Sincronización entre procesos (cont.)

Sistemas Operativos

Segundo Cuatrimestre 2024

Repaso

Vimos hasta ahora los patrones de sincronización:

- Signaling (Cuando thread A ocurre antes que B).
- Rendezvous (Cuando A1 pasa antes que B2 y B1 pasa antes que A2).
- Mutex (Exclusión mutua del acceso a la sección crítica).
- Barrera (Turnstile)
- Barrera Reutilizable (esta clase).

Ejercicio 4 (Recapitulando)

Enunciado

Un grupo de N estudiantes se dispone a hacer un TP de su materia favorita.
Cada estudiante conoce a la perfección cómo `implementarTp()` y cómo `experimentar()`. Curiosamente, cada etapa puede ser llevada acabo de manera independiente por cada une, así que decidieron dividirse el trabajo. Sin embargo, acordaron que para que alguien pudiera `experimentar()` todes deberían haber terminado de `implementarTp()`.
Se pide diseñar un programa concurrente que utilice procesos y que modele esta situación utilizando semáforos.

Solución con turnstile:

```
barrera = sem(0)
mutex = sem(1)
int counter = 0

ProcesoEstudiantes():

    implementarTp()

    mutex.wait()
    counter++

    if (counter == n): barrera.signal()

    mutex.signal()

    barrera.wait()
    barrera.signal()

    experimentar()
```

Alternativa con multiple signals:

```
barrera.signal(unsigned int n):
        for(i = 0; i < n; i++):
                barrera.signal()
```

Solución con multiple signals:

```
barrera = sem(0)
mutex = sem(1)
int counter = 0

ProcesoEstudiantes():

    implementarTp()

    mutex.wait()
    counter++

    if (counter == n): barrera.signal(n)

    mutex.signal()

    barrera.wait()

    experimentar()
```

Ejercicio 4 bis

Enunciado

Supongamos que el grupo se da cuenta de que su esquema de trabajo es una pésima idea. Proponer un esquema iterativo con etapas ahora más cortas de implementación y experimentación de manera iterativa. ¿Podemos reutilizar nuestra solución anterior?

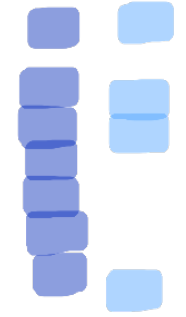
Ejemplo:

```
while(true):
    implementarTp()
    // Inserte código para sincronizar
    experimentar()
```

Pista: El patrón de esta solución es una barrera (o punto de encuentro) de N procesos reusable.

NO Solución

```
mutex = sem(1)
barrera = sem(0)

while(true):

    implementarTp()

    mutex.wait()
    n = n + 1
    if n == N :
        barrera.signal()
        barrera.wait()
    mutex.signal()

    barrera.wait()
    barrera.signal()

    experimentar()

    n = n - 1
    if n == 0:
        barrera.signal()

    barrera.wait()
    barrera.signal()
```



Veamos N = 2

P1 ejecuta primero hasta llegar a la barrera

P2 entra en el if con n == N y se queda bloqueado en la barrera más adelante

Deadlock

Problema: Se accede a n fuera del mutex, posibilidad de race condition.

NO Solución

```
mutex = sem(1)
barrera = sem(0)

while(true):

    implementarTp()

    mutex.wait()
    n = n + 1
    if n == N :
        barrera.signal()
    mutex.signal()

    barrera.wait()
    barrera.signal()

    experimentar()

    mutex.wait()
    n = n - 1
    if n == 0:
        barrera.signal()
    mutex.signal()

    barrera.wait()
    barrera.signal()
```

P2 se adelanta

Veamos P1 y P2

P1 avanza hasta la barrera

P2 entra en el if con n==N

y sigue de largo, en la 2da iteración del while es capaz de experimentar por 2da vez antes de que P1 implementeTP por 2da vez.

Problema: Se puede adelantar un proceso.

Notar que se hacen n+1 signals, entonces puede ocurrir que un proceso vuelva a empezar, pase el mutex, la barrera, y experimente.

Solución

```
mutex = sem(1)
barrera_entrada = sem(0)
barrera_salida = sem(1)

while(true):

  implementarTp()

  mutex.wait()
  n = n + 1
  if n == N :
       barrera_salida.wait()
       barrera_entrada.signal()
  mutex.signal()

  barrera_entrada.wait()
  barrera_entrada.signal()

  experimentar()

  mutex.wait()
  n = n - 1
  if n == 0:
       barrera_entrada.wait()
       barrera_salida.signal()
  mutex.signal()

  barrera_salida.wait()
  barrera_salida.signal()
```

¿Como saber que la solución es correcta?

Intuición

- Solo el proceso N puede bloquear o desbloquear el molinete.
- Antes de que un proceso pueda desbloquear el primer molinete, se tiene que cerrar el segundo, asi que es imposible que un proceso se adelante como en el caso anterior.

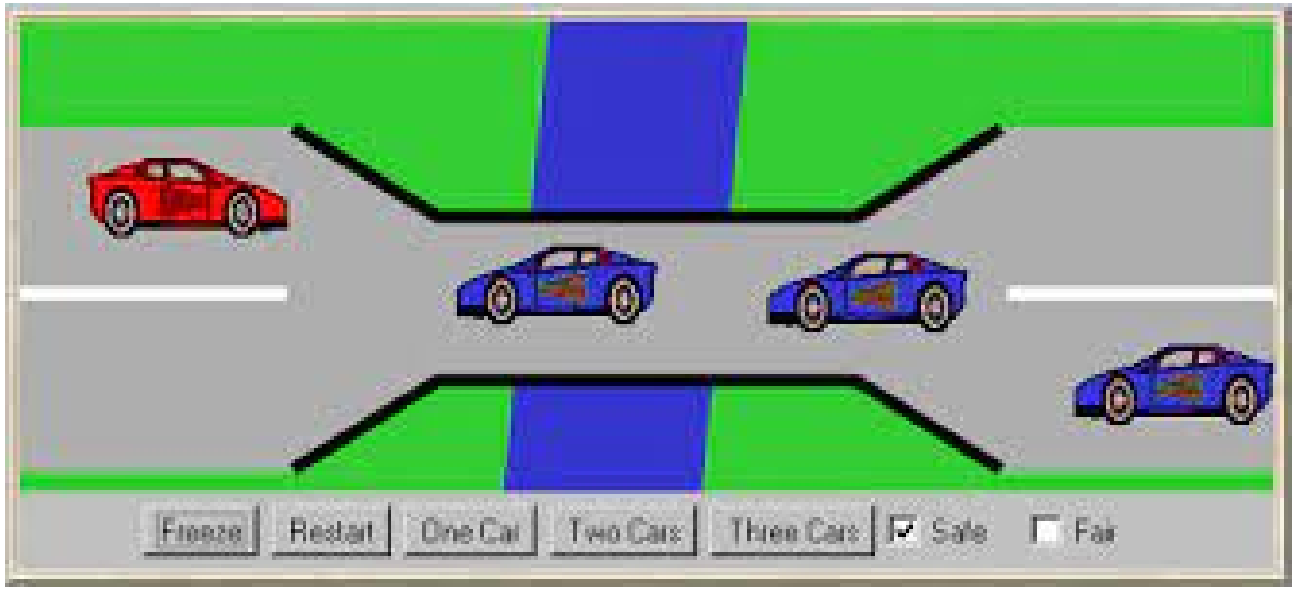
Ejercicio 5

Enunciado

Se desea modelar el control de tránsito de un puente que conecta dos ciudades. Dado que el puente es muy estrecho, se debe evitar que dos autos circulen al mismo tiempo en dirección opuesta. Tener en cuenta que atravesar el puente no es una acción atómica, y por lo tanto, requiere de cierto tiempo.
Por simplicidad podemos clasificar los autos que van en cada sentido como autos azules y rojos.

Pregunta: ¿La solución cumple con la propiedad starvation-freedom?

Solución



Pista 1: Utilizar variable autosCruzando = [0,0]
Pista 2: Utilizar los semáforos
permisoPuente = sem(1)
mutexOrillas = [sem(1), sem(1)]

Solución

```
permisoPuente = sem(1)
autosCruzando = [0,0]
mutexOrillas = [sem(1), sem(1)]

enum color {
    AZUL = 0,
    ROJO = 1
}

auto(color):
    mutexOrilla[color].wait()
    if (autosCruzando[color] == 0):
    // El primer auto para cada color/direccion debe esperar el permiso del puente
        permisoPuente.wait()
    autosCruzando[color]++
    mutexOrilla[color].signal()

    cruzar()

    mutexOrilla[color].wait()
    autosCruzando[color]--
    if (autosCruzando[color] == 0):
    // El ultimo auto en la misma direccion libera el puente
        permisoPuente.signal()
    mutexOrilla[color].signal()
```

Esta solución permite inanición!!

No hay garantías de que si empiezan a entrar los autos azules, eventualmente los autos rojos esperando entrar puedan pasar.

Ejercicio 6

Enunciado

Modelar con semáforos un micro de larga distancia directo entre Buenos Aires y Mendoza. El micro tiene capacidad para N personas y funciona de la siguiente manera:

- Empieza en Buenos Aires.
- Espera a llenarse.
- Viaja hasta Mendoza.
- Estaciona en una terminal, permitiendo que los pasajeros desciendan.
- Repite el procedimiento desde el principio pero desde la otra terminal.

Notar que para subir al micro no importa el orden de llegada. Además el micro permite que pasajeros puedan subir y bajar al mismo tiempo.

Solución

Pista 1: Definir dos tipos de procesos, bus y pasajero (con hasta n pasajeros)

Pista 2: Utilizar variables asientosOcupados (modela la cantidad de asientos ocupados) y terminal

Pista 3: Empiecen utilizando los semáforos

permisoSubir = [sem(0), sem(0)]

permisoBajar = sem(0)

asientos = sem(N)

mutex = sem(1)

Solución

```
enum Terminal {
    BUENOS_AIRES = 0,
    MENDOZA = 1
}

permisoSubir   = [sem(0), sem(0)]
asientos = sem(N)
permisoBajar = sem(0)
mutex = sem(1)
permisoViajar = sem(0)
asientosOcupados = [0,0]
```

```
bus(terminal):
  while true:
    repeat N:
        permisoSubir[terminal].signal()
    permisoViajar.wait()
    viajar()
    repeat N:
        permisoBajar.signal()
    terminal = 1 - terminal
```

```
pasajero(i, terminal):

  permisoSubir[terminal].wait()
  asientos.wait()
  subir()
  mutex.wait()
  asientosOcupados[terminal]++
  if (asientosOcupados[terminal] == N)
    permisoViajar.signal()
  mutex.signal()
  permisoBajar.wait()
  bajar()
  mutex.wait()
  asientosOcupados[terminal]--
  mutex.signal()
  asiento.signal()
```

Gracias por su atención!