Asignación de Prácticas Número 9

Programación Concurrente y de Tiempo Real

Departamento de Ingeniería Informática
Universidad de Cádiz

PCTR, 2022

Contenido I

Objetivos de la Práctica

- Aprender a controlar secciones críticas con cerrojos `ReentrantLock` y con `Semaphore`.
- Construir objetos seguros con cerrojos `ReentrantLock`.
- Revisitar brevemente la clase `CyclicBarrier`.
- Implementar monitores ya conocidos con el API de alto nivel: cerrojos `ReentrantLock`+`Condition`.
- Comparar las diferentes técnicas de control de la exclusión mutua en términos de «rapidez».

Control De Exclusión Mutua con Cerrojos ReentrantLock I

- Poseen igual comportamiento y semántica que los cerrojos implícitos disponibles mediante synchronized, pero añaden capacidades nuevas.
- La posesión de estos cerrojos sigue siendo por hebra.
- Permiten generar instancias Condition para lograr monitores con una arquitectura más parecida al modelo teórico estándar con variables de condición.

Ejemplo de Control con Cerrojos ReentrantLock I

```
1  import java.util.concurrent.locks.ReentrantLock;
2
3  public class secureCriticalSectionRLock
4    extends Thread{
5    public static long iterations = 4000000;
6    public static long n              = 0;
7    public static ReentrantLock lock = new ReentrantLock();
8
9    public secureCriticalSectionRLock(){}
10
11   public void run(){
12     for(long i=0; i<iterations; i++){
13       lock.lock();
14       try {n++;} finally {
15        lock.unlock();
16      }
17    }
18   }
19
20   public static void main(String[] args) throws Exception{
21     secureCriticalSectionRLock A = new
           secureCriticalSectionRLock();
```

Ejemplo de Control con Cerrojos ReentrantLock II

```
22        secureCriticalSectionRLock B = new
              secureCriticalSectionRLock();
23        A.start(); B.start();
24        A.join(); B.join();
25        System.out.println(n);
26      }
27    }
```

Objetos Seguros con Cerrojos ReentrantLock I

```
1  import java.util.concurrent.locks.*;
2
3  public class secureObject{
4    public long       n = 0;
5    public ReentrantLock lock = new ReentrantLock();
6
7    public secureObject(){}
8
9    public void inc(){
10     lock.lock();
11     try{n++;}
12     finally{lock.unlock();}
13   }
14
15   public long get(){
16     lock.lock();
17     try{return this.n;}
18     finally{lock.unlock();}
19   }
20 }
```

Usando un Objeto Seguro con Cerrojos ReentrantLock I

```
1  public class usingSecureObject
2    extends Thread{
3    public static long iterations = 1000000;
4    public secureObject obj;
5
6    public usingSecureObject(secureObject obj){this.obj = obj;}
7    public void run(){
8      for(long i=0; i<iterations; i++)obj.inc();
9    }
10
11   public static void main(String[] args) throws Exception{
12     secureObject o      = new secureObject();
13     usingSecureObject A = new usingSecureObject(o);
14     usingSecureObject B = new usingSecureObject(o);
15     A.start(); B.start();
16     A.join(); B.join();
17     System.out.println(o.get());
18   }
19 }
```

Ejemplo de Control con Semaphore I

```
1  import java.util.concurrent.Semaphore;
2
3  public class secureCriticalSectionSem
4    extends Thread{
5    public static long iterations = 4000000;
6    public static long n          = 0;
7    public static Semaphore sem  = new Semaphore(1);
8
9    public void run(){
10     for(long i=0; i<iterations; i++){
11         try{sem.acquire();}catch(InterruptedException e){}
12   try{n++;}
13   finally{sem.release();}
14     }
15   }
16
17   public static void main(String[] args) throws Exception{
18     secureCriticalSectionSem A = new secureCriticalSectionSem();
19     secureCriticalSectionSem B = new secureCriticalSectionSem();
20     A.start(); B.start();
21     A.join(); B.join();
22     System.out.println(n);
```

Ejemplo de Control con Semaphore II

```
23          }
24      }
```

¿Qué Hago Ahora? I

- Recuperamos el modelo de cuenta corriente que diseñó en asignaciones anteriores.
- Rediseñamos la clase para que sus objetos sean seguros frente a concurrencia con cerrojos ReentrantLock y con Semaphore
- Escribimos un diseño de hebras de prueba...
- ... comprobamos que el saldo final es coherente.

Barreras Cíclicas I

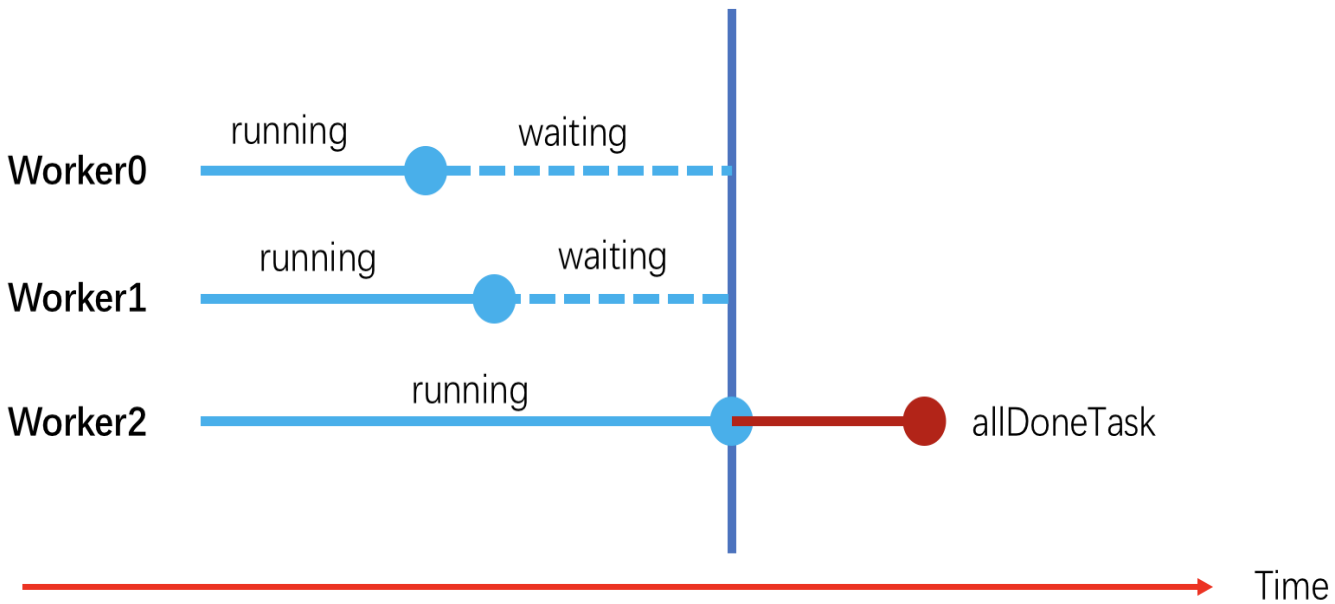

Figura: Barrera para tres hebras...

¿Qué Hago Ahora? I

- Escribimos un programilla que implante el gráfico anterior.
- Tres hebras compartirán una barrera que se abrirá cuando todas lleguen a la misma
- Comprobamos que la barrera se abre cuando debe (llegan tres hebras)...
- ... pero permenece cerrada cuando llegan menos de tres.

Monitores Java con API de Alto Nivel I

- Son objetos de una clase en la cuál todos los métodos está en exclusión mutua bajo el control de un cerrojo ReentranLock común a todos los métodos.
- Proveen sincronización mediante variables de condición que se pueden instanciar a partir del cerrojo ReentrantLock mediante el método `newCondition()`
- Las variables de condición se gestionan mediante los métodos `await()`, `signal()` y `signalAll()`. Vemos en API.
- Esto permite construir monitores más cercanos al concepto teórico.
- Sigue siendo necesario el uso de condiciones de guarda.

Simulando un Semáforo con un Monitor Java y el API de Alto Nivel I

```
1  import java.util.concurrent.locks.*;
2
3  public class Sem{
4    private int s;
5    final ReentrantLock lock = new ReentrantLock();
6    final Condition  semZero = lock.newCondition();
7
8    public Sem(int s){
9      this.s = s;
10   }
11   public void waitS() throws InterruptedException{
12     lock.lock();
13     try{
14       while(s==0)semZero.await();
15       s=s-1;
16     }finally {lock.unlock();}
17   }
18
19   public void signalS() throws InterruptedException{
20     lock.lock();
```

Simulando un Semáforo con un Monitor Java y el API de Alto Nivel II

```
21        try{
22          s=s+1;
23          semZero.signalAll();
24         }finally {lock.unlock();}
25      }
26    }
```

Y Usándolo Para Controlar Una Sección Crítica I

```
1  public class usaSem
2    extends Thread{
3
4    public static long n = 0;
5    public Sem semaforo;
6
7    public usaSem(Sem semaforo){this.semaforo=semaforo;}
8
9    public void run(){
10     for(long i=0; i<40000000; i++){
11       try{semaforo.waitS();}catch (InterruptedException e){}
12       n++;
13       try{semaforo.signalS();}catch (InterruptedException e){}
14     }
15   }
16
17   public static void main(String[] args) throws Exception{
18     Sem mon_semaforo = new Sem(1);
19     usaSem A         = new usaSem(mon_semaforo);
20     usaSem B         = new usaSem(mon_semaforo);
21     A.start(); B.start();
22     A.join(); B.join();
```

Y Usándolo Para Controlar Una Sección Crítica II

```
23          System.out.println(n);
24        }
25    }
```

¿ Qué Hago Ahora? I

- **Leer el protocolo para escribir monitores en Java con el API de alto nivel, disponible en la carpeta de la práctica.**
- Recuperamos el código ya desarrollado para el problema de los lectores-escritores (práctica 8).
- Lo reescribimos utilizando el API de alto nivel, con cerrojos `ReentrantLock` y variables de condición `Condition`.
- Observamos que el «mapping» monitor teórico $\rightarrow$ Monitor Java es ahora más directo con este API.

Comparando Técnicas de Bloqueo I

```
1  import java.util.concurrent.*;
2
3  public class timing{
4
5    public static long f(long iter){
6      int n = 0;
7      Semaphore s = new Semaphore(1);
8      long ini=System.nanoTime();
9      for(long i=0; i<iter; i++){
10   try{s.acquire();}catch(InterruptedException e){}
11   try{n++;}
12   finally{s.release();}
13     }
14    long fin=System.nanoTime();
15    return(fin-ini);
16   }
17
18 public static long g(long iter){
19     int n = 0;
20     Object o = new Object();
21     long ini=System.nanoTime();
22     for(long i=0; i<iter; i++){
```

Comparando Técnicas de Bloqueo II

```
23      synchronized(o){n++;}
24        }
25       long fin=System.nanoTime();
26       return(fin-ini);
27      }
28      public static void main(String[] args){
29        long it = 100000000;
30        System.out.println("Tiempo para semaforos   : "+f(it)+"
              nanosegundos...");
31        System.out.println("Tiempo para synchronized: "+g(it)+"
              nanosegundos...");
32      }
33
34  }
```

¿Qué Hago Ahora? I

- Añadimos al código anterior lo necesario para evaluar el resto de técnicas de bloqueo pedidas.
- Tomamos tiempos para un número creciente de iteraciones.
- Construimos curvas con los puntos obtenidos, a fin de determinar qué técnica se «comporta mejor».