Silas Garrido - sgtcs@cin.ufpe.br
Vinicius Lira – vcml@cin.ufpe.br

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

O algoritmo é livre de deadlock pois só é possível conseguir o lock de um nó somente após conseguir o lock do nó anterior. No fine-grained uma thread A adquire o lock em um tipo de ordem "hand-over-hand". Para conseguir os locks, os métodos seguem sempre essa sequência. Ou seja, não há espera circular. Esse protocolo de locking também é chamado de lock coupling.

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

* Sabendo que os pontos de linearizabilidade de add() estão nas linhas 7 ou 13 do código, uma vez que a posição correta de "curr.key" foi encontrada e a inserção foi realizada, após os desbloqueio, qualquer outra thread que vier imediantamente depois já levará em conta a modificação na estrutura de dados. Isso acontece pelo fato das travas "segurarem" as outras threads que tentarem acessar a região que já está sendo modificada, garantindo a consistência e, consequentemente, a linearizabilidade.

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

No optimistic lock o algoritmo procura pelo nó sem adquirir o lock, uma vez encontrado faz lock de seu predecessor e de seu nó atual, em seguida verifica se o lock sucedeu da forma correta, caso contrário reinicia o procedimento.

No lazy locking o algoritmo é semelhante porém utilizando de uma variável booleana para constatar a existência lógica no conjunto, mesmo que fisicamente ainda esteja incluso.

Em ambos os algoritmos, a lista sempre é lida na mesma sequência (pred => curr) e o lock de um nó só é obtido após conseguir o lock de seu predecessor. Assim não há dependência cíclica.

105. Provide the code for the contains() method missing from the finegrained 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();
         }
         return (curr.key == key);
      } finally {
           curr.unlock();
      }
   } finally {
        pred.unlock();
   }
}
```

Bloquear os atributos "curr" e "pred" garantirá que não haverá inconsistências durante a consulta, da mesma forma que acontece com add() e com remove().

106. Is the optimistic list implementation still correct if we switch the order in which add() locks the pred and curr entries?

Não. Essa troca ordem não garante a ausência de dependência cíclica entre as operações de add e remove. Com essa troca de ordem poderiam ser geradas listas inconsistente visto que os métodos de inserção e remoção utilizariam estratégias diferentes de lock. Para garantir que essa troca não gerasse filas inconsistentes, ambos os métodos (inserção e remoção) deveriam seguir a essa nova ordem de lock.

108. Show that in the optimistic algorithm, the add() method needs to lock only pred.

* Se somente "pred" for bloqueado, possíveis inconsistências de "curr" serão corrigidas pelo método "validate()". Exemplo: duas threads paralelas -- uma no método add() e outra no método remove() -- são designadas para adicionar e remover o mesmo item e que já existe no "set". Ambas encontram "pred" e "curr" no mesmo intervalo de tempo, porém, a thread com o remove() consegue o lock() do "pred" e remove o item com sucesso. Posteriormente, quando o lock() de "pred" é liberado, a threads com o add(), que aguardava, passa a bloqueá-lo. Quando o método "validate()" for executado, ele retornará false pelo fato de "pred" ser

diferente de "curr" (removido pela outra thread). Com isso, no lugar do add() retornar, de forma incorreta, "false", ele irá correr toda a lista novamente, encontrar os atributos "pred" e "curr" corretos, e conseguir adicionar o item com sucesso.

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. Em ambos os casos ocorreria um erro em tempo de execução no momento em que a lista fosse percorrida.

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.

* Supor que duas threads estejam sendo executadas ao mesmo tempo. A thread A deseja remover o nodo "a" e a thread B deseja adicionar um novo nodo exatamente antes de "a". Ambas as threads encontram o mesmo "pred" e "cur", porém, a thread B adquire o lock(), adiciona o novo nó com sucesso, e libera o lock(). Posteriormente a thread A (que aguardava) adquire o lock(), e executa o método "validate". Como nenhum nó foi marcado na inserção e como a condição "pred.next == curr" foi retirada, a método retornará "true". Com isso, a thread A removerá o nó incorreto e pred.next irá apontar para um local incorreto, ignorando o novo nodo adicionado pela thread B e comprometendo a coerência do "set". Dessa forma, fica evidente que a presença da condição "pred.nex == curr" é essencial para manter a consistência do "set".

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?

Não. Para tal cenário serio o caso de terem sido adicionados um ou mais elementos entre o *pred* e o *curr* ou o *curr* é removido. Dessa forma seria é necessário percorrer a lista a partir do *pred*, visto que alterações nos elementos anteriores ao *pred* não interferem na adição do elemento em questão.