Programación Concurrente Y De Tiempo Real

Guión De Prácticas 6: Programación En Java De Algoritmos De Control De La Exclusión Mutua Con Variables Comunes

Natalia Partera Jaime
Manuel Francisco
Alumnos Colaboradores De La Asignatura

Índice

1. Introducción **2**

2. Búsqueda De Un Algoritmo De Control De La Exclusión Mutua **2**
2.1. Primer Intento 2
2.2. Segundo Intento 4
2.3. Tercer Intento 5
2.4. Cuarto Intento 7
2.5. Algoritmo De Dekker 9

3. Variables `volatile` **10**

4. Ejercicios **11**

5. Soluciones De Los Ejercicios **12**
5.1. Ejercicio 1 12

1. Introducción

En ocasiones, al ejecutar procesos concurrentemente, pueden aparecer errores derivados de no controlar la concurrencia en ciertos fragmentos críticos del programa. Para conseguir que dichos hilos trabajen como está previsto y aporten el resultado esperado, debemos ejecutarlos en exclusión mutua, y una forma de lograr ésta es mediante variables comunes.

Se establecen para ello unas *condiciones de exclusión mutua*, que habrán de cumplirse para la correcta ejecución de la *sección crítica*. Para un número N de hilos, es necesario:

- Identificar y diferenciar la sección crítica del resto del código.
- Asegurar que, en un mismo instante de tiempo, sólo haya un hilo ejecutando su sección crítica. Para ello, cada proceso debe ejecutar un *pre-protocolo* antes de entrar en su sección crítica y un *post-protocolo* al salir.

Las condiciones se mantienen inmutables aunque cambie el número de hilos o procesos concurrentes. Lo único que podemos necesitar es cambiar nuestro código para satisfacerlas.

Uno de los algoritmos que solucionan el problema de la exclusión mutua es el algoritmo de *Dekker*, el cual le detallamos más adelante (para 2 tareas, pero perfectamente extrapolable a N hebras).

2. Búsqueda De Un Algoritmo De Control De La Exclusión Mutua

Utilizaremos variables comunes a ambos para controlar la exclusión mutua. Trabajaremos [1] con 2 procesos para que el código sea más sencillo.

2.1. Primer Intento

Ya hemos mencionado que para controlar la exclusión mutua es necesario que los procesos concurrentes tengan un *pre-protocolo* y un *post-protocolo* que se ejecuten antes y después de su sección crítica. Lo más sencillo entonces es pensar en una variable global que identifique el siguiente hilo que ha de entrar (o lo que es lo mismo, un turno).

Pues bien, podemos crear una variable entera global `turno` que indique si es el turno del hilo 1 o del hilo 2. Cada hilo comprueba en su *pre-protocolo* si es su turno, observando la variable `turno`, y, si as así, continúa con la ejecución de su sección crítica y el *post-protocolo*. Si no fuera su turno, entraría en un bucle hasta que llegase suyo (*condición de guarda*). Tras ejecutar la sección crítica, hará lo mismo con el *post-protocolo*, en el que cambiará el valor de la variable turno para que dar paso a la siguiente hebra.

Veamos el código:

```
1  /* Adaptado de M. Ben-Ari por Manuel Francisco */
2
3  /* Primer intento */
4  class First {
5      /* Iteraciones que dara cada hilo */
6      static final int iteraciones = 2000000;
7      /* Recurso compartido */
8      public static volatile int enteroCompartido = 0;
9      /* Representa de quien es el turno */
10     static volatile int turn = 1;
11
12     class P extends Thread {
13         public void run() {
14             for (int i=0; i<iteraciones; ++i) {
15                 /* Seccion no critica */
16                 while (turn != 1)
17                     Thread.yield();
18
19                 /* Seccion critica */
20                 ++enteroCompartido;
21                 /* Fin Seccion critica */
22
23                 turn = 2;
24             }
25         }
26     }
27
28     class Q extends Thread {
29         public void run() {
30             for (int i=0; i<iteraciones; ++i) {
31                 /* Seccion no critica */
32                 while (turn != 2)
33                     Thread.yield();
34
35                 /* Seccion critica */
36                 --enteroCompartido;
37                 /* Fin Seccion critica */
38
39                 turn = 1;
40             }
41         }
42     }
43
44     First() {
45         Thread p = new P();
46         Thread q = new Q();
47         p.start();
48         q.start();
49
50         try {
51             p.join();
52             q.join();
53             System.out.println("El valor del recurso compartido es " + enteroCompartido);
54             System.out.println("Deberia ser 0.");
55         }
56         catch (InterruptedException e) {}
57     }
58
59     public static void main(String[] args) {
60         new First();
61     }
62 }
```

volatile: para cuando una hebra lo modifique, el resto de las hebras vean el cambio

variable compartida turno, y antes de hacer nada el preprotocolo es mientras no sea mi turno, estaremos en while. yield es mas efidiente ya que no tenemos ocupado el procesador todo el while.

dos hebras p y q que heredan de thread y pueden tener un metodo run

Tiene un bucle que realiza un n++ y n-- respectivamente. Asi entrarian en conflicto sin preprotocolo

la entrada a la seccion critica se realiza en exclusion mutua, obligamos a la alternancia, pero eso es un problema, ya que si el numero de iteraciones es distinto, si uno termina antes que otro, no devuelve el turno a el otro hilo, y este no terminaria todas las iteraciones que tiene.
Cuando haya un conflicto de ENTRADA, se podria hacer una alternancia.

El principal problema que esta aproximación presenta es que obliga a que haya alternancia entre ambos hilos (o que se ejecuten en un cierto orden, en el caso de tener N hilos). Por si fuera poco, si uno de los dos hilos acaba, el otro se bloquea, ya que el primero no le devolverá el turno.

2.2. Segundo Intento

Intentaremos solucionar el problema anterior añadiendo más variables de control. Ahora cada hilo tendrá su propia variable que reflejará si los demás pueden usar el recurso crítico o no. Las variables de control son ahora lógicas (`boolean`).

Cuando un hilo quiere entrar en su sección crítica, coloca su variable a `false`, indicando así a los demás hilos que no pueden acceder a la sección crítica. Una vez que el hilo llega al *post-protocolo* coloca su variable de control a `true`, indicando que el recurso crítico está libre. Si un hilo desea ejecutar su sección crítica, comprobará que todas las demás variables de control sean distintas de 0. Si no es así, espera.

Veamos cómo quedaría el código:

```
1  /* Adaptado de M. Ben-Ari por Manuel Francisco */
2
3  /* Segundo intento */
4  class Second {
5          /* Iteraciones que dara cada hilo */
6      static final int iteraciones = 2000000;
7      /* Recurso compartido */
8      static volatile int enteroCompartido = 0;
9      /* Representa el deseo del hilo P de entrar en la seccion critica */
10     static volatile boolean wantp = false;
11     /* Representa el deseo del hilo Q de entrar en la seccion critica */
12     static volatile boolean wantq = false;
13
14     class P extends Thread {
15         public void run() {
16             for (int i=0; i<iteraciones; ++i) {
17                 /* Seccion no critica */
18                 while (wantq)
19                     Thread.yield();
20                 wantp = true;
21
22                 /* Seccion critica */
23                 ++enteroCompartido;
24                 /* Fin Seccion critica */
25
26                 wantp = false;
27             }
28         }
29     }
30
31     class Q extends Thread {
32         public void run() {
33             for (int i=0; i<iteraciones; ++i) {
34                 /* Seccion no critica */
35                 while (wantp)
36                     Thread.yield();
37                 wantq = true;
38
39                 /* Seccion critica */
40                 --enteroCompartido;
41                 /* Fin Seccion critica */
42
43                 wantq = false;
```

Variables logicas en vez de las anteriores

la hebra se pregunta si hay alguien que quiere entrar (q en este caso), si es falso, esta entra aqui cede al procesador

el planificador decide a quien decede la ALU y podria ocurrir que lo haga antes de que a wantp se le de el valor TRUE

podria pasar que p aun no no haya colocado que quiere entrar, y este hilo entre sin que p haya cambiado la variable logica.

```
44              }
45          }
46      }
47
48      Second() {
49          Thread p = new P();
50          Thread q = new Q();
51          p.start();
52          q.start();
53
54          try {
55              p.join();
56              q.join();
57              System.out.println("El valor del recurso compartido es " + enteroCompartido);
58              System.out.println("Deberia ser 0.");
59          }
60          catch (InterruptedException e) {}
61      }
62
63      public static void main(String[] args) {
64          new Second();
65      }
66  }
```

Desgraciadamente, este intento no cumple con los requisitos de exclusión mutua. Puede darse el caso de que los hilos se intercalen al comprobar las demás variables de control, no viendo así las modificaciones que pueden estar haciendo otros hilos sobre ellas, es decir:

1. P_1 comprueba C_2 y encuentra que $C_2 = 1$.
2. P_2 comprueba C_1 y encuentra que $C_1 = 1$.
3. P_1 pone C_1 a 0.
4. P_2 pone C_2 a 0.
5. Ambos procesos entran en sus secciones críticas.

2.3. Tercer Intento

Para evitar la situación anterior, se puede cambiar el orden en el que se espera en el bucle y se cambia la variable de control. Primero, los hilos declararán su intención (cambiando su variable) y luego esperarán a poder entrar en la sección crítica. De este modo sí se asegura que se cumplan los requisitos.

Cuando un hilo llega a su *pre-protocolo* anuncia su intención de ejecutar su sección crítica cambiando el valor de su variable de control a `false`, queda a la espera en un bucle hasta que las demás variables de control valgan `true`. En ese momento el hilo entra en su sección crítica. En el *post-protocolo*, cambiará el valor de su variable de control a `true` para indicar que ha dejado libre el recurso crítico.

Veamos el código.

```
1  /* Adaptado de M. Ben-Ari por Manuel Francisco */
2
3  /* Tercer intento */
4  class Third {
5      /* Iteraciones que dara cada hilo */
6      static final int iteraciones = 2000000;
7      /* Recurso compartido */
```

```
8       static volatile int enteroCompartido = 0;
9       /* Representa el deseo del hilo P de entrar en la seccion critica */
10      static volatile boolean wantp = false;
11      /* Representa el deseo del hilo Q de entrar en la seccion critica */
12      static volatile boolean wantq = false;
13
14      class P extends Thread {
15          public void run() {
16              for (int i=0; i<iteraciones; ++i) {
17                  /* Seccion no critica */
18                  wantp = true;
19                  while (wantq)
20                      Thread.yield();
21
22                  /* Seccion critica */
23                  ++enteroCompartido;
24                  /* Fin Seccion critica */
25
26                  wantp = false;
27              }
28          }
29      }
30
31      class Q extends Thread {
32          public void run() {
33              for (int i=0; i<iteraciones; ++i) {
34                  /* Seccion no critica */
35                  try{Thread.sleep(10);}catch(Exception e){}
36                  wantq = true;
37                  while (wantp)
38                      Thread.yield();
39
40                  /* Seccion critica */
41                  --enteroCompartido;System.out.println("Q");
42                  /* Fin Seccion critica */
43
44                  wantq = false;
45              }
46          }
47      }
48
49      Third() {
50          Thread p = new P();
51          Thread q = new Q();
52          p.start();
53          q.start();
54
55          try {
56              p.join();
57              q.join();
58              System.out.println("El valor del recurso compartido es " + enteroCompartido);
59              System.out.println("Deberia ser 0.");
60          }
61          catch (InterruptedException e) {}
62      }
63
64      public static void main(String[] args) {
65          new Third();
66      }
67  }
```

Aqui wantp esta antes del while, para que la otra hebra lo tenga en cuenta. Podria generar un problema de interbloqueo

los blucles se quedarian en espera infinitua, ya que los dos se podrian poner a verdadero a la misma vez

Hemos conseguido así que sólo un hilo ejecute su sección crítica a la vez. Sin embargo, el programa puede bloquearse para ciertos entrelazados, por ejemplo:

1. P_1 pone C_1 a `false`.
2. P_2 pone C_2 a `false`.
3. P_1 comprueba C_2 y encuentra que $C_2 = false$, por lo que permanece en el bucle.
4. P_2 comprueba C_1 y encuentra que $C_1 = false$, por lo que permanece en el bucle.

Es decir, pueden producirse bloqueos si dos hilos declaran a la vez su intención de entrar en la sección crítica.

2.4. Cuarto Intento

Este intento continua el razonamiento del anterior, y obliga a un hilo a abandonar su intención de ejecutar la sección crítica si descubre que puede existir un estado de contención con otro proceso, donde se bloquearán mutuamente.

Para obligar a un hilo que abandone su intención momentáneamente, se cambia el valor de su variable de control dos veces mientras se encuentre esperando en el bucle. De este modo, el otro hilo verá la alternancia, ejecutará la sección crítica, y establecerá su variable a `true`, con lo que el hilo anterior podrá entrar, evitando así el interbloqueo.

El código sería el siguiente:

```
1  /* Adaptado de M. Ben-Ari por Manuel Francisco */
2
3  /* Cuarto intento */
4  class Fourth {
5      /* Iteraciones que dara cada hilo */
6      static final int iteraciones = 2000000;
7      /* Recurso compartido */
8      static volatile int enteroCompartido = 0;
9      /* Representa el deseo del hilo P de entrar en la seccion critica */
10     static volatile boolean wantp = false;
11     /* Representa el deseo del hilo Q de entrar en la seccion critica */
12     static volatile boolean wantq = false;
13
14     class P extends Thread {
15         public void run() {
16             for (int i=0; i<iteraciones; ++i) {
17                 /* Seccion no critica */
18                 wantp = true;
19                 while (wantq) {
20                     wantp = false;
21                     Thread.yield();
22                     wantp = true;
23                 }
24
25                 /* Seccion critica */
26                 ++enteroCompartido;
27                 /* Fin Seccion critica */
28
29                 wantp = false;
30             }
31         }
32     }
33
34     class Q extends Thread {
35         public void run() {
36             for (int i=0; i<iteraciones; ++i) {
```

Nos aseguramos de que wantp esté en verdadero / falso antes de y despues del while, volviendolo a verdadero despues

```
37                  /* Seccion no critica */
38                  wantq = true;
39                  while (wantp) {
40                      wantq = false;
41                      Thread.yield();
42                      wantq = true;
43                  }
44
45                  /* Seccion critica */
46                  --enteroCompartido;
47                  /* Fin Seccion critica */
48
49                  wantq = false;
50              }
51          }
52      }
53
54      Fourth() {
55          Thread p = new P();
56          Thread q = new Q();
57          p.start();
58          q.start();
59
60          try {
61              p.join();
62              q.join();
63              System.out.println("El valor del recurso compartido es " + enteroCompartido);
64              System.out.println("Deberia ser 0.");
65          }
66          catch (InterruptedException e) {}
67      }
68
69      public static void main(String[] args) {
70          new Fourth();
71      }
72  }
```

En este caso, existe un problema que crea procesos ansiosos, que los dos se ceden paso siempre, y justo antes de entrar entran en conflicto. Ahora la alternancia serviria para comprobar quien entra, pero solo en el momento de entrar

Con este algoritmo tenemos garantizada la exclusión mutua. Sin embargo puede dar lugar a procesos ansiosos. Veamos una secuencia en la que esto puede ocurrir:

1. P_1 pone C_1 a 0.
2. P_2 pone C_2 a 0.
3. P_2 comprueba C_1 y encuentra que $C_1 = 0$, por lo que permanece en su bucle y cambia C_2 a 1.
4. P_1 entra en su sección crítica y su *post-protocolo*, donde cambia C_1 a 1.
5. P_2 sigue en el bucle y cambia C_2 a 0.
6. En este punto, si P_1 no desea volver a entrar en su sección crítica, P_2 puede por fin entrar. Pero si P_1 entra en su *pre-protocolo* y vuelve a solicitar entrar en su sección crítica, P_2 se encontraría atrapado en el bucle.

Otro defecto del algoritmo es que puede provocar un *livelock*. Esto puede ocurrir cuando ambos hilos no lleguen a ver la alternancia de la variable de control del otro, por lo que ninguno llega a ejecutar la sección crítica.

1. P_1 pone C_1 a 0.
2. P_2 pone C_2 a 0.

3. P_2 comprueba C_1 y encuentra que $C_1 = 0$, por lo que permanece en el bucle y cambia C_2 a 1.
4. P_2 sigue en el bucle y cambia C_2 a 0.
5. P_1 comprueba C_2 y encuentra que $C_2 = 0$, por lo que permanece en el bucle y cambia C_1 a 1.
6. P_1 sigue en el bucle y cambia C_1 a 0.
7. P_2 comprueba C_1 y encuentra que $C_1 = 0$, por lo que permanece en el bucle.
8. P_1 comprueba C_2 y encuentra que $C_2 = 0$, por lo que permanece en el bucle.

2.5. Algoritmo De Dekker

El problema del último intento es que no se insiste en la adquisición del turno para no ocasionar un bloqueo. Si lo combinásemos con el primer intento, obligaríamos a que alguno de los hilos entrase en su sección crítica en el caso de que los dos estuviesen cediéndose entre sí el recurso crítico compartido, es decir, si hay contención.

Primero, un hilo anuncia su deseo de entrar en la sección crítica, por ejemplo el hilo 1. Antes de entrar, comprueba si hay posibilidad de insistir en ello, comprobando el valor de la variable `turno`. Si no tiene derecho, desactiva el anuncio de entrada en la sección crítica, poniendo su variable de control (C_1) a `true`, y quedando a la espera hasta que reciba el turno. Cuando el otro hilo (hilo 2) complete su sección crítica, el hilo 1 recibirá el turno. En ese momento, volvería a ajustar su variable de control a `false` y entra en la sección crítica. Al salir, cambia el valor de su variable de control y le devuelve el turno al otro hilo.

Veámoslo:

```
1  /* Adaptado de M. Ben-Ari por Manuel Francisco */
2
3  /* Algoritmo de Dekker */
4  class Dekker {
5      /* Iteraciones que dara cada hilo */
6      static final int iteraciones = 2000000;
7      /* Recurso compartido */
8      static volatile int enteroCompartido = 0;
9      /* Representa el deseo del hilo P de entrar en la seccion critica */
10     static volatile boolean wantp = false;
11     /* Representa el deseo del hilo Q de entrar en la seccion critica */
12     static volatile boolean wantq = false;
13     /* Representa de quien es el turno */
14     static volatile int turn = 1;
15
16     class P extends Thread {
17         public void run() {
18             for (int i=0; i<iteraciones; ++i) {
19                 /* Seccion no critica */
20                 wantp = true;
21                 while (wantq) {
22                     if (turn == 2) {
23                         wantp = false;
24                         while (turn == 2)
25                             Thread.yield();
26                         wantp = true;
27                     }
28                 }
29
30                 /* Seccion critica */
31                 ++enteroCompartido;
```

La misma idea de la alternancia, pero solo cuando los procesos estén bloqueados
Con este preprotocolo nos aseguramos la entrada en exclusion mutua que no haya interbloqueos
y que los procesos no esten ansiosos cuando haya interbloqueos.

```
32                  /* Fin Seccion critica */
33
34                  turn = 2;
35                  wantp = false;
36              }
37          }
38      }
39
40      class Q extends Thread {
41          public void run() {
42              for (int i=0; i<iteraciones; ++i) {
43                  /* Seccion no critica */
44                  wantq = true;
45                  while (wantp) {
46                      if (turn == 1) {
47                          wantq = false;
48                          while (turn == 1)
49                              Thread.yield();
50                          wantq = true;
51                      }
52                  }
53
54                  /* Seccion critica */
55                  --enteroCompartido;
56                  /* Fin Seccion critica */
57
58                  turn = 1;
59                  wantq = false;
60              }
61          }
62      }
63
64      Dekker() {
65          Thread p = new P();
66          Thread q = new Q();
67          p.start();
68          q.start();
69
70          try {
71              p.join();
72              q.join();
73              System.out.println("El valor del recurso compartido es " + enteroCompartido);
74              System.out.println("Deberia ser 0.");
75          }
76          catch (InterruptedException e) {}
77      }
78
79      public static void main(String[] args) {
80          new Dekker();
81      }
82  }
```

El algoritmo de Dekker es correcto y satisface los requerimientos de exclusión mutua y ausencia de bloqueos. Ningún hilo puede volverse ansioso y en ausencia de contención, un proceso entra en su sección crítica inmediatamente, si así lo desea.

3. Variables `volatile`

Como habrá observado en los ejemplos anteriores, las variables globales han sido definidas como `volatile`. Esta palabra reservada puede usarse en tipos primitivos o en objetos.

Si una variable es `volatile` el proceso está obligado a leer y escribir su valor en memoria cada vez que sea accedida, en lugar de hacerlo desde o sobre la memoria caché. Esto permite que una variable global sea accedida por distintos hilos que modifiquen su valor y que sea sincronizada automáticamente en cada acceso. No usar la caché provoca peores rendimientos; sin embargo, es un mal necesario.

4. Ejercicios

Ejercicio 1 Implemente el algoritmo de la panadería de Lamport para dos variables. Ejecútelo varias veces para comprobar que funciona correctamente.

---

5. Soluciones De Los Ejercicios

En esta sección encontrará las soluciones a los ejercicios propuestos a lo largo del guión.

5.1. Ejercicio 1

A continuación puede ver la implementación del algoritmo de Lamport para 2 hilos:

```
/* Adaptado de M. Ben-Ari por Manuel Francisco */

/* Algoritmo de la panaderia para dos procesos */
class BakeryTwo {
    /* Iteraciones que dara cada hilo */
    static final int iteraciones = 2000000;
    /* Recurso compartido */
    static volatile int enteroCompartido = 0;
    /* Numero para el proceso p */
    static volatile int np = 0;
    /* Numero para el proceso q */
    static volatile int nq = 0;

    class P extends Thread {
        public void run() {
            for (int i=0; i<iteraciones; ++i) {
                /* Seccion no critica */
                np = nq + 1;
                /* esperar nq = 0 o np <= nq */
                while(nq != 0 && np > nq)
                    Thread.yield();

                Thread.yield();
                /* Seccion critica */
                ++enteroCompartido;
                /* Fin Seccion critica */

                np = 0;
            }
        }
    }

    class Q extends Thread {
        public void run() {
            for (int i=0; i<iteraciones; ++i) {
                /* Seccion no critica */
                nq = np + 1;
                while(np != 0 && nq > np)
                    Thread.yield();

                Thread.yield();
                /* Seccion critica */
                --enteroCompartido;
                /* Fin Seccion critica */

                nq = 0;
            }
        }
    }

    BakeryTwo() {
        Thread p = new P();
        Thread q = new Q();
        p.start();
        q.start();

```

```
57          try {
58              p.join();
59              q.join();
60              System.out.println("El valor del recurso compartido es " + enteroCompartido);
61              System.out.println("Deberia ser 0.");
62          }
63          catch (InterruptedException e) {}
64      }
65
66      public static void main(String[] args) {
67          new BakeryTwo();
68      }
69  }
```

Referencias

[1] M. Ben-Ari. *Principles of concurrent and distributed programming.* Addison-Wesley, 2006.