Sistemas Operativos

Práctica 3: Sincronización entre procesos

Notas preliminares

- Los ejercicios marcados con el símbolo ★ constituyen un subconjunto mínimo de ejercitación. Sin embargo, aconsejamos fuertemente hacer todos los ejercicios.

Parte 1 – Sincronización entre procesos

Ejercicio 1

A continuación se muestran dos códigos de procesos que son ejecutados concurrentemente. La variable `X` es compartida y se inicializa en 0.

- Proceso A:

```
X = X + 1;
printf("%d", X);
```

- Proceso B:

```
X = X + 1;
```

Las variables `X` e `Y` son compartidas y se inicializan en 0.

- Proceso A:

```
for (; X < 4; X++) {
  Y = 0;
  printf("%d", X);
  Y = 1;
}
```

- Proceso B:

```
while (X < 4) {
  if  (Y == 1)
    printf("a");
}
```

No hay información acerca de cómo serán ejecutados por el *scheduler* para ninguno de los dos.

a) ¿Hay una única salida en pantalla posible para cada código?
b) Indicar todas las salidas posibles para cada caso.

Ejercicio 2

Se tiene un sistema con cuatro procesos accediendo a una variable compartida x y un *mutex*, el siguiente código lo ejecutan los cuatro procesos. Del valor de la variable dependen ciertas decisiones que toma cada proceso. Se debe asegurar que cada vez que un proceso lee de la variable compartida, previamente solicita el *mutex* y luego lo libera. ¿Estos procesos cumplan con lo planteado? ¿Pueden ser víctimas de *race condition*?

```
x = 0; // Variable compartida
mutex(1); // Mutex compartido

while (1) {
  mutex.wait();
    y = x; // Lectura de x
  mutex.signal();
  if (y <= 5) {
    x++;
  } else {
    x--;
  }
}
```

Ejercicio 3

La operación `wait()` sobre semáforos suele utilizar una cola para almacenar los pedidos que se encuentran en espera. Si en lugar de una cola utilizara una pila (LIFO), determinar si habría inanición o funcionaría correctamente.

Ejercicio 4 ★

Demostrar que, en caso de que las operaciones de semáforos `wait()` y `signal()` no se ejecuten atómicamente, entonces se viola la propiedad de exclusión mutua, es decir que un recurso no puede estar asignado a más de un proceso.

Pista: Revise el funcionamiento interno del `wait()` y del `signal()` mostrados en clase, el cual no se haría de forma atómica, y luego piense en una traza que muestre lo propuesto.

Ejercicio 5

Se tienen n procesos: $P_1, P_2, \cdots, P_n$ que ejecutan el siguiente código. Se espera que todos los procesos terminen de ejecutar la función `preparado()` antes de que alguno de ellos llame a la función `critica()`. ¿Por qué la siguiente solución permite inanición? Modificar el código para arreglarlo.

```
preparado()

mutex.wait()
count = count + 1
mutex.signal()

if (count == n)
  barrera.signal()

barrera.wait()

critica()
```

Ejercicio 6 ★

Cambie su solución del ejercicio anterior con una solución basada solamente en las herramientas atómicas vista en las clases, que se implementan a nivel de hardware, y responda las siguientes preguntas:

a) ¿Cuál de sus dos soluciones genera un código más legible?
b) ¿Cuál de ellas es más eficiente? ¿Por qué?
c) ¿Qué soporte requiere cada una de ellas del SO y del HW?

Ejercicio 7 ★

Se tienen N procesos, $\mathrm{P}_0, \mathrm{P}_1, ..., \mathrm{P}_{N-1}$ (donde N es un parámetro). Se requiere sincronizarlos de manera que la secuencia de ejecución sea $\mathrm{P}_i, \mathrm{P}_{i+1}, ..., \mathrm{P}_{N-1}, \mathrm{P}_0, ..., \mathrm{P}_{i-1}$ (donde i es otro parámetro). Escriba el código que deben ejecutar cada uno de los procesos para cumplir con la sincronización requerida utilizando semáforos (no olvidar los valores iniciales).

Ejercicio 8 ★

Considere cada uno de los siguientes enunciados, para cada caso, escriba el código que permita la ejecución de los procesos según la forma de sincronización planteada utilizando semáforos (no se olvide de los valores iniciales). Debe argumentar porqué cada solución evita la inanición:

1. Se tienen tres procesos (A, B y C). Se desea que el orden en que se ejecutan sea el orden alfabético, es decir que las secuencias normales deben ser: ABC, ABC, ABC, ...
2. Idem anterior, pero se desea que la secuencia normal sea: BBCA, BBCA, BBCA, ...
3. Se tienen un productor (A) y dos consumidores (B y C) que actúan no determinísticamente. La información provista por el productor debe ser retirada siempre 2 veces, es decir que las secuencias normales son: ABB, ABC, ACB o ACC. **Nota**: *¡Ojo con la exclusión mutua!*
4. Se tienen un productor (A) y dos consumidores (B y C). Cuando C retira la información, la retira dos veces. Los receptores actúan en forma alternada. Secuencia normal: ABB, AC, ABB, AC, ABB, AC...

Ejercicio 9

Suponer que se tienen N procesos P_i, cada uno de los cuales ejecuta un conjunto de sentencias a_i y b_i. ¿Cómo se pueden sincronizar estos procesos de manera tal que los b_i se ejecuten después de que se hayan ejecutado todos los a_i?

Ejercicio 10

Se tienen los siguientes dos procesos, `foo` y `bar`, que son ejecutados concurrentemente. Además comparten los semáforos `S` y `R`, ambos inicializados en 1, y una variable global `x`, inicializada en 0.

```
void foo( ) {
  do {
    semWait(S);
    semWait(R);
    x++;
    semSignal(S);
    semSignal(R);
  } while (1);
}
```

```
void bar( ) {
  do {
    semWait(R);
    semWait(S);
    x--;
    semSignal(S);
    semSignal(R);
  } while (1);
}
```

a) ¿Puede alguna ejecución de estos procesos terminar en *deadlock*? En caso afirmativo, describir una traza de ejecución.

b) ¿Puede alguna ejecución de estos procesos generar inanición para alguno de los procesos? En caso afirmativo, describir una traza.

Ejercicio 11 (*Read y Write*)

Se quiere simular la comunicación mediante pipes entre dos procesos usando las syscalls **read()** y **write()**, pero usando memoria compartida (sin usar **file descriptors**). Se puede pensar al pipe como si fuese un buffer de tamaño N, donde en cada posición se le puede escribir un cierto mensaje. El **read()** debe ser bloqueante en caso que no haya ningún mensaje y si el buffer está lleno, el **write()** también debe ser bloqueante. No puede haber condiciones de carrera y se puede suponer que el buffer tiene los siguientes métodos: **pop()** (saca el mensaje y lo desencola), **push()** (agrega un mensaje al buffer).