Paralelizando uma estrutura de dados do tipo lista ligada

Alexandro Baldassin

2sem, 2016

Memória Compatilhada

- Processo cria várias threads de execução
- Threads compartilham espaço de endereçamento
- Comunicação é feita diretamente através de leituras e escritas em memória compartilhada
- Acessos concorrentes de leitura e escrita podem causar inconsistências (condições de corrida)
- Sincronização é usada para evitar tais cenários

Sincronização

- Evitar intercalações inconsistentes de execução

```
shared counter;
void work()
{
  counter++;
}
```

```
shared counter;
void work()
{
  counter++;
}
```

Qual o resultado esperado após a primeira execução?

Sincronização

- Evitar intercalações inconsistentes de execução

```
shared counter;
void work()
{
  counter++;
}
```

```
i1: temp = load(counter);
i2: temp = temp + 1;
i3: store(counter, temp);
```

Sincronização

- Evitar intercalações inconsistentes de execução

t_1 (counter++)

```
i1: temp = load(counter);
i2: temp = temp + 1;
i3: store(counter, temp);
```

t_2 (counter++)

```
i1: temp = load(counter);
i2: temp = temp + 1;
i3: store(counter, temp);
```

Sincronização

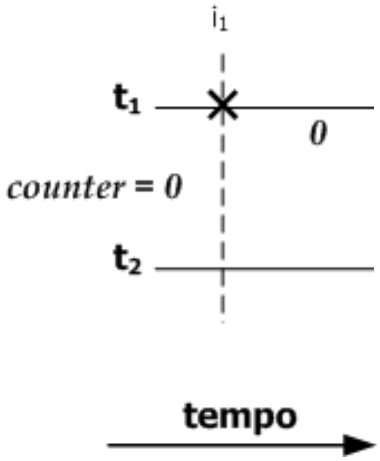
- Evitar intercalações inconsistentes de execução

t_1 (counter++)

```
i1: temp = load(counter);
i2: temp = temp + 1;
i3: store(counter, temp);
```

t_2 (counter++)

```
i1: temp = load(counter);
i2: temp = temp + 1;
i3: store(counter, temp);
```



Sincronização

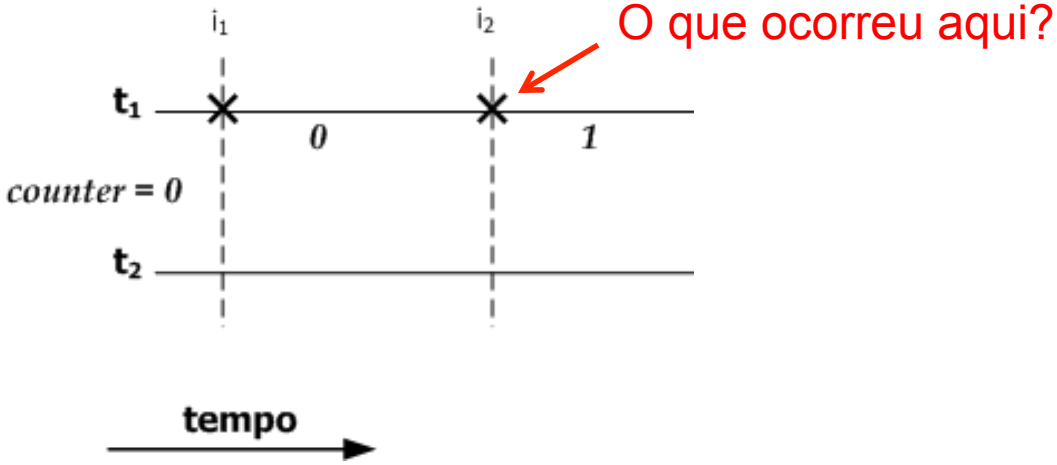
- Evitar intercalações inconsistentes de execução

t_1 (counter++)

```
i1: temp = load(counter);
i2: temp = temp + 1;
i3: store(counter, temp);
```

t_2 (counter++)

```
i1: temp = load(counter);
i2: temp = temp + 1;
i3: store(counter, temp);
```



O que ocorreu aqui?

Sincronização

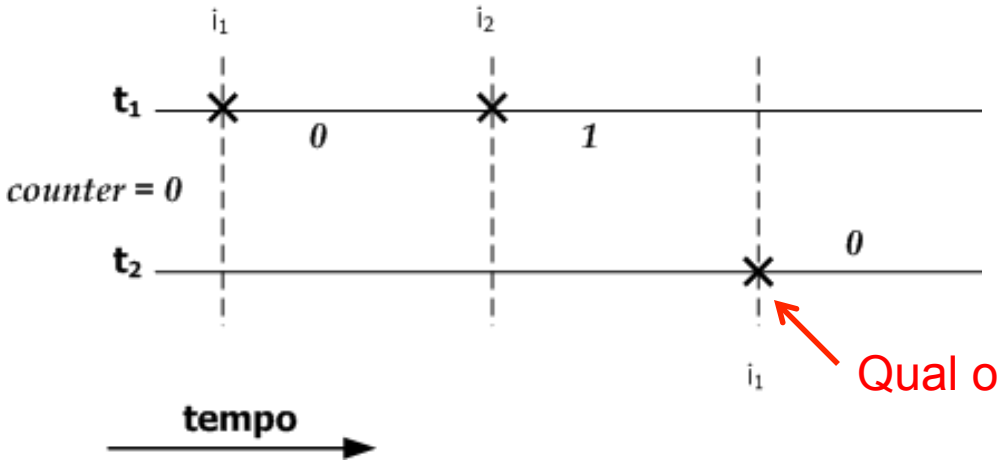
- Evitar intercalações inconsistentes de execução

t_1 (counter++)

```
i1: temp = load(counter);
i2: temp = temp + 1;
i3: store(counter, temp);
```

t_2 (counter++)

```
i1: temp = load(counter);
i2: temp = temp + 1;
i3: store(counter, temp);
```



Qual o valor lido?

Sincronização

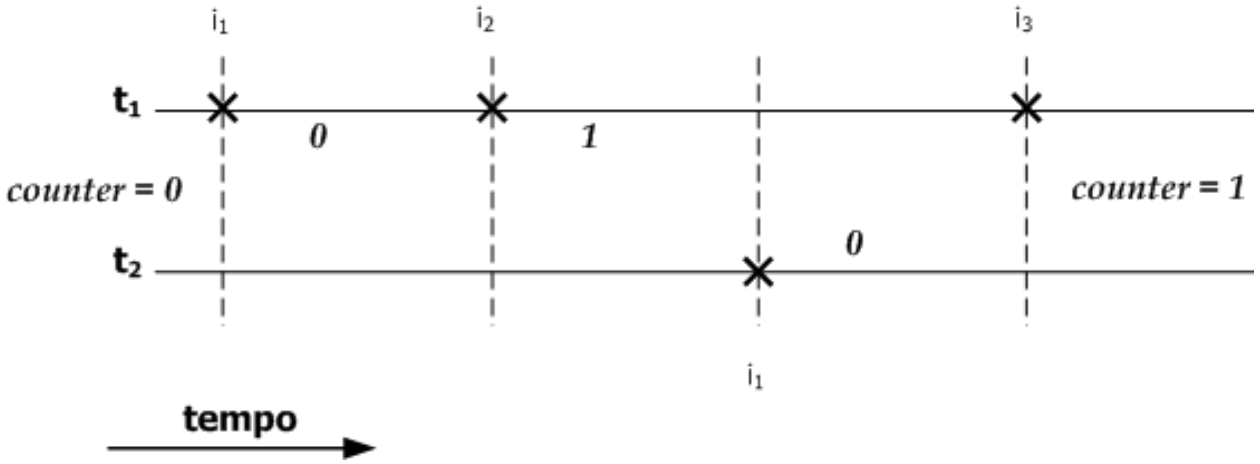
- Evitar intercalações inconsistentes de execução

t_1 (counter++)

```
i1: temp = load(counter);
i2: temp = temp + 1;
i3: store(counter, temp);
```

t_2 (counter++)

```
i1: temp = load(counter);
i2: temp = temp + 1;
i3: store(counter, temp);
```



Sincronização

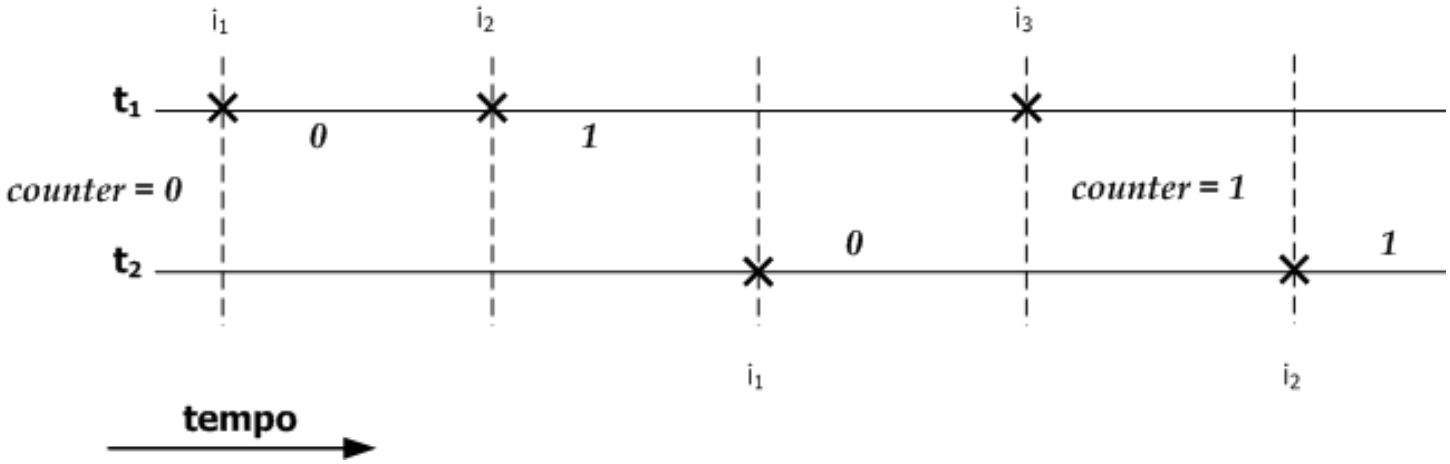
- Evitar intercalações inconsistentes de execução

t_1 (counter++)

```
i1: temp = load(counter);
i2: temp = temp + 1;
i3: store(counter, temp);
```

t_2 (counter++)

```
i1: temp = load(counter);
i2: temp = temp + 1;
i3: store(counter, temp);
```



Sincronização

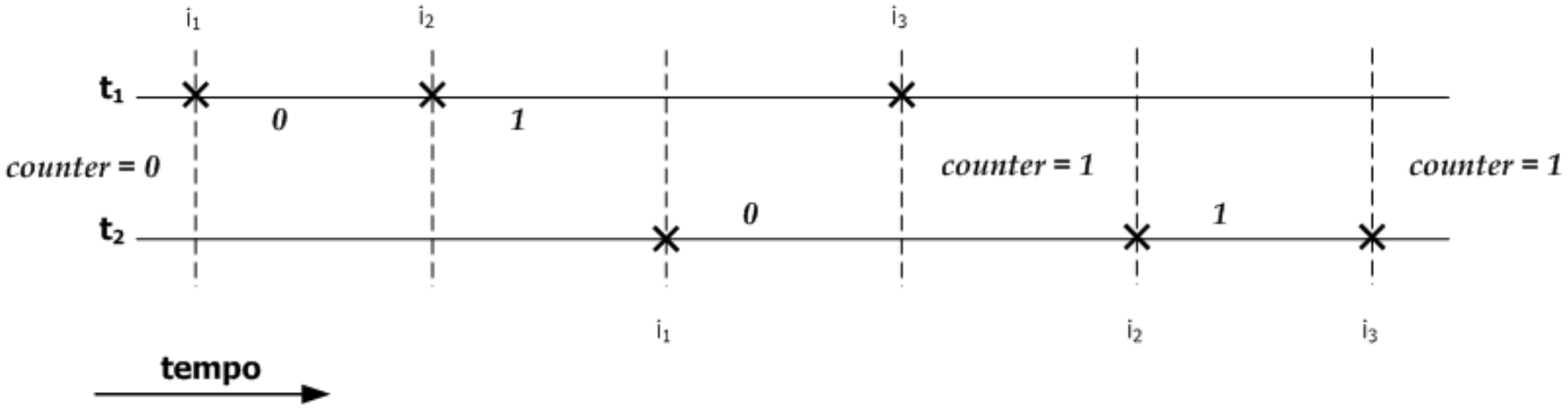
- Evitar intercalações inconsistentes de execução

t_1 (counter++)

```
i1: temp = load(counter);
i2: temp = temp + 1;
i3: store(counter, temp);
```

t_2 (counter++)

```
i1: temp = load(counter);
i2: temp = temp + 1;
i3: store(counter, temp);
```



Sincronização

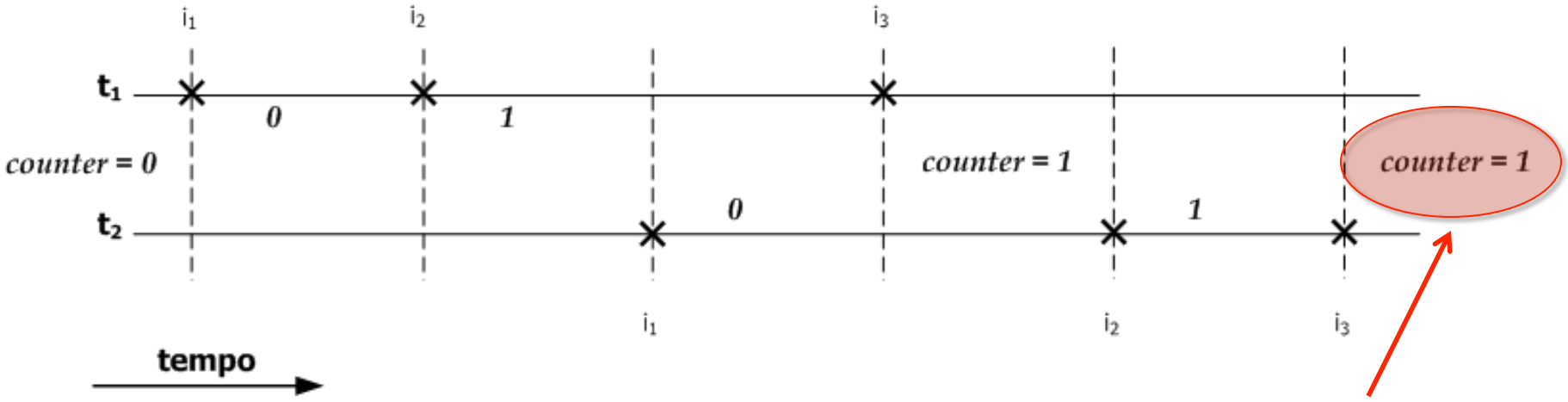
- Evitar intercalações inconsistentes de execução

t_1 (counter++)

```
i1: temp = load(counter);
i2: temp = temp + 1;
i3: store(counter, temp);
```

t_2 (counter++)

```
i1: temp = load(counter);
i2: temp = temp + 1;
i3: store(counter, temp);
```



Está correto?

Mecanismos de Sincronização

- Bloqueantes
  - travas (*locks*) ←
  - variáveis de condição (*condition variables*)
  - semáforos/monitores
- Não-bloqueantes
  - livre de espera (*wait-free*)
  - livre de trava (*lock-free*)
  - livre de obstrução (*obstruction-free*)

Exemplo

- Considere o seguinte problema: implementar uma **lista** de inteiros ordenados em ordem crescente, admitindo operações como inserção, remoção e consulta

- Esta tarefa pode ser desenvolvida facilmente por alunos de disciplinas de introdução à computação

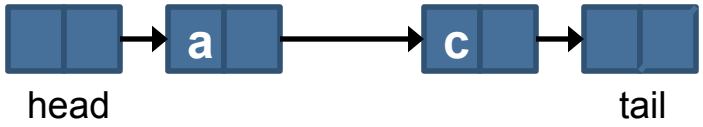
- Desejamos uma versão paralela, que permita operações concorrentes na lista

Lista ordenada – sequencial

```
class Node {
  int key;
  Node next;
}
```

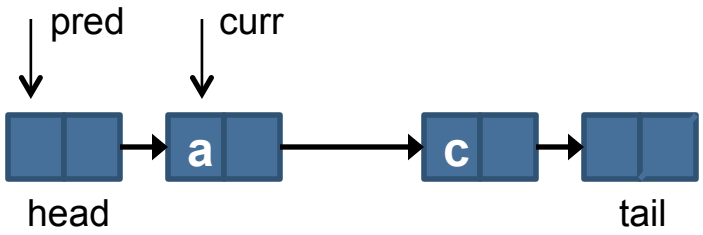
```
class List {
  private Node head;
  public List() {
    head = new Node(Integer.MIN_VALUE);
    head.next = new Node(Integer.MAX_VALUE);
  }
}
```

Lista ordenada – sequencial

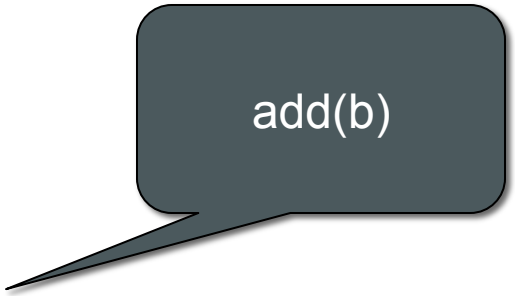



```
public boolean add(int item) {
   Node pred, curr;

   pred = head;
   curr = pred.next;
   while (curr.key < item) {
     pred = curr;
     curr = curr.next;
   }
   if (item != curr.key) {
     Node node = new Node(item);
     node.next = curr;
     pred.next = node;
     return true;
   }
   else return false;
}
```

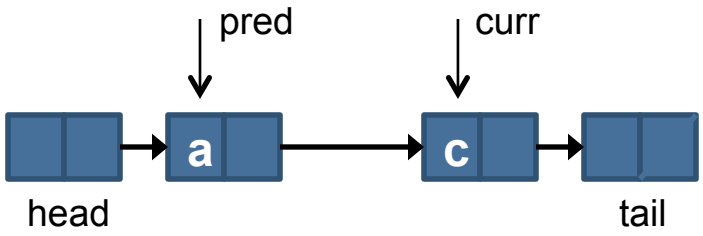
Lista ordenada – sequencial



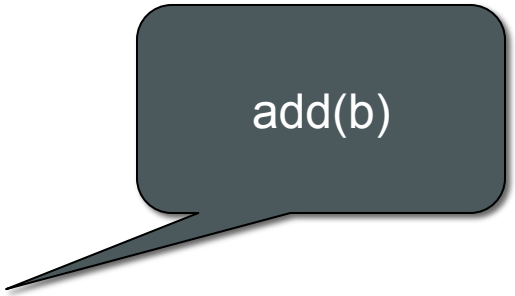

```
public boolean add(int item) {
   Node pred, curr;

   pred = head;
   curr = pred.next;
   while (curr.key < item) {
     pred = curr;
     curr = curr.next;
   }
   if (item != curr.key) {
     Node node = new Node(item);
     node.next = curr;
     pred.next = node;
     return true;
   }
   else return false;
}
```



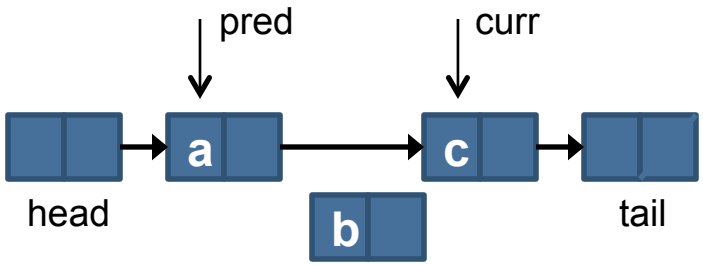
Lista ordenada – sequencial



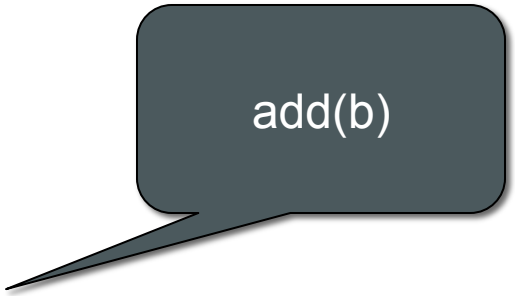

```
public boolean add(int item) {
   Node pred, curr;

   pred = head;
   curr = pred.next;
   while (curr.key < item) {
     pred = curr;
     curr = curr.next;
   }
   if (item != curr.key) {
     Node node = new Node(item);
     node.next = curr;
     pred.next = node;
     return true;
   }
   else return false;
}
```



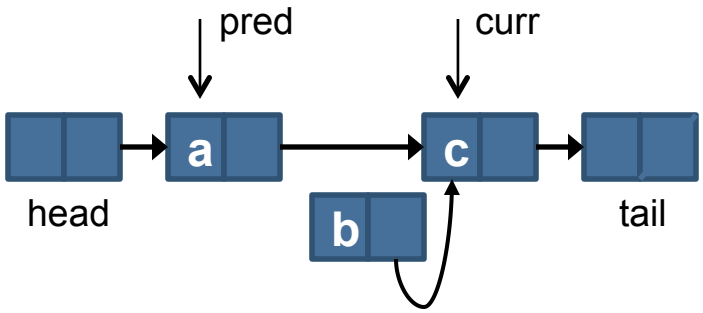
Lista ordenada – sequencial



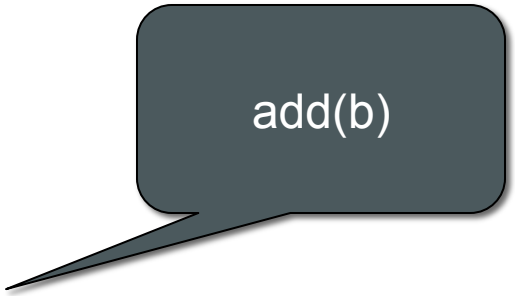

```
public boolean add(int item) {
   Node pred, curr;

   pred = head;
   curr = pred.next;
   while (curr.key < item) {
     pred = curr;
     curr = curr.next;
   }
   if (item != curr.key) {
     Node node = new Node(item);
     node.next = curr;
     pred.next = node;
     return true;
   }
   else return false;
}
```



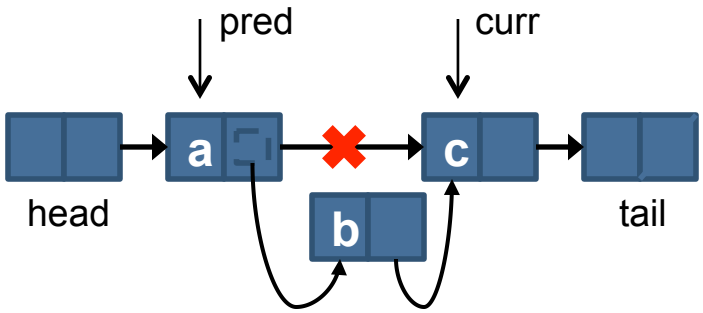
Lista ordenada – sequencial



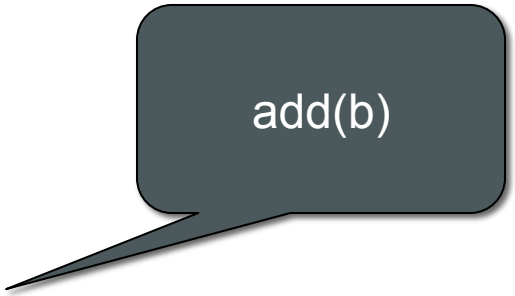

```
public boolean add(int item) {
   Node pred, curr;

   pred = head;
   curr = pred.next;
   while (curr.key < item) {
     pred = curr;
     curr = curr.next;
   }
   if (item != curr.key) {
     Node node = new Node(item);
     node.next = curr;
     pred.next = node;
     return true;
   }
   else return false;
}
```



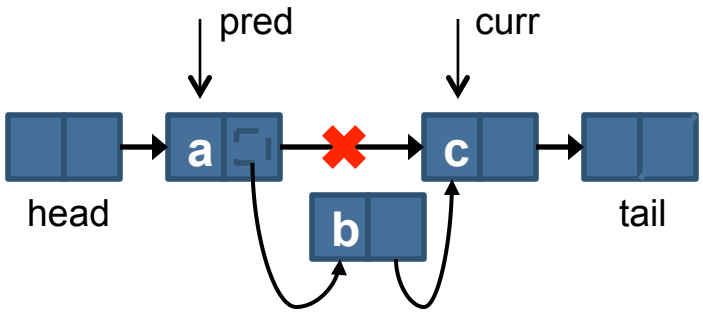
Lista ordenada – sequencial

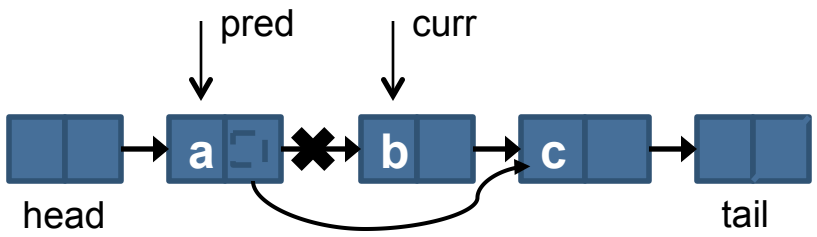



```
public boolean add(int item) {
    Node pred, curr;

    pred = head;
    curr = pred.next;
    while (curr.key < item) {
      pred = curr;
      curr = curr.next;
    }
    if (item != curr.key) {
      Node node = new Node(item);
      node.next = curr;
      pred.next = node;
      return true;
    }
    else return false;
}
```



Lista ordenada – sequencial





remove(b)

```
public boolean add(int item) {
    Node pred, curr;

    pred = head;
    curr = pred.next;
    while (curr.key < item) {
      pred = curr;
      curr = curr.next;
    }
    if (item != curr.key) {
      Node node = new Node(item);
      node.next = curr;
      pred.next = node;
      return true;
    }
    else return false;
}
```

```
public boolean remove(int item) {
    Node pred, curr;

    pred = head;
    curr = pred.next;
    while (curr.key < item) {
      pred = curr;
      curr = curr.next;
    }
    if (item == curr.key) {
      pred.next = curr.next;
      return true;
    }
    else return false;
}
```

Paralelizando

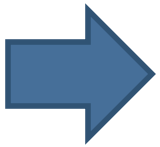
- Como desenvolver uma versão paralela do exemplo anterior?

Lista ordenada – lock global

```
public boolean add(int item) {
   Node pred, curr;

   pred = head;
   curr = pred.next;
   while (curr.key < item) {
     pred = curr;
     curr = curr.next;
   }
   if (item != curr.key) {
     Node node = new Node(item);
     node.next = curr;
     pred.next = node;
     return true;
   }
   else return false;
}
```

Lista ordenada – lock global

```
public boolean add(int item) {
   Node pred, curr;

   pred = head;
   curr = pred.next;
   while (curr.key < item) {
     pred = curr;
     curr = curr.next;
   }
   if (item != curr.key) {
     Node node = new Node(item);
     node.next = curr;
     pred.next = node;
     return true;
   }
   else return false;
}
```



```
public boolean add(int item) {
   Node pred, curr;
   boolean valid = false;

   lock.lock();
     pred = head;
     curr = pred.next;
     while (curr.key < item) {
       pred = curr;
       curr = curr.next;
     }
     if (item != curr.key) {
       Node node = new Node(item);
       node.next = curr;
       pred.next = node;
       valid = true;
     }
   lock.unlock();
   return valid;
}
```

Lista ordenada – lock global

- Ideia do lock global
  - Antes de iniciar o trecho de código que altera a lista, adquirir a trava (**lock**)
  - Após trecho de código, liberar a trava (**unlock**)
  - Funciona?
- Solução simples, mas não escala!
  - Operações são serializadas

Serializando

```
public boolean add(int item) {
   Node pred, curr;
   boolean valid = false;

   lock.lock();
     pred = head;
     curr = pred.next;
     while (curr.key < item) {
       pred = curr;
       curr = curr.next;
     }
     if (item != curr.key) {
       Node node = new Node(item);
       node.next = curr;
       pred.next = node;
       valid = true;
     }
   lock.unlock();
   return valid;
}
```

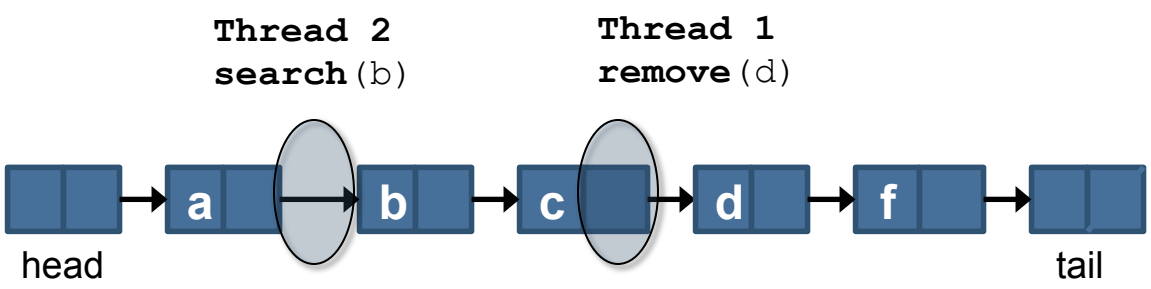
- Como melhorar a solução?
  - Sugestões?

```
public boolean add(int item) {
   Node pred, curr;
   boolean valid = false;

   lock.lock();
     pred = head;
     curr = pred.next;
     while (curr.key < item) {
       pred = curr;
       curr = curr.next;
     }
     if (item != curr.key) {
       Node node = new Node(item);
       node.next = curr;
       pred.next = node;
       valid = true;
     }
   lock.unlock();
   return valid;
}
```

Paralelizando

- Como desenvolver uma versão paralela do exemplo anterior?

- Sendo otimista
  - Operações de **inserção**, **remoção** e **busca** podem "potencialmente" ser executadas em paralelo

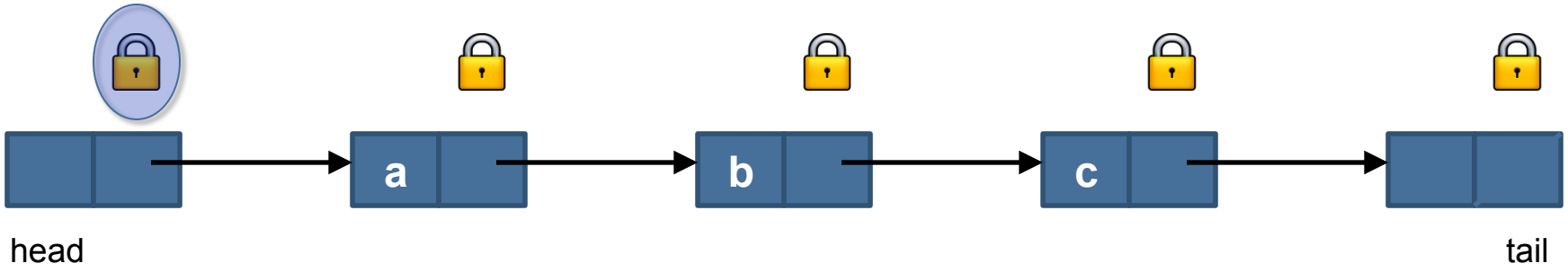


Lista ordenada – locks finos

- Ideia
  - Associar um lock a cada nó da lista
  - Antes do conteúdo do nó ser acessado, adquirimos seu respectivo lock (liberando-o após o acesso)

- Essa abordagem funciona?
  - Considere duas operações concorrentes para remoção dos itens ‘b’ e ‘a’, por duas threads distintas (T_1 e T_2)

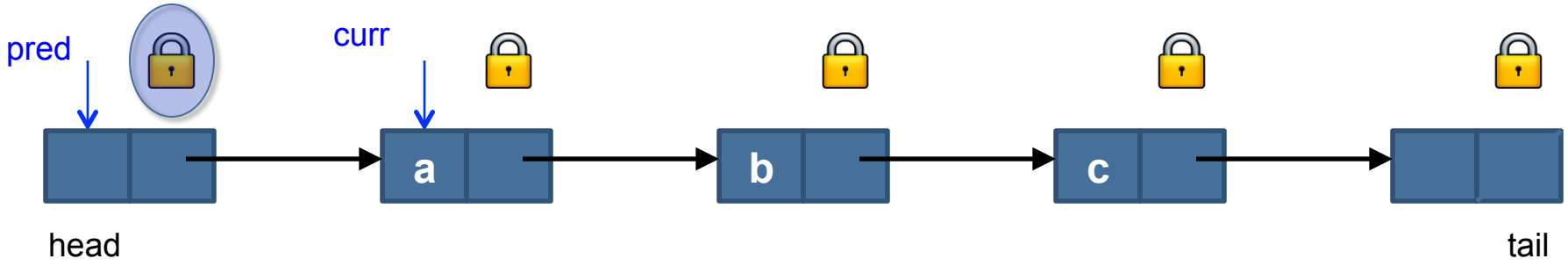
Lista ordenada – locks finos




```
remove(b)
  ...
    head.lock();
    pred = head; curr = pred.next;
    while (curr.key < item) {
      pred.unlock();
      pred = curr; curr = curr.next;
      pred.lock();
    }
    if (item == curr.key) {
      pred.next = curr.next;
      valid = true;
    }
    pred.unlock();
    return valid;
}
```

T_1

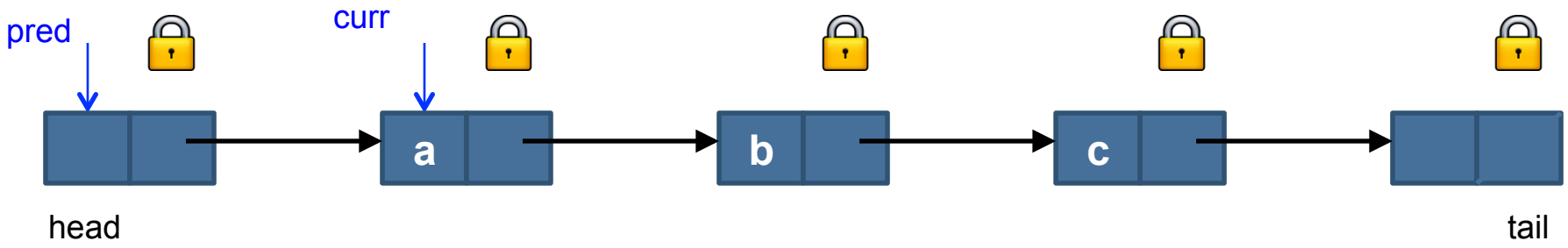
Lista ordenada – locks finos




```
remove(b)
    ...
    head.lock();
    pred = head; curr = pred.next;
    while (curr.key < item) {
      pred.unlock();
      pred = curr; curr = curr.next;
      pred.lock();
    }
    if (item == curr.key) {
      pred.next = curr.next;
      valid = true;
    }
    pred.unlock();
    return valid;
}
```

T_1

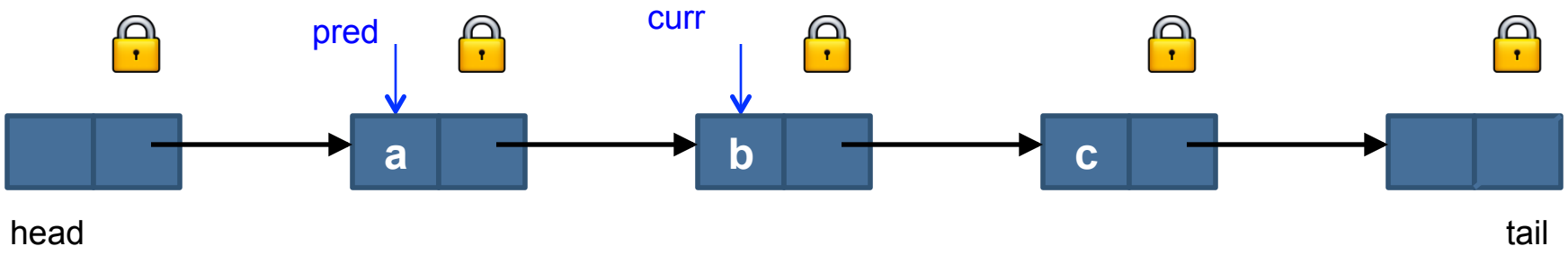
Lista ordenada – locks finos




```
remove(b)
    ...
    head.lock();
    pred = head; curr = pred.next;
    while (curr.key < item) {
      pred.unlock();
      pred = curr; curr = curr.next;
      pred.lock();
    }
    if (item == curr.key) {
      pred.next = curr.next;
      valid = true;
    }
    pred.unlock();
    return valid;
}
```

T_1

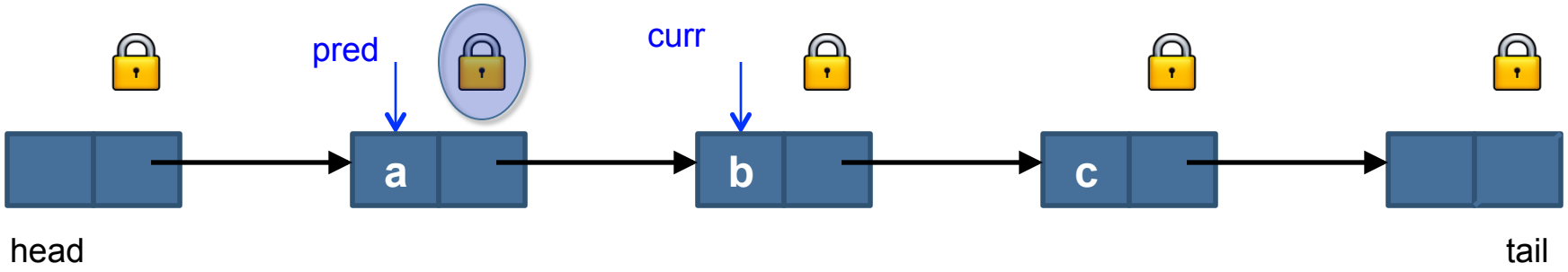
Lista ordenada – locks finos




```
remove(b)
    ...
    head.lock();
    pred = head; curr = pred.next;
    while (curr.key < item) {
      pred.unlock();
      pred = curr; curr = curr.next;
      pred.lock();
    }
    if (item == curr.key) {
      pred.next = curr.next;
      valid = true;
    }
    pred.unlock();
    return valid;
}
```

T_1

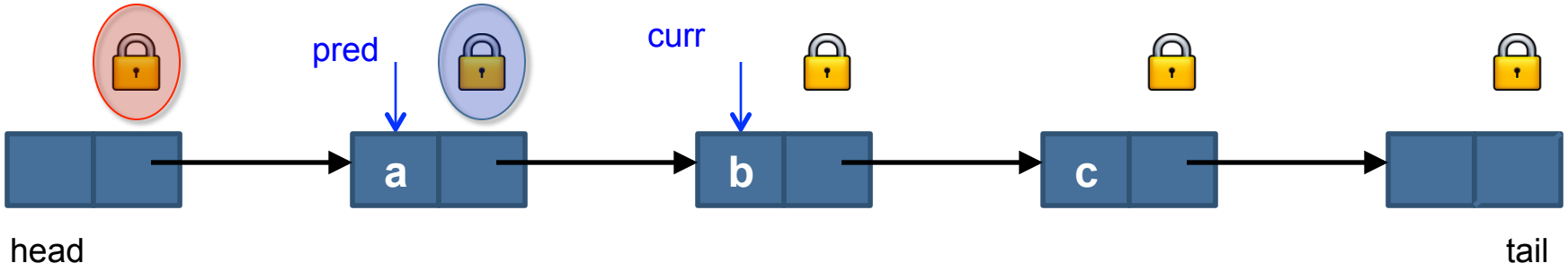
Lista ordenada – locks finos




```
remove(b)
    ...
    head.lock();
    pred = head; curr = pred.next;
    while (curr.key < item) {
      pred.unlock();
      pred = curr; curr = curr.next;
T1    pred.lock();
    }
    if (item == curr.key) {
      pred.next = curr.next;
      valid = true;
    }
    pred.unlock();
    return valid;
}
```

Assuma um “page fault” aqui!!

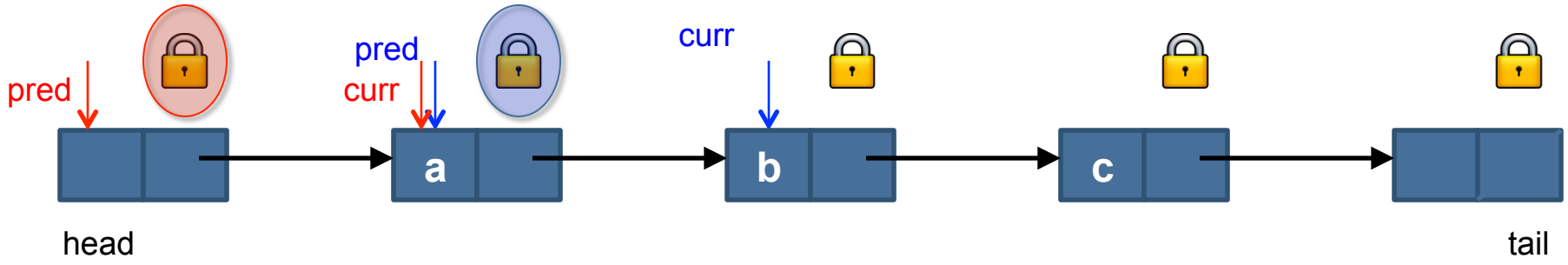
Lista ordenada – locks finos




```
remove(b)
    ...
    head.lock();
    pred = head; curr = pred.next;
    while (curr.key < item) {
      pred.unlock();
      pred = curr; curr = curr.next;
T1    pred.lock();
    }
    if (item == curr.key) {
      pred.next = curr.next;
      valid = true;
    }
    pred.unlock();
    return valid;
}
```

```
remove(a)
    ...
    head.lock();
    pred = head; curr = pred.next;
    while (curr.key < item) {
      pred.unlock();
      pred = curr; curr = curr.next;
T2    pred.lock();
    }
    if (item == curr.key) {
      pred.next = curr.next;
      valid = true;
    }
    pred.unlock();
    return valid;
}
```

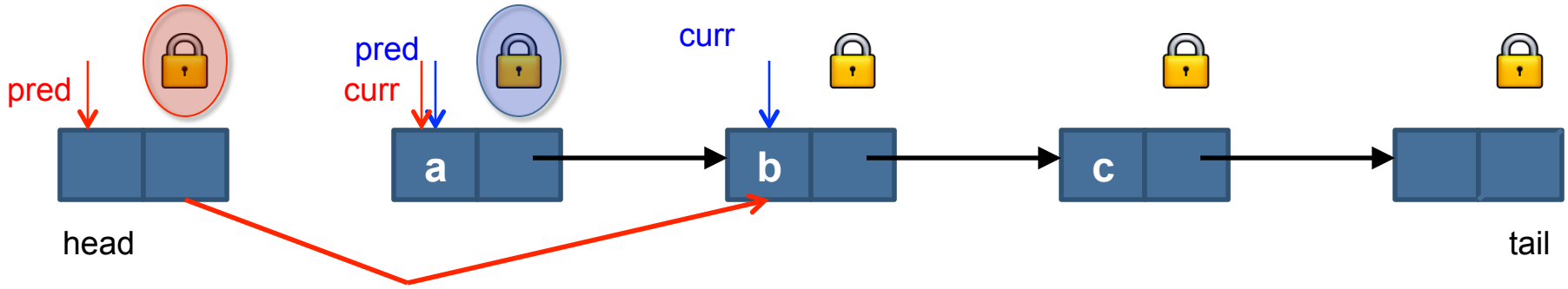
Lista ordenada – locks finos




```
remove(b)
    ...
    head.lock();
    pred = head; curr = pred.next;
    while (curr.key < item) {
      pred.unlock();
      pred = curr; curr = curr.next;
T1    pred.lock();
    }
    if (item == curr.key) {
      pred.next = curr.next;
      valid = true;
    }
    pred.unlock();
    return valid;
}
```

```
remove(a)
    ...
    head.lock();
    pred = head; curr = pred.next;
    while (curr.key < item) {
      pred.unlock();
      pred = curr; curr = curr.next;
T2    pred.lock();
    }
    if (item == curr.key) {
      pred.next = curr.next;
      valid = true;
    }
    pred.unlock();
    return valid;
}
```

Lista ordenada – locks finos




```
remove(b)
    ...
    head.lock();
    pred = head; curr = pred.next;
    while (curr.key < item) {
      pred.unlock();
      pred = curr; curr = curr.next;
      pred.lock();
    }
    if (item == curr.key) {
      pred.next = curr.next;
      valid = true;
    }
    pred.unlock();
    return valid;
}
```

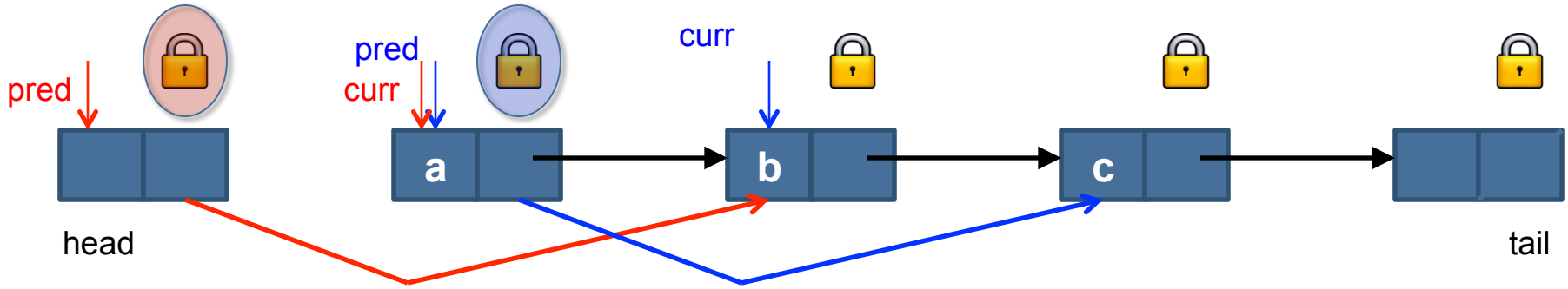
T_1

```
remove(a)
    ...
    head.lock();
    pred = head; curr = pred.next;
    while (curr.key < item) {
      pred.unlock();
      pred = curr; curr = curr.next;
      pred.lock();
    }
    if (item == curr.key) {
      pred.next = curr.next;
      valid = true;
    }
    pred.unlock();
    return valid;
}
```

T_2

Thread azul volta

Lista ordenada – locks finos



T_1

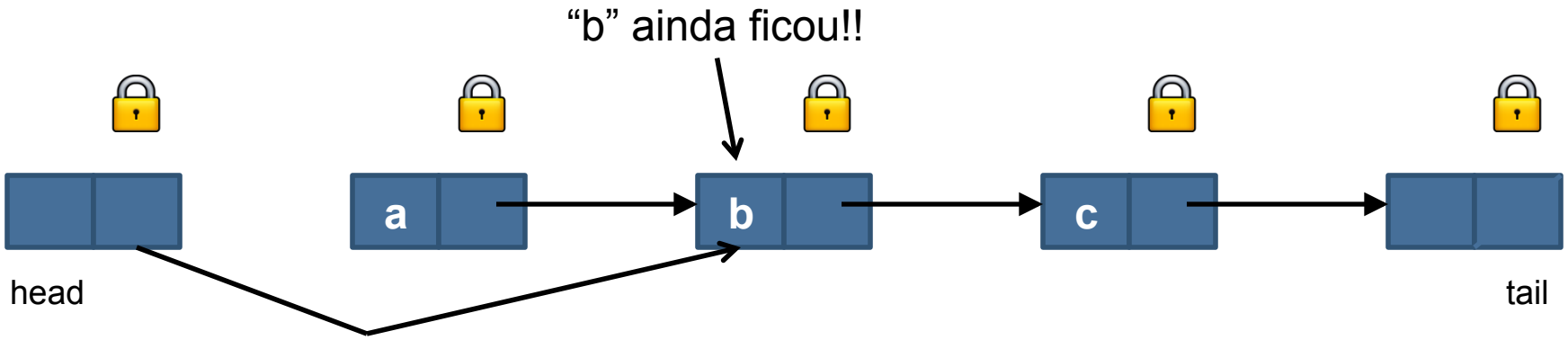
```
remove(b)
    ...
    head.lock();
    pred = head; curr = pred.next;
    while (curr.key < item) {
      pred.unlock();
      pred = curr; curr = curr.next;
      pred.lock();
    }
    if (item == curr.key) {
      pred.next = curr.next;
      valid = true;
    }
    pred.unlock();
    return valid;
}
```

T_2

```
remove(a)
    ...
    head.lock();
    pred = head; curr = pred.next;
    while (curr.key < item) {
      pred.unlock();
      pred = curr; curr = curr.next;
      pred.lock();
    }
    if (item == curr.key) {
      pred.next = curr.next;
      valid = true;
    }
    pred.unlock();
    return valid;
}
```

Resultado?

Lista ordenada – locks finos



T_1

```
remove(b)
    ...
    head.lock();
    pred = head; curr = pred.next;
    while (curr.key < item) {
      pred.unlock();
      pred = curr; curr = curr.next;
      pred.lock();
    }
    if (item == curr.key) {
      pred.next = curr.next;
      valid = true;
    }
    pred.unlock();
    return valid;
}
```

T_2

```
remove(a)
    ...
    head.lock();
    pred = head; curr = pred.next;
    while (curr.key < item) {
      pred.unlock();
      pred = curr; curr = curr.next;
      pred.lock();
    }
    if (item == curr.key) {
      pred.next = curr.next;
      valid = true;
    }
    pred.unlock();
    return valid;
}
```

Lista ordenada – locks finos

- Antes de alterar um nó, uma *thread* necessita adquirir as travas para o nó atual e o próximo
- Note que as threads envolvidas precisam adquirir os locks na **mesma ordem** para evitar o risco de **deadlock**
- Não é trivial provar a corretude!
- Exemplo de código para a operação de inserção ...

Lista ordenada – locks finos

```
public boolean add(int item) {
   boolean valid = false;
   head.lock();
     Node pred = head;
     Node curr = pred.next;
     curr.lock();
       while (curr.key < item) {
         pred.unlock();
         pred = curr;
         curr = curr.next;
         curr.lock();
       }
       if (item != curr.key) {
         Node newNode = new Node(item);
         newNode.next = curr;
         pred.next = newNode;
         valid = true;
       }
     curr.unlock();
   pred.unlock();
   return valid;
}
```

Lista ordenada – locks finos

```
public boolean add(int item) {
   boolean valid = false;
   head.lock();
     Node pred = head;
     Node curr = pred.next;
     curr.lock();
       while (curr.key < item) {
         pred.unlock();
         pred = curr;
         curr = curr.next;
         curr.lock();
       }
       if (item != curr.key) {
         Node newNode = new Node(item);
         newNode.next = curr;
         pred.next = newNode;
         valid = true;
       }
     curr.unlock();
   pred.unlock();
   return valid;
}
```

Grande parte do código é específico para sincronização (6 de 18 linhas = ~33%)

Problemas com lock finos

- Risco alto de deadlock
  - Diferentes locks adquiridos em diferentes ordens
- Operações **lock** e **unlock** custosas
  - Geralmente envolvem alguma forma de *syscall*
- Dificuldade relacionada a engenharia de software
  - Como encapsular um método com locks?
  - Como compor código?

Readers/Writer Lock

- Uma outra abordagem para sincronização das operações da lista ligada é permitir que threads que somente leem possam executar em paralelo
- Locks do tipo *Readers/Writer* permitem especificar a forma de travamento: se para leitura ou escrita
- Threads que usam o travamento no modo leitura podem executar em paralelo
- Threads que usam o modo escrita adquirem acesso exclusivo

Readers/Writer Lock - Pthreads

Tipo: **pthread_rwlock_t**

Operação	Pthreads
Inicializa um read-write lock	int **pthread_rwlock_init**(pthread_rwlock_t *rwlock, const pthread_rwlockattr_t *attr);
Adquire trava de leitura (compartilhada entre leitores)	int **pthread_rwlock_rdlock**(pthread_rwlock_t *rwlock);
Adquire trava de escrita (exclusiva)	int **pthread_rwlock_wrlock**(pthread_rwlock_t *rwlock);
Destrava	int **pthread_rwlock_unlock**(pthread_rwlock_t **rwlock*);
Destrói read-write lock	int **pthread_rwlock_destroy**(pthread_rwlock_t *rwlock);

Composição de código com locks

- Imagine que nossa aplicação precise utilizar as operações da lista ligada para implementar uma outra operação de nível mais alto, como mover um elemento de uma lista para outra

- Não temos acesso ao código fonte
  - Apenas sabemos que cada operação é atômica

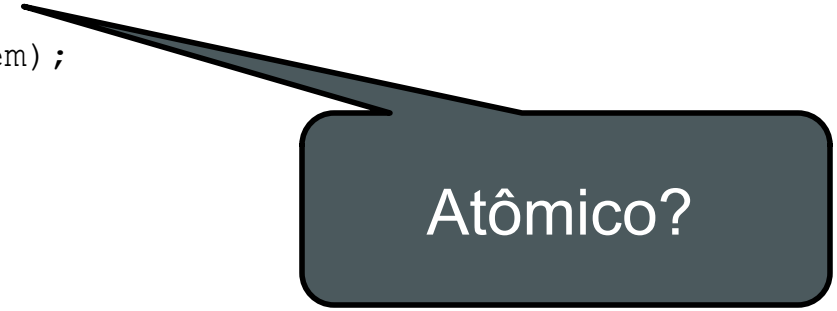
```
public boolean move(List from, List to, int item)
{
   from.remove(item);
   to.add(item);
}
```

Composição de código com locks

- Imagine que nossa aplicação precise utilizar as operações da lista ligada para implementar uma outra operação de nível mais alto, como mover um elemento de uma lista para outra

- Não temos acesso ao código fonte
  - Apenas sabemos que cada operação é atômica

```
public boolean move(List from, List to, int item)
{
   from.remove(item);
   to.add(item);
}
```



Composição de código com locks

- Colocar um lock global?

```
public boolean move(List from, List to, int item)
{
    newlock.lock();
    from.remove(item);
    to.add(item);
    newlock.unlock();
}
```

E se ocorrer *busca(item,from)* em outra thread neste ponto?

Funciona?

Composição de código com locks

- Colocar um lock global?

```
public boolean move(List from, List to, int item)
{
    newlock.lock();
    from.remove(item);
    to.add(item);
    newlock.unlock();
}
```

- Esta solução requer que todas as operações atômicas da lista sejam envoltas pelo novo lock

- Uma solução alternativa seria quebrar o encapsulamento e expor a implementação da lista (quais locks foram usados)

Composição de código com locks

- Cada lista expõe seu lock global

```
public boolean move(List from, List to, int item)
{
    from.lock(); to.lock();
    from.remove(item);
    to.add(item);
    from.unlock(); to.unlock();
}
```

- Esta solução funciona?

Composição de código com locks

- Cada lista expõe seu lock global

```
public boolean move(List from, List to, int item)
{
    from.lock(); to.lock();
    from.remove(item);
    to.add(item);
    from.unlock(); to.unlock();
}
```

- Esta solução funciona?

Thread 1

```
move(lista_clientes, lista_devedores, USER1);

from.lock();
to.lock();
from.remove(USER1);
to.add(USER1);
from.unlock();
to.unlock();
```

Thread 2

```
move(lista_devedores, lista_clientes, USER2);

from.lock();
to.lock();
from.remove(USER1);
to.add(USER1);
from.unlock();
to.unlock();
```

Composição de código com locks

- Cada lista expõe seu lock global

```
public boolean move(List from, List to, int item)
{
    from.lock(); to.lock();
    from.remove(item);
    to.add(item);
    from.unlock(); to.unlock();
}
```

- Esta solução funciona?

Thread 1

```
move(lista_clientes, lista_devedores, USER1);
    from.lock();
    to.lock();
    from.remove(USER1);
    to.add(USER1);
    from.unlock();
    to.unlock();
```

Thread 2

```
move(lista_devedores, lista_clientes, USER2);
    from.lock();
    to.lock();
    from.remove(USER1);
    to.add(USER1);
    from.unlock();
    to.unlock();
```

Composição de código com locks

- Cada lista expõe seu lock global

```
public boolean move(List from, List to, int item)
{
    from.lock(); to.lock();
    from.remove(item);
    to.add(item);
    from.unlock(); to.unlock();
}
```

- Esta solução funciona?

Thread 1

```
move(lista_clientes, lista_devedores, USER1);
    from.lock();
    to.lock();
    from.remove(USER1);
    to.add(USER1);
    from.unlock();
    to.unlock();
```

Thread 2

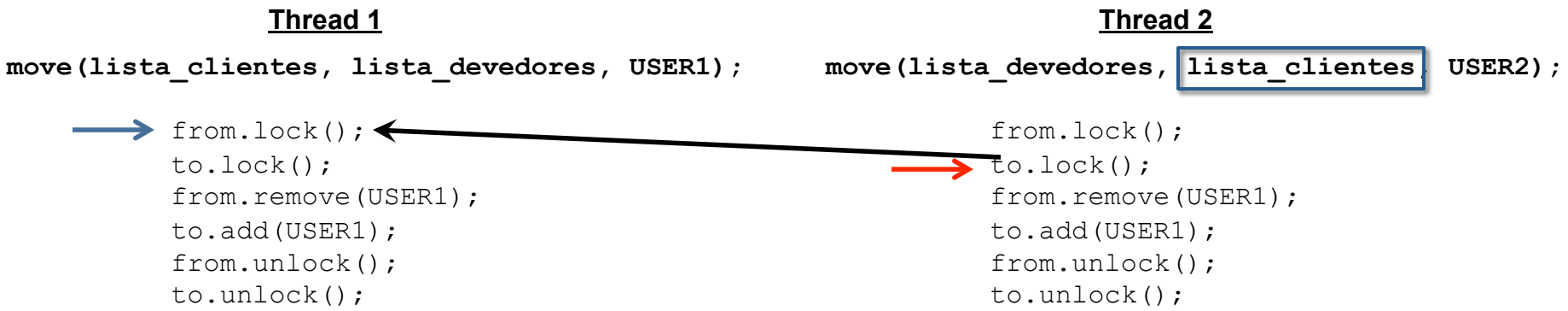
```
move(lista_devedores, lista_clientes, USER2);
    from.lock();
    to.lock();
    from.remove(USER1);
    to.add(USER1);
    from.unlock();
    to.unlock();
```

Composição de código com locks

- Cada lista expõe seu lock global

```
public boolean move(List from, List to, int item)
{
    from.lock(); to.lock();
    from.remove(item);
    to.add(item);
    from.unlock(); to.unlock();
}
```

- Esta solução funciona?



Composição de código com locks

- Cada lista expõe seu lock global

```
public boolean move(List from, List to, int item)
{
    from.lock(); to.lock();
    from.remove(item);
    to.add(item);
    from.unlock(); to.unlock();
}
```

- Esta solução funciona?