Programación Concurrente

Práctica 3: Atomicidad

Ejercicio 1. Considere el siguiente fragmento de código (donde la variable `x` comienza inicializada en cero):

```
thread T1: {
  while(entrar());
  x = x + 1;
  salir();
}
```

```
thread T2: {
  while(entrar());
  x = x + 2;
  salir();
}
```

```
thread T3: {
  while(entrar());
  x = x + 3;
  salir();
}
```

a) Dadas las siguientes implementaciones triviales de las funciones `entrar` y `salir`, determinar todos los valores posibles de la variable `x` al final de la ejecución de los threads.

```
entrar() {
  return false;
}
```

```
salir() { }
```

b) Considere ahora las siguientes implementaciones de las funciones `entrar` y `salir` que utilizan la variable booleana global `ocupado` (inicializada en `false`) e indique si al usarlas se delimita correctamente una sección crítica (en caso contrario indicar al menos una propiedad que no se cumpla y mostrar una traza que ejemplifique el problema).

```
entrar() {
  if (ocupado)
    return true;
  ocupado = true;
  return false;
}
```

```
salir() {
  ocupado = false;
}
```

c) Analice el problema considerando que las funciones `entrar` y `salir` son atómicas ¿Se resuelve en este caso el problema de la exclusión mutua? Justifique su respuesta.

Ejercicio 2. Considere la siguiente propuesta para resolver el problema de exclusión mutua para 3 threads que utiliza las operaciones atómicas `PedirTurno` y `LiberarTurno`:

```
global int actual = 0, turnos = 0;
```

```
PedirTurno() {
  int turno = turnos;
  turnos = turnos + 1;
  return turno;
}
```

```
LiberarTurno() {
  turnos = turnos - 1;
}
```

Considere, también, que cada thread ejecuta el siguiente protocolo:

```
// seccion no critica
miTurno = PedirTurno();
while (actual != miTurno);
// seccion critica
LiberarTurno();
// seccion no critica
```

Muestre que esta propuesta **no** resuelve el problema de la exclusión mutua. Explique cuáles condiciones se cumplen y cuáles no, justificando o mostrando una traza según corresponda.

Ejercicio 3. Considere la siguiente operación:

```
tomarFlag(mia, otro) {
  flags[mia] = !flags[otro];
}
```

Se propone el siguiente algoritmo para resolver el problema de la exclusión mutua entre dos procesos que utilizan el array compartido:

```
global boolean[] flags = {false, false};

thread { // threadId=0
  while (!flags[0])
    tomarFlag(0,1);
  // seccion critica
  flags[0] = false;
}
```

```
thread { // threadId=1
  while (!flags[1])
    tomarFlag(1,0);
  // seccion critica
  flag[1] = false;
}
```

Responda si la propuesta anterior resuelve el problema de la exclusión mutua en los siguientes casos:

a) La operación `tomarFlag` **no** es atómica.

b) La operación `tomarFlag` **sí** es atómica.

Ejercicio 4. Considere la siguiente operación atómica que implementa un swap entre dos referencias booleanas:

```
void Exchange(Ref sref, Ref lref) {
   temp       = sref.value;
   sref.value = lref.value;
   lref.value = temp;
}
```

Y la siguiente propuesta para resolver el problema de la exclusión mutua:

```
global Ref shared = new Ref(false);
thread {
  Ref local = new Ref(true);
  // seccion no critica
  do { Exchange(shared, local); } while (local.value);
  // seccion critica
  shared.value = false;
  // seccion no critica
}
```

Indique si la propuesta resuelve el problema de la exclusión mutua para una cantidad no acotada de threads. Justifique (en caso negativo indicar que condición no se cumple y mostrar una traza). Recuerde analizar si la solución es libre de *starvation*.

Nota: Tenga en cuenta que la estructura de control `do-while` ejecuta el cuerpo del ciclo al menos una vez, y luego verifica la condición antes de repetir la ejecución.

Ejercicio 5. Considere el siguiente fragmento de código:

```
global int ticket = 0;
global int turn = 0;


thread T1: {                    thread T2: {
  // non-critical section       // non-critical section
  int myTurn = getTurn();       int myTurn = getTurn();
  while (myTurn != turn);       while (myTurn != turn);
  // critical section           // critical section
  releaseTurn();                releaseTurn();
  // non-critical section       // non-critical section
}                               }
```

Dadas las siguientes implementaciones de las funciones `getTurn` y `releaseTurn`.

```
getTurn() {                     releaseTurn() {
  return ticket++;                turn++;
}                               }
```

a) Determine si al utilizar las funciones `getTurn` y `releaseTurn` (no atómicas) se delimita correctamente una sección crítica (en caso contrario indicar al menos una propiedad que no se cumpla y mostrar una traza que ejemplifique el problema).

b) Ahora considere como atómicas las funciones `getTurn` y `releaseTurn` e indique si en este caso se resuelve el problema de la exclusión mutua.

c) Consiedere si la respuesta se mantiene en caso de que se tenga una cantidad no acotada de threads.

d) Si esta solución se ejecuta en un entorno donde los enteros son representados con un byte (8 bits), es decir, el número sin signo más significativo representable es 255, ¿Afecta esto su respuesta anterior?

Ejercicio 6. Considere la operaciones atómicas `push`, `peek` y `pop` que implementan una cola FIFO sobre un array de n posiciones. Es decir, `push` agrega un elemento al final de la cola, `pop` remueve el elemento al principio de la cola, y `peek` retorna el elemento al principio de la cola sin removerlo. Más el siguiente algoritmo para resolver el problema de la exclusión mutua para n threads, que utiliza la variable compartida `queue`.

```
global int[] queue = new int[n];

thread {
  id = 1; // ids de 1 a n
  // seccion no critica
```

```
    push(queue, id);
    while (id != peek(queue));
    // SECCION CRITICA
    pop(queue);
    // seccion no critica
}
```

- Indique si el algoritmo resuelve el problema de la exclusión mutua para n threads. Justifique (en caso negativo indicar que condición no se cumple y mostrar una traza).

Ejercicio 7. Considere la siguiente propuesta para resolver el problema de exclusión mutua para N threads:

```
global LLSC lock = new LLSC(false, -1);

thread {
    //Seccion No Critica
    LLSC local;
    do {
        do { local = lock.loadLink(); }
        while (local.flag);
    } while (!lock.storeConditional(local, true));
    //Seccion Critica
    lock.reset();
    //Seccion No Critica
}
```

Dada la siguiente implementación de la clase `LLSC`.

```
class LLSC {

    private boolean flag;
    private long timestamp;

    public LLSC(boolean flag, long timestamp) {
        this.flag = flag;
        this.timestamp = timestamp;
    }

    public LLSC loadLink() {
        this.timestamp = System.currentTimeMillis();
        return new LLSC(this.flag, this.timestamp);
    }

    public boolean storeConditional(LLSC other, boolean value) {
        if (this.timestamp != other.timestamp)
            return false;
        this.flag = value;
        return true;
    }

    public void reset() {
        this.flag = false;
    }
```

```
}
```

Asumiendo que `System.currentTimeMillis()` nunca retorna dos veces el mismo valor, responda:

a) Muestre que la propuesta **no** cumple la propiedad de Mutex.

b) Muestre que la propuesta **no** cumple la propiedad de Garantía de Entrada.

c) ¿Cambian en algo sus respuestas si `loadLink` y `storeConditional` son atómicas? Justifique apropiadamente.