Grado en Ingeniería Informática

Programación Concurrente y Tiempo Real

Departamento de Ingeniería Informática

Práctica 4

Autor:

Pablo Velicias Barquín

Fecha:

12 de Noviembre de 2021

Índice

1. Ejercicio 1. Refinamiento sucesivo

1.1. Tercer Intento

El tercer intento corrige el fallo que teníamos en el segundo intento (El problema era que desde que P1 indica que está en la sección critica hasta que realmente ocurre, pasa un tiempo arbitrario) indicando que c1 = 0 (Es decir, indica que P1 está en la sección critica) incluso antes de comprobar el valor de c2. Por lo tanto, P1 está en la sección critica en el instante que la condición del while se cumpla. De manera teórica, este algoritmo resuelve el problema de exclusión mutua. Sin embargo, el comportamiento que obtenemos es un **Deadlock** muy sencillo que podemos ver con el siguiente ejemplo:

Instrucción	***c1***	***c2***
Valores iniciales	1	1
P1 cambia c1	0	1
P2 cambia c2	0	0
P1 comprueba a c2	0	0
P2 comprueba a c1	0	0

...

P1 y P2 podrían estar comprobados infinitamente las variables c1 y c2, pero nunca llegarían a progresar. Esta bloqueado en este punto. Por lo tanto, se descarta.

1.2. Cuarto Intento

En el intento anterior, cuando P1 marcaba c1=0, indicando su intención de entrar a la sección critica, también insistía en su derecho de entrar en la sección critica. Cambiar c1 antes de comprobar c2 evita el entrelazado y la violación de exclusión

mutua, pero si P2 no cede su lugar de la sección critica, entonces P1 si debería cederlo.

Para ello se realizó el algoritmo del cuarto intento. En este caso, el problema se resuelve haciendo que un proceso ceda su intención de entrar en la sección critica por un breve periodo de tiempo para dejar al otro proceso la oportunidad de salir de la sección critica.

Es decir, P1 cambia c1=0 y posteriormente comprueba c2. Si c2=0, entonces P1 vuelve a cambiar c1=1 y espera un periodo de tiempo. Luego, vuelve a colocar c1=0 y comprueba de nuevo c2.

De esta manera, podemos comprobar que evitamos entrelazados y la violación de exclusión mutua. Sin embargo, puede suceder un caso muy raro, donde los dos procesos estén continuamente cediendo su puesto en la sección critica y ninguno entre. El ejemplo sería:

Instrucción	***c1***	***c2***
Valores iniciales	1	1
P1 cambia c1	0	1
P2 cambia c2	0	0
P1 comprueba a c2	0	0
P2 comprueba a c1	0	0
P1 cambia c1	1	0
P2 cambia c2	1	1
P1 cambia c1	0	1
P2 cambia c2	0	0

...

Como decimos, es un caso muy extraño y muy difícil de que ocurra (Tiene que haber

una sincronización perfecta continua en el programa para que ocurra). Aun así, este intento también se ha rechazado como solución correcta, ya que no existe una manera fiable de dar un número de pasos a priori que realizará los bucles antes de acabar correctamente el programa, es decir, no tenemos ninguna manera de garantizar el rendimiento del sistema.

2. Ejercicio 2. Algoritmo de Dekker

El algoritmo de Dekker es una combinación del intento uno y el intento cuatro realizados anteriormente. Recordemos que el primer intento tenia el problema del fuerte acoplamiento que el mecanismo de turnos impone a los procesos, y el cuarto la posibilidad de un estado de bucle infinito.

Esto se evita añadiendo una variable llamada **turno**. Ahora cuando P1 cambia c1=0, comprueba si turno=1. Si eso es cierto, entonces comprueba continuamente que P1 este fuera de la sección critica. Si no fuera su turno, entonces pasaría c1=1, y esperaría hasta que el turno pase a ser suyo.

Con este algoritmo, la exclusión mutua se cumple, esta libre de interbloqueos y no hay procesos ansiosos. Los resultados que obtenemos son los correctos.

3. Ejercicio 3. Algoritmo de Eisenberg-McGuire

El algoritmo de Eisenberg-McGuire cumple la exclusión mutua para N-procesos. Con la utilización de un vector de estados (INACTIVO, ACTIVO, ESPERANDO) y turnos, podemos tratar el problema. Sin embargo, no es el mejor algoritmo para utilizar para dos procesos, ya que un algoritmo complejo, y existen opciones mucho más sencillas para dos procesos, como puede ser el algoritmo de Dekker.

Además, utilizamos un ejecutor de tamaño fijo para los hilos, de esta manera solo creamos un número especifico de hilos, mientras que el resto se mantienen bloqueados a la espera que se quede un hueco libre. En este caso el algoritmo de Eisenberg-McGuire obtiene los resultados correctos.

4. Ejercicio 4. Algoritmo de Hyman.

El algoritmo de Hyman es otro intento de solución para garantizar la exclusión mutua, la limitación en la espera y el progreso en la ejecución. Utiliza al igual que el algoritmo de Dekker, dos variables para mostrar la intención del proceso de entrar en la sección critica y una variable para indicar el turno. Sin embargo, termino siendo otro de los intentos fallidos que ya hemos estudiado. Siguiendo el siguiente esquema (El proceso 2 es igual cambiando los valores correspondientes):

Código para Proceso 1

```
1 c1 = 0;
2 while(turno != 1)
3 {
4     while(c2 == 0);
5     turno = 1;
6 }
7 ZonaCritica
8 c1 = 1;
```

El problema de este algoritmo llega cuando un proceso llega a la linea 5 del código y se para. Supongamos que el proceso 1 cambia a c1=0, y turno=2. Entrará en el primer bucle, y pasará el bucle de la linea 4, esperando en la linea 5. Si el proceso 2 cambia c2=0 y el turno=2, entonces el proceso 2 pasa del bucle y entra en la sección

critica del programa. Si luego, el proceso 1, continua su ejecución, el turno cambiará a 1, y entonces el proceso 1 también entrará en la sección critica, dando la violación de exclusión mutua.

Por lo tanto, los resultado que obtenemos, son incorrectos.