Trabajo Práctico N°1

**1.-** Dadas las siguientes propuestas de pseudocódigos de solución al problema de la sección crítica, analice detalladamente si cumplen con cada una de las condiciones para que sean solución.

**Condiciones a cumplir por la Solución de la Sección Crítica**

1. Los procesos P0, P1, …Pn no deben estar simultáneamente en sus secciones críticas (debe proveerse exclusión mutua).

2. Un proceso que está fuera de su SC - ni siquiera intentando ingresar a su SC[ \*] - no debe evitar que otro proceso entre a su SC (puede generar bloqueo indefinido en el 2º proceso).

3. No debe ocurrir que un proceso ingrese a su SC en repetidas oportunidades y que otro proceso nunca tenga la posibilidad de ingresar a su SC (puede generar inanición).

4. Dos o más procesos que están por ingresar a sus respectivas SC´s no deben entrar en un lazo de espera infinito (si así sucediera, se puede generar bloqueo indefinido).

**a)**

**int** turno = 1; // *corresponde ingresar a P1.*

**P0: P1:**

**while(true) while(true)**

**{ {**

**while(**turno = 1); **while(**turno = 0);

/\*SeccCritica\_0\*/; /\*SeccCritica\_1\*/;

turno = 1; turno = 0;

/\*programa0\*/; /\*programa1\*/;

**} }**

**Condición 1:** esta solución si provee exclusión mutua mediante la variable compartida turno. Como es una única variable que toma los valores 0 y 1. En un momento dado solo podrá tomar uno de dichos valores por lo que no podrán ingresar simultáneamente en la sección crítica.

**Condición 2:** No se cumple esta condición ya que si suponemos que el proceso 0 sale de la sección crítica poniendo la variable turno en 1 y continúa ejecutando su programa, el proceso 1 podrá entrar en la sección crítica si lo desea. Al salir pone la variable turno en 0. Posteriormente, si el proceso 1 desea entrar nuevamente a la sección crítica sin que P0 haya entrado antes porque sigue ejecutando su programa0 no lo podrá hacer porque la variable turno sigue en 0. Por lo tanto, a pesar de P0 no desea entrar en la sección crítica aun así impide que el otro entre.

**Condición 3:** Si se cumple. Un proceso no puede entrar en su sección critica repetidamente ya que apenas sale de ella pone la variable turno en el valor que le impide entrar a la misma. Por lo tanto, hasta que el otro proceso no entra en la sección crítica no puede volver a entrar.

**Condición 4:** Si se cumple esta condición. La variable turno es compartida y es la que provee la exclusión mutua. Esta variable toma el valor 0 o 1. No existe momento en el que ésta no tome un valor por lo que siempre alguno (el que primero haya salido de la sección crítica) de los procesos entrará en la sección crítica.

**b)**

**boolean** estado[2] = {false, false}; // valor inicial de variables compartidas.

**P0:**

**while(true)**

**{**

estado[0] = true;

**while(**estado[1]);

/\*SeccCritica\_0\*/; estado[0] = false;

/\*programa0\*/;

**}**

**P1:**

**while(true)**

**{**

estado[1] = true;

**while(**estado[0]);

/\*SeccCritica\_1\*/; estado[1] = false;

/\*programa1\*/

**}**

**Condición 1:** esta solución si provee exclusión mutua mediante el vector compartido estado que contiene valores booleanos. Cada proceso cuando desea entrar a la sección crítica coloca el valor del vector estado correspondiente al otro proceso en true para que este se mantenga en el bucle de espera si desea entrar en la sección crítica. Esto provee la exclusión mutua.

**Condición 2:** Si se cumple esta condición ya que cualquiera sea el proceso que sale de la sección crítica pone la variable correspondiente al otro proceso el false para que pueda entrar en la sección crítica. Hasta que el proceso no desea entrar nuevamente en la sección crítica no modifica esa variable por lo que no podrá impedir que el otro entre.

**Condición 3:** No pasará ya que apenas un proceso sale de la sección crítica, habilita al otro proceso a entrar si este se encuentra esperando porque pone la variable estado en false para que salga del bucle.

**Condición 4:** No se cumple esta condición. En el caso que tanto P0 como P1 quieran entrar en la sección crítica al mismo tiempo, cada uno de ellos pondrá la variable estado que maneja en true, impidiendo que el otro entre a la sección crítica. Por lo tanto, ambos se encontrarán en un lazo de espera. Como ninguna entra a la sección crítica, la variable estado nunca podrá cambiar su estado. Luego, ambos procesos quedarán en un lazo de espera infinito.

**c)**

**boolean** estado[2] = {false, false}; // valor inicial de variables compartidas.

**P0:**

**while(true)**

**{**

estado[0] = true;

**if**(estado[1]) estado[0] = false;

**else**

**{**

/\*SeccCritica\_0\*/; estado[0] = false;

/\*programa0\*/;

**}**

**}**

**P1:**

**while(true)**

**{**

estado[1] = true;

**if**(estado[0]) estado[1] = false;

**else**

**{**

/\*SeccCritica\_1\*/; estado[1] = false;

/\*programa1\*/;

**}**

**}**

**Condición 1:** esta solución si provee exclusión mutua mediante el vector compartido estado que contiene valores booleanos. Cada proceso cuando desea entrar a la sección crítica coloca el valor del vector estado correspondiente al otro proceso en true para que éste no pueda entrar a su sección crítica si asi lo desea. Esto provee la exclusión mutua.

**Condición 2:** Si se cumple esta condición ya que cualquiera sea el proceso que sale de la sección crítica pone la variable correspondiente al otro proceso el false para que pueda entrar en la sección crítica. Hasta que el proceso no desea entrar nuevamente en la sección crítica no modifica esa variable por lo que no podrá impedir que el otro entre.

**Condición 3:** No se cumple. Por ejemplo si, P0 es muy rápido, ejecuta todo su bucle (fija estado[0]=true, comprueba estado[1] que está false, entra a la SC, sale, pone estado[0]=false y vuelve a empezar) repetidas veces antes de que P1 tenga oportunidad de ejecutar incluso la instrucción estado[1] = true. Si esto sucede repetidamente, P1 puede quedarse siempre detrás y nunca entrar,

**Condición 4:** La condición 4 no se cumple porque puede darse un caso de **bloqueo indefinido**: si ambos procesos intentan entrar a sus secciones críticas de manera simultánea, cada uno establece su variable de estado en true y, al verificar la del otro, encuentran que también está en true. Entonces, ambos procesos retroceden a la vez, restableciendo su estado en false y repitiendo inmediatamente el intento de entrada. Esto puede prolongarse indefinidamente si la ejecución sigue siendo intercalada de esta forma, generando un lazo de espera infinito en el que ninguno de los dos logra acceder a su sección crítica.

**2.-** Considere un sistema operativo en cual se encuentran en ejecución dos procesos concurrentes, P1 y P2, que intentan entrar en su sección crítica la cual, en ambos procesos, tiene una cantidad de tres instrucciones. Luego de finalizar la sección crítica, el proceso P1 ejecuta en su programa siete instrucciones y P2, cinco. Se indican en la tabla a continuación los instantes en que los procesos invocan a la primitiva P(s) al intentar ingresar en su sección crítica.

Utilizando las implementaciones de las primitivas P y V que se indican, complete la tabla indicando los valores que toman en cada paso las variables ***mutex***, ***delay*** y ***s***, e indique en cada caso qué proceso modifica éstas variables y cuál de ellos queda en espera. Complete, además, el código que ejecuta cada proceso, asumiendo que cada instrucción se ejecuta en un sólo paso, que la primitiva P tiene prioridad sobre la V, y que las primitivas tienen prioridad frente a las otras instrucciones. Para indicar código de sección crítica utilice SC1/SC2, y para programa, Prog1/Prog2.

**Operación P(s):**

Pb(mutex); s = s – 1;

**if** s < 0 **then**

Vb(mutex); Pb(delay);

Vb(mutex);

**Operación V(s):**

Pb(mutex); s = s + 1;

**if** s <= 0 **then**

Vb(delay);

**else**

Vb(mutex);

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Paso** | **Proceso 1** | **Proceso 2** | **mutex** | **delay** | **s** | **Espera** |
| 0 |  |  | True | False | 1 |  |
| 1 | P(s) = Pb(mutex) |  | False | false | 1 |  |
| 2 | S = s - 1 |  | false | false | 0 |  |
| 3 |  | P(s) = Pb(mutex) | false | false | 0 | EspP2(atomicidad) |
| 4 | If(s<0) |  | false | false | 0 | EspP2(atomicidad) |
| 5 | Vb(mutex) |  | true | false | 0 | Esp(P2atomicidad) |
| 6 |  | Pb(mutex) | false | false | 0 |  |
| 7 |  | S = s - 1 | false | false | -1 |  |
| 8 |  | If(s<0) | false | false | -1 |  |
| 9 |  | Vb(mutex) | true | False | -1 |  |
| 10 |  | Pb(delay) | True | False | -1 | EspP2(EM) |
| 11 | SC1 |  |  |  |  | EspP2(EM) |
| 12 | SC1 |  |  |  |  | EspP2(EM) |
| 13 | SC1 |  |  |  |  | EspP2(EM) |
| 14 | V(s) = Pb(mutex) |  | False | False | -1 | EspP2(EM) |
| 15 | S = s +1 |  | False | False | 0 | EspP2(EM) |
| 16 | If(s <= 0) |  | False | False | 0 | EspP2(EM) |
| 17 | Vb(delay) |  | False | True | 0 | EspP2(EM) |
| 18 |  | Pb(delay) | False | False | 0 |  |
| 19 |  | Vb(mutex) | True | False | 0 |  |
| 20 |  | SC2 |  |  |  |  |
| 21 | Prog1 |  |  |  |  |  |
| 22 | Prog1 |  |  |  |  |  |
| 23 |  | SC2 |  |  |  |  |
| 24 |  | SC2 |  |  |  |  |
| 25 |  | V(s)= Pb(mutex) | False | False | 0 |  |
| 26 |  | S = s + 1 | False | False | 1 |  |
| 27 |  | If(s<=0) | False | False | 1 |  |
| 28 |  | Vb(mutex) | True | False | 1 |  |
| 29 | Prog1 |  | True | False | 1 |  |
| 30 | Prog1 |  | True | False | 1 |  |
| 31 |  | Prog2 | True | False | 1 |  |
| 32 |  | Prog2 | True | False | 1 |  |
| 33 | Prog1 |  | True | False | 1 |  |
| 34 |  | Prog2 | True | False | 1 |  |
| 35 | Prog1 |  | True | False | 1 |  |
| 36 |  | Prog2 | True | False | 1 |  |
| 37 |  | Prog2 | True | False | 1 |  |
| 38 | Prog1 |  | True | False | 1 |  |

**3.-** Resolver con el mismo criterio del ejercicio anterior, pero para tres procesos, P1, P2 y P3, donde todos tienen tres instrucciones en la sección crítica, P1 tiene seis instrucciones de programa, P2 tiene cuatro y P3 tiene cinco.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Paso** | **Proceso 1** | **Proceso 2** | **Proceso 3** | **mutex** | **delay** | **s** | **Espera** |
| 0 |  |  |  | True | False | 1 |  |
| 1 | P(s) = Pb(mutex) |  |  | false | false | 1 |  |
| 2 | S = s-1 |  |  | false | false | 0 |  |
| 3 | If(s < 0) |  |  | false | false | 0 |  |
| 4 | Vb(mutex) |  |  | true | false | 0 |  |
| 5 | SC1 |  |  |  |  |  |  |
| 6 |  |  | P(s) = Pb(mutex) | false | false | 0 |  |
| 7 |  |  | S = s -1 | false | false | -1 |  |
| 8 | SC1 |  |  |  |  |  |  |
| 9 |  |  | If(s<0) | false | false | -1 |  |
| 10 |  |  | Vb(mutex) | true | false | -1 |  |
| 11 |  |  | Pb(delay) | true | false | -1 | EspP3(EM) |
| 12 | SC1 |  |  |  |  |  | EspP3(EM) |
| 13 |  | P(s)=Pb(mutex) |  | false | false | -1 | EspP3(EM) |
| 14 | V(s) = Pb(mutex) |  |  | false | False | -1 | EspP3(EM), EspP1(atom) |
| 15 |  | S = s-1 |  | False | false | -2 | EspP3(EM), EspP1(atom) |
| 16 |  | If(s<0) |  | false | false | -2 | EspP3(EM), EspP1(atom) |
| 17 |  | Vb(mutex) |  | true | false | -2 | EspP3(EM), EspP1(atom) |
| 18 | Pb(mutex) |  |  | false | false | -2 | EspP3(EM) |
| 19 |  | Pb(delay) |  | false | false | -2 | EspP3(EM), EspP2(EM) |
| 20 | S = s + 1 |  |  | false | false | -1 | EspP3(EM), EspP2(EM) |
| 21 | If(s <= 0) |  |  | false | false | -1 | EspP3(EM), EspP2(EM) |
| 22 | Vb(delay) |  |  | false | true | -1 | EspP3(EM), EspP2(EM) |
| 23 |  |  | Pb(delay) | false | false | -1 | EspP2(EM) |
| 24 |  |  | Vb(mutex) | true | false | -1 | EspP2(EM) |
| 25 |  |  | SC3 |  |  |  |  |
| 26 |  |  | SC3 |  |  |  |  |
| 27 |  |  | SC3 |  |  |  |  |
| 28 |  |  | V(s) = Pb(mutex) | false | false | -1 | EspP2(EM) |
| 29 |  |  | s = s + 1 | false | false | 0 | EspP2(EM) |
| 30 |  |  | If(s<=0) | false | false | 0 | EspP2(EM) |
| 31 |  |  | Vb(delay) | false | true | 0 | EspP2(EM) |
| 32 |  | Pb(delay) |  | false | False | 0 |  |
| 33 |  | Vb(mutex) |  | true | false | 0 |  |
| 34 |  | SC2 |  |  |  |  |  |
| 35 |  | SC2 |  |  |  |  |  |
| 36 |  | SC2 |  |  |  |  |  |
| 37 |  | V(s) = Pb(mutex) |  | false | false | 0 |  |
| 38 |  | s=s+1 |  | false | false | 1 |  |
| 39 |  | If(s<=0) |  | false | false | 1 |  |
| 40 |  | Vb(mutex) |  | true | false | 1 |  |
| 41 | Prog1 |  |  | true | false | 1 |  |
| 42 | Prog1 |  |  | true | false | 1 |  |
| 43 |  |  | Prog3 | true | false | 1 |  |
| 44 |  |  | Prog3 | true | false | 1 |  |
| 45 |  | Prog2 |  | true | false | 1 |  |
| 46 |  | Prog2 |  | true | false | 1 |  |
| 47 | Prog1 |  |  | true | false | 1 |  |
| 48 | Prog1 |  |  | true | false | 1 |  |
| 49 |  |  | Prog3 | true | false | 1 |  |
| 50 |  | Prog2 |  | true | false | 1 |  |
| 51 | Prog1 |  |  | true | false | 1 |  |
| 52 |  |  | Prog3 | true | false | 1 |  |
| 53 |  |  | Prog3 | true | false | 1 |  |
| 54 | Prog1 |  |  | true | false | 1 |  |
| 55 |  | Prog2 |  | true | false | 1 |  |

**4.-** Considere la solución propuesta en el ejemplo del buffer limitado visto en teoría. Asuma además que cada instrucción o primitiva se ejecuta en un instante de tiempo, y que las primitivas tienen prioridad frente a las otras instrucciones. Teniendo en cuenta que para el proceso productor la generación del próximo registro y el agregado en el buffer requiere en ambos casos una instrucción mientras que, en el proceso consumidor, extraer un registro del buffer requiere una instrucción y procesar el registro extraído, tres instrucciones, realice las siguientes consignas:

1. Identifique las rutinas *Comienzo\_SC* y *Fin\_SC*.
2. Identifique la sección crítica de cada proceso, productor y consumidor.

**Productor**

While(true)

{

<generar registro>

P(e)

P(s)

* *Comienzo de la sección crítica*

<Agregar registro>

* *Fin de la sección crítica*

V(s)

V(f)

}

**Productor**

While(true)

{

P(f)

P(s)

* *Comienzo de la sección crítica*

<extraer registro>

* *Fin de la sección crítica*

V(s)

V(e)

<Procesar registro> *-- 3 instrucciones*

}

1. Complete las tablas a continuación, indicando los valores que toman los semáforos en cada paso, qué proceso modifica cada uno de ellos y cuál de ellos se queda en espera. Determine si se producen errores por buffer desbordado o buffer vacío.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Paso** | **Proceso 1** | **Proceso 2** | **e** | **f** | **s** | **Espera** |
| 0 |  |  | 3 | 0 | 1 |  |
| 1 | Productor: <Prod> |  | 3 | 0 | 1 |  |
| 2 | P(e) |  | 2 | 0 | 1 |  |
| 3 | P(s) |  | 2 | 0 | 0 |  |
| 4 | <Ag> |  | 2 | 0 | 0 |  |
| 5 | V(s) |  | 2 | 0 | 1 |  |
| 6 | V(f) |  | 2 | 1 | 1 |  |
| 7 |  | Consumidor:P(f) | 2 | 0 | 1 |  |
| 8 | Productor: <Prod> |  | 2 | 0 | 1 |  |
| 9 |  | C: P(s) | 2 | 0 | 0 |  |
| 10 | P(e) |  | 1 | 0 | 0 |  |
| 11 | P(s) |  | 1 | 0 | 0 | Esp(Productor) |
| 12 |  | C: <ext> | 1 | 0 | 0 | Esp(Productor) |
| 13 |  | V(s) | 1 | 0 | 1 | Esp(Productor) |
| 14 |  | V(e) | 2 | 0 | 1 | Esp(Productor) |
| 15 | P(s) |  | 2 | 0 | 0 |  |
| 16 | <Agr> |  | 2 | 0 | 0 |  |
| 17 |  | Consumidor:<proc> | 2 | 0 | 0 |  |
| 18 |  | Consumidor:<proc> | 2 | 0 | 0 |  |
| 19 |  | Consumidor:<proc> |  |  |  |  |
| 20 | V(s) |  | 2 | 0 | 1 |  |
| 21 | V(f) |  | 2 | 1 | 1 |  |

**5.-** Dada la implementación de la primitiva V(s) sin espera ocupada:

**V(s):**

Pb(mutex);

s.value = s.value + 1;

**if** s <=0 **then** //activar un proceso.

q = delete(s.L); //sacar proceso de cola asociada a s.

**if** (hay CPU libre) **then** //apropiado para SMP. comenzar a ejecutar q en un CPU libre;

**else**

insert(q, Cola\_Ready); //si CPU’s ocupados, a cola ready.

Vb(mutex);

Proponga un control para evitar errores al acceder a la cola de procesos asociados al semáforo *s*, por ejemplo, en el caso que se intente sacar un elemento y la misma esté vacía. ¿Qué ventajas y desventajas tendría respecto de la implementación original?

Una forma de evitar errores al acceder a la cola de procesos asociados al semáforo es agregar un control que verifique si la cola está vacía antes de intentar extraer un elemento, de modo que la operación delete solo se ejecute si efectivamente hay procesos esperando. La principal ventaja de este control es que incrementa la robustez y seguridad de la implementación, ya que previene fallos por accesos inválidos y hace más claro el manejo de casos extremos. Como desventaja, introduce una verificación adicional que agrega un costo mínimo en tiempo de ejecución y cierta redundancia, dado que en una implementación teóricamente correcta la condición s.value <= 0 ya debería garantizar que la cola contiene al menos un proceso bloqueado.