**Trabalho 11**

02 de Junho de 2015 - Exercícios do AMP Capitulo 9

Exercise 101. Explain why the fine-grained locking algorithm is not subject to deadlock.

Porque todos os metodos (add() e remove()) adquirem o lock sempre na mesma ordem, de head para tail, por isso o algoritmo garante progresso. Caso os locks fossem adquiridos em ordem diferente o deadlock poderia acontecer. *Fig 9.9 e Fig 9.10*

Exercise 102. Explain why the fine-grained list's add() method is linearizable.

Para identificar se o código é linearizable basta identificar um ponto de linearização, uma única etapa atômica onde a chamada de método produz efeitos é suficiente (*199 p4*). Ou seja em um cenário onde a Thread A adquire o lock nas linha 3 e 7 um outra Thread B não consegue acessar até que seja dado o unlock(), já dentro de um finally (é sempre executado) e após o retorno do metodo, nas linha 23 e 26.

Exercise 103. Explain why the optimistic and lazy locking algorithms are not subject to deadlock.

Assim como os algoritmo anteriores do capitulo 9 do livro o optimistic e lazy locking os métodos adquirem o bloqueio sempre na mesma ordem. o lock é sempre de predA (anterior) para depois currA (atual).

Exercise 105. Provide the code for the contains() method missing from the fine-grained algorithm. Explain why your implementation is correct.

```
public boolean contains(T item) {
   int key = item.hashCode();
   head.lock();
   Node pred = head;
   try {
      Node curr = pred.next;
      curr.lock();
      try {
         while (curr.key < key) {
            pred.unlock();
            pred = curr;
            curr = curr.next;
            curr.lock();
         }
         if (curr.key == key) {
```

```
            return true;
          }
        } finally {
          curr.unlock();
        }
      } finally {
        pred.unlock();
      }
    }
```

A implementação é correta pois segue a mesma linha de pensamento do método *remove*(), obtendo e liberando os locks na ordem (de head a tail) para cada iteração na lista.

Exercise 106. Is the optimistic list implementation still correct if we switch the order in which add() locks the pred and curr entries?
Não, pois deixaria o algoritmo suscetível a deadlock.

Exercise 108. Show that in the optimistic algorithm, the add() method needs to lock only pred.
Não pode usar apenas o lock de pred pois uma outra thread pode remover o curr, fazendo com que a lista passe a apontar para um ítem que não mais existe.

Exercise 110. Would the lazy algorithm still work if we marked a node as removed simply by setting its next field to null? Why or why not? What about the lock-free algorithm?
Não poderia apenas colocar o next para null pois impediria outras threads de continuarem a iteração naquele elemento. Por exemplo, uma verificação de que contém algum elemento poderia se deparar com um nó nulo em algum momento.

Exercise 112. Your new employee claims that the lazy list's validation method (Fig. 9.16) can be simplified by dropping the check that pred.next is equal to curr: After all, the code always sets pred to the old value of curr, and before pred.next can be changed, the new value of curr must be marked, causing the validation to fail. Explain the error in this reasoning.

```
1    private boolean validate(Node pred, Node curr) {
2      return !pred.marked && !curr.marked && pred.next == curr;
3    }
```

Figure 9.16 The LazyList class: validation checks that neither the pred nor the curr node has been logically deleted, and that pred points to curr.

Alguma operação de adição de um elemento após pred ou remoção de curr poderia fazer com que a lista fosse alterada antes dessa verificação.

Exercise 115. In the lock-free algorithm, if an add() method call fails because pred does not point to curr, but pred is not marked, do we need to traverse the list again from head in order to attempt to complete the call?

Sim, para garantir que nenhuma outra operação modificou a lista, fazendo todas as verificações atomicamente.