

PRACTICA 2

Programación Paralela

Pablo Fernández del Amo

Tanto para la versión teniendo en cuenta la iniciación, como la que sólo tiene en cuenta la seguridad, el cliente es el siguiente:

coche-sur

sección.no-critica
monitor.wantS-cross()
coche cruzando
monitor.exitS()

coche-norte

sección.no-critica
monitor.wantN-cross()
coche cruzando
monitor.exitN()

peatón

sección.no-critica
monitor.wantP-cross()
peatón cruzando
monitor.exitP()

Ahora, escribiremos la solución sencilla con iniciación:

comentario
monitor (Puente)

ncars-S : int = 0
ncars-N : int = 0
npedestrians : int = 0

OKtoN.VC
OKtoS.VC
OKtoP.VC

{ INV = ncars-S ≥ 0 ∧ ncars-N ≥ 0 ∧ npedestrians ≥ 0 ∧

(npedestrians > 0 → (ncars-S = 0 ∧ ncars-N = 0)) ∧

(ncars-S > 0 → (npedestrians = 0 ∧ ncars-N = 0)) ∧

(ncars-N > 0 → (npedestrians = 0 ∧ ncars-S = 0)) }

} Variables de condición que indican a coches dirección norte, sur y peatones, respectivamente, cuando pueden cruzar. Siguiendo la notación del libro sería condition OKtoN

wantN-cross()

{INV}

OKtoN.wait(npedestrians == 0 ∧ ncars-S == 0)

ncars-N = ncars-N + 1

(Análogo con wantS-cross() y wantP-cross())

exitN()

{INV ∧ ncars-N > 0}

ncars-N = ncars-N - 1

wantOKtoS.notify-all()

OKtoP.notify-all()

(Análogo con exitP() y exitS())

Ahora, claramente esta versión cumple la seguridad del puente (ya que para que un proceso entre al puente, este tiene que satisfacer que no hay ningún proceso de otro tipo que no sea el suyo cruzando). Además, la estructura de monitor hace satisfacer la exclusión mutua.

Esta implementación, genera problemas de incompatibilidad entre procesos, ya que se puede dar que solo estén pasando coches hacia el sur y hacia el norte en turnos consecutivos, sin dejar pasar a los peatones de manera indefinida.

Por lo tanto, la siguiente implementación asegura que cualquier proceso que quiera cruzar el puente, lo acabará haciendo. Para ello, utilizaremos turnos de paso para cada tipo de proceso, con lo que evitaremos deadlock y la famosa incompatibilidad.

monitor:

ncars-N:int = 0

ncars-S:int = 0

npedestrians:int = 0

ncars-S-waiting:int = 0

ncars-N-waiting:int = 0

npedestrians-waiting:int = 0

turn:int = 0

(turno 0 → Coches hacia Norte
1 → Coches hacia Sur
2 → Peatones)

Ok to N . VC
Ok to S . VC
Ok to P . VC } igual que antes

wantN-cross()

{ INV $\{ \text{ncars}_N-\text{waiting} = \text{ncars}-N-\text{waiting} + 1 \}$ } INV $\{ \text{ncars}-N-\text{waiting} > 0 \}$

$\{ \text{INV} \{ \text{ncars}_N-\text{waiting} = \text{ncars}-N-\text{waiting} + 1 \} \}$ } INV $\{ \text{ncars}_S = 0 \wedge \text{npedestrians} = 0 \wedge (\text{turn} = 0 \vee (\text{ncars}_S-\text{waiting} = 0 \wedge \text{npedestrians}-\text{waiting} = 0)) \}$

$\{ \text{INV} \{ \text{ncars}_N-\text{waiting} = \text{ncars}-N-\text{waiting} + 1 \} \}$

$\{ \text{ncars}-N-\text{waiting} = \text{ncars}_N-\text{waiting} - 1 \}$

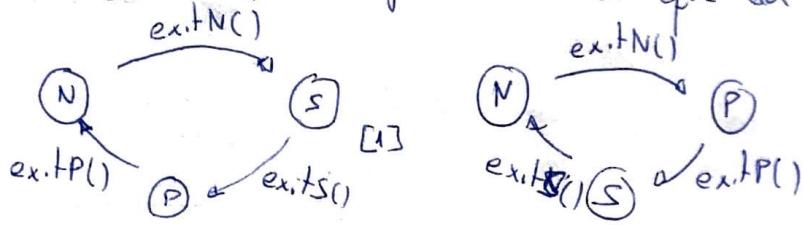
$\{ \text{ncars}_N = \text{ncars}-N + 1 \}$

$\{ \text{INV} \{ \text{ncars}_N-\text{waiting} \geq 0 \wedge \text{ncars}_N > 0 \} \neq \text{INV}$

wantS-cross() y wantP-cross() son análogos cambiando las variables de los coches hacia el sur y peatones respectivamente -así como los turnos)

$\text{ext.N}()$ \rightarrow Ya que, viendo el cliente, solo se invoca un exit cuando el proceso previamente ha entrado y salido del wait-cross, incrementa la ncars-N = ncars-N + 1 en 1 el valor de ncars-N
 $\{ \text{INV } \wedge \text{ncars_N} > 0 \}$
 $\{ \text{INV } \wedge \text{ncars_N} \geq 0 \}$
 if ncars-S.waiting > 0 :
 turn = 1 $\{ \text{INV } \wedge \text{ncars_N} \geq 0 \}$ { Cuando el primer coche dirección norte ~~aterriza~~ el puente, este cambia de turno.
 else turn = 2 $\{ \text{turn} = 1 \wedge \text{ncars_S.waiting} \neq 0 \}$ Cambia en los coches dirección sur s. hay alguno esperando, y si: no, a los peatones
 oktoS.notify-call() $\{ \text{INV } \wedge \text{ncars_N} \geq 0 \}$
 okton.notify-call() $\{ \text{INV } \wedge \text{ncars_N} \geq 0 \wedge \text{turn} = 2 \wedge \text{ncars_S.waiting} = 0 \}$

(análogo para $\text{exit.S}()$ y $\text{exit.P}()$, siguiendo este diagrama de paso de turnos en caso de que haya procesos esperando [1], y si no los hay) en el paso siguiente tipo de proceso al que da paso E23:
 [1]



El invariante de este monitor es:

$\{ \text{INV} \wedge \text{ncars_S} \geq 0 \wedge \text{ncars_N} \geq 0 \wedge \text{npedestrians} \geq 0 \wedge \text{ncars_S.waiting} \geq 0 \wedge \text{ncars_N.waiting} \wedge \text{turn} \in \{0, 1, 2\} \wedge ((\text{npedestrians} > 0 \rightarrow (\text{ncars_S} = 0 \wedge \text{ncars_N} = 0)) \wedge (\text{ncars_S} > 0 \rightarrow (\text{ncars_S} = 0 \wedge \text{npedestrians} = 0)) \wedge (\text{ncars_N} > 0 \rightarrow (\text{ncars_N} = 0 \wedge \text{npedestrians} = 0))) \}$

Las siguientes fórmulas, siguiendo la notación del libro sobre variables condicionales tambien son invariantes:

$\neg \text{empty}(\text{okton}) \rightarrow \text{ncars_S} \neq 0 \vee \text{npedestrians} \neq 0 \vee (\text{turn} \neq 0 \wedge (\text{ncars_S.waiting} \neq 0 \vee \text{npedestrians.waiting} \neq 0))$

Similar para $\neg \text{empty}(\text{oktoS})$ y $\neg \text{empty}(\text{oktoP})$.

• Para demostrar la seguridad equivale a que el invariante mencionado al principio es correcto, por las conjunciones e implicaciones ① ② \wedge ③, ya que son las que garantizan el paso exclusivo por el prente.

Para demostrar que este invariante es cierto, hay que fijarnos que si en las operaciones $\text{wait?}-cross()$ y $\text{exit?}()$, se sigue manteniendo claramente, que las variables son mayores iguales que 0 se sigue cumpliendo en cada función; lo hemos ido explicando a medida que hemos desarrollado los invariantes en cada func. (3)

Para las implicaciones, demostraremos la primera, siendo las otras dos análogas pero con las variables como pendientes.

La variable npedestrians solo cambia del valor 0 cuando un proceso llama a wantP-cross().

Supongamos que $o \cdot \text{hours} - 5 \neq 0 \cdot \text{hours} - N \neq 0$. Al ser uno de estos dos distintos de 0, oktoP.wait (esta variable de condición) al invocarla la función wait comprueba que el predado de su argumento es falso, por lo que almacena este proceso y lo bloquea. Por tanto, basta que esta condición no cambie y esta VC sea invocada, este proceso no alterará el valor de npedestrians. Así que solo será mayor que 0 cuando hubo coches en el puente.

Con esto queda demostrada la seguridad del puente

- Esta implementación evita también la aparición de

Supongamos los siguientes casos:

- 1 proceso quiere cruzar el puente:

Será cierto que este proceso, el resto de tipos no quieren entrar (es decir, si un coche hacia el sur quiere cruzar, no habrá problemas ni coches hacia el norte esperando)

Por tanto, la condición de oktos.wait será verdadera, dejando a este coche pasar el puente (que acabará saliendo ya que la sección crítica tiene fin y exit si está bien definido).

- 2 procesos ^{diferentes} quieren cruzar el puente:

Uno de estos habrá llegado primero a la deseable condición de su función respectiva (estamos considerando 2 coches en distintas direcciones o coche-pareja), por lo que no habrá ningún otro proceso esperando y el resto es verdadero de la condición. Por lo que se ejecutará. En el caso que no estemos en la inicialización del puente, implica que hay un proceso cruzando el puente, este al salir, cambiará el turno a otro que no sea el suyo, por lo tanto, ningún proceso más de ese tipo podrá entrar (ya que además hay otros tipos esperando). Cuando todos los procesos de este último tipo abandonen el puente, el tipo de procesos al que se le ha dado el turno entrará al puente (por la variable de condición okto ^{se le da turno} _{sistema del tipo al que})

- 3 procesos:

Gracias a las ideas que hemos explicado en el punto anterior, y al diagrama [1] que mostramos al explicar como funciona bien los turnos, se demuestra que habiendo 3 procesos distintos esperando, cuando el puente se quede vacío (que por la definición de sincronización, como está implementado el cliente la función exit pasará), el proceso que tenga el turno entrará al puente.

- Por todo lo explicado antes se intuye como está implementada esta sincronización para evitar la inanición.

Si un proceso sufre inanición, implica que ~~un p~~ está bloqueando indebidamente en la variable de condición de su función wait.

Demostaremos que la función notify(1) que lo despierta (en este caso, notify-all()) se acaba ejecutando, junto a que el predicado del atributo de la variable es positivo.

Por que el proceso que está bloqueado es un cache con dirección ~~sur~~ (al ser la implementación simétrica para los 3 tipos de procesos, ~~por lo tanto~~ razoneamiento válido)

- Supongamos que la condición que hace falso el mencionado predicado es que el puente está lleno. Entonces, (i.e. ncars.N > 0 o npedestrians > 0). Entonces, sea cual sea el tipo de proceso que está pasando, cuando salga el primero del puente cambiará el turno, cerrando así el paso a más procesos del mismo tipo.

Entonces, antes de salir de la función exit, despertaría a nuestro proceso que sufre inanición. ~~Este~~ Este no podrá desbloquearse hasta que su predicado se cumpla.

Por ello, con el puente cerrado a procesos del mismo tipo, esperamos a que el último abandone y notify (o signal según el libro) a nuestro proceso con inanición. - ya que con notify-all() al final de la función exit lo conseguimos. Aquí podemos tener tres casos:

1. Se ha dado paso al turno de nuestro proceso \Rightarrow

Predicado es verdadero y proceso está despierto \Rightarrow proceso desbloqueado

2. Si el turno se ha cambiado al otro tipo de procesos que no es el nuestro, volvemos a estar en la casuística que hemos descrito en este apartado, pero otra vez acabando en la conclusión ① o ③ en vez de en esta (la ②)

3. Puede ser que el turno se cambie al tipo de nuestro proceso, pero como los procesos se almacenan y bloquean en la LC con estructura FIFO, puede ser que antes de dar paso y desbloquear este proceso, el turno haya cambiado otra vez (porque el primero en desbloquearse ya haya salido) y haga otro tipo de procesos esperando.

Entonces, por las propiedades de una estructura FIFO, y realizando un número finito de veces el proceso descrito y acabando en ② o ③, llegaremos a la conclusión ④, desbloqueando el proceso.

Por tanto, no existe inanición.