char c) {
         // Generate a random number between 0 & 50
         // The random nbr is used to simulate the "unplanned"
         // execution of the concurrent code
         std::uniform_int_distribution<int> dist(0, 50);
         std::random_device rd;
         int randomNbr = dist(rd);
         //std::cout << randomNbr << '\n';

        for(int j=0; j<10; ++j) {
           if(lock) mtx.lock();
           for (int i=0; i<5; ++i) {
              std::cout << c << std::flush;
              std::this_thread::sleep_for(std::chrono::milliseconds(randomNbr));
           }
           std::cout << '|';
           if(lock) mtx.unlock();
        }
      }
};

std::mutex MutextTest::mtx;

int main()
{
  std::cout << "Without Locking:\n";
  std::thread th1 (MutextTest::run, false, '$');
  std::thread th2 (MutextTest::run, false, '#');

  th1.join();
  th2.join();

  std::cout << "\n\n";

  std::cout << "With Locking:\n";
  std::thread th3 (MutextTest::run, true, '$');
  std::thread th4 (MutextTest::run, true, '#');

  th3.join();
  th4.join();

  std::cout << '\n';

  return 0;
}
```

Run `g++ --std=c++11 MutexTest.cpp; ./a.out`, and you will get something like this:

Without Locking:
$#$#$#$#$#|$|#$#$#$#$#|$$|#$#$#$#|$#$|#$#$#$#|$$#$|#$#$#|$#$#$|#$#$#|$#$$#$|#$#|$#$#$#

With Locking:
$$$$$|#####|$$$$$|#####|$$$$$|#####|$$$$$|#####|$$$$$|#####|$$$$$|#####|$$$$$|#####|$$$

Read Semaphores & Mutexes online:
https://riptutorial.com/multithreading/topic/10861/semaphores---mutexes

Credits

S. No	Chapters	Contributors
1	Getting started with multithreading	alain, Amit Gujarathi, Boo Radley, Community, Gul Md Ershad, james large, Jim, John Odom, Mert Gülsoy, RamenChef, Thomas Krieger, vvtx, Zim-Zam O'Pootertoot
2	Executors	Francisco Durdin Garcia, Guillermo Orellana Ruiz, vvtx
3	Semaphores & Mutexes	John DiFini