Asignación de Prácticas Número 4

Programación Concurrente y de Tiempo Real

Antonio J. Tomeu[1]

[1]Departamento de Ingeniería Informática
Universidad de Cádiz

PCTR, 2021

Contenido I

Objetivos de la Práctica

- Plantear de manera rigurosa el «problema de la exclusión mutua».
- Utilizar variable cualificadas como `volatile` y conocer su significado y utilidad.
- Conocer las técnicas de sincronización basadas en variables comunes y espera ocupada para tratar el problema de la exclusión mutua.
- Estudiar e implementar algunos algoritmos clásicos que aplican estas técnicas: *Dekker*, *Eisenberg-McGuire*, etc.

El Problema de la Exclusión Mutua I

- Clásicamente, al problema de eliminar entrelazados indeseables en código concurrente se le ha denominado genéricamente el «problema de la exclusión mutua».
- El objetivo es identificar los puntos del código (secciones críticas) que pueden dar lugar a entrelazados indeseables, forzando a que en tiempo t haya una única hebra en ejecución en esos puntos.
- Para ello, es necesario desarrollar preprotocolos y postprotocolos para evitar que dos o más hebras ejecuten sus secciones críticas al mismo tiempo.
- La estructura de las tareas concurrentes desde un punto de vista teório será, en general, la siguiente:

El Problema de la Exclusión Mutua II

```
while(true){
  código_no_crítico;
pre-protocolo;
sección_crítica;
post-protocolo
}
```

- El problema de la exclusión mutua para n procesos con la estructura descrita se concreta en:
  - n procesos ejecutan indefinidamente una secuencia de instrucciones divida en dos partes: código no crítico y sección crítica; los procesos deben cumplir la propiedad de exclusión mutua: no puede haber entrelazado de instrucciones de las seccciones críticas de dos o más procesos.

El Problema de la Exclusión Mutua III

- Para lograrlo, se introducen pre-protocolos y post-protocolos que protegen a la sección crítica, que pueden requerir variables adicionales. Los procesos se comunican sus intenciones intercambiando información a través de estas variables.
- Un proceso puede detenerse (E/S, etc.) en su código no crítico, pero no en la sección crítica. Si un proceso se detiene en su código no crítico, ello no debe afectar al funcionamiento del resto de procesos.
- Los procesos deben estar libres de interbloqueos (deadlocks). Si un proceso desea entrar a su sección crítica, eventualmente lo logrará.
- No debe haber procesos «ansiosos»; si un proceso indica su intención de ejecutar la sección crítica comenzando la ejecución de su pre-protocolo, eventualmente tendrá éxito.
- En ausencia de contención en el acceso a las secciones críticas, un único proceso que desea ejecutar la suya, podrá hacerlo tantas veces como quiera.

Las Variables `volatile` de Java I

- Las analizamos dado que las vamos a emplear al intentar resolver el problema de la exclusión mutua con variables compartidas en Java.
- Sintácticamente, una variable es volátil cuando se cualifica como tal al declararla mediante el cualificador `volatile`.
- Semánticamente, esto significa que todas las lecturas/escrituras de tales variables se hacen en memoria principal de forma atómica, no generándose copias locales a las hebras en su memoria de trabajo.
- Esto garantiza una visión única y coherente del valor de la variable a todas las tareas que la usan...
- ... pero no resuelve el problema de la exclusión mutua, ya que algo tan simple como n++ no es atómico, aunque n se haya declarado como `volatile`.

Condición de Concurso y Variables volatile I

```
1  public class tryVolatile
2    extends Thread
3  {
4    private int tipoHilo;
5    private static volatile int Turno = 1;
6    private static volatile int nVueltas = 10000;
7    private static volatile int n = 0;
8
9      public tryVolatile(int tipoHilo)
10     {this.tipoHilo=tipoHilo;}
11
12     public void run()
13     {
14       switch(tipoHilo){
15         case 1:{for(int i=0; i<nVueltas; i++)n++;break;}
16         case 2: {for(int i=0; i<nVueltas;i++)n--;break;}
17       }
18     }
19
20     public static void main(String[] args)
21       throws InterruptedException
22     {
```

Condición de Concurso y Variables volatile II

```
23          tryVolatile h1 = new tryVolatile(1);
24          tryVolatile h2 = new tryVolatile(2);
25          h1.start(); h2.start();
26          h1.join(); h2.join();
27          System.out.println(n);
28      }
29  }
```

Primer Intento: Tomando Turnos I

- Los procesos acceden a sus secciones críticas por turnos.
- Hay una variable compartida que da explícitamente el turno a los procesos.
- Los protocolos pre y post van pasando el turno en forma explícita.
- El proceso que no posee el turno, hace espera ocupada hasta que lo recibe.

Primer Intento: Tomando Turno en Java I

```
1  public class tryOne extends Thread{
2    private int tipoHilo;
3    private static volatile int Turno = 1;
4    private static volatile int nVueltas = 10000;
5    private static volatile int n = 0;
6
7      public tryOne(int tipoHilo)
8      {this.tipoHilo=tipoHilo;}
9
10     public void run()    {
11       switch(tipoHilo){
12         case 1:{for(int i=0; i<nVueltas; i++){
13                  while(Turno!=1);
14                  n++;
15                  Turno = 2;
16                }
17                  break;}
18         case 2: {for(int i=0; i<nVueltas;i++){
19                  while(Turno!=2);
20                  n--;
21                  Turno = 1;
22                    }
```

Primer Intento: Tomando Turno en Java II

```
23              }break;
24          }
25      }
26
27      public static void main(String[] args) throws
            InterruptedException{
28          tryOne h1 = new tryOne(1);
29          tryOne h2 = new tryOne(2);
30          h1.start(); h2.start();
31          h1.join(); h2.join();
32          System.out.println(n);
33      }
34  }
```

Primer Intento: Análisis I

- Es evidente que preserva la exclusión mutua, que está libre de interbloqueos y que no hay procesos ansiosos. Todo ellos puede ser demostrado matemáticamente de forma rigurosa (lógica temporal, CSP, Redes de Petri, etc.).
- Sin embargo, la solución falla cuando en ausencia de contención un proceso quiere pasar por su sección crítica tantas veces como quiera. Si la hebra 2 se para en su código no crítico, la hebra 1 podrá ejecutar su sección crítica una vez a lo sumo, y luego quedará detenida en su pre-protocolo.
- El problema es causado por el fuerte acomplamiento (dependencia) que el mecanismo de turnos impone a los procesos.
- Solución: debilitar el acoplamiento.

Segundo Intento: *Flags* de Estado I

- Cada proceso `i` posee un *flag* booleano `Ci` con el que anuncia sus intenciones a los demás procesos.
- Un proceso debe esperar en su pre-protocolo si otro proceso ya ha anunciado con su flag que va ejecutar la sección crítica.
- Finalizada la espera, indica con su propio *flag* que pasa a ejecutar su sección crítica, y lo hace.
- Cuando termina de ejecutar la sección crítica, indica que sale de ella, de nuevo con su *flag*.

Segundo Intento: *Flags* de Estado en Java I

```
1  public class tryTwo extends Thread{
2    private int tipoHilo;
3    private static volatile int nVueltas = 10000;
4    private static volatile int n = 0;
5    private static volatile boolean C1 = false;
6    private static volatile boolean C2 = false;
7
8      public tryTwo(int tipoHilo)
9      {this.tipoHilo=tipoHilo;}
10
11     public void run(){
12       switch(tipoHilo){
13         case 1:{for(int i=0; i<nVueltas; i++){
14                  while(C2==true);
15                  C1 = true;
16                  n++;
17                  C1 = false;
18
19                }
20                break;}
21         case 2: {for(int i=0; i<nVueltas;i++){
22                  while(C1==true);
```

Segundo Intento: *Flags* de Estado en Java II

```
23                      C2 = true;
24                      n--;
25                      C2 = false;
26                      }
27                    }break;
28          }
29        }
30
31        public static void main(String[] args) throws
              InterruptedException{
32          tryTwo h1 = new tryTwo(1);
33          tryTwo h2 = new tryTwo(2);
34          h1.start(); h2.start();
35          h1.join(); h2.join();
36          System.out.println(n);
37        }
38    }
```

Segundo Intento: Análisis I

- No preserva la exclusión mutua sobre la sección crítica.
- Por tanto, hay algún entrelazado indeseable.
- Analizar el cumplimiento del resto de condiciones deseadas es inútil.
- Ejercicio: búsquese el entrelazado que rompe la exclusión mutua.

¿Qué hago ahora? I

- Programamos los intentos de solución y algoritmos solicitados en el documento de asignación.
- Lo primero de todo: **documéntese a fondo**. Para ello, lea los capítulos que tratan los algoritmos de exclusión mutua con variables compartidas de las referencias siguientes:
  - Ben-Ari, M. Principles of Concurrent and Distributed Programming, segunda edición **(lectura preferente)**.
  - Palma et al., Programación Concurrente, segunda edición (lectura alternativa).
- Ejercicio 1: Implemente en Java los intentos tercero y cuarto descritos en las mismas, y estudie por qué son incorrectos y no resuelven el problema de la exclusión mutua adecuadamente.
- Ejercicios 2-3: Implemente en Java los algoritmos de exclusión mutua de *Dekker* y *Eisenberg-McGuire*.

¿Qué hago ahora? II

- Ejercicio 4: Implemente en Java el algoritmo incorrecto de *Hyman*.
- Escriba un documento `analisis.pdf` que recoja el resultado de sus pruebas de ejecución para todo ello, y analice el comportamiento de todos los códigos pedidos.