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 **EXCL** (exclusión mutua: 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 (*Problema de los prisioneros*)

P prisioneros están encarcelados. Para salir de prisión se les propone el siguiente problema.

- Los prisioneros tienen un día para planear una estrategia. Después, permanecerán en celdas aisladas *sin ninguna* comunicación.
- Hay una sala con una luz y un interruptor. La luz puede estar prendida (interruptor *on*) o apagada (interruptor *off*).
- De vez en cuando, un prisionero es llevado a esa sala y tiene derecho cambiar el estado del interruptor o dejarlo como está.
- Se garantiza que *todo* prisionero va a entrar a la sala *infinitas* veces.
- En cualquier momento, cualquier prisionero puede declarar "*todos los prisioneros hemos visitado la sala al menos una vez*".
- Si la declaración es correcta, los prisioneros serán liberados. Si no, quedarían encerrados para siempre.

El problema de los prisioneros consiste en definir una estrategia que permita liberar a los prisioneros sabiendo que el estado inicial del interruptor es *off* (luz apagada) y considerando que no todos los prisioneros tienen por qué hacer lo mismo en cada momento.

Considere el siguiente código:

```
void PrisioneroCero(){

  contador = 0;
  while (!libres) {

    // Esperar que la sala esté libre
    ...

    if (!luz){
      luz = true
      contador++
      if(contador == N)  libres = true;
    }
  }
}

// Resto de los prisioneros
void RestoDeLosPrisionero(int i){

  entreASala = false;

  while (!libres) {

    // Esperar que la sala esté libre
    ...
```

```
      if (luz && !entreASala){
        entreASala = true
        luz = false;
      }
    }
  }
```

a) Defina el tipo de las variables utilizadas (considere además los tipos atómicos) para el correcto funcionamiento de la solución.
b) Modificar el código de los prisioneros para que entren a la sala de a uno por vez. Explicar porqué su solución garantiza la exclusión mutua sobre la sala.

Ejercicio 12 (*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).

Ejercicio 13 (*Game Pool Party!*) ★

Se tiene un bar de juegos de mesa donde se encuentran **N** mesas con capacidad para 4 personas. Los clientes entran constantemente y van ocupando las mesas hasta llenar su capacidad, preferentemente aquellas que ya tienen clientes en la mesa. Cuando recien una mesa tiene 4 personas, cada cliente debe invocar la función **jugar()**. Cuando las **N** mesas están ocupadas, los nuevos clientes tienen que esperar a que se libere alguna mesa. Las mesas solo se liberan todas juntas, es decir, los clientes terminan de jugar y abandonan la mesa. Escribir un código que reproduzca este comportamiento. Suponga que tiene una función **conseguirMesa()** que cuando es ejecutada, devuelve el número de mesa que el cliente tiene que ir. La función es bloqueante cuando ya no hay más mesas libres. Cuando los jugadores dejan la mesa, se debe llamar a una función **abandonarMesa(i)**, donde **i** es el número de mesa actual.

Ejercicio 14 (*El problema del barbero,* recargado[1].)

Se tiene un negocio con tres barberos, con sus respectivas tres sillas. Al igual que en el ejemplo clásico, se tiene una sala de espera, pero en la sala se encuentra un sofá para cuatro personas. Además, las disposiciones municipales limitan la cantidad de gente dentro del negocio a 20 personas.

Al llegar un cliente nuevo, si el negocio se encuentra lleno, se retira. En caso contrario, entra y una vez adentro se queda parado hasta que le toque turno de sentarse en el sofá. Al liberarse un lugar en el sofá, el cliente que lleva más tiempo parado se sienta. Cuando algún barbero se libera, aquel que haya estado por más tiempo sentado en el sofá es atendido. Al terminar su corte de pelo, el cliente le paga a **cualquiera** de los barberos y se retira. Al haber una única caja registradora, los clientes pueden pagar de a uno por vez.

Resumiendo, los clientes deberán hacer en orden: `entrar`, `sentarseEnSofa`, `sentarseEnSilla`, `pagar` y `salir`. Por otro lado, los barberos: `cortarCabello` y `aceptarPago`. En caso que no hayan clientes, los barberos se duermen esperando que entre un cliente.

Escribir un código que reproduzca este comportamiento utilizando las primitivas de sincronización vistas en la materia.

[1] Extraido de Stallings, William. *Operating Systems: Internals and Design Principles*, Edition 8. Pearson, 2014.

Ejercicio 15 (*El crucero de Noel*) ★

En el crucero de Noel queremos guardar parejas de distintas especies (no sólo un solo individuo por especie). Hay una puerta por cada especie. Los animales forman fila en cada puerta, en dos colas, una por sexo. Queremos que entren en parejas. Programar el código de cada proceso (animal) P(i, sexo). Pista: usar dos semáforos por especie y la función `entrar(i)`.

Ejercicio 16 (*La cena de los antropófagos* - The Dining Savages[2])

Una tribu de antropófagos cena usando una gran cacerola que puede contener M porciones de misionero asado. Cuando un antropófago quiere comer se sirve de la cacerola, excepto que esté vacía. Si la cacerola está vacía, el antropófago despierta al cocinero y espera hasta que éste rellene la cacerola.

Pensar que, sin sincronización, el antropófago hace:

```
while (true) {
  tomar_porcion();
  comer();
}
```

La idea es que el antropófago no pueda comer si la cacerola está vacía y que el cocinero sólo trabaje si está vacía la cacerola.

a) Complete el código mostrado para que los procesos tipo antropófagos, y el proceso tipo cocinero, cumplan con la idea establecida. El código del proceso cocinero debe escribirse aparte de los antropófagos.

Ejercicio 17 ★

Somos los encargados de organizar una fiesta, y se nos encomendó llenar las heladeras de cerveza. Cada heladera tiene capacidad para 15 botellas de 1 litro y 10 porrones. Los porrones no pueden ser ubicados en el sector de botellas y viceversa.

Para no confundirnos, las heladeras hay que llenarlas en orden. Hasta no llenar completamente una heladera (ambos tipos de envases), no pasamos a la siguiente. Además, debemos enchufarlas antes de empezar a llenarlas. Una vez llena, hay que presionar el botón de enfriado rápido.

Al bar llegan los proveedores y nos entregan cervezas de distintos envases al azar, no pudiendo predecir el tipo de envase.

El modelo por computadora de este problema tiene dos tipos de procesos: *heladera* y *cerveza*. La operaciones disponibles en los procesos *heladera* son: `EnchufarHeladera()`, `AbrirHeladera()`, `CerrarHeladera()` y `EnfriadoRapido()`.

Por otro lado, los procesos *cerveza* tienen las operaciones: `LlegarABar()` y `MeMetenEnHeladera()`. La función `MeMetenEnHeladera()` debe ejecutarse de a una cerveza a la vez (con una mano sostenemos la puerta y con la otra acomodamos la bebida).

Una vez adentro de la *heladera*, el proceso *cerveza* puede terminar. Al llenarse el proceso *heladera* debemos continuar a la siguiente luego de presionar el botón de enfriar (`EnfriadoRapido()`).

Utilizando las primitivas de sincronización vistas en clase, escribir el pseudocódigo de los procesos H(i) (heladera) y C(i, tipoEnvase) (cerveza) que modelan el problema. Cada heladera está representada por una instancia del proceso H y cada cerveza por una instancia del proceso C. Definir las variables globales necesarias (y su inicialización) que permitan resolver el problema y diferenciar entre los dos tipos de cervezas.

[2]Sacado del libro Andrews, Gregory R. *Concurrent Programming: Principles and Practice*. Addison-Wesley, 1991.

Ejercicio 18

Se tiene un único lavarropas que puede lavar 10 prendas y para aprovechar al máximo el jabón nunca se enciende hasta estar *totalmente lleno*. Suponer que se tiene un proceso L para simular el lavarropas y un conjunto de procesos P(i) para representar a cada prenda.

Escribir el pseudocódigo que resuelva este problema de sincronización teniendo en cuenta los siguientes requisitos:

- El proceso L invoca `estoyListo()` para indicar que la ropa puede empezar a ser cargada.
- Un proceso P(i) invoca `entroAlLavarropas()` una vez que el lavarropas está listo. No pueden ingresar dos prendas al lavarropas al mismo tiempo. Ver aclaración.
- El lavarropas invoca `lavar()` una vez que está totalmente lleno. Al terminar el lavado invoca a `puedenDescargarme()`.
- Cada prenda invoca `saquenmeDeAquí()` una vez que el lavarropas indicó que puede ser descargado y termina su proceso. Las prendas **sí** pueden salir todas a la vez.
- Una vez vacío, el lavarropas espera nuevas prendas mediante `estoyListo()`.

Aclaración: no es necesario tener en cuenta el orden de llegada de las prendas para introducirlas en el lavarropas. Cualquier orden es permitido.

Ejercicio 19 (*Babuinos Cruzando* - Baboon crossing problem[3]) ★

En el parque nacional Kruger en Sudáfrica hay un cañón muy profundo con una simple cuerda para cruzarlo. Los babuinos necesitan cruzar constantemente en ambas direcciones mediante esta cuerda.

Como los babuinos son muy agresivos, si dos de ellos se encuentran en cualquier punto de la cuerda yendo en direcciones opuestas, estos se pelearán y terminarán cayendo por el cañón.

La cuerda no es muy resistente y aguanta a un máximo de cinco babuinos simultáneamente. Si en cualquier instante hay más de cinco babuinos en la cuerda, ésta se romperá y los babuinos caerán también al vacío.

a) Asumiendo que le podemos enseñar a los babuinos a usar semáforos, diseñar un esquema de sincronización con las siguientes propiedades:

   - Una vez que un babuino ha comenzado a cruzar, se garantiza que llegará al otro lado sin encontrarse con un babuino que vaya en la dirección opuesta.
   - Nunca hay más de 5 babuinos en la cuerda.

b) ¿La solución propuesta permite inanición? Como la modificaría para cumplir con:

   - Un flujo continuo de babuinos cruzando en una dirección no debe impedir indefinidamente a los babuinos que vayan en la dirección opuesta (sin inanición).

[3]Sacado del libro Tanenbaum's *Operating Systems: Design and Implementation*. Pearson, 1987.