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.

Ejercicio 1

Se tienen dos procesos A y B que ejecutan concurrentemente. No se tiene información sobre cómo serán ejecutados por el *scheduler*. Para cada par A-B detallado a continuación, responder:

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

```
// variables compartidas
x = 0;

A():
  x = x + 1;
  printf("%d", x)

B():
  x = x + 1;
```

```
// variables compartidas
x = 0;
y = 0;

A():
  for (; x < 4; x++) {
    y = 0;
    printf("%d", x);
    y = 1;
  }

B():
  while (x < 4) {
    if  (y == 1)
      printf("a");
  }
```

Ejercicio 2 ★

Se tiene un sistema con 4 procesos accediendo a una variable compartida x y un *mutex*. Los 4 procesos ejecutan el siguiente código. Ciertas decisiones que toma cada proceso dependen del valor de la variable compartida. Se debe asegurar que cada vez que un proceso lee 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 operació `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 habrá inanición o funcionará 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: Revisar el funcionamiento interno del `wait()` y del `signal()` mostrados en clase, el cual no se hará de forma atóica, y luego pensar 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 ★

Cambiar la solución del ejercicio anterior por una solución basada solamente en las herramientas atómicas vistas en las clases, que se implementan a nivel de hardware, y responder las siguientes preguntas:

a) ¿Cuál de las 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, $P_0, P_1, ..., P_{N-1}$ (donde N es un parámetro). Se requiere sincronizarlos de manera que la secuencia de ejecución sea $P_i, P_{i+1}, ..., P_{N-1}, P_0, ..., P_{i-1}$ (donde i es otro parámetro). Escribir 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 ★

Considerar cada uno de los siguientes enunciados. Para cada caso, escribir el código que permita la ejecución de los procesos según la forma de sincronización planteada utilizando semáforos (no olvidar los valores iniciales). Se 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 mediante las syscalls **read()** y **write()**, pero usando memoria compartida (sin usar **file descriptors**). Se puede pensar al pipe como 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).

Ejercicio 12 (*Dividir y conquistar el TP*) ★

Un grupo de N estudiantes se dispone a hacer un TP de su materia favorita (Sistemas Operativos).

Cada estudiante conoce a la perfección cómo `implementarTp()` y cómo `experimentar()`. Curiosamente, cada una de estas acciones puede ser llevada acabo de manera independiente por cada uno, así que decidieron dividirse el trabajo.

Acordaron dividir el trabajo en varias etapas. En cada etapa, todos los estudiantes deben primero `implementarTp()`, y recién cuando todos hayan terminado, pueden empezar a `experimentar()`. Luego, para poder comenzar la siguiente etapa y volver a implementar, todos deben haber terminado de experimentar con la etapa anterior.

Se pide diseñar un programa concurrente que utilice procesos y que modele esta situación utilizando semáforos.

Ejercicio 13 (*Lavarropas automático*) ★

Se tiene un único lavarropas que puede lavar 10 prendas y para aprovechar al máximo el jabón nunca se enciende hasta estar **totalmente lleno**.

Escribir el pseudocódigo que resuelva este problema de sincronización, indicando los semáforos utilizados y sus respectivos valores iniciales, teniendo en cuenta los siguientes requisitos:

a) El lavarropas invoca `estoyListo()` para indicar que la ropa puede empezar a ser cargada, e invoca `lavar()` una vez que está totalmente lleno. Al terminar el lavado invoca a `puedenDescargarme()`. Una vez vacío, el lavarropas espera prendas nuevamente.

b) Cada prenda invoca `entroAlLavarropas()` una vez que el lavarropas está listo, y no pueden ingresar dos prendas al lavarropas al mismo tiempo. No es necesario tener en cuenta el orden de llegada de las prendas para introducirlas en el lavarropas, cualquier orden es permitido. Una vez que el lavarropas indicó que puede ser descargado, cada prenda invoca `saquenmeDeAquí()` y termina su proceso. Las prendas pueden salir todas a la vez.