PRÁCTICA 3 - SINCRONIZACIÓN

Cabe aclarar que este PDF solo incluye los ejercicios con ⋆

4)

Supongamos *wait()* no atómico.
(sacado de las slides de las teóricas) Un *wait()* tiene el siguiente funcionamiento:

wait(s): while(s <= 0) dormir(); s--;

Es decir, como supusimos *wait()* no atómico, puede suceder que el scheduler desaloje al proceso durante la ejecución misma del *wait()*. Veamos:
Si tenemos el siguiente código:

```
wait(s):
        while(s <= 0) dormir();
            s--;
// CRIT
s++;
– fin código –
```

Si inicializamos el semáforo en 1, puede suceder que una tarea ingrese al *wait()* no atómico, pase el while() sin problemas (porque es el primer proceso en ingresar), pero el scheduler desaloja la tarea antes de ejecutar la instrucción s--. Si ahora viene otra tarea, puede "pasar" el *wait()* sin ningún drama (porque el semáforo sigue estando en 1), entonces ejecuta la tarea crítica. Devuelta al primer proceso, continúa desde el s-- y ejecuta el código de la sección crítica también.
Esto sucede únicamente porque consideramos que *wait()* no era atómico.

5) y 6)

En el código del 5, se produce *inanición* pues tenemos un único *signal* que activa la barrera. Es decir, un único proceso podrá ingresar a la sección crítica y luego los *n-1* procesos restantes se quedan esperando en el *barrera.wait()*.
Una solución implementable sería colocar un *barrera.signal()* luego de la sección crítica, así cada proceso avanza de uno a la vez.
También, no estoy seguro de que la comparación que hay en el *if* se pueda realizar de manera atómica. Colocaría eso dentro del *mutex* para ahorrar complicaciones.

```
preparado();
count.getAndInc();
while (count < n);
critica();
```

// Estoy asumiendo FUERTEMENTE que "critica()" puede ejecutarse de a muchos o que la auxiliar se encarga de manejar correctamente el acceso de cada proceso.

a) La solución actual es más legible. Con esos mutex.wait(), signals y barreras no se entiende nada.
b) La nueva solución es más eficiente. Ningún proceso de queda “durmiendo”. Están todos activos constantemente dentro de un while.
c) No tengo idea.

7)

En este ejercicio queremos darle un orden a la ejecución de cada proceso.
Tenemos: $\mathbf{P_1, P_2, \ldots, P_{N-1}}$
Queremos $\mathbf{P_i, P_{i+1}, \ldots, P_{N-1}, P_0, \ldots, P_i}$
N es pasado por parámetro, al igual que i.

Me gustaría implementar una barrera tal que el i-ésimo proceso pueda activar el que le sigue, y en el caso del proceso n-1, que ese pueda activar al proceso 0.
Para eso, podemos armar un arreglo de tamaño n de semáforos. La idea es que el proceso i+1 haga un *wait()* sobre su misma posición. Veamos:

i+1-ésimo proceso → sem[i+1].wait();
Está esperando a que suceda:
i-ésimo proceso → sem[i+1].signal();

```
int N;
int i;
semaphore sem[N];

int main(int argc, char* argv []) {
        for (int j = 0; j < N; j++) {
                sem[j] = 0;
                thread(process, j);
        }
        sem[i].signal();
}

void process(int i) {
        sem[i].wait();
        // CÓDIGO
        if (i == N-1) {
                sem[0].signal();
        } else {
                sem[i+1].signal();
        }
}
```

8)

1) Es como el anterior, pero con un semáforo que va del 1 al 3. Además, se ejecuta todo en orden.
2) No voy a codearlo, así que voy a escribir lo que deberíamos tener en el código.
   Arrancamos con:
   sem[3] = {1, 0, 0}; // El primero es el semáforo de A, el segundo es el de B y el tercero el de C.
   Luego, por cada proceso:
   - **Proceso A:**
     sem[A].wait() // Solo lo puede activar el proceso C
     – Todo lo que tenga que ejecutar
     sem[B].signal(2)
   - **Proceso B:**
     sem[B].wait() // Solo lo puede activar el proceso A
     – Todo lo que tenga que ejecutar
     sem[C].signal()
   - **Proceso C:**
     sem[C].wait()
     – Todo lo que tenga que ejecutar
     sem[A].signal()

Es importante tener en cuenta que vamos a tener que tener 3 hijos, uno sobre cada proceso. A su vez, cada proceso debe tener un *while(true)* para volver a iniciar el *loop* **BBCA**, sino, no tiene gracia enviar sem[B].signal(2) desde A.

3) Acá tenemos que tener cuidado porque no queremos determinar un orden para la secuencia. El único orden que debemos marcar es que B o C se ejecutan SI O SI después de A.

   Entonces, es necesario tener a nuestra disposición:
   - 1 semáforo para A (que debe ser activado 2 veces)
   - 1 semáforo compartido entre B y C (puede ser activado por cualquier proceso)

PROCESO A	PROCESO B	PROCESO C

sem[A].wait()
sem[A].wait()
// PRODUCE
sem[BC].signal()
	sem[BC].wait()
	// Lee lo que produjo A
	count++
	if (count < 2)
	{sem[BC].signal()}
	else
	{count = 0;
	sem[A].signal(2);}
		sem[BC].wait()
		// Lee lo que produjo A
		count++
		if (count < 2)
		{sem[BC].signal()}
		else
		{count = 0;
		sem[A].signal(2)}

4) Acá tenemos que marcar un orden de ejecución **ABB, AC, ABB, AC, ABB, …**
El proceso C produce el doble de B, por eso se ejecuta una única vez.

Inicia con sem[A] = 1 y turnoB = true

PROCESO A	PROCESO B	PROCESO C

sem[A].wait()
// Produce
if (turnoB) sem[B].signal(2)
else sem[C].signal()
	sem[B].wait()
	// Consume
	count++
	if (count == 2) {
	count = 0;
	turnoB = false;
	sem[A].signal;}
		sem[C].wait()
		// Consume
		turnoB = true
		sem[A].signal()

13) *Game Pool Party!*

```
mutex mtx
sem[N] = sem(0)

persona() {
        mutex.wait();
        int mesa = conseguirMesa();
        mesas[mesa]++
        if (mesas[mesa] == 4) sem[i].signal(4)
        mutex.signal()

        sem[mesa].wait()
        jugar()
        abandonarMesa(mesa)
}
```

No me queda claro si abandonarMesa() la invocan todos los jugadores o el último en levantarse de la mesa. Sino, habría que armar un if( ) como el de arriba, pero cuando queda un único jugador por irse, hace abandonarMesa() y listo.
El resto de jugadores debería de tomar la cantidad de jugadores que hay en la i-ésima mesa y decrementar ese contador a medida que se van levantando.

15) *El Crucero de Noel*

Lo que está bueno de este ejercicio es que tenemos una puerta para CADA ESPECIE, entonces no hay que hacer muchos malabares a la hora de hacer que pasen los animales.

Como queremos que entren de a parejas, antes de hacer pasar una tanda de animales, tenemos que chequear que haya tanto un macho como una hembra.

Supongamos que tenemos N especies.

```
semaforo[N][2];

animal[N][2];  // Matriz donde la i-ésima posición nos dice cuantos animales de cada sexo hay en fila.

mutex mutex[N];

P(i, sexo) {
        int otroSexo = (sexo+1) % 2
        animal[i][sexo].getAndInc();  // Llega uno nuevo
        mutex[i].wait();
        if (animal[i][(otroSexo] >= 1) {        // Me fijo si hay uno del sexo contrario
                semaforo[i][sexo].signal();     // Si lo hay, los dejo pasar
                semaforo[i][otroSexo].signal();
                animal[i][sexo].getAndDec();            // Los saco de la fila porque embarcan
                animal[i][otroSexo].getAndDec();
        }
        mutex[i].signal();

        sem[i][sexo].wait();
        entrar(i)         // No sé si esta función la tienen que invocar ambos o no.
}
```

17)

Tenemos que modelar dos funciones distintas, una es la heladera y la otra son las cervezas. Hay que estar atentos porque antes de meter una cerveza en la heladera, vamos a tener que chequear 2 cosas:

1) El tipo de envase que tenemos
2) Si la heladera ya no está full con ese tipo de envase

Notemos que como las 2 funciones se ejecutan en paralelo, es necesario poner semáforos para abrir la heladera únicamente cuando llega una cerveza, y cerrar la heladera únicamente cuando ya metimos la cerveza en la heladera.

Estaría bueno que los tipos de envases vengan determinados por ceros y unos.

```
#define BOTELLA 0
#define PORRON 1

semaphore hayLugar[N][2];
```

Son 2 posiciones, una que indica el lugar que queda para las *botellas* (posición 0), y la otra posición nos dice cuanto lugar queda para los *porrones* (posición 1).
Luego, tenemos N heladeras en total.

Empiezo entonces a modelar la función de la heladera.

```
semaphore llegaCerveza = 0;

H(i) {
        EnchufarHeladera();

        hayLugar[i][BOTELLAS].signal(15);
        hayLugar[i][PORRON].signal(10);

        // Tenemos 25 lugares disponibles
        int espacio = 25;
        while (espacio--) {
                llegaCerveza.wait();
                AbrirHeladera();
                meMetenEnHeladera.wait();
                CerrarHeladera();
        }

        // Se  llenó la heladera
        EnfriadoRapido();
}
```

Empiezo ahora entonces a modelar las cervezas.

```
C(i, tipoCerveza) {
        LlegarABar();           // Nos acercamos a las heladeras

        hayLugar[i][tipoCerveza].wait();

        llegaCerveza.signal();          // Hay espacio, entonces abro la heladera

        heladeraAbierta.wait();
```

```
        MeMetenEnHeladera();          // Función del enunciado
        meMetenEnHeladera.signal();

        // Mando esas señales para coordinar el abrir y cerrar de la heladera.
}
```

19) *Baboon Crossing Problem*

a) Ok, lo que nos interesa acá es hacer cruzar a los babuinos y siempre garantizar que no hay más de 5 a la vez en la cuerda.
El ejercicio nos garantiza que, una vez que un babuino empiece a cruzar, no se va a cruzar a ningún otro que vaya en dirección opuesta.
Entonces, deberíamos de poder modelar al babuino y a la cuerda sin ningún drama.
Veamos:

BABUINO	**CUERDA**

llegaBabuino();
if (babuino.getAndInc() > 5)
{babuino.wait()}
 	
cruza.signal()
	cruza.wait()
	cruzando();
	// Terminó de cruzar uno
	babuino.signal()

Así estaría bien? No me termina de cerrar, siento que podría modelar únicamente al babuino o a la cuerda y chau. Pero bueno, que se yo.

b) La solución genera inanición, ya que puede suceder que estén constantemente cruzando babuinos de izquierda a derecha, mientras que los babuinos que quieren cruzar al lado izquierdo de quedan esperando como boludos.
Pasa que acá tampoco me gasté en determinar hacia qué lado cruzaban los lokos…

Primero, determinaría hacia qué lado están cruzando los babuinos, así podemos diferenciarlos. Luego, cada 5 que cruzan hacia un lado, hago que crucen 5 hacia el otro.