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 procesos que son ejecutados concurrentemente. No hay información acerca de cómo serán ejecutados por el scheduler. Indicar la salida en cada uno de los casos. Si existen varias opciones, indicar todas las salidas posibles. La variable X es compartida y se inicializa en 0.

- Proceso A:

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

- Proceso B:

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

Ejercicio 2

Idem ejercicio anterior. 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");
}
```

Ejercicio 3

Idem ejercicio anterior.

- Proceso A:

```
while (X == 0) {
  // no hacer nada
}
printf("a");
Y = 1;
Y = 0;
printf("d");
Y = 1;
```

- Proceso B:

```
printf("b");
X = 1;
while (Y == 0) {
  // no hacer nada
}
printf("c");
```

Ejercicio 4

Se tiene un sistema con cuatro procesos accediendo a una variable compartida y un mutex. Del valor de la variable dependerán ciertas decisiones que tome cada proceso. Nos aseguran que cada vez que un proceso accede a la variable compartida, previamente se solicita el mutex. ¿Es posible escribir procesos que cumplan con estas características, pero que puedan ser víctimas de una *race condition*?

Ejercicio 5

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), ¿qué problemas podrían suceder?

Ejercicio 6 ★

Demostrar que, en caso de que las operaciones de semáforos `wait()` y `signal()` no se ejecuten atómicamente, entonces se viola el principio de exclusión mutua. Pista: mostrar un scheduling posible.

Ejercicio 7 ★

Dekker desarrolló la primera solución correcta para el problema de la sección crítica para dos procesos. Los dos procesos, P_0 y P_1, comparten las variables siguientes:

```
boolean flag[2]; /* Inicializado en FALSE */
int turn = 0;
```

La estructura del proceso P_i (con $i = 0$ o $i = 1$) es la siguiente:

```
int other = 1 - i;
do {
  flag[i] = TRUE;
```

```
  while (flag[other]) {
    if (turn == other) {
      flag[i] = FALSE;
      while (turn == other); // Espero, no hago nada
        flag[i] = TRUE;
    }
  }

  // Sección crítica.

  turn = other;
  flag[i] = FALSE;

  // Sección restante.
} while (TRUE);
```

Demostrar que el algoritmo satisface los tres requisitos del problema de la sección crítica vistos en la teórica. Pista: demostrar por el absurdo.

Ejercicio 8

El siguiente código es una solución al problema de exclusión mutua, que fuera publicado en *Communications of the ACM*. Encontrar un contraejemplo, en donde esta solución falla.

```
boolean blocked[2];
int turn;
void P(int id){
  while(true) {
    blocked[id] = true;
    while(turn != id) {
      while(blocked[1-id]) {
        /*skip*/
      }
      turn = id;
    }
    /* critical section */
    blocked[id] = false;
    /* remainder */
  }
}

void main() {
  blocked[0] = false;
  blocked[1] = false;
  turn = 0;
  parbegin(P(0), P(1));
}
```

Pista: suponer que el primer proceso en ejecutarse es P(1) y es desalojado **justo antes** de ejecutar la línea 8.

Parte 2 – Primitivas de sincronización

Ejercicio 9 ★

Escriba el código con semáforos (no se olvide de los valores iniciales) para los siguientes problemas:

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 consumidor 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 10

Construya, utilizando semáforos, el código que permita realizar lo siguiente, dos emisores A y C y un receptor B que debe retirar la información depositada por cada emisor dos veces seguidas y actúan en forma alternada (secuencia normal esperada es ABBCBBABBCBB ....).

Ejercicio 11

Dada la siguiente secuencia lógica y los valores iniciales de los semáforos contadores :

X	Y	Z
P(S)	P(R)	P(R)
...	P(C)	P(B)
...	...	...
V(R)	V(B)	V(C)
	V(S)	V(S)

con valores iniciales S = 1, R = 0, B = 0 y C = 1 y cuya secuencia normal de ejecución sería: `XYXZXYXZ`

a) ¿Qué sucede si se presenta la secuencia : XZY ?
b) ¿Podría agregarse alguna rutina (X,Y, o Z) a continuación que permitiera salvar la situación?
c) Si se cambian las rutinas Y y Z de la siguiente manera:

Y	Z
P(C)	P(B)
P(R)	P(R)
...	...
V(S)	V(S)
V(B)	V(C)

¿qué sucede ahora con la secuencia del punto a) ?
d) ¿Cuál es la razón por la que esta simple inversión solucionó el problema?

Ejercicio 12 ★

Se tienen N procesos, $P_0, P_1, ..., P_{N-1}$ (donde N es un parámetro). Se los quiere sincronizar 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).

Escriba el código utilizando semáforos (no se olvide de los valores iniciales), para solucionar este problema de sincronozación.

Ejercicio 13

Suponga que se tienen N procesos P_i, cada uno de los cuales ejecuta un conjunto de sentencias a_i y b_i. Se los quiere sincronizar de manera tal que los b_i se ejecuten después de que se hayan ejecutado todos los a_i.

Parte 3 – Deadlock

Ejercicio 14 ★

Mostrar cómo se cumplen las cuatro condiciones necesarias para el *deadlock* en el problema de los filósofos, cuando éstos agarran un palillo cada uno.

Ejercicio 15 ★

¿Es posible tener un *deadlock* existiendo un único proceso?

Ejercicio 16

En un sistema conviven 3 procesos y 2 recursos. Uno de los recursos (R2) es de uso exclusivo y el otro (R1) puede ser compartido por hasta dos procesos. ¿Puede haber deadlock? ¿ Y si ahora R1 puede ser compartido por hasta tres procesos?

Ejercicio 17

Considere un sistema con 4 recursos del mismo tipo, que son compartidos por 3 procesos donde cada uno de ellos necesita a lo sumo 2. En cuanto los tiene, procesa por una cantidad de tiempo y termina. Muestre que está libre de *deadlock*.

Ejercicio 18

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);
}
```

1. ¿Puede alguna ejecución de estos procesos terminar en *deadlock*? En caso afirmativo, describir la secuencia y mostrar que se cumplen las 4 condiciones de Coffman.
2. ¿Puede alguna ejecución de estos procesos generar inanición para alguno de los procesos? En caso afirmativo, describir la secuencia.

Ejercicio 19 ★

Se tiene un sistema que en un determinado momento tiene 5 procesos que están compartiendo 4 recursos. Dada las tablas de asignación y necesidad de recursos:

Asigación	R1	R2	R3	R4
P1	0	1	0	0
P2	2	0	0	1
P3	3	0	3	0
P4	2	1	1	1
P5	0	0	2	0

Necesidad	R1	R2	R3	R4
P1	0	0	0	0
P2	2	0	2	0
P3	0	0	0	0
P4	1	0	0	0
P5	0	0	2	1

Teniendo en cuenta que existen 7 instancias de R1, 2 de R2, 6 de R3 y 2 de R4. Se pide:

1. Pruebe que el sistema se encuentra libre de *deadlock*.
2. Si se altera la tabla de necesidad para reflejar que el proceso P3 requiere dos instancias más del recurso R2, determine ahora si el sistema está en *deadlock*. De ser así, indique los procesos involucrados. Justifique.

Parte 4 – Problemas de sincronización

Ejercicio 20 *(El problema del barbero,* reloaded[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 uno.

Escribir un código que reproduzca este comportamiento utilizando semáforos.

Ejercicio 21 *(El Crucero de Noel)* ★

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

Ejercicio 22 *(La cena de los antropófagos (o* The Dinning Savages*)*[2]*)*

Una tribu de antropófagos cena de una gran carerola 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:

[1] Extraido del libro de Stallings.

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

```
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.

Ejercicio 23 ★

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 semáforos, 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.

Ejercicio 24

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.

Escriba el pseudocódigo que resuelva este problema de sincronización, indicando los semáforos utilizados y sus respectivos valores iniciales 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 SI 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 25 ★

La CNRT recibió una denuncia reclamando que muchas líneas de colectivos no recogen a los pasajeros. Ellos sospechan que al no haber suficientes colectivos, estos se llenan muy rápidamente. Debido

a esto, pidieron construir un simulador que comprenda a los actores e interacciones involucradas.

El simulador debe contar con dos tipos de procesos, colectivero y pasajero. Cada parada está representada por su número en el recorrido. Hay N paradas y M colectivos.

Cada pasajero comienza esperando en una parada (la cual recibe por parámetro) detrás de las personas que ya se encontraban en ella (de haberlas). Una vez que el colectivo llega y el pasajero logra subir, le pide al colectivero que le marque la tarifa con la función `pedirle125()`. Esta función devuelve el número de colectivo. Luego espera que el colectivero haga `marcarTarifa()` y finalmente el pasajero ejecuta `pagarConSUBE()` (se asume que todos los pasajeros tienen SUBE). Luego de pagar, procede a `viajar()`. Cuando el pasajero termina de `viajar()`, efectúa `dirigirseAPuertaTrasera()` y una vez que el colectivo se detiene, los pasajeros que estan agrupados en la puerta trasera realizan `bajar()` de a uno por vez, sin importar el orden.

El colectivero recibe como parámetro la capacidad (cantidad de de pasajeros) del colectivo, que, ni bien comienza, está vacío y el identificador del colectivo (entre 0 y M).

Al llegar a una parada, el colectivero, se detiene con `detener()`. Si hay pasajeros esperando para bajar, este abre su puerta trasera para indicar que ya pueden hacerlo (`abrirPuertaTrasera()`). Mientras esto sucede, si hay pasajeros en la parada, abre la puerta delantera (`abrirPuertaDelantera()`) y las personas que esperaban en la parada comienzan a ascender en orden siempre y cuando haya capacidad (notar que a esta altura ya se sabe cuántos pasajeros van a bajar).

Las personas proceden a subir y el colectivero, amablemente, los atiende de a uno marcando en la máquina con `marcarTarifa()`, pero nunca lo hace antes de que el anterior haya terminado de `pagarConSUBE()`. Si no hay más pasajeros para subir o se llegó al límite de capacidad, el colectivero no duda en `cerrarPuertaDelantera()`, impidiendo que el resto de las personas en la parada ascienda.

Una vez que los pasajeros terminan de ascender, este espera a que terminen de descender todos los pasajeros que así lo desean, procede a `cerrarPuertaTrasera()` y `avanzar()` nuevamente a la siguiente parada, donde la dinámica será la misma.

Puede asumir:

- El colectivero se detiene en todas las paradas y abre la puerta delantera, sin importar si hay pasajeros esperando para bajar o subir.
- El cálculo de la cantidad máxima de pasajeros que el colectivero deja subir puede realizarse considerando la cantidad de pasajeros que ya se sabe que descenderán (no es un problema que brevemente se vea superada la capacidad del colectivo mientras se espera que terminen de descender).