Práctica 4: Problemas de sincronización

1. Problemas introductorios

1. Se tienen un proceso que leerá una línea de un archivo (hará *readline()* y luego seguirá haciendo otra cosa, no nos interesa) y otro que mostrará dicha línea por pantalla(y luego seguirá haciendo otra cosa, no nos interesa). Para esto, hay que sincronizarlos de manera que la instrucción *showline()* se haga siempre después que se hizo *readline()*)

2. Los semáforos pueden utilizarse, también, para garantizar *exclusión mutua*. Es decir, que en un momento dado, solo un proceso tenga acceso a la sección crítica donde podrá modificar recursos compartidos. En estos casos, se lo llama **mutex** y la idea es que si un proceso adquiere el **mutex**, tenga acceso exclusivo a la sección crítica. Luego el proceso deberá soltar el **mutex**, para liberar el acceso.
   Utilizar semáforos para generar exclusión mutua entre dos threads que incrementan una variable *count* compartida

3. Un **multiplex** permite el accesso a la sección crítica a más de un proceso al mismo tiempo. Además, establece un límite en la cantidad de procesos que pueden estar ejecutando concurrentemente en la sección crítica.
   Generalizar la solución anterior para **mutex** para resolver este problema.

4. Se quiere generalizar el problema del ejercicio 1. Es decir, dado un thread A que ejecuta A1(), A2() y otro B que ejecuta B1(), B2(), queremos que para toda ejecución ocurra que A2 se ejecuta después de B1 y que B2 se ejecute después de A1.
   La idea es crear un *rendezvous* entre ambos procesos y que ninguno pueda continuar la ejecución si el otro no llego a dicho punto. Notar que el orden entre A1 y B1 no importa y que la solución no debe ser demasiado restrictiva.

5. Se tienen tres procesos (A, B y C). Se desea que el orden en que se ejecutan sea el orden afabético, es decir que las secuencias normales deben ser: ABC, ABC, ABC,..

2. Problemas avanzados

6. Una de las limitaciones del ejercicio cuatro, donde se lograba un punto de encuentro, es que sirve para dos procesos.
   Supongamos que se tienen **n** procesos que deben ejecutar las instrucciones *rendezvous()* y luego *criticalPoint()*. Queremos que ninguno de ellos ejecute *criticalPoint()* sin que todos hayan ejecutado *rendezvous()*. En otras palabras, cuando los primeros $n-1$ procesos llegan al punto de encuentro, deben bloquearse hastaque el n-ésimo proceso lo haga, momento en el cual todos pueden continuar. A este patrón se le suelle llamar **turnstile**(molinete o barrera). Se puede asumir que existe una variable compartida n.

7. Es posible resolver el ejercicio anterior y que al terminar, la barrera no quede bloqueada. Si tuviésemos que sincronizar los procesos más de una vez (por ejemplo, al estar en un loop), necesitamos mejorar la solución anterior. Entonces, se quiere escribir una solución de manera que después de que todos pasen la barrera, la misma quede bloqueada nuevamente y fuerce un nuevo punto de encuentro.

8. 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).
   Escribir el código para solucionar este problema de sincronización utilizando semáforos (no olvidar los valores iniciales).

9. Hay situaciones donde los semáforos pueden usarse para representar colas. En tal caso, su valor inicial es cero, y debe verificarse que no reciba un *signal* a menos que haya un procesos esperando, de forma que el valor del semáforo sea siempre positivo. Sirve, entonces, interpretar al *wait* como "esperá en la colaz signal como "desencolá alguien de la cola".
   En una sala de baile llegan dos tipos de personas: líderes y seguidores. Antes de entrar al lugar propiamente dicho, esperan en colas diferenciadas por tipo. Cuando un líder llega, verifica si hay un seguidor esperando. Si es así, ambos tienen permitido continuar a la sala. Si no, debe esperar. De igual forma, cuando un seguidor llega debe verificar si hay un líder y proceder en tal caso o esperaren caso contrario.

10. En el ejercicio anterior, no había restricción sobre la instrucción para bailar. Es decir, si bien los lideres y seguidores tenían permitido entrar de a pares, esto podría no ocurrir. Por lo tanto, podría pasar que varios líderes bailen sin que ningún seguidor lo haya hecho.
    Modificar la solución para que tenga en cuenta la restricción adicional que haga que cada líder pueda bailar concurrentemente con un único seguidor y viceversa.

11. El problema del productor-consumidor para n consumidores y n productores puede ser resuelto utilizando semáforos. Dada una estructura que representa un buffer, un consumidor debe hacer un *get()* para obtener un item y luego procesarlo. A su vez, un productor debe hacer *produceItem()* y luego haer un *add(item)* sobre el buffer.
    Recordar que si un consumidor llega cuando el buffer está vacío, se bloquea hasta que un productor aparezca. Además, tener en cuenta que las funciones *add()* y *get()* son no atómicas, por lo que la estructura interna es inconsistente durante la ejecución de las mismas.