Programação Paralela (Prof. Fernando Castor)
Alunos: Edwin Marinho e Thomas Cristanis

Trabalho 04

Do livro AMP, devem ser feitos (no mínimo!) os seguintes exercícios:
Capítulo 2. 11, 13, 15, 16

Exercise 11. Programmers at the Flaky Computer Corporation designed the protocol shown in Fig. 2.15 to achieve n-thread mutual exclusion. For each question, either sketch a proof, or display an execution where it fails.

```
class Flaky implements Lock {
  private int turn;
  private boolean busy = false;
  public void lock() {
    int me = ThreadID.get();
    do {
      do {
        turn = me;
      } while (busy);
      busy = true;
    } while (turn != me);
  }
  public void unlock() {
    busy = false;
  }
}
```

Figure 2.15 The Flaky lock used in Exercise 11.

- Does this protocol satisfy mutual exclusion?

Seguindo a notação de *prova* aplicada no livro, dada uma execução de duas Threads A e B, onde A executa primeiro que B, é possivel dizer que o mesmo satisfaz a exclusão mútua (mutex)
$write_A(turn = me(A)) \rightarrow read_A(busy == false) \rightarrow write_A(busy == true) \rightarrow read_A(turn == me(A)) \rightarrow CS_A$
$write_B(turn = me(B)) \rightarrow read_B(busy == false) \rightarrow write_B(busy == true) \rightarrow read_B(turn == me(B)) \rightarrow CS_B$

$read_A(turn == me(A)) \rightarrow write_B(turn = me(B))$ //Exclusão Mútua

Como é usado *Do… While* é testado primeiro quem é *turn.* A Thread A e B não acessaram a área critica ao mesmo tempo.

- Is this protocol starvation-free?

Não, pelo fato de protocolos que podem ser deadlock não serem starvation-free.
Se uma thread está tentando entrar em sua seção crítica, então a thread deve eventualmente entrar em sua seção crítica, no exemplo a Thread A pode não entra em sua região critica.

#Thread A: $write_A$(turn = me(A))
#Thread B: $write_B$(turn = me(B))
#Thread B: $read_B$(busy == false) → $write_B$(busy == true) → $read_B$(turn == me (B)) → CS_B
#Thread A: $write_A$(turn = me(A)) → $read_A$(busy == false)
#Thread B: CS_B → $write_B$(busy == true)
#Thread B: $write_B$(turn = me(B))

```
class Flaky implements Lock {
  private int turn;
  private boolean busy = false;
  public void lock() {
    int me = ThreadID.get();
    do {
      do {
        turn = me;
      } while (busy);
      busy = true;
    } while (turn != me);
  }
  public void unlock() {
    busy = false;
  }
}
```

Figure 2.15 The Flaky lock used in Exercise 11.

- Is this protocol deadlock-free?

O loop interno continua a ser executado, já que busy == true. Como mencionado no Capítulo 1 a exclusão mútua exige deadlock-free (página 11).

#Thread A: $write_A$(turn = me(A))
#Thread B: $write_B$(turn = me(B))
#Thread B: $read_B$(busy == false) → $write_B$(busy == true)
#Thread A: $write_A$(turn = me(A))
#Thread B: $read_B$(busy == false) → $write_B$(busy == true)

Exercise 13. Another way to generalize the two-thread Peterson lock is to arrange a number of 2-thread Peterson locks in a binary tree. Suppose n is a power of two. Each thread is assigned a leaf lock which it shares with one other thread. Each lock treats one thread as thread 0 and the other as thread 1.

In the tree-lock's acquire method, the thread acquires every two-thread Peterson lock from that thread's leaf to the root. The tree-lock's release method for the tree-lock unlocks each of the 2-thread Peterson locks that thread has acquired, from the root back to its leaf. At any time, a thread can be delayed for a finite duration. (In other words, threads can take naps, or even vacations, but they do not drop dead.)

For each property, either sketch a proof that it holds, or describe a (possibly infinite) execution where it is violated.

1. mutual exclusion.
2. freedom from deadlock.
3. freedom from starvation.

Is there an upper bound on the number of times the tree-lock can be acquired and released between the time a thread starts acquiring the tree-lock and when it succeeds?

Exercise 15. In practice, almost all lock acquisitions are uncontended, so the most practical measure of a lock's performance is the number of steps needed for a thread to acquire a lock when no other thread is concurrently trying to acquire the lock.

Scientists at Cantaloupe-Melon University have devised the following "wrapper" for an arbitrary lock, shown in Fig. 2.16. They claim that if the base Lock class provides mutual exclusion and is starvation-free, so does the FastPath lock, but it can be acquired in a constant number of steps in the absence of contention. Sketch an argument why they are right, or give a counterexample.

```
1  class FastPath implements Lock {
2    private static ThreadLocal<Integer> myIndex;
3    private Lock lock;
4    private int x, y = -1;
5    public void lock() {
6      int i = myIndex.get();
7      x = i;                    // I'm here
8      while (y != -1) {}        // is the lock free?
9      y = i;                    // me again?
10     if (x != i)               // Am I still here?
11       lock.lock();            // slow path
12   }
13   public void unlock() {
14     y = -1;
15     lock.unlock();
16   }
17 }
```

Figure 2.16 Fast path mutual exclusion algorithm used in Exercise 15.

Line 4: $\text{write}_A(x = A) \rightarrow \text{write}_B(x = B) \rightarrow$
Line 8: $\text{read}_A(y == A) \rightarrow \text{read}_B(y == B) \rightarrow$
Line 4: $\text{write}_C(x = C)$
Line 10: $\text{read}_A(x == A) \rightarrow \text{read}_B(x == A)$ // Os dois podem entrar no lock ao mesmo tempo

Como não há garantia de exclusão mútua duas Threads podem ficar impedidas de continuar a execução e o acesso a CS pode nunca acontecer. Então não é startvation-free onde se um thread está tentando entrar em sua seção crítica, então esta thread deve eventualmente entrar em sua seção crítica.

Exercise 16. Suppose n threads call the visit() method of the Bouncer class shown in Fig. 2.17. Prove that:

- At most one thread gets the value STOP.
- At most n − 1 threads get the value DOWN.
- At most n − 1 threads get the value RIGHT.

Note that the last two proofs are not symmetric.

```
class Bouncer {
  public static final int DOWN = 0;
  public static final int RIGHT = 1;
  public static final int STOP = 2;
  private boolean goRight = false;
  private ThreadLocal<Integer> myIndex;
  private int last = -1;
  int visit() {
    int i = myIndex.get();
    last = i;
    if (goRight)
      return RIGHT;
    goRight = true;
    if (last == i)
      return STOP;
    else
      return DOWN;
  }
}
```

Figure 2.17 The Bouncer class implementation.

[...]