EBook Gratis

APRENDIZAJE multithreading

Free unaffiliated eBook created from
Stack Overflow contributors.

#multithreading

ing

Tabla de contenido

Acerca de

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

Capítulo 1: Empezando con multihilo

Observaciones

El subprocesamiento múltiple es una técnica de programación que consiste en dividir una tarea en subprocesos separados de ejecución. Estos subprocesos se ejecutan de forma simultánea, ya sea mediante la asignación a diferentes núcleos de procesamiento o mediante el corte de tiempo.

Al diseñar un programa de multiproceso, los hilos deben hacerse lo más independientes posible entre sí, para lograr la mayor velocidad.
En la práctica, los hilos rara vez son totalmente independientes, lo que hace que la sincronización sea necesaria.
La máxima aceleración teórica se puede calcular utilizando la ley de Amdahl .

Ventajas

- Acelere el tiempo de ejecución utilizando los recursos de procesamiento disponibles de manera eficiente
- Permita que un proceso siga respondiendo sin la necesidad de dividir cálculos largos o operaciones de E / S costosas
- Priorizar fácilmente ciertas operaciones sobre otras

Desventajas

- Sin un diseño cuidadoso, se pueden introducir errores difíciles de encontrar
- Crear hilos implica algo de sobrecarga

Examples

Propósito

Los hilos son las partes de bajo nivel de un sistema informático en el que se produce el procesamiento de comandos. Es compatible / proporcionado por el hardware de CPU / MCU. También hay métodos de software. El propósito de los subprocesos múltiples es hacer cálculos en paralelo entre sí, si es posible. Por lo tanto, el resultado deseado se puede obtener en una porción de tiempo más pequeña.

Puntos muertos

Se produce un punto muerto cuando cada miembro de un grupo de dos o más subprocesos debe esperar a que uno de los otros miembros haga algo (por ejemplo, para liberar un bloqueo) antes de que pueda continuar. Sin intervención, los hilos esperarán por siempre.

Un ejemplo de pseudocódigo de un diseño propenso a interbloqueo es:

```
thread_1 {
```

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

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

Se puede producir un punto muerto cuando el `thread_1` ha adquirido `A` , pero aún no `B` , y `thread_2` ha adquirido `B` , pero no `A` Como se muestra en el siguiente diagrama, ambos hilos esperarán por siempre.

Cómo evitar los puntos muertos

Como regla general, minimice el uso de bloqueos y minimice el código entre el bloqueo y el desbloqueo.

Adquirir cerraduras en el mismo orden

Un rediseño de `thread_2` resuelve el problema:

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

Ambos hilos adquieren los recursos en el mismo orden, evitando así los interbloqueos.

Esta solución se conoce como la "solución de jerarquía de recursos". Fue propuesto por Dijkstra como una solución al "problema de los filósofos que comen".

A veces, incluso si especifica un orden estricto para la adquisición de bloqueos, dicho orden de adquisición de bloqueo estático se puede dinamizar en el tiempo de ejecución.

Considere el siguiente código:

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

        }
```

```
    }
}
```

Aquí incluso si la orden de adquisición de bloqueo parece segura, puede causar un interbloqueo cuando thread_1 accede a este método con, digamos, Object_1 como parámetro A y Object_2 como parámetro B y thread_2 lo hace en orden opuesto, es decir, Object_2 como parámetro A y Object_1 como parámetro B.

En tal situación, es mejor tener alguna condición única derivada utilizando tanto Object_1 como Object_2 con algún tipo de cálculo, por ejemplo, utilizando el código hash de ambos objetos, por lo que cada vez que un hilo diferente ingrese en ese método en cualquier orden paramétrico, cada vez que esa condición única se derive bloqueo de orden de adquisición.

por ejemplo, Say Object tiene una clave única, por ejemplo, accountNumber en el caso del objeto Account.

```
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){

            }
        }
    }
}
```

Condiciones de carrera

Una condición de carrera o carrera de datos es un problema que puede ocurrir cuando un programa de multiproceso no está sincronizado correctamente. Si dos o más subprocesos acceden a la misma memoria sin sincronización, y al menos uno de los accesos es una operación de "escritura", se produce una carrera de datos. Esto conduce a un comportamiento del programa dependiente, posiblemente incoherente. Por ejemplo, el resultado de un cálculo podría depender de la programación de subprocesos.

Problema de los lectores-escritores :

```
writer_thread {
    write_to(buffer)
}

reader_thread {
    read_from(buffer)
}
```

Una solución simple:

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

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

Esta solución simple funciona bien si solo hay un subproceso del lector, pero si hay más de uno, ralentiza la ejecución innecesariamente, porque los subprocesos del lector pueden leer simultáneamente.

Una solución que evita este problema podría ser:

```
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)
}
```

Tenga en cuenta que `reader_count` está bloqueado durante toda la operación de escritura, por lo que ningún lector puede comenzar a leer mientras la escritura no haya terminado.

Ahora muchos lectores pueden leer simultáneamente, pero puede surgir un nuevo problema: el `reader_count` nunca puede alcanzar `0` , por lo que el hilo del escritor nunca puede escribir en el búfer. Esto se llama inanición , existen diferentes soluciones para evitarlo.

Incluso los programas que pueden parecer correctos pueden ser problemáticos:

```
boolean_variable = false

writer_thread {
    boolean_variable = true
}
```

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

Es posible que el programa de ejemplo nunca finalice, ya que el subproceso del lector nunca verá la actualización del subproceso del escritor. Si, por ejemplo, el hardware utiliza cachés de CPU, es posible que los valores se almacenen en caché. Y dado que una escritura o lectura en un campo normal, no lleva a una actualización de la memoria caché, el hilo de lectura nunca verá el valor modificado.

C ++ y Java definen en el llamado modelo de memoria, lo que correctamente sincronizado significa: C ++ Memory Model , Java Memory Model .

En Java, una solución sería declarar el campo como volátil:

```
volatile boolean boolean_field;
```

En C ++, una solución sería declarar el campo como atómico:

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

Una carrera de datos es un tipo de condición de carrera. Pero no todas las condiciones de carrera son carreras de datos. Lo siguiente llamado por más de un hilo conduce a una condición de carrera pero no a una carrera de datos:

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

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

Se sincroniza correctamente de acuerdo con la especificación del modelo de memoria Java, por lo que no es una carrera de datos. Pero todavía conduce a condiciones de carrera, por ejemplo, el resultado depende de la intercalación de los hilos.

No todas las razas de datos son errores. Un ejemplo de una condición llamada raza benigna es 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);
    }
    ...
}
```

Aquí el rendimiento del código es más importante que la corrección del recuento de numInvocation.

Hello Multithreading - Creando nuevos hilos.

Este sencillo ejemplo muestra cómo iniciar múltiples hilos en Java. Tenga en cuenta que no se garantiza que los subprocesos se ejecuten en orden, y el orden de ejecución puede variar para cada ejecución.

```
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);
        }

    }

}
```

¿Puede el mismo hilo correr dos veces?

La pregunta más frecuente es que un mismo hilo puede ejecutarse dos veces.

La respuesta para esto es saber que un hilo solo puede ejecutarse una vez.

si intenta ejecutar el mismo hilo dos veces, se ejecutará por primera vez, pero dará un error por segunda vez y el error será IllegalThreadStateException.

ejemplo :

```
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();
 }
}
```

salida :

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

Lea Empezando con multihilo en línea:
https://riptutorial.com/es/multithreading/topic/1229/empezando-con-multihilo

Capítulo 2: Ejecutores

Sintaxis

- **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)`

Parámetros

Parámetro	Detalle
corePoolSize	Número mínimo de hilos a mantener en la piscina.
maximumPoolSize	Número máximo de hilos para permitir en la piscina.
keepAliveTime	Cuando el número de subprocesos es mayor que el núcleo, los

Parámetro	Detalle
	subprocesos no centrales (exceso de subprocesos inactivos) esperarán el tiempo definido por este parámetro para nuevas tareas antes de terminar.
unidad	Unidad de tiempo para `keepAliveTime` .
se acabó el tiempo	el tiempo máximo para esperar
workQueue	El tipo de cola que nuestro Ejecutor va a utilizar.
threadFactory	La fábrica para usar al crear nuevos hilos.
nhilos	El número de hilos en la piscina.
albacea	La implementación subyacente
tarea	la tarea a ejecutar
resultado	El resultado para volver.
acción	La acción privilegiada para ejecutar.
llamable	La tarea subyacente

Observaciones

Los diferentes tipos de Threadpools y Colas que se explican a continuación, se han tomado de la información y los conocimientos de [la documentación de Oracle] [1] y [Jakob Jenkov] [2] donde puede aprender mucho sobre la concurrencia en Java.

Diferentes tipos de ThreadPools

SingleThreadExecutor: Ejecutor que usa un solo subproceso de trabajo que opera en una cola sin límites, y usa el ThreadFactory proporcionado para crear un nuevo subproceso cuando sea necesario. A diferencia del newFixedThreadPool (1, threadFactory) equivalente, se garantiza que el ejecutor devuelto no se puede volver a configurar para usar subprocesos adicionales.

FixedThreadPool: grupo de subprocesos que reutiliza un número fijo de subprocesos que operan en una cola compartida sin límites, utilizando el ThreadFactory proporcionado para crear nuevos subprocesos cuando sea necesario. En cualquier momento, como máximo, las hebras nThreads serán tareas de procesamiento activas. Si se envían tareas adicionales cuando todos los subprocesos están activos, esperarán en la cola hasta que haya un subproceso disponible. Si alguna de las hebras termina debido a una falla durante la ejecución antes del cierre, una nueva tomará su lugar si es necesario para ejecutar tareas posteriores. Los subprocesos en el grupo existirán hasta que se cierre explícitamente.

CachedThreadPool: grupo de subprocesos que crea nuevos subprocesos según sea necesario, pero reutilizará los subprocesos construidos previamente cuando estén disponibles y utiliza el ThreadFactory proporcionado para crear nuevos subprocesos cuando sea necesario.

SingleThreadScheduledExecutor: ejecutor de un solo hilo que puede programar comandos para que se ejecuten después de un retraso determinado o para que se ejecuten periódicamente. (Sin embargo, tenga en cuenta que si este único hilo termina debido a una falla durante la ejecución antes del cierre, uno nuevo tomará su lugar si es necesario para ejecutar las tareas subsiguientes). Se garantiza que las tareas se ejecutarán de forma secuencial, y no más de una tarea estará activa en cualquier momento dado. A diferencia del newScheduledThreadPool (1, threadFactory) equivalente, se garantiza que el ejecutor devuelto no se puede volver a configurar para usar subprocesos adicionales.

ScheduledThreadPool: grupo de subprocesos que puede programar comandos para que se ejecuten después de un retraso determinado o que se ejecuten periódicamente. Diferentes tipos de colas de trabajo

Examples

Definiendo un nuevo ThreadPool

Un `ThreadPool` es un `ExecutorService` que ejecuta cada tarea enviada usando uno de posiblemente varios subprocesos agrupados, normalmente configurados usando los métodos de fábrica de los Ejecutores.

Aquí hay un código básico para iniciar un nuevo ThreadPool como un singleton para usar en su aplicación:

```
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;
    }
}
```

Tiene dos formas de llamar a su método ejecutable, usar `execute()` o `submit()` . la diferencia entre ellos es que `submit()` devuelve un objeto `Future` que le permite una manera de cancelar programáticamente el subproceso en ejecución cuando el objeto T se devuelve de la `Callable` llamada `Callable` . Puedes leer más sobre el `Future` aquí

Futuro y callables

Una de las características que podemos usar con Threadpools es el método `submit()` que nos permite saber cuándo el hilo termina su trabajo. Podemos hacerlo gracias al objeto `Future` , que nos devuelve un objeto de Callable que podemos utilizar para nuestros propios objetivos.

Aquí hay un ejemplo sobre cómo usar la instancia invocable:

```
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();
```

```
    }
}
}
```

Como puedes ver, creamos un Threadpool con 5 hilos, esto significa que podemos lanzar 5 callables paralelos. Cuando los hilos terminen, obtendremos un objeto Future del llamable, en este caso el nombre del hilo.

ADVERTENCIA

En este ejemplo, solo usamos los Futuros como un objeto dentro de la matriz para saber cuántos subprocesos estamos ejecutando e imprimimos muchas veces un registro en la consola con los datos que deseamos. Pero, si queremos usar el método `Future.get()` , para devolvernos los datos que guardamos antes en el llamable, bloquearemos el hilo hasta que se complete la tarea. Tenga cuidado con este tipo de llamadas cuando desee realizar esto lo más rápido posible

Runnables personalizados en lugar de Callables

Otra buena práctica para verificar cuándo nuestros hilos han finalizado sin bloquear el hilo en espera de recuperar el objeto Future de nuestro Callable es crear nuestra propia implementación para Runnables, usándola junto con el método `execute()` .

En el siguiente ejemplo, muestro una clase personalizada que implementa Runnable con una devolución de llamada interna, que nos permite saber cuándo finalizan los runnables y usarlos más adelante en nuestro 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);
    }
}
```

Y aquí está nuestra implementación ThreadExecutor:

```
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);

}
```

Hice este ejemplo para verificar la velocidad de mis hilos en milisegundos cuando se ejecutan, sin usar Future. Puede tomar este ejemplo y agregarlo a su aplicación para controlar el trabajo

simultáneo de la tarea y las tareas completadas / terminadas. Comprobando en todo momento, el tiempo que necesitabas para ejecutar esos hilos.

Agregando ThreadFactory al Ejecutor

Usamos ExecutorService para asignar subprocesos desde el grupo de subprocesos internos o crearlos a pedido para realizar tareas. Cada ExecutorService tiene un ThreadFactory, pero El ExecutorService usará siempre uno predeterminado si no configuramos uno personalizado. ¿Por qué deberíamos hacer esto?

- Para establecer un nombre de hilo más descriptivo. El ThreadFactory predeterminado proporciona nombres de subprocesos en la forma de pool-m-thread-n, como pool-1-thread-1, pool-2-thread-1, pool-3-thread-1, etc. Si está intentando depurar o monitorear algo, es difícil saber qué hacen esos hilos
- Establezca un estado de Daemon personalizado, el ThreadFactory predeterminado produce resultados que no son de daemon.
- Establecer prioridad para nuestros subprocesos, el ThreadFactory predeterminado establece una prioridad media para todos sus subprocesos.
- Puede especificar `UncaughtExceptionHandler` para nuestro hilo usando `setUncaughtExceptionHandler()` en el objeto de hilo. Esto se recupera cuando el método de ejecución de Thread lanza una excepción no detectada.

Aquí hay una implementación fácil de un ThreadFactory sobre un 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);

}
```

Este ejemplo simplemente modifica el nombre del subproceso con un contador, pero podemos modificarlo tanto como queramos.

Lea Ejecutores en línea: https://riptutorial.com/es/multithreading/topic/6710/ejecutores

Capítulo 3: Semáforos y Mutexes

Introducción

Los semáforos y Mutexes son controles de concurrencia que se utilizan para sincronizar el acceso de múltiples subprocesos a los recursos compartidos.

Observaciones

Semáforo

Aquí hay una brillante explicación de esta pregunta de Stackoverflow :

> Piense en los semáforos como en un club nocturno. Hay un número dedicado de personas que se permiten en el club a la vez. Si el club está lleno, a nadie se le permite entrar, pero tan pronto como una persona abandona, otra persona puede ingresar.
>
> Es simplemente una forma de limitar el número de consumidores para un recurso específico. Por ejemplo, para limitar el número de llamadas simultáneas a una base de datos en una aplicación.

Mutex

Un mutex es un semáforo de 1 (es decir, solo un hilo a la vez). Usando la metáfora del club nocturno, piense en una exclusión mutua en términos de un puesto de baño en el club nocturno. Sólo se permite un ocupante a la vez.

Examples

Mutex en Java y C ++

Aunque Java no tiene una clase Mutex, puede imitar un Mutex con el uso de un Semáforo de 1. El siguiente ejemplo ejecuta dos subprocesos con y sin bloqueo. Sin bloqueo, el programa escupe un orden algo aleatorio de caracteres de salida ($ o #). Con el bloqueo, el programa escupe juegos de caracteres agradables y ordenados de ##### o $$$$$, pero nunca una mezcla de # & $.

```
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');
   }
}
```

Ejecutar `javac MutexTest.java; java MutexTest` , y obtendrás algo como esto:

Sin bloqueo: # $$$$$ | $$$$$ | $$ # $$$ | $$$$$ | $$$$ # $ | $$$$$ | $$$$$ | $ # $$$$ | $$$$$ | $$$ # $$ || ##### | ##### | ##### | ##### | ##### | # #### | ##### | ##### | ##### |

Con bloqueo: $$$$$ | ##### | $$$$$ | ##### | $$$$$ | #### | | $$$$$ | ##### | $$$$$ | ##### | $$$$$ | ##### | $$$$$ | ##### | $$$$$ | ##### | $ $$$$ | ##### | $$$$$ | ##### |

Aquí está el mismo ejemplo en 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;
}
```

Ejecute `g++ --std=c++11 MutexTest.cpp; ./a.out` , y obtendrás algo como esto:

> Sin bloqueo: $ # $ # $ # $ # $ # | $ | # $ # $ # $ # $ # | # | # $ # $ # $ # | $ # $ # $ # $ # $ # | $ $ # $ | # $ # $ # | $ # $ # $ | # $ # $ # | $ # $$ # $ | # $ # | $ # $ # $ # $ | # $ # | $ # $ # # $$ | # | $ # $ # $ # $ # $ | # | #### |
>
> Con bloqueo: $$$$$ | ##### | $$$$$ | ##### | $$$$$ | #### | | $$$$$ | ##### | $$$$$ | ##### | $$$$$ | ##### | $$$$$ | ##### | $$$$$ | ##### | $ $$$$ | ##### | $$$$$ | ##### |

Lea Semáforos y Mutexes en línea:
https://riptutorial.com/es/multithreading/topic/10861/semaforos-y-mutexes

Creditos

S. No	Capítulos	Contributors
1	Empezando con multihilo	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	Ejecutores	Francisco Durdin Garcia, Guillermo Orellana Ruiz, vvtx
3	Semáforos y Mutexes	John DiFini