Laboratorio 4 - Sincronización

Nota: Pueden escribir las respuestas a las preguntas en el archivo `resultados.txt`.

Ejercicio 1

El programa `glob.c` crea dos hilos que incrementan repetidamente la variable global `glob`. Esto lo realizan copiando primero el valor de `glob` en una variable automática (local) del hilo, luego incrementando el valor de la variable local, y finalmente copiando el nuevo valor en `glob`. Cada hilo incrementa `glob` el número de veces indicado en la línea de comandos.

- Compilar y ejecutar el programa, probando valores hasta que se encuentre una *condición de carrera*.
- ¿Por qué ocurre esta situación de carrera? ¿Cómo se podría evitar?
- Evitar la condición de carrera mediante el uso de un *mutex* (`pthread_mutex_t`). Utilizar un *mutex* inicializado estáticamente, con `PTHREAD_MUTEX_INITIALIZER`. Para tomar y liberar el *mutex*, usar las funciones `pthread_mutex_lock()` y `pthread_mutex_unlock()` respectivamente.

Ejercicio 2

Completar los siguientes programas:

- `sem_open.c`: crea un semáforo, con el nombre indicado como parámetro.
- `sem_wait.c`: realiza una operación *down* sobre el semáforo indicado.
- `sem_post.c`: realiza una operación *up* sobre el semáforo indicado.
- `sem_getvalue.c`: imprime el valor del semáforo indicado.
- `sem_unlink.c`: elimina el semáforo indicado.

Se deben utilizar las siguientes funciones:

- `sem_open()`: abre o crea un semáforo.
- `sem_post()`: realiza una operación up sobre el semáforo.
- `sem_wait()`: realiza una operación down sobre el semáforo.
- `sem_getvalue()`: obtiene el valor actual de un semáforo.
- `sem_close()`: cierra un semáforo.
- `sem_unlink()`: elimina un semáforo del sistema.

Una vez completados los programas, tendrían que poder ejecutar la siguiente serie de comandos: primero, se crea un semáforo con un valor inicial de cero. Luego, se realiza una operación *down*, luego un *up*, y finalmente se lo elimina.

```
$ sem_open /semaforo 0
$ sem_wait /semaforo &
$ sem_post /semaforo
$ sem_unlink /semaforo
```

Responder:

1. ¿Qué es lo que sucede con el proceso que ejecuta `sem_wait`, en el segundo comando?

Ejercicio 3

El programa `buf.c` implementa un ejemplo del problema del productor-consumidor, haciendo uso de un *buffer limitado*. El programa no utiliza mecanismos de sincronización para el acceso al *buffer*. Esto puede ocasionar problemas, por ejemplo una condición de carrera. Modificar el programa para sincronizar el acceso al recurso compartido (el *buffer*) por parte de los hilos productor y consumidor, empleando semáforos y *mutexs*. En este ejercicio, y en los que siguen, crear los *mutexs* necesarios con la funcion `pthread_mutex_init()`.

Ejercicio 4

El programa `philo.c` implementa un ejemplo del problema de la *cena de los filósofos*. Durante la ejecución del programa puede ocurrir una condición de carrera.

1. Modificar el programa para evitar esta condición, mediante el uso de semáforos y mutexes.
2. Agregar también una solución para evitar el *bloqueo mutuo* o *abrazo mortal*. Explicarla.

Ejercicio 5

Modificado de: https://pdos.csail.mit.edu/6.828/2017/homework/lock.html

En este ejercicio exploraremos la programación paralela, utilizando hilos, exclusión mutua y una tabla *hash*. Para lograr un paralelismo real, se debe ejecutar este programa en una computadora con dos o más núcleos.

En primer lugar, compilar y ejecutar el programa `ph.c`:

```
$ make ph
$ bin/ph 2
```

El número 2 especifica el número de hilos que realizaran operaciones *put* y *get* sobre una tabla *hash*. Cuando termine de ejecutar, el programa debe generar una salida similar a la siguiente:

```
0: put time = 2.871728
1: put time = 2.957073
1: get time = 12.731078
1: 1 keys missing
0: get time = 12.731874
0: 1 keys missing
completion time = 15.689165
```

Cada hilo ejecuta dos fases. En la primera, almacena claves en la tabla, y en la segunda fase trata de recuperar dichas claves de la tabla. La salida del programa indica cuanto tiempo duro cada fase para cada hilo. La última linea ("*completion time*") indica el tiempo total de ejecución del programa. En la salida de ejemplo anterior, el programa ejecutó durante aproximadamente 16 segundos.

Por ejemplo, si ejecutaramos nuevamente el programa, pero con un único hilo:

```
$ bin/ph 1
0: put time = 5.350298
0: get time = 11.690395
0: 0 keys missing
completion time = 17.040894
```

El tiempo total de ejecución es levemente mayor que para el caso de ejecución con dos hilos (~17s contra ~15.6s), aunque podría ser también levemente menor. Sin embargo, notar que al utilizar dos hilos se realizó el doble de trabajo en la fase *get*, lo que representa un mejora de casi 2x (¡nada mal!). En cambio, para la fase *put* se logró una mejora mucho más pequeña, ya que entre ambos hilos guardaron el mismo número de claves en algo menos de la mitad de tiempo (~2.9s contra ~5.3s).

Independientemente de si al ejecutar el programa logran un incremento de velocidad, o la magnitud del mismo, notarán que el programa **no funciona correctamente**. Al ejecutarlo utilizando dos o más hilos, algunas claves posiblemente no puedan ser recuperadas. En el ejemplo anterior, una de las claves se perdió ("*1 keys missing*").

Esto empeora cuando incrementamos el número de hilos:

```
$ bin/ph 4
2: put time = 1.516581
1: put time = 1.529754
0: put time = 1.816878
3: put time = 2.113230
2: get time = 15.635937
2: 21 keys missing
3: get time = 15.694796
3: 21 keys missing
1: get time = 15.714341
1: 21 keys missing
0: get time = 15.746386
0: 21 keys missing
completion time = 17.866878
```

Dos consideraciones:

- El tiempo total de ejecución es aproximadamente el mismo que para el caso de dos hilos. Sin embargo, se realizó casi el doble de operaciones *get*, lo que indica que se esta obteniendo una buena paralelización.
- Más claves se han perdido. Es posible, sin embargo, que en una ejecución particular se pierdan más o menos claves, o incluso que no se pierda ninguna.

Para evitar la pérdida de claves, es necesario emplear *exclusión mutua*, durante las operaciones *put* y *get*. Las funciones a utilizar son:

```
pthread_mutex_t lock;     // declare a lock
pthread_mutex_init(&lock, NULL);   // initialize the lock
pthread_mutex_lock(&lock);  // acquire lock
pthread_mutex_unlock(&lock);  // release lock
```

Se pide:

1. Identificar la secuencia de eventos que pueden llevar a que se pierda una clave en el caso de dos o más hilos.
2. Modificar el código del programa de manera que no se pierdan claves al utilizar dos o más hilos. ¿Es aún la versión de dos hilos más rápida que la versión con un único hilo? De no ser así, ¿por qué?
3. Modificar el código para que las operaciones *get* puedan ejecutarse en paralelo. (Tip: ¿Es necesario utilizar exclusión mútua al realizar una operación *get*?)
4. Modificar el código para que algunas de las operaciones *put* puedan ejecutar en paralelo.

Ejercicio 6 (Opcional)

Un problema clásico de IPC es el problema del peluquero dormido. Una peluquería tiene *n* peluqueros, y *m* sillas donde los clientes esperan su turno. Si no hay clientes, los peluqueros duermen (se *bloquean*). Cuando arriba un cliente, alguno de los peluqueros se despierta, y realiza el corte de pelo. Si todos los peluqueros estuvieran ocupados, y hubiera sillas disponibles, el cliente se sienta a esperar su turno (es decir, se *bloquea* a la espera de su turno). Caso contrario, se retira. Implementar en `peluquero.c` un ejemplo de este problema. Emplear semáforos para manejar **sincronización** y *mutexs* para garantizar la **exclusión mutua**.

¡Fin del Laboratorio 4!