Práctica 2

David Gutiérrez Villar
2º DAM ESICA
PSP Práctica 2

Ejercicio 1

Se denomina sección crítica o región crítica, en programación concurrente de ciencias de la computación, a la porción de código de un programa de ordenador en la que se accede a un recurso compartido que no debe ser accedido por más de un proceso o hilo en ejecución. La región crítica termina en un tiempo determinado y el hilo, proceso o tarea sólo tiene que esperar ese período de tiempo para entrar.

El uso adecuado de la concurrencia entre procesos exige la capacidad de definir secciones críticas y hacer cumplir la exclusión mutua. Cuando ningún proceso está en su sección crítica, cualquier proceso que solicite entrar en la suya debe poder hacerlo sin dilatación. Que los procesos sólo puedan estar en su región crítica por un tiempo finito es para evitar que se queden con un recurso por mucho tiempo y para que un recurso no se quede trabado sin sentido.

Ejercicio 2

La instrucción test-and-set (TAS) implementa una comprobación del contenido de una variable en la memoria al mismo tiempo que varía su contenido en caso que la comprobación de éxito. En caso de un sistema multi-procesador hay que tener cuidado que la operación test-and-set esté realizada cerca de la memoria compartida.

Para conseguir una espera limitada se implementa un protocolo de paso de tal manera que un proceso saliendo de su sección crítica da de forma explícita paso a un proceso esperando.

Ejercicio 3

Es un algoritmo de programación concurrente para exclusión mutua, que permite a dos o más procesos o hilos de ejecución compartir un recurso sin conflictos, utilizando la memoria compartida para la comunicación. Un ejemplo de principio de la bandera es el algoritmo de Peterson o solución de Peterson. Que utiliza dos variables para resolver el problema de la entrada a la zona crítica, el turno y la bandera. Ejemplificando con dos procesos, p0 y p1, cada uno se ejecuta en un hilo distinto, cuando p0 levanta la bandera, p1 la baja y espera a que p0 la baje para entrar él en la zona crítica.

Ejercicio 4

La espera activa es una técnica en la cual un proceso comprueba continuamente si una condición se cumple o si se produce un evento. En esta espera se hace la comprobación contraria a la esperada, por ejemplo, para comprobar si es nuestro turno comprobamos en un bucle que no es nuestro turno:

While (turno!=myTurno){}

De esta forma mientras se cumpla que no es nuestro turno no continua el código porque no sale del bucle.

En el caso de tener dos procesos podemos tener un ejemplo como este:

Proceso 1	Proceso 2
while (turno!=turno1){}	While (turno!=turno2){}
//Sección crítica	//Sección crítica
turno=turno2;	turno=turno1;

Este ejemplo se podría aplicar a más de dos procesos.

Esta espera produce esperas innecesarias entre los procesos que no tienen el turno, además de que la alternancia estricta es un problema grave y provoca un buen número de bloqueos. Esta alternancia estricta se debe a que el ritmo lo marcará el proceso más lento y si los procesos tienen distinto número de accesos a la sección crítica, al terminar el que menos accesos solicita y no entregar turno, el resto quedan bloqueados.

Además del ejemplo propuesto, hay más posibles ejemplo cada cual un poco más complejo sobre la espera activa.

Ejercicio 5

La condición de carrera es un comportamiento del software en el cual la salida depende del orden de ejecución de eventos que no se encuentran bajo control.

Este comportamiento se puede producir cuando varios hilos no acceden en exclusión mutua a un recurso compartido. Es un fallo siempre que el orden de ejecución no sea el esperado. Este concepto viene de la idea de dos procesos compitiendo en una carrera para acceder al recurso compartido. El comportamiento de un programa con una condición de carrera es imprevisible, es difícil de detectar,

difícil de reproducir y difícil de depurar. La solución más común es prevenirlas mediante una sincronización adecuada.

Existen dos tipos de condiciones de carrera:

- Check-then-act:

```
public Singleton getInstance (){
        if(instance == null){
                instance = new Singleton ();
        }
}
```

- read-modify-write:

```
if (x == 0) {
        x++;
}
```

Las condiciones de carrera se producen al acceder o modificar la memoria compartida.

Ejercicio 6

La palabra volatile es útil cuando existen muchos hilos intentando acceder a un mismo valor de una variable. Cuando una variable es declarada como volatile, el valor de esa variable es escrita y es leída desde la memoria principal, es decir, los hilos no hacen una copia en su memoria caché de la variable sino que leen y escriben directamente en la memoria principal. Hay que resaltar que el escribir y leer de la memoria principal es más costoso por tanto impacta en el performance de la aplicación, por tanto hay que saber cuando y donde usar volatile.

La forma de declarar una variable volatile es:

```
public volatile int contador=0;
```

Ejercicio 7

Los monitores son un mecanismo de alto nivel para programación concurrente, podríamos decir que un monitor es una instancia de una clase que puede ser usada de forma segura por múltiples hilos. Todos los métodos de un monitor deben ejecutarse en exclusión mutua, es decir, habrá un solo proceso accediendo a memoria compartida.

Los monitores presentan ventajas frente a mecanismo de bajo nivel como la espera activa, etc.

Algunas de las ventajas que presenta son:

- No depende del número de procesos que accedan.
- Todos los accesos de memoria compartida se encuentran dentro de una misma clase.
- El “cliente” del monitor solo necesita conocer el interfaz del recurso.

La clase que implementa un monitor debe ser de este estilo:

```
class Parking {
        private Monitor mutex;
        public Parking () {
                mutex = new Monitor ();
                ...
        }
        …
}
```

Ejercicio 8

Clase future:

La clase future es una interfaz que se usa para poder evitar bloqueos. Un futuro representa el resultado de un cálculo asincrónico. Se proporcionan métodos para verificar si el cálculo se ha completado, esperar su finalización y recuperar el resultado del cálculo. El resultado solo se puede recuperar utilizando el método get cuando el cálculo se ha completado, bloqueando si es necesario hasta que esté listo. La cancelación se realiza por el método de cancelación . Se proporcionan métodos adicionales para determinar si la tarea se completó normalmente o se canceló. Una vez que se ha completado un cálculo, el cálculo no se puede cancelar.

Ejemplo de uso:

```
interface ArchiveSearcher { String search(String target); }
        class App {
         ExecutorService executor = ...
        ArchiveSearcher searcher = ...
        void showSearch(final String target)
                 throws InterruptedException {
         Future<String> future
         = executor.submit(new Callable<String>() {
                 public String call() {
                 return searcher.search(target);
         }});
         displayOtherThings(); // do other things while searching
         try {
                 displayText(future.get()); // use future
         } catch (ExecutionException ex) { cleanup(); return; }
         }
}
```

Clase CountDownLatch:

Esta clase permite sincronizar entre hilos, lo que permite esperar por uno o más hilos, también permite hacer lo mismo que con el wait and notify pero de una forma más sencilla. CountDownLatch bloquea todos los hilos hasta que el conteo llegue a cero, cuando esto ocurre todos los hilos de espera se vuelven a habilitar.

Ejemplo de uso:

```
Conductor de clase {// ...
  void main () lanza InterruptedException {
    CountDownLatch startSignal = new CountDownLatch (1);
    CountDownLatch doneSignal = new CountDownLatch (N);

    for (int i = 0; i <N; ++ i) // crea y comienza hilos
      Nuevo hilo (nuevo trabajador (startSignal, doneSignal)). start ();

    hacer algo más(); // no dejes correr todavía
    startSignal.countDown (); // deja que todos los hilos continúen
    hacer algo más();
    doneSignal.await (); // espera a que todos terminen
  }
}

El trabajador de clase implementa Runnable {
  Privado final CountDownLatch startSignal;
  Privado final CountDownLatch doneSignal;
  Trabajador (CountDownLatch startSignal, CountDownLatch doneSignal) {
    this.startSignal = startSignal;
    this.doneSignal = doneSignal;
  }
  public void run () {
    tratar {
      startSignal.await ();
      hacer trabajo();
```

```
      doneSignal.countDown ();
    } catch (InterruptedException ex) {} // return;
  }

  anular doWork () {...}
}
```

Clase ExecutorService:

Proporciona métodos para administrar la terminación y métodos que pueden producir un Future por el seguimiento del progreso de una o más tareas asincrónicas.

Un ExecutorService puede cerrarse, lo que hará que rechace nuevas tareas. Se proporcionan dos métodos diferentes para cerrar un ExecutorService . El método shutdown() permitirá que las tareas enviadas previamente se ejecuten antes de finalizar, mientras que el método shutdownNow() evita que las tareas en espera comiencen e intenta detener las tareas actualmente en ejecución. Al finalizar, un ejecutor no tiene tareas ejecutándose activamente, no hay tareas en espera de ejecución y no se pueden enviar nuevas tareas. Un ExecutorService no utilizado debe cerrarse para permitir la recuperación de sus recursos.

El envío de método extiende el método base Executor.execute (java.lang.Runnable) creando y devolviendo un Future que se puede utilizar para cancelar la ejecución y / o esperar a que se complete. Los métodos invokeAny e invokeAll realizan las formas más útiles de ejecución masiva, ejecutan una colección de tareas y luego esperan que se complete al menos una o todas. (La clase ExecutorCompletionServicese puede usar para escribir variantes personalizadas de estos métodos).

La clase Executors proporciona métodos de fábrica para los servicios de ejecución proporcionados en este paquete.

Ejemplo de uso:

```
class NetworkService implementa Runnable {
  privado final ServerSocket serverSocket;
  grupo privado final de ExecutorService;
```

```
  Public NetworkService (int port, int poolSize)
      lanza IOException {
    serverSocket = new ServerSocket (port);
    pool = Executors.newFixedThreadPool (poolSize);
  }
  public void run () {// ejecuta el servicio
    try {
      for (;;) {
        pool.execute (new Handler (serverSocket.accept ()));
      }
    } catch (IOException ex) {
      pool.shutdown ();
    }
  }
}
 class Handler implementa Runnable {
  socket de socket final privado;
  Handler (Socket socket) {this.socket = socket; }
  public void run () {
    // solicitud de lectura y servicio en el socket
  }
}
```

Clase Semaphore:

Conceptualmente, un semáforo mantiene un conjunto de permisos. Cada adquirir () bloquea si es necesario hasta que haya un permiso disponible, y luego lo toma. Cada lanzamiento () agrega un permiso, potencialmente liberando a un adquirente de bloqueo. Sin embargo, no se utilizan objetos de permiso reales; el semáforo sólo mantiene un recuento del número disponible y actúa en consecuencia.

Ejemplo de uso:

```
grupo de clase {
  privado estático final int MAX_AVAILABLE = 100;
```

```
      semáforo final privado disponible = nuevo semáforo (MAX_AVAILABLE, verdadero);
  public Object getItem () lanza InterruptedException {
    disponible.acquire ();
    return getNextAvailableItem ();}
  public void putItem (Object x) {
    if (markAsUnused (x))
      disponible.release (); }
  // No es una estructura de datos particularmente eficiente; solo para demostración
   Objetos protegidos [] elementos = ... cualesquiera que sean los tipos de elementos gestionados
  booleano protegido [] usado = nuevo booleano [MAX_AVAILABLE];
  Objeto sincronizado protegido getNextAvailableItem () {
    para (int i = 0; i <MAX_AVAILABLE; ++ i) {
      if (! used [i]) {
        utilizado [i] = verdadero;
        devolver artículos [i];
      } }
    volver nulo; // no alcanzado
  }
  marca booleana sincronizada protegidaAsUnused (elemento de objeto) {
    para (int i = 0; i <MAX_AVAILABLE; ++ i) {
      if (item == items [i]) {
        if (usado [i]) {
          utilizado [i] = falso;
          volver verdadero;
        } más
          falso retorno;
      } }
    falso retorno;
  } }
```

Clase BlockingQueue:

Esta clase representa una cola que es segura para subprocesos para colocar elementos y sacar elementos de ellos. En otras palabras, múltiples hilos pueden estar insertando y tomando elementos simultáneamente de un Java BlockingQueue, sin que surjan problemas de concurrencia.

El término cola de bloqueo proviene del hecho de que Java es capaz de bloquear los hilos que intentan insertar o tomar elementos de la cola. Por ejemplo, si un subproceso intenta tomar un elemento y no queda ninguno en la cola, el subproceso puede bloquearse hasta que haya un elemento para tomar. Si el hilo de llamada está bloqueado o no depende de los métodos que llame en el .

Ejemplo de uso:

```
El productor de clase implementa Runnable {
  cola privada final de BlockingQueue;
  Productor (BlockingQueue q) {queue = q; }
  public void run () {
    tratar {
      while (verdadero) {queue.put (produce ()); }
    } catch (InterruptedException ex) {... manejar ...}
  }
  Objeto producir () {...}
}
El consumidor de clase implementa Runnable {
  cola privada final de BlockingQueue;
  Consumidor (BlockingQueue q) {queue = q; }
  public void run () {
    tratar {
      while (verdadero) {consume (queue.take ()); }
    } catch (InterruptedException ex) {... manejar ...}
  }
  vacío consumir (Objeto x) {...}
}
Configuración de clase {
```

```
  vacío principal() {
    BlockingQueue q = new SomeQueueImplementation ();
    Productor p = nuevo productor (q);
    Consumidor c1 = nuevo Consumidor (q);
    Consumidor c2 = nuevo Consumidor (q);
    nuevo hilo (p) .start ();
    Hilo nuevo (c1) .start ();
    Hilo nuevo (c2) .start (); } }
```

Clase AtomicInteger:

Esta clase representa un valor entero sobre el que se definen unos cuantos métodos con la garantía de que están sincronizados. Hay dos usos principales para esta clase:

1. Como contador que puede ser usado por muchos hilos simultáneamente.
2. Como una primitiva que admite la instrucción compareAndSet() para implementar algoritmos de no bloqueo.

Ejemplo de uso como generador de números aleatorios sin bloqueo:

```
public class AtomicPseudoRandom extends PseudoRandom {
   private AtomicInteger seed;
   AtomicPseudoRandom(int seed) {
      this.seed = new AtomicInteger(seed);
   }
   public int nextInt(int n) {
      while (true) {
         int s = seed.get();
         int nextSeed = calculateNext(s);
         if (seed.compareAndSet(s, nextSeed)) {
            int remainder = s % n;
            return remainder > 0 ? remainder : remainder + n;
         }
      }
   }
   ...
}
```