

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. La variable **X** es compartida y se inicializa en 0. No hay información acerca de cómo serán ejecutados por el *scheduler*.

- Proceso A:

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

- Proceso B:

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

- a) ¿Hay una única salida en pantalla posible?
- b) Si existen varias opciones, indicar todas las salidas posibles.

Ejercicio 2

A continuación se muestran procesos que son ejecutados concurrentemente. Las variables **X** e **Y** son compartidas y se inicializan en 0. No hay información acerca de cómo serán ejecutados por el *scheduler*.

- Proceso A:

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

- Proceso B:

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

- a) ¿Hay una única salida en pantalla posible?
- b) Si existen varias opciones, indicar todas las salidas posibles.

Ejercicio 3

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 cumplen 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 4

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), ¿cuáles de las siguientes propiedades podrían afectarse: **EXCL**, **FAIRNESS**, **WAIT-FREEDOM**?

Ejercicio 5 ★

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).

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 6

Se tienen n procesos: P_1, P_2, \dots, 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 rompe la propiedad **WAIT-FREEDOM**? Solucionar el código y justificar por qué su solución cumple con la propiedad.

```
preparado()

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

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

barrera.wait()

critica()
```

Ejercicio 7 ★

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:

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

Ejercicio 8 ★

Escribir un procedimiento llamado `TryToLock()` que intenta tomar un lock al estilo CAS (Compare_And_Swap) pero retorna el control si no lo puede lograr, en lugar de quedarse haciendo espera activa (busy waiting). Ayuda: Revise el comportamiento de CAS.

Ejercicio 9 ★

Se tienen N procesos, P_0, P_1, \dots, P_{N-1} (donde N es un parámetro). Se los quiere sincronizar de manera que la secuencia de ejecución sea $P_i, P_{i+1}, \dots, P_{N-1}, P_0, \dots, P_{i-1}$ (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 10 ★

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 por qué cada solución cumple con la propiedad de **LOCK-FREEDOM**:

- 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, ...
- Idem anterior, pero se desea que la secuencia normal sea: BBBCA, BBBCA, BBBCA, ...
- 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!
- 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 11

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 los puede sincronizar de manera tal que los b_i se ejecuten después de que se hayan ejecutado todos los a_i ?

Ejercicio 12 ★

Una definición de *deadlock* muy difundida en la literatura es la siguiente: “En computación concurrente, un *deadlock* es un estado en el cual cada miembro de un grupo de acciones está esperando que otro miembro libere un *lock*”.¹

Considerar los siguientes procesos que comparten el mutex r .

¹Fuente: Wikipedia que cita a su vez a Coulouris, George (2012). *Distributed Systems Concepts and Design*. Pearson. p. 716. ISBN 978-0-273-76059-7.

```
proceso p1 { r.lock; while (true) {sleep();} }  
proceso p2 { r.lock;}
```

- Estos procesos, ¿pueden completar su ejecución?
- Tal situación, ¿está dentro de la definición de *deadlock* propuesta?

Considere ahora un proceso adicional cuyo código es el siguiente y vuelva a responder las preguntas anteriores:

```
void f() { r.lock; f(); r.unlock; }  
proceso p3 { f(); }
```

Ejercicio 13 ★

En el año 1971 Edward Coffman y colaboradores propusieron cuatro condiciones necesarias para que se dé un *deadlock*², que se conocen como condiciones de Coffman:

- Condición de exclusión mutua: existencia de al menos de un recurso compartido por los procesos, al cual solo puede acceder uno simultáneamente.
- Condición de retención y espera: al menos un proceso P_i ha adquirido un recurso R_i , y lo retiene mientras espera al menos un recurso R_j que ya ha sido asignado a otro proceso.
- Condición de no expropiación: los recursos no pueden ser expropiados por los procesos, es decir, los recursos solo podrán ser liberados voluntariamente por sus propietarios.
- Condición de espera circular: dado el conjunto de procesos P_0, \dots, P_m (subconjunto del total de procesos original), P_0 está esperando un recurso adquirido por P_1 , que está esperando un recurso adquirido por P_2 , ..., que está esperando un recurso adquirido por P_m , que está esperando un recurso adquirido por P_0 . Esta condición implica la condición de retención y espera.

Considerar los conjuntos de procesos del ejercicio anterior y responder:

- ¿Se cumplen las condiciones de Coffman?
- ¿Bajo qué condiciones considera que estas condiciones son realmente necesarias para que haya *deadlock*?

Ejercicio 14

En un sistema conviven 3 procesos y 2 recursos. Uno de los recursos (R_2) es de uso exclusivo y el otro (R_1) puede ser compartido por hasta dos procesos. ¿Puede haber *deadlock*? ¿Y si ahora R_1 puede ser compartido por hasta tres procesos? Al explicar su razonamiento explicita las condiciones que deben cumplirse para que sea válido.

Ejercicio 15

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 de estos recursos. Cuando un proceso los tiene, procesa por una cantidad de tiempo y termina. Explicar por qué cumpliría la propiedad de **DEADLOCK-FREEDOM**, explicitando las condiciones que supone que se cumplen para que su razonamiento sea válido.

²Coffman, Edward G., Melanie Elphick, and Arie Shoshani. *System deadlocks*. ACM Computing Surveys (CSUR) 3.2 (1971): 67-78.

Ejercicio 16

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);
}
```

- ¿Puede alguna ejecución de estos procesos terminar en *deadlock*? En caso afirmativo, describir una traza.
- ¿Puede alguna ejecución de estos procesos generar inanición para alguno de los procesos? En caso afirmativo, describir una traza.

Ejercicio 17 (*Problema de los prisioneros*)

P prisioneros son 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 *infinitamente* seguido.
- 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.

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;
        }
    }
}

```

- Defina el tipo de las variables utilizadas (considere además los tipos atómicos) para el correcto funcionamiento de la solución.
- Modificar el código de los prisioneros para que entren a la sala de a uno por vez. Explicar por qué su solución garantiza la propiedad **EXCL** sobre la sala.
- ¿Por qué es importante la propiedad de **FAIRNESS** para la solución? ¿Cómo puede garantizar que se cumpla?
- Sea $\mathcal{P} = \{0, \dots, N\}$ el conjunto de prisioneros y $E \in \{on, off\}$ el estado del interruptor, utilice el modelo de Lynch para demostrar que la secuencia τ de eventos termina en la declaración de libertad sólo cuando todos los prisioneros entraron en la sala. Para ello divida τ en N segmentos $\tau^1 \dots \tau^N$, tal que el último estado de τ^i es $(E = on, contador = i)$ y el estado previo a éste es $(E = off, contador = i - 1)$.

Ejercicio 18 (*El problema del barbero*, recargado³.)

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.

³Extraído de Stallings, William. *Operating Systems: Internals and Design Principles*, Edition 8. Pearson, 2014.

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

Ejercicio 19 (*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 $P(i, \text{sexo})$. Pista: usar dos semáforos y la función `entrar(i)`.

Ejercicio 20 (*La cena de los antropófagos* - The Dining Savages⁴)

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.

- 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 21 ★

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*. Las operaciones disponibles en los procesos *heladera* son: `EnchufarHeladera()`, `AbrirHeladera()`, `CerrarHeladera()` y `EnfriadoRapido()`.

Por otro lado, los procesos *cerveza* tienen las operaciones: `LlegarABar()` y `MeMettenEnHeladera()`. La función `MeMettenEnHeladera()` 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 $C(i, \text{tipoEnvase})$ (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.

⁴Sacado del libro Andrews, Gregory R. *Concurrent Programming: Principles and Practice*. Addison-Wesley, 1991.

Ejercicio 22

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 resuelve 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 23 ★

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**. Los colectivos realizan un recorrido cíclico, donde cada parada está representada por un número. Hay N paradas y M colectivos.

Cada pasajero comienza esperando en una parada (que 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 indica su destino al colectivero, con la función `indicarDestino()`. Esta función devuelve el número de colectivo. Luego espera que el colectivero haga `marcarTarifa()` y, finalmente, el pasajero ejecuta `pagarConSUBE()`.

Después de pagar, el pasajero procede a `viajar()`, y cuando termina, se dispone a bajar del colectivo. Para ello, efectúa `dirigirseAPuertaTrasera()` y una vez que el colectivo se detiene, los pasajeros que están agrupados en la puerta trasera realizan `bajar()` de a uno por vez, sin importar el orden.

Por su parte, el colectivero recibe como parámetro la capacidad (cantidad máxima de pasajeros) del colectivo, y el identificador del colectivo (entre 0 y $M - 1$). El colectivo comienza su recorrido desde la parada número 0, e inicialmente está vacío.

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, abre la puerta delantera (`abrirPuertaDelantera()`) y, si hay pasajeros en la parada, estos comienzan a ascender en orden, siempre y cuando haya capacidad.

Las personas proceden a subir y el colectivero, amablemente, los atiende de a uno marcando en la máquina con `marcarTarifa()`. Ningún pasajero puede `indicarDestino()` 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, el colectivero espera a que terminen de descender todos los pasajeros que así lo desean, y procede a `cerrarPuertaTrasera()` y `avanzar()` hacia la siguiente parada, donde la dinámica será la misma.