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.

```
1   class Flaky implements Lock {
2     private int turn;
3     private boolean busy = false;
4     public void lock() {
5       int me = ThreadID.get();
6       do {
7         do {
8           turn = me;
9         } while (busy);
10        busy = true;
11      } while (turn != me);
12    }
13    public void unlock() {
14      busy = false;
15    }
16  }
```

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 possível dizer que o mesmo satisfaz a exclusão mútua (mutex)

$write_A(turn = me(A)) \rightarrow read_A(busy == false) \rightarrow write_A(busy == true) \rightarrow read_A(turn == me (A)) \rightarrow CS_A$
$write_B(turn = me(B)) \rightarrow read_B(busy == false) \rightarrow write_B(busy == true) \rightarrow read_B(turn == me (B)) \rightarrow CS_B$

$read_A(turn == me(A)) \rightarrow read_B(turn == me(B))$ // CASO NAO GARANTA EXCLUSAO MUTUA

// PARA QUE O CASO ACIMA ACONTECE SERA NECESSARIO QUE SE PRECEDA ($\rightarrow$)
$write_B(turn = me(B)) \rightarrow read_B(busy == false) \rightarrow write_A(busy == true) \rightarrow read_A(turn == me (A)) \rightarrow read_B(turn == me (B))$

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

```
1   class Flaky implements Lock {
2     private int turn;
3     private boolean busy = false;
4     public void lock() {
5       int me = ThreadID.get();
6       do {
7         do {
8           turn = me;
9         } while (busy);
10        busy = true;
11      } while (turn != me);
12    }
13    public void unlock() {
14      busy = false;
15    }
16  }
```

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

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: $write_A(x = A) \rightarrow write_B(x = B) \rightarrow$
Line 8: $read_A(y == A) \rightarrow read_B(y == B) \rightarrow$
Line 4: $write_C(x = C)$
Line 10: $read_A(x == A) \rightarrow read_B(x == A)$ // 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.

A única possibilidade de uma thread retornar STOP é caso *last* == *i*. A partir disso entre o momento em que a Linha 10 é executada e o tempo que *if* (last == i) é verificado, nenhuma outra thread executa a Linha 10 e por isso apenas uma thread pode ser STOP.

- At most n − 1 threads get the value DOWN.

Duas são as possibilidades:
- Pelo menos uma thread retornado RIGHT. Então, eventualmente, não haverá mais do que n - 1 retornando DOWN.
- As threads passam pela Linha 11 antes de outra thread executar a Linha 13.

- At most n − 1 threads get the value RIGHT.

$read_A(goRight == false) \rightarrow write_A(goRight = true) \rightarrow return\ RIGHT$

Note that the last two proofs are not symmetric.

```
1   class Bouncer {
2     public static final int DOWN = 0;
3     public static final int RIGHT = 1;
4     public static final int STOP = 2;
5     private boolean goRight = false;
6     private ThreadLocal<Integer> myIndex;
7     private int last = -1;
8     int visit() {
9       int i = myIndex.get();
10      last = i;
11      if (goRight)
12        return RIGHT;
13      goRight = true;
14      if (last == i)
15        return STOP;
16      else
17        return DOWN;
18    }
19  }
```

Figure 2.17 The Bouncer class implementation.