Sincronización entre procesos

Sistemas Operativos

Segundo Cuatrimestre 2024

Estructura de la clase

Repaso

- Conceptos de sincro
- Ejecución concurrente y race conditions
- Memoria compartida

Atomicidad

- ¿Qué es una variable atómica?
- Uso de variables atómicas

Semáforos

- ¿Qué es un semáforo?
- Uso de semáforos
- Ejercicios

Conceptos de sincro

- Supongamos que tenemos dos **procesos** corriendo en simultáneo: `proceso A` y `proceso B`. En estos procesos ocurren **eventos**: `evento A` y `evento B`.

  ¿Cómo sabemos si `evento A` ocurre antes o después que `evento B`?

- En general, no tenemos control sobre el orden de ejecución de los procesos, eso es tarea del scheduler del SO.

Ejecución concurrente y race conditions

Concurrencia

Dos eventos son **concurrentes** si no podemos decir a simple vista en qué orden ejecutarán.

Condición de carrera

Defecto de un programa concurrente por el cual su correctitud depende del orden de ejecución de ciertos eventos.

Memoria compartida

- La mayoría de las variables en un proceso son **locales** al proceso, y otros procesos no las pueden ver ni acceder.
- Pero a veces tenemos **variables compartidas** entre dos o más procesos.
- Problemas de sincronización:
  - Escrituras concurrentes
  - Actualizaciones concurrentes
  - Lecturas concurrentes (?)

Ejercicio

¿Cuál es la salida esperada de los procesos **A** y **B** corriendo concurrentemente y asumiendo que la variable **x** reside en memoria compartida?

$x = 0$

A

```
x = x + 1;
print(x);
```

B

```
x = x + 1;
print(x);
```

Ojo con la atomicidad

- Pensemos qué pasa a nivel de máquina.
- Esta operación sobre memoria compartida involucra:
  1. Leer valor de x
  2. Operar (sumarle 1 a este valor)
  3. Escribir nuevo valor en x
- El scheduler puede interrumpir la operación en medio de alguna de estas 3 partes.
- Ejemplo de ejecución:
  - Proceso A: lee x, el valor es 0
  - Proceso A: suma 1 a x, el valor es ahora 1
  - Proceso B: lee x, el valor es 0
  - Proceso A: almacena 1 en x
  - Proceso B: suma 1 a x, el valor es ahora 1
  - Proceso B: almacena 1 en x

Ojo con la atomicidad

- En general no sabemos qué operaciones se realizan en un paso y cuáles pueden ser interrumpidas.
- Una operación que no puede ser interrumpida se llama **atómica**.
- ¿Cómo podemos solucionar el problema de race conditions en el problema de recién?

¿Qué es una variable atómica?

- Es un objeto que nos permite realizar operaciones de escritura y lectura de forma atómica.
- Almacenan un valor entero. Se puede interactuar con la variable mediante algunas primitivas como `getAndInc()` y `getAndAdd()`.
- Estas primitivas son atómicas a efectos de los procesos.

Uso de variables atómicas

Primitivas

- `getAndInc()`: Devuelve el entero atómico sumado 1.
- `getAndAdd(unsigned int value)|`: Devuelve el entero atómico sumado la cantidad especificada.
- `set(unsigned int value)`: Asigna al objeto un valor pasado por parámetro.

Más info en
`https://en.cppreference.com/w/cpp/atomic/atomic`.

¿Cómo solucionamos la pérdida de sumas del ejemplo?

Esquema de la solución

atomic<int> x

x.set(0)

A	B
x.getAndInc()	x.getAndInc()

¿Qué es un semáforo?

- Es un tipo abstracto de datos que permite controlar el acceso de múltiples procesos a un recurso común.
- Tiene un valor entero, al cuál no podemos acceder. La única manera de interactuar con el semáforo es mediante las primitivas `wait()` y `signal()`.
- Estas primitivas son atómicas a efectos de los procesos.

Uso de semáforos

Primitivas

- `sem(unsigned int value)`: Devuelve un nuevo semáforo inicializado en `value`.
- `wait()`: Mientras el valor sea menor o igual a 0 se bloquea el proceso esperando un `signal`. Luego decrementa el valor de `sem`.
- `signal()`: Incrementa en uno el valor del semáforo y despierta a **alguno** de los procesos que están esperando en ese semáforo.

Idea detrás de wait() y signal()

```
wait(s):
    while (s<=0) dormir();
    s--;

signal(s):
    s++;
    if (alguien espera por s) despertar a alguno;
```

Importante antes de arrancar

- ¿Puedo saber cuál es el proceso que se despierta en un signal?

  ¡No! Es determinístico pero depende de demasiadas variables.

- ¿Puedo asumir que el proceso que se despierta es el próximo en correr?

  ¡No! Es determinístico pero depende de demasiadas variables.

- ¿Puedo consultar el valor de un semáforo?

  ¡No! Revisar la interfaz, no hay observador.

Usando semáforos para el problema de sumas del ejemplo

Esquema de la solución

mutex = sem(1)

A	B
NoCrit()	NoCrit()
mutex.wait()	mutex.wait()
x = x + 1	x = x + 1
mutex.signal()	mutex.signal()
NoCrit()	NoCrit()

Usando semáforos para el problema de sumas del ejemplo

Esquema de la solución

mutex = sem(1)

A	**B**
NoCrit()	NoCrit()
mutex.wait()	mutex.wait()
Crit()	**Crit()**
mutex.signal()	mutex.signal()
NoCrit()	NoCrit()

Nota: Se puede generalizar este esquema para resolver el **problema de la sección crítica**.

Ejercicio 1

Enunciado

Se tienen un proceso productor P que hace `producir()` y dos procesos consumidores C_1, C_2 que hacen `consumir1()` y `consumir2()` respectivamente.
Se desea sincronizarlos tal que las secuencias de ejecución sean:
`producir, producir, consumir1, consumir2, producir, producir, consumir1, consumir2,...`

Solución:

permisoC1 = sem(0)
permisoC2 = sem(0)
permisoP = sem(1)

P	C1	C2
permisoP.wait()	permisoC1.wait()	permisoC2.wait()
producir()	consumir1()	consumir2()
producir()	permisoC2.signal()	permisoP.signal()
permisoC1.signal()		

Ejercicio 2

Enunciado

Se tienen 2 procesos A y B.
El proceso A tiene que ejecutar $A1()$ y luego $A2()$.
B debe ejecutar $B1()$ y después $B2()$.
En cualquier ejecución, $A1()$ tiene que ejecutarse antes que $B2()$.
Escribir el código con semáforos tal que cualquier ejecución cumpla lo pedido.

Solución:

permisoB = sem(0)

A	B
A1()	B1()
permisoB.signal()	permisoB.wait()
A2()	B2()

Ejercicio 3

Enunciado

Usando los procesos A y B del ejercicio anterior, ahora se quiere que $A1()$ y $B1()$ ejecuten antes de $B2()$ y $A2()$.

Solución?:

permisoB = sem(0)

permisoA = sem(0)

A	B
A1()	B1()
permisoA.wait()	permisoB.wait()
permisoB.signal()	permisoA.signal()
A2()	B2()

Pista: **No**

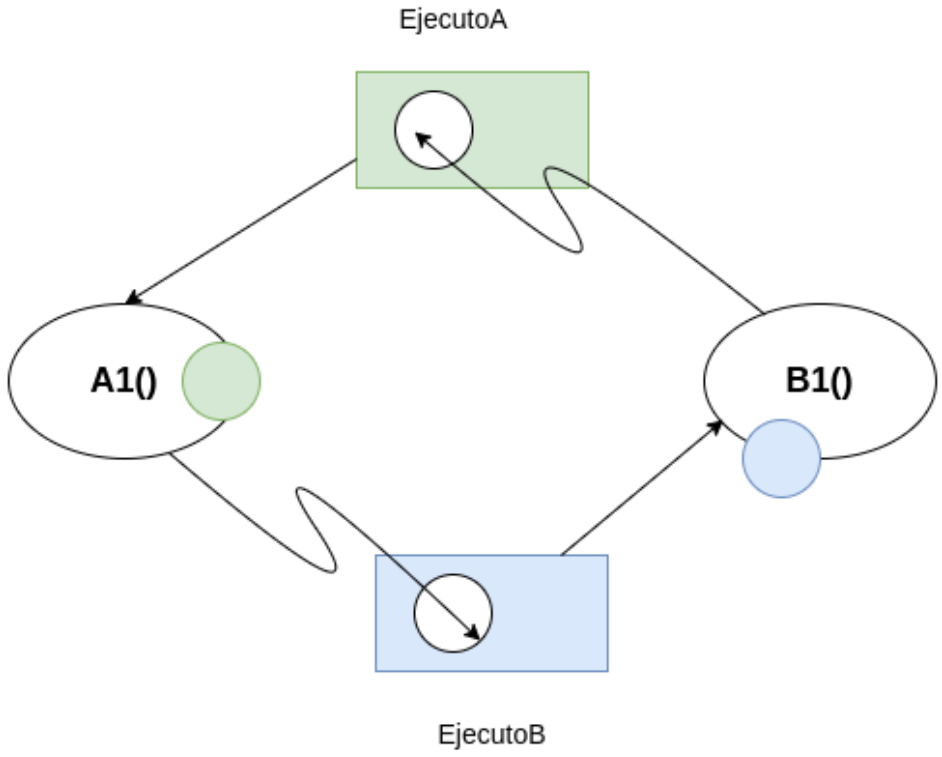
Problema: Existe un DEADLOCK

Algunos tips para identificar deadlocks:

- **Recursos Bloqueados:** Los procesos esperan indefinidamente por recursos que están siendo retenidos por otros procesos.
- **Espera Circular:** Existe una cadena circular de procesos, donde cada uno está esperando el recurso del siguiente proceso en la cadena.
- **No Liberación:** Un proceso mantiene recursos mientras espera otros, sin liberar los recursos que ya posee.
- **Tiempo de Espera Infinito:** Un proceso espera indefinidamente para entrar a una sección crítica.

Análisis de Grafos

Se puede pensar como un grafo bipartito donde la detección de ciclos en el grafo indica un deadlock.



Herramienta útil pero no definitiva (no escala). En Inge2 se ven Lógicas Lineales Temporales para analizar estas situaciones.

Ejercicio 3

Enunciado

Usando los procesos A y B de los ejemplos anteriores, pero quiero que A1 y B1 ejecuten antes de B2 y A2.

Solución:

permisoB = sem(0)

permisoA = sem(0)

A	B
A1()	B1()
permisoB.signal()	permisoA.signal()
permisoA.wait()	permisoB.wait()
A2()	B2()

Nota: Este patrón se suele llamar **rendezvous (punto de encuentro) o barrera**.

Ejercicio 4

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 uno, así que decidieron dividirse el trabajo. Sin embargo, acordaron que para que alguien pudiera `experimentar()` todos 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 repetitivas ahora más cortas de implementación y experimentación. Es decir, si luego de experimentar() queremos volver a implementarTP() y el resto del proceso sucesivamente. ¿Podemos reutilizar nuestra solución anterior?

Solución

Próxima clase...