Programação Paralela e Concorrente

Locks: alternativas

2016

Custo de Threads

- custo de chamadas ao sistema

Alternativas de locks em listas encadeadas

- sincronização em granularidade fina
  - criação de locks para cada componente independente
- sincronização otimista
  - primeira fase sem obtenção de lock (exemplo busca)
- sincronização adiada (*lazy*)
  - divisão da tarefa em parte que não precisa de sincronização e parte que precisa
- sincronização sem bloqueio
  - eliminação de locks com uso de operações como *compareAndSet()*

Conjuntos implementados por listas

- interface do conjunto:

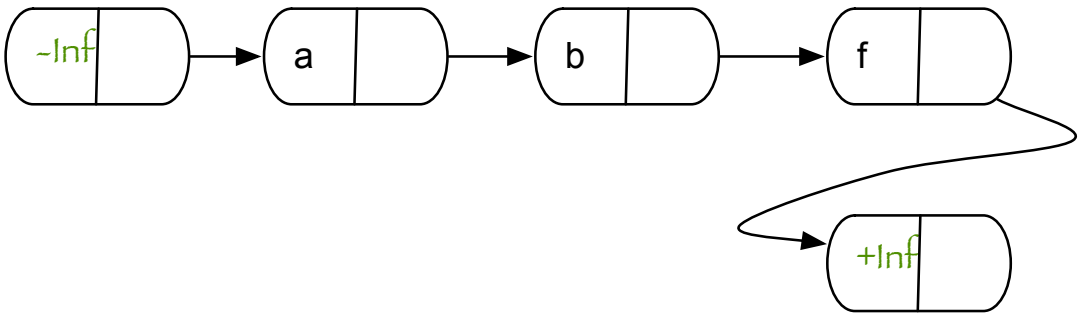
```
public interface Set<T> {
 public boolean add(T x);
 public boolean remove(T x);
 public boolean contains(T x);
}
```

- classe interna para implementar lista encadeada:

```
public class Node {
 public T item;
 public int key;
 public Node next;
}
```

implementação do conjunto

- lista ordenada com sentinelas:



implementações alternativas de conjunto

- slides Herlihy

Estruturas lock-free

- pelo menos uma thread "progride" a cada fatia de tempo

uso de primitivas CompareAndSet

```
boolean compareAndSet(int expect, int update);
```

Uso antigo de primitivas atômicas...

```
1  public class TTASLock implements Lock {
2    AtomicBoolean state = new AtomicBoolean(false);
3    public void lock() {
4      while (true) {
5        while (state.get()) {};
6        if (!state.getAndSet(true))
7          return;
8      }
9    }
10   public void unlock() {
11     state.set(false);
12   }
13 }
```

- testandset instrução comumente presente em processadores
- espera ocupada

implementação lock-free de uma pilha

```
1  public class LockFreeStack<T> {
2    AtomicReference<Node> top = new AtomicReference<Node>(null);
3    static final int MIN_DELAY = ...;
4    static final int MAX_DELAY = ...;
5    Backoff backoff = new Backoff(MIN_DELAY, MAX_DELAY);
6
7    protected boolean tryPush(Node node){
8      Node oldTop = top.get();
9      node.next = oldTop;
10     return(top.compareAndSet(oldTop, node));
11   }
12   public void push(T value) {
13     Node node = new Node(value);
14     while (true) {
15       if (tryPush(node)) {
16         return;
17       } else {
18         backoff.backoff();
19       }
20     }
21   }
```

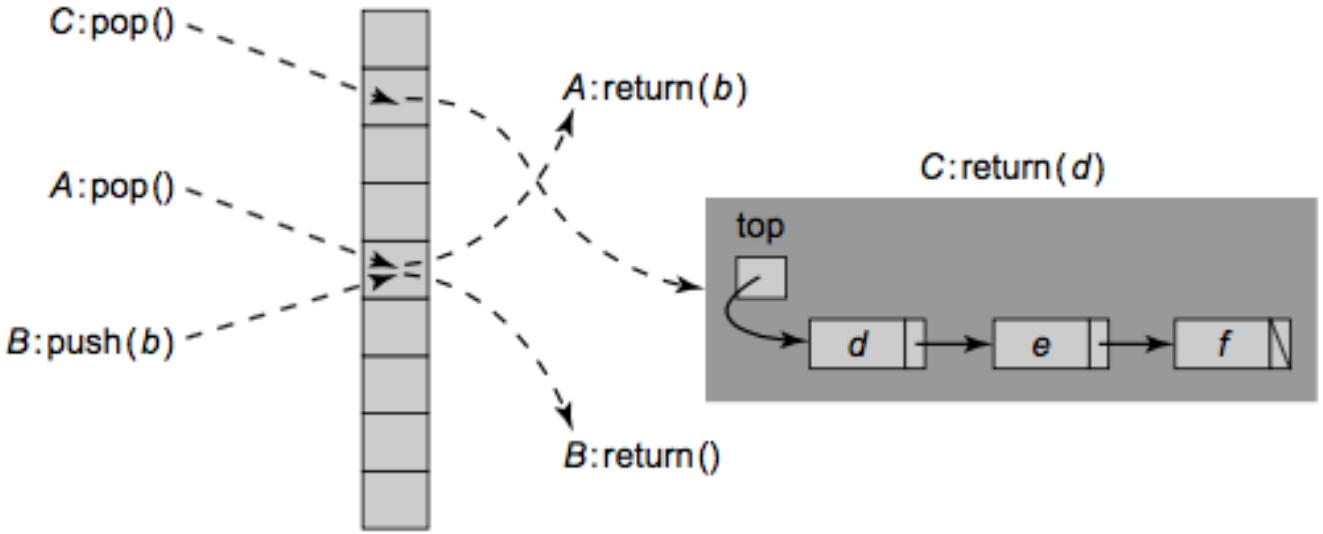
implementação lock-free de uma pilha

```
1   protected Node tryPop() throws EmptyException {
2     Node oldTop = top.get();
3     if (oldTop == null) {
4       throw new EmptyException();
5     }
6     Node newTop = oldTop.next;
7     if (top.compareAndSet(oldTop, newTop)) {
8       return oldTop;
9     } else {
10      return null;
11    }
12  }
13  public T pop() throws EmptyException {
14    while (true) {
15      Node returnNode = tryPop();
16      if (returnNode != null) {
17        return returnNode.value;
18      } else {
19        backoff.backoff();
20      }
21    }
22  }
```

implementação lock-free de uma pilha

- espera ocupada!
- contenção no acesso ao topo da pilha
- idéia: uso de estrutura de dados aumentada

implementação lock-free de uma pilha



implementação de exchange

```
public class AtomicStampedReference<V>
extends Object

An AtomicStampedReference maintains an object reference along with an integer "stamp",
that can be updated atomically.

...

boolean compareAndSet(V expectedReference, V newReference, int expectedStamp, int newStamp);

Atomically sets the value of both the reference and stamp to the given update values if
the current reference is == to the expected reference and the current stamp is equal to
the expected stamp.
```

implementação lock-free do exchanger

```
1  public class LockFreeExchanger<T> {
2    static final int EMPTY = ..., WAITING = ..., BUSY = ...;
3    AtomicStampedReference<T> slot = new AtomicStampedReference<T>(null, 0);
4    public T exchange(T myItem, long timeout, TimeUnit unit)
5      throws TimeoutException {
6      long nanos = unit.toNanos(timeout);
7      long timeBound = System.nanoTime() + nanos;
8      int[] stampHolder = {EMPTY};
9      while (true) {
10       if (System.nanoTime() > timeBound)
11         throw new TimeoutException();
12       T yrItem = slot.get(stampHolder);
13       int stamp = stampHolder[0];
14       switch(stamp) {
15       case EMPTY:
16         if (slot.compareAndSet(yrItem, myItem, EMPTY, WAITING)) {
17           while (System.nanoTime() < timeBound){
18             yrItem = slot.get(stampHolder);
19             if (stampHolder[0] == BUSY) {
20               slot.set(null, EMPTY);
21               return yrItem;
22             }
23           }
24           if (slot.compareAndSet(myItem, null, WAITING, EMPTY)) {
25             throw new TimeoutException();
26           } else {
27             yrItem = slot.get(stampHolder);
28             slot.set(null, EMPTY);
29             return yrItem;
30           }
31         break;
32       case WAITING:
33         if (slot.compareAndSet(yrItem, myItem, WAITING, BUSY))
34           return yrItem;
35         break;
36       case BUSY:
37         break;
38       default: // impossible
39         ...
40       }
```

array de exchangers

```
1  public class EliminationArray<T> {
2    private static final int duration = ...;
3    LockFreeExchanger<T>[] exchanger;
4    Random random;
5    public EliminationArray(int capacity) {
6      exchanger = (LockFreeExchanger<T>[]) new LockFreeExchanger[cap
7      for (int i = 0; i < capacity; i++) {
8        exchanger[i] = new LockFreeExchanger<T>();
9      }
10     random = new Random();
11   }
12   public T visit(T value, int range) throws TimeoutException {
13     int slot = random.nextInt(range);
14     return (exchanger[slot].exchange(value, duration,
15             TimeUnit.MILLISECONDS));
16   }
17 }
```

pilha estendida

```
1  public class EliminationBackoffStack<T> extends LockFreeStack<T> {
2    static final int capacity = ...;
3    EliminationArray<T> eliminationArray = new EliminationArray<T>(capacity);
4    static ThreadLocal<RangePolicy> policy = new ThreadLocal<RangePolicy>() {
5      protected synchronized RangePolicy initialValue() {
6        return new RangePolicy();
7      }
8
9    public void push(T value) {
10     RangePolicy rangePolicy = policy.get();
11     Node node = new Node(value);
12     while (true) {
13       if (tryPush(node)) {
14         return;
15       } else try {
16         T otherValue = eliminationArray.visit
17                              (value, rangePolicy.getRange());
18         if (otherValue == null) {
19           rangePolicy.recordEliminationSuccess();
20           return; // exchanged with pop
21         }
22       } catch (TimeoutException ex) {
23         rangePolicy.recordEliminationTimeout();
24       }
25     }
26   }
27 }
```