Programación Concurrente

Práctica 6: Monitores

Ejercicio 1. Implementar un monitor `Contador` que permita incrementar y decrementar concurrentemente de forma segura.

Ejercicio 2. Definir un monitor Semáforo que implemente las operaciones de un semáforo (`acquire` y `release`). ¿Garantiza su solución ausencia de starvation?

Ejercicio 3. Implementar lo siguiente utilizando monitores:

a) Un buffer (FIFO) de números enteros cuya dimensión está prefijada al momento de su creación.

b) Una clase productor (que extienda de `Thread`) que agregue números naturales consecutivos a un buffer dado al momento de creación.

c) Una clase consumidor (que extienda de `Thread`) que muestre por pantalla los valores que toma de un buffer pasado al momento de su creación.

d) Un programa que cree un buffer de dimensión 2 y active concurrentemente un consumidor y un productor.

Ejercicio 4. Se desea implementar usando monitores una barrera para coordinar a N threads. Una barrera provee una única operación denominada `esperar`. La idea es que cada uno de los N threads invocarán la operación `esperar` una vez y el efecto de invocarla es que el thread se bloquea y no puede continuar hasta tanto los restantes threads invoquen a la operación esperar. Por ejemplo, si miBarrera es una barrera para coordinar 3 threads, el uso de miBarrera en los siguientes threads

```
thread T1: { print('a'); miBarrera.esperar(); print(1); }
thread T2: { print('b'); miBarrera.esperar(); print(2); }
thread T3: { print('c'); miBarrera.esperar(); print(3); }
```

garantiza que todas las letras se mostrarán antes de los números. Dar una implementación para el monitor barrera tal que una vez alcanzado el cupo, la barrera quede levantada, es decir, que no vuelva a bloquear invocaciones al método esperar.

Ejercicio 5. Se desea implementar el monitor `Event` con métodos `publish` y `suscribe`. Los threads suscriptores a un evento deben esperar hasta que otro thread publique su ocurrencia, momento en el cual todos los suscriptores bloqueados continúan su ejecución. Tenga en cuenta que un suscriptor siempre debe bloquearse hasta que se ejecute el siguiente `publish`, es decir, ignorando las invocaciones a `public` previas.

Ejercicio 6. Se desea implementar el monitor `Promise`, que representa un cómputo que concluirá en algún momento y devolverá un resultado. El monitor posee los siguientes métodos:

- `get()`, que devuelve el resultado del cómputo, bloqueando al thread que lo invoca mientras el resultado no está disponible; y

- `set(object)`, que asigna el resultado, desbloqueando a cualquier thread esperando en el monitor.

Además se desea que `Promise` implemente la interfaz `Future`, que sólo declara el método `get`. De esta manera una función que desea devolver su resultado de manera asincrónica, puede crear un `Promise` e iniciar un thread para que sete el valor una vez haya concluido el computo necesario. La función puede devolver el `Promise` como un `Future`, evitando así que alguien más pueda setear el valor.

Ejemplo:

```
Future asynch(Object x) {
  promise = new Promise();
  thread {
    // computo costo en funcion de x que genera un resultado
    promise.set(resultado);
  }
  return promise;
}
```

Ejercicio 7. Se requiere implementar lo siguiente:

a) Un monitor `RW` con un atributo `Serializable` (i.e., que se puede transformar en una representación binaria) y métodos `beginRead`, `endRead`, `beginWrite` y `endWrite`. De forma tal que múltiples threads pueda "leer" el dato serializable concurrentemente, pero que asegurando acceso en exclusión mutua el las escrituras.

b) Una clase `Writer` que extienda de `Thread` que escriba un dato en el campo del monitor (asuma la existencia de un método `serialize` no-synchronized, cuya ejecución toma algún tiempo).

c) Una clase `Reader` que extienda de `Thread` que lea el dato del campo del monitor (asuma la existencia de una método `deserialize` no-synchronized, cuya ejecución toma algún tiempo).

d) Un programa que cree un monitor y active concurrentemente dos escritores y cuatro lectores.

e) Si su solución le da la prioridad a uno de los dos tipos de threads, modifíquelo de forma tal de que sea el otro thread el que tenga la prioridad de entrada.

Ejercicio 8. Diseñe un monitor `ThreadPool` encargado de administrar la asignación de tareas entre múltiples threads `Worker`, implementando los siguientes puntos:

a) Un monitor `ThreadPool`, encargado de crear un `Buffer` de una capacidad total dada (como el del ejercicio 3), e iniciar una cantidad dada de threads `Worker` asignándole a cada el mismo `Buffer` de tareas. El monitor debe implementar un método `launch` que envía una tarea a un `Worker` ocioso encolándola en el buffer.

b) Un thread `Worker` que por siempre resuelve tareas obtenidas de un `Buffer` dado en su construcción (asuma que las tareas son un objeto `Runnable`, es decir, implementa el método `run`).

c) Una tarea `DummyTask` (i.e., implementa `Runnable`) cuyo método `run` simplemente imprime un string por consola.

d) Una tarea `PoisonPill` (i.e., implementa `Runnable`) cuyo método `run` arroja una excepción de tipo `PoisonException` cuyo objetivo es terminar la ejecución de un `Worker`. Extienda, además, la interfaz del `ThreadPool` con un método `stop` que, luego de agotadas las tareas encoladas, termina la ejecución de todos los workers mediante el uso de `PosionPill`s.

e) Un programa que instancia un `ThreadPool` para administrar 8 threads `Worker` y lance 100 tareas de tipo `DummyTask`. Al terminar la ejecución de las 100 tareas el programa debe garantizar que todos los threads creados terminan su ejecución exitosamente.