Priority Queues & Barriers

Christoph Kösters & Marinko Krajina

Priority Queues

Überblick

- **Allgemeines**
- Array-Based Bounded PQ
- Tree-Based Bounded PQ
- Unbounded Heap-Based PQ
- Skiplist-Based Unbounded PQ

Priority Queues: Allgemeines

- Multiset von Objekten
- Jedes Objekt besitzt einen Prioritätswert
- Konvention: kleinerer Wert -> höhere Priorität

- 2 Methoden:

```
public interface PQueue<T> {
  void add(T item, int priority);
  T removeMin();
}
```

Concurrent Priority Queues

- parallel ausgeführte Befehle können sich überschneiden
- 2 Consistency Conditions:
  - linearizability
  - quiescent consistency (schwächer, meist effektiver)

→ genaue Überlegung vor der Implementierung, welcher Ansatz benötigt wird

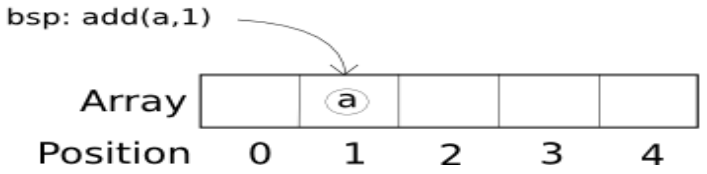
Überblick

- Allgemeines
- **Array-Based Bounded PQ**
- Tree-Based Bounded PQ
- Unbounded Heap-Based PQ
- Skiplist-Based Unbounded PQ

Array-Based Bounded PQ: Überblick

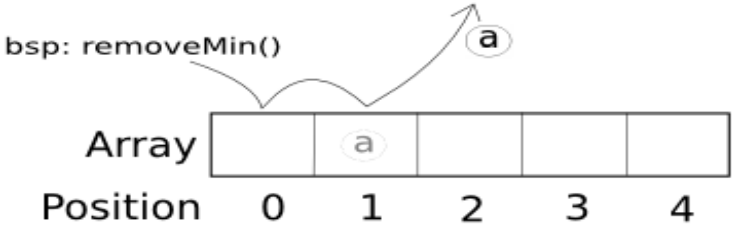
- Array von *m* Bins
- jeder Bin hält eine Priorität (*0..m-1*)

- add(*item, key*)
  - bin[*key*].put(*item*);



Array-Based Bounded PQ: removeMin()

- removeMin()
  - Array wird aufsteigend durchgegangen
  - erstes gefundene Item wird zurückgegeben



- linearisierbar, wenn Bins linearisierbar
- add() & removeMin() lock-free, falls Bin-Methoden lock-free

Array-Based Bounded PQ: Code

```
public class SimpleLinear<T> implements PQueue<T> {
  int range;
  Bin<T>[] pqueue;    // für jede Priorität 1 Bin
  public SimpleLinear(int range) {
    this.range = range;
    pqueue = (Bin<T>[])new Bin[range];
    for (int i = 0; i < pqueue.length; i++){
      pqueue[i] = new Bin<T>();
    }
  }
  public void add(T item, int key) {
    pqueue[key].put(item);
  }
  public T removeMin() {
    for (int i = 0; i < range; i++) {    // gehe Bins von vorne bis hinten durch
       T item = pqueue[i].get();
       if (item != null) {
         return item;              // gib Item zurück, sobald gefunden
       }
    }
    return null;
  }
}
```

Überblick

- Allgemeines
- Array-Based Bounded PQ
- **Tree-Based Bounded PQ**
- Unbounded Heap-Based PQ
- Skiplist-Based Unbounded PQ

Tree-Based Bounded PQ: Überblick

- Binärbaum aus treeNode Objekten

```
public class TreeNode {
  AtomicInteger counter;    // bounded counter
  TreeNode parent;    // reference to parent
  TreeNode right;     // right child
  TreeNode left;      // left child
  Bin<T> bin;         // non-null for leaf
  public boolean isLeaf() {
    return right == null;
  }
}
```

- *m* Blätter halten eigentliche Items
- *m-1* innere Knoten zählen die Items des jeweils *linken* Teilbaums
- add() & removeMin() benutzen die Counter, um einen Pfad zu legen bzw. zu lesen

Tree-Based Bounded PQ: Details

- Blätter und Wurzel werden gespeichert, um direkten Zugriff zu ermöglichen
  - TreeNode root;
  - List<TreeNode> leaves;
- add(*item, key*)
  - leaves.get(*key)* wird ausgewählt und *item* dessen bin hinzugefügt
  - Pfad wird bis zur Wurzel gelegt, indem der Counter des Elternknoten erhöht wird, falls von links aufgestiegen wird
- removeMin()
  - Wähle Wurzel aus
  - solange noch kein Blatt erreicht ist gehe
    - links & decrement für Counter ≥ 1
    - rechts sonst
  - bei Blatt: return node.bin.get();

Tree-Based Bounded PQ: Code

```
public class SimpleTree<T> implements PQueue<T> {
  int range;

  List<TreeNode> leaves; // array of tree leaves
  TreeNode root;     // root of tree
  public SimpleTree(int logRange) {
    range = (1 << logRange);
    leaves = new ArrayList<TreeNode>(range);
    root = buildTree(logRange, 0);
  }

  TreeNode buildTree(int height, int slot) {
    ...
  }

  public void add(T item, int priority) {
    TreeNode node = leaves.get(priority);
    node.bin.put(item);
    while(node != root) {
      TreeNode parent = node.parent;
      if (node == parent.left) { // increment if ascending from left
        parent.counter.getAndIncrement();
      }
      node = parent;
    }
  }

  public T removeMin() {
    TreeNode node = root;
    while(!node.isLeaf()) {
      if (node.counter.getAndDecrement() > 0 ) {
        node = node.left;
      } else {
        node = node.right;
      }
    }
    return node.bin.get(); // if null pqueue is empty
  }
```

// entsprechendes Blatt wird ausgewählt
// *Item* dessen *bin* hinzugefügt
// Pfad bis zur Wurzel legen

// bei Wurzel anfangen
// bis zu einem Blatt:
bei Counter > 0 -> links entlang
bei Counter = 0 -> rechts entlang

Tree-Based Bounded PQ: Details

- nicht linearisierbar
- quiescently consistent
- add() und removeMin() lock-free, falls die Bins und Counter lock-free sind
- Effizienz: logarithmisch zur niedrigsten Priorität

Überblick

- Allgemeines
- Array-Based Bounded PQ
- Tree-Based Bounded PQ
- **Unbounded Heap-Based PQ**
- Skiplist-Based Unbounded PQ

Unbounded Heap-Based PQ: Idee Heap

- linearisierbare Queue, dargestellt als Array von HeapNodes
- Wurzel hat Index 1, rechtes bzw. linkes Kind von *Node i* ist *2*i* bzw. *(2*i) + 1*
- HeapNodes halten Item und Score
- Heap: Binärerer Baum mit folgenden Eigenschaften
  - H1: Knoten und Blätter des Baums sind mit Objekten beschriftet (hier: priority scores)
  - H2: Alle Schichten sind gefüllt bis auf den rechten Teil der Untersten (hier: alle gefüllt)
  - **H3: Die Beschriftungen der Nachfolger eines Knotens sind kleiner oder gleich den Beschriftungen des Knotens (hier: je höher das Item im Baum, desto höher die Priorität)**

  → wichtigstes Item befindet sich in der Wurzel

- Problem: Einfügen & Entfernen können H3 verletzen

  → Eigenschaft muss durch vertauschen wieder hergestellt werden

Quelle: Algorithmen & Datenstrukturen, Skript SoSe 2009, Martin Hofmann

Unbounded Heap-Based PQ: Sequentieller Heap

- *next*-Feld speichert Index des ersten leeren HeapNodes
- add(*item, key*)
  - wähle Knoten an Position next++
  - speichere *item* und *key* in diesem Knoten
  - Heap-Eigenschaft bis zur Wurzel korrigieren:
    - swap(parent,child) und weitergehen — falls parent.priority < child.priority
    - return; — sonst

```
public void add(T item, int priority) {
  int child = next++;                          // erster unbenutzter Knoten
  heap[child].init(item, priority);
  while (child > ROOT) {                       // Bis zur Wurzel kontrollieren:
    int parent = child / 2;
    int oldChild = child;
    if (heap[child].priority < heap[parent].priority) {   // Heap-Eigenschaft verletzt???
      swap(child, parent);      // falls ja: swap() und
      child = parent;           // weitergehen
    } else {
      return;                   // falls nein: return
    }
  }
}
```

Unbounded Heap-Based PQ: Sequentieller Heap

- removeMin()
  - holt sich *item* aus der Wurzel
  - vertauscht Wurzel mit letztem Blatt (--next)
  - ehemalige Wurzel ist nun logisch leer
  - neue Wurzel verletzt evtl. Heapeigenschaft
  - Schleife: Stelle Heapeigenschaft wieder her

    Wähle ein Kind zum Vergleich
    - break, falls rechtes Kind *right* & linkes Kind *left* leer
    - links, falls *right* leer oder *left.priority* > *right.priority*
    - rechts sonst

    Vergleiche Prioritäten
    - Falls child.priority > parent.priority swap() & weitergehen
    - sonst break;
  - return *item*;

Unbounded Heap-Based PQ: Sequentieller Heap: Code

```
public T removeMin() {
  int bottom = --next;                    // letzter benutzter Knoten
  T item = heap[ROOT].item;               // Item speichern
  swap(ROOT, bottom);                     // Wurzel mit letztem Knoten ersetzen
  if (bottom == ROOT) {
    return item;
  }
  int child = 0;
  int parent = ROOT;
  while (parent < heap.length / 2) {      // Heap-Eigenschaft verletzt? nach unten gehen & prüfen
    int left = parent * 2; int right = (parent * 2) + 1;
    if (left >= next) {                   // beide Kinder leer
      break;
    } else if (right >= next || heap[left].priority < heap[right].priority) {
                                          // nur rechts leer oder linke Priorität > rechte
      child = left;
    } else {                              // linker Weg
      child = right;
    }                                     // sonst rechter Weg
    // If child higher priority than parent swap then else stop
    if (heap[child].priority < heap[parent].priority) {
      swap(parent, child);
      parent = child;
    } else {
      break;
    }
  }
  return item;
}
```

Unbounded Heap-Based PQ: Sequentieller Heap: Code

```
public class SequentialHeap<T> implements PQueue<T> {
  private static final int ROOT = 1;
  int next;
  HeapNode<T>[] heap;

  public SequentialHeap(int capacity) {
    next = 1;
    heap = (HeapNode<T>[]) new HeapNode[capacity + 1];
    for (int i = 0; i < capacity + 1; i++) {
      heap[i] = new HeapNode<T>();
    }
  }
```

// Wurzel
// erster unbenutzter Knoten
// Heap als Array dargestellt

// Initialisiert das gesamte Array

```
  ...
  private static class HeapNode<S> {
    int priority;
    S item;
    public void init(S myItem, int myPriority) {
      item = myItem;
      priority = myPriority;
    }
  }
}
```

// hält Priorität & Item

// existierende HeapNodes werden mit
// Werten belegt

Unbounded Heap-Based PQ: FineGrained Heap

- Erweiterung des Sequentiellen Heaps auf Nebenläufigkeit
- FineGrained Locking für Synchronisation
  - jeder Knoten besitzt eigenen Lock
  - Lock für gesamte Queue (*heapLock*)
- add() & removeMin() logisch gleich dem sequentiellen Heap

Unbounded Heap-Based PQ: FineGrained Heap: removeMin()

- Schritt 1: Knoten auswählen
  - *heapLock* aktivieren (kritische Bereiche: *root* & *next*)
  - Wurzel und letztes Blatt locken
  - *heapLock* freigeben
- Schritt 2: Wurzel extrahieren
  - *item* speichern, Wurzel und letztes Blatt vertauschen
  - leere, ehemalige Wurzel wird jetzt freigegeben
- Schritt 3: Heap-Eigenschaft herstellen
  - Beide Kinder locken
  - auswählen, ob links oder rechts verglichen wird (Knoten: *child*)
  - nicht benötigtes Kind freigeben
  - Vergleich auf Priorität durchführen und ggf. tauschen
  - *child* freigeben

Unbounded Heap-Based PQ: FineGrained Heap: removeMin()

- Schritt 4: *Item zurückgeben*
  - Falls getauscht wurde: wiederhole Schritt 3
  - sonst: gebe Knoten frei und *item* zurück

- Beobachtungen:
  - Wurzel ist bis zum Schluss gelockt
  - Bei jedem Vergleich sind genau 2 Knoten gelockt

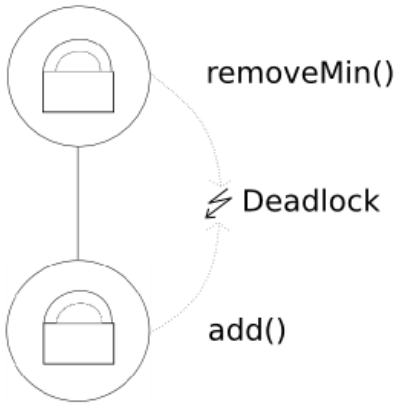
- ➔ Sequenz von einzelnen atomaren Schritten, die sich verschachteln dürfen

Unbounded Heap-Based PQ: FineGrained Heap: Code

```
public T removeMin() {
  heapLock.lock();
  int bottom = --next;
  heap[bottom].lock();
  heap[ROOT].lock();          // lock des zu korrigierenden Knotens, bis zum Schluss gelockt
  heapLock.unlock();
  if (heap[ROOT].tag == Status.EMPTY) {          // Heap leer?
    heap[ROOT].unlock();
    heap[bottom].lock();
    return null;
  }
  T item = heap[ROOT].item;                      // Item temporär gespeichert
  heap[ROOT].tag = Status.EMPTY;
  swap(bottom, ROOT);                            // Node ‚zurückgesetzt'
  heap[bottom].owner = NO_ONE;
  heap[bottom].unlock();
  if (heap[ROOT].tag == Status.EMPTY) {          // Heap jetzt leer?
    heap[ROOT].unlock();
    return item;
  }
  int child = 0;
  int parent = ROOT;
  while (parent < heap.length / 2) {
    int left = parent * 2;
    int right = (parent * 2) + 1;
    heap[left].lock();
    heap[right].lock();
    if (heap[left].tag == Status.EMPTY) {        // keine Kinder – fertig
      heap[right].unlock();
      heap[left].unlock();
      break;
    } else if (heap[right].tag == Status.EMPTY || heap[left].score < heap[right].score) {
      heap[right].unlock();                      // links auswählen, rechts unlock
      child = left;
    } else {
      heap[left].unlock();     // rechts auswählen, links unlock
      child = right;
    }
    if (heap[child].score < heap[parent].score) {   // Vergleich der Prioritäten, hier: immer 2 Knoten gelockt
      swap(parent, child);                          // vertauschen, falls Heap-Eigenschaft verletzt
      heap[parent].unlock();
      parent = child;
    } else {
      heap[child].unlock();
      break;
    }
  }
  heap[parent].unlock();       // unlock des korrigierten Knotens & return des Items
  return item;
}
```

Unbounded Heap-Based PQ: FineGrained Heap

- Problem:



- Lösung:
  beide von add() gelockten Knoten werden nach jedem Tausch wieder freigegeben
- removeMin() wird dadurch nicht unterbrochen, tauscht add() aber vielleicht dessen Knoten nach oben hin weg
- add() muss diesen Fall behandeln!

Unbounded Heap-Based PQ: FineGrained Heap

- Zusätzliche Felder in HeapNode helfen dabei, den Knoten wieder zu finden:
  - Status tag*;* kann sein:
    - EMPTY: Knoten ist leer
    - AVAILABLE: Knoten hält *item* und Wert
    - BUSY: Knoten wird gerade von einem add() Befehl korrigiert
  - int owner; hält ThreadID des bewegenden Knotens gdw. wenn *tag* = BUSY

Unbounded Heap-Based PQ: FineGrained Heap: add()

- Schritt 1: Knoten auswählen
  - *heapLock* aktivieren
  - erstes freies Blatt an Position next++ locken
  - heapLock freigeben
- Schritt 2: Knoten initialisieren
  - *item* und Priorität abspeichern
  - tag = Status.BUSY und owner = ThreadID
  - Knoten wieder freigeben ➔ jetzt keine Locks mehr aktiv

Unbounded Heap-Based PQ: FineGrained Heap: add()

- Schritt 3: Heap-Eigenschaft wieder herstellen
  - zuerst *parent*, dann sich selber (*child*) locken
  - Falls (parent.tag == AVAILABLE) && (child.amOwner())
    - vertausche, falls Position nicht stimmt
    - setze child.tag und child.owner zurück falls Position stimmt
  - falls (!child.amOwner())
    - gehe einen Knoten nach oben
  - beide Knoten freigeben
- Wiederhole Schritt 3, falls Position noch nicht gefunden

Unbounded Heap-Based PQ: FineGrained Heap: Code

```
public class FineGrainedHeap<T> implements PQueue<T> {
  private static int ROOT = 1;
  private static int NO_ONE = -1;// wird für owner-Feld benutzt
  private Lock heapLock;          // Lock auf gesamten Heap
  int next;
  HeapNode<T>[] heap;

  public FineGrainedHeap(int capacity) {
    heapLock = new ReentrantLock();
    next = ROOT;
    heap = (HeapNode<T>[]) new HeapNode[capacity + 1];
    for (int i = 0; i < capacity + 1; i++) {
      heap[i] = new HeapNode<T>();
    }
  }
  ...                                  // EMPTY: Knoten unbenutzt
  private static enum Status {         // AVAILABLE: Knoten hält Item und Wert
    EMPTY, AVAILABLE, BUSY             // BUSY: Node wird bewegt, owner-Feld hält ausführende Thread ID, siehe init()
  }

  private static class HeapNode<S> {   // Inner Class
    Status tag;
    int score;
    S item;
    int owner;                         // Lock für kurze Änderungen und während der Knoten
    Lock lock;                         // nach unten bewegt wird

    public void init(S myItem, int myPriority) {
      item = myItem;
      score = myPriority;
      tag = Status.BUSY;
      owner = ThreadID.get();
    }

    public HeapNode() {
      tag = Status.EMPTY;
      lock = new ReentrantLock();
    }
    ...
  }
}
```

Unbounded Heap-Based PQ: FineGrained Heap: Code

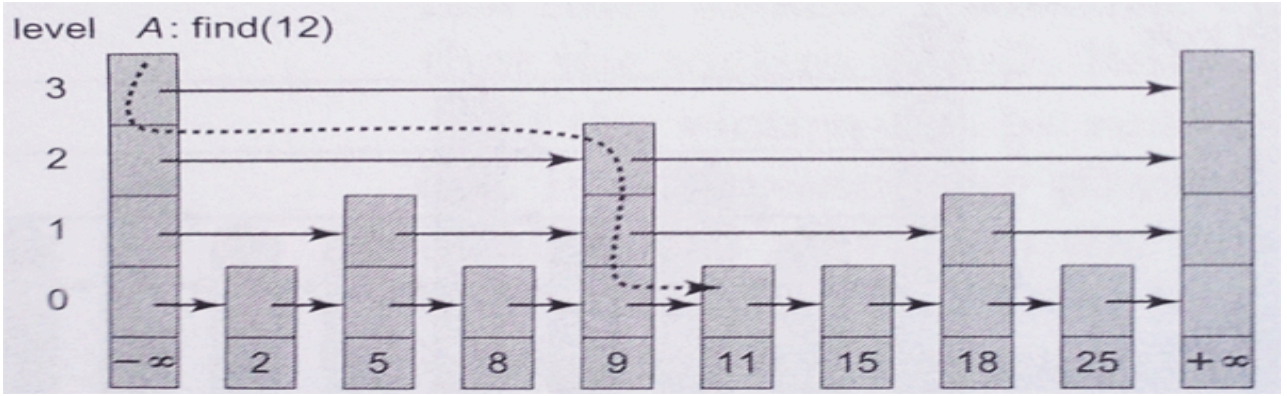
```
public void add(T item, int priority) {
  heapLock.lock();
  int child = next++;
  heap[child].lock();
  heapLock.unlock();
  heap[child].init(item, priority);   // tag = Status.BUSY; owner = ThreadID.get()
  heap[child].unlock();
  while (child > ROOT) {
    int parent = child / 2;
    heap[parent].lock();
    heap[child].lock();
    int oldChild = child;
    try {
      if (heap[parent].tag == Status.AVAILABLE && heap[child].amOwner()) {
        if (heap[child].score < heap[parent].score) {
          swap(child, parent);                          // vertauschen, falls Position nicht stimmt
          child = parent;
        } else {
          heap[child].tag = Status.AVAILABLE;           // Position gefunden: owner und tag-Felder ‚zurücksetzen‘
          heap[child].owner = NO_ONE;
          return;
        }
      } else if (!heap[child].amOwner()) {              // Knoten wurde von anderem removeMin() Befehl vertauscht
        child = parent;                                 // => muss sich nun also irgendwo oberhalb befinden, dort weitersuchen
      }
    } finally {
      heap[oldChild].unlock();
      heap[parent].unlock();
    }
  }
  if (child == ROOT) {                                  // falls Knoten bis ganz nach oben bewegt wurde
    heap[ROOT].lock();                                  // (noch nicht im oberen Vergleich behandelt)
    if (heap[ROOT].amOwner()) {
      heap[ROOT].tag = Status.AVAILABLE;
      heap[child].owner = NO_ONE;
    }
    heap[ROOT].unlock();
  }
}
```

Überblick

- Allgemeines
- Array-Based Bounded PQ
- Tree-Based Bounded PQ
- Unbounded Heap-Based PQ
- **Skiplist-Based Unbounded PQ**

Skiplist-Based Unbounded PQ: Überblick

- Sammlung von sortierten Listen von *Nodes*
- unterste Liste (*level 0*) enthält alle Knoten
- jede höhere nur noch einen Teil (*ungefähr ½*)
- effizientes einfügen&entfernen: O(log n)
- Bsp:



Skiplist-Based Unbounded PQ: PrioritySkiplist

- basiert auf LockFreeSkipList
- sortiert die *Nodes* nach Priorität
  ➔wichtigstes Objekt ganz vorne
- für korrekte, nebenläufige Ausführung:
  AtomicBoolean *marked*; *in inner-class Node*
- Knoten werden *lazily* entfernt:
  - findAndMarkMin() sucht und markiert ersten freien Knoten
  - remove() löscht diesen physisch aus der Liste
- add() und remove() sind bereits von *LockFreeSkipList* implementiert

Skiplist-Based Unbounded PQ: Code

```
public final class PrioritySkipList<T> {  // basiert auf LockFreeSkipList
  public static final class Node<T> {
    final T item;
    final int priority;
    AtomicBoolean marked;   // marked-Feld ermöglicht lazy-deletion
    final AtomicMarkableReference<Node<T>>[] next;
     // sentinel node Constructor
    public Node(int myPriority) { ... }
     // ordinary node constructor
    public Node(T x, int myPriority) { ... }
  }
  boolean add(Node node) { ... }    // Node als Parameter
  boolean remove(Node<T> node) { ... }
  public Node<T> findAndMarkMin() {
    Node<T> curr = null, succ = null;
    curr = head.next[0].getReference();
    while (curr != tail) {
      if (!curr.marked.get()) {
        if (curr.marked.compareAndSet(false, true)) {
          return curr;
        } else {
          curr = curr.next[0].getReference();
        }
      }
    }
    return null; // no unmarked nodes
  }

}
```

// Wrapper-Klasse ruft Methoden der Skiplist auf

```
public class SkipQueue<T> {
  PrioritySkipList<T> skiplist;
  public SkipQueue() {
    skiplist = new PrioritySkipList<T>();
  }
  public boolean add(T item, int priority) {
    Node<T> node = (Node<T>)new Node(item, priority);
    return skiplist.add(node);
  }
  public T removeMin() {
    Node<T> node = skiplist.findAndMarkMin();
    if (node != null) {
      skiplist.remove(node);   // erst markiert,
      return node.item;        // dann wirklich gelöscht
    } else{
      return null;
    }
  }

}
```

Skiplist-Based Unbounded PQ: Details

- quiescently consistent
- nicht linearisierbar
- lock-free
- i.A. effektiver als heap-based queue
  - minimales Item befindet sich bei *n Threads* unter den ersten *n Nodes*
  - dadurch schlechtestenfalls in O(log k) gelöscht
- aber auch hier Konfliktstellen:
  - Flaschenhals bei findAndMarkMin(): nur einer von mehreren Threads gewinnt
  - bei remove(): zu entfernende Knoten sind Nachbarn am Anfang der Liste, was zu compareAndSet()-Fehlern beim Entfernen der Referenzen führen kann

Fazit

- den einen perfekten Algorithmus gibt es (noch) nicht
- Verschiedene Algorithmen für verschiedene Zwecke

Barriers

Gliederung

- **1. Einführung**
- 2. Simple Barrier
- 3. Sense-Reversing Barrier
- 4. Combining Tree Barrier
- 5. Static Tree Barrier
- 6.Termination Detecting Barrier

1.Einführung

- Grafikdarstellung für ein Computerspiel

- Bsp. 1:
- while (true) {

```
frame.prepare();
frame.display();
}
```

1.Einführung

- Bsp. 2:
-
```
int me = ThreadID.get();
while (true) {
    frame[me].prepare();
    frame[me].display();
    }
```

  - Was für ein Problem könnte hier auftreten?

1.Einführung

- Synchronisationsprobleme
- Lösung: Barriers

- ```
  public interface Barrier {
      public void await();
      }
  ```
  ```

1.Einführung

```
private Barrier b;

    ...

int me = ThreadID.get();
while (true) {
    frame[me].prepare();
    b.await();
    frame[me].display();
    }
```

Gliederung

- 1. Einführung
- **2. Simple Barrier**
- 3. Sense-Reversing Barrier
- 4. Combining Tree Barrier
- 5. Static Tree Barrier
- 6.Termination Detecting Barrier

2. Simple Barrier

```
public class SimpleBarrier implements Barrier
{
  AtomicInteger count;
  int size;
  public SimpleBarrier(int n){
    this.count = new AtomicInteger(n);
    this.size = n;
  }
  public void await() {
    int position = count.getAndDecrement();
    if (position == 1) {         // If I'm last ...
      count.set(size);          // reset for next use
    } else {                    // otherwise spin
      while (count.get() != 0) {}
    }
  }
}
```

- AtomicInteger count
- await()
- getAndDecrement()

2. Simple Barrier

- Funktioniert nicht wenn die Barriere mehr als einmal benutzt wird.
- Schon bei zwei Threads kann es zum verhungern kommen.
- Einfache Lösung: Wechsel zwischen 2 Barrieren.
- Problem: Speicheraufwand.

Gliederung

- 1. Einführung
- 2. Simple Barrier
- **3. Sense-Reversing Barrier**
- 4. Combining Tree Barrier
- 5. Static Tree Barrier
- 6.Termination Detecting Barrier

3. Sense-Reversig Barrier

```
public class SenseBarrier implements Barrier {
  AtomicInteger count;     // how many threads
                              have arrived
  int size;                // number of threads
  volatile boolean sense;  // object's sense
  ThreadLocal<Boolean> threadSense;

  public SenseBarrier(int n) {
    count = new AtomicInteger(n);
    size = n;
    sense = false;
    threadSense = new ThreadLocal<Boolean>()
    {
     protected Boolean initialValue()
           { return !sense; };
    };
  }
}
```

- Eine elegante und praktische Lösung für das Problem der Wiederverwendung von Barrieren
- sense-field: boolescher Wert; true für geradzahlige Phasen, false

3. Sense-Reversing Barrier

- Veränderte await()-Methode :

```
public void await() {
    boolean mySense = threadSense.get();
    int position = count.getAndDecrement();
    if (position == 1) {      // I'm last
      count.set(this.size);     // reset counter
      sense = mySense;          // reverse sense
    } else {
      while (sense != mySense) {} // busy-wait
    }
    threadSense.set(!mySense);
  }
```

3. Sense-Reversing Barrier

- Hier kann es auch zu Problemen kommen beim gleichzeitigen Zugriff auf den counter – memory contention
- Ist gut geignet für cache-kohärenz Architekturen

Gliederung

- 1. Einführung
- 2. Simple Barrier
- 3. Sense-Reversing Barrier
- **4. Combining Tree Barrier**
- 5. Static Tree Barrier
- 6.Termination Detecting Barrier

4. Combining Tree Barrier

```
public class TreeBarrier implements Barrier {
  int radix;              // tree fan-in
  Node[] leaf;            // array of leaf nodes
  int leaves;                     // used to build tree
  ThreadLocal<Boolean> threadSense; // thread-local
                                                  // sense
  public TreeBarrier(int n, int r) {
    radix = r;
    leaves = 0;
    this.leaf = new Node[n / r];
    int depth = 0;
    threadSense = new ThreadLocal<Boolean>() {
      protected Boolean initialValue() { return true; };
    };
    while (n > 1) {       // compute tree depth
      depth++;
      n = n / r;
    }
    Node root = new Node();
    build(root, depth – 1); }          …
```

- vermeidet die memory contention
- Barriere als Baumstruktur mit kombinierten Thread-Zugriffen
- Radix r – Anzahl der

4. Combining Tree Barrier

```
// recursive tree constructor

   void build(Node parent, int depth) {
       if(depth == 0) {
            leaf[leaves++] = parent;
       } else {
         for(int i=0; i < radix; i++) {
           Node child = new Node(parent);
            build(child, depth – 1);
         }
       }
   }
```

- Erzeugt eine Baum-Struktur
- Wenn die Tiefe Null ist, dann werden die Blattknoten in ein Array von Knoten positioniert.

4. Combining Tree Barrier

```
private class Node {
  AtomicInteger count;
  Node parent;
  volatile boolean sense;

      . . .

 public void await() {
    boolean mySense = threadSense.get();
    int position = count.getAndDecrement();
    if (position == 1) {        // I'm last
      if (parent != null) { // root?
        parent.await();
      }
      count.set(radix);      // reset counter
      sense = mySense;
    } else {
      while (sense != mySense) {};
    }
    threadSense.set(!mySense);
  }
}
```

- Die TreeBarrier class hat eine *inner class* – die Klasse Node
- Jeder Knoten hat eine sense und einen counter, und noch seinen parent. (außer er ist die Wurzel) .

4. Combining Tree Barrier

```
public void await() {
  int me = ThreadID.get();
  Node myLeaf = leaf[me / radix];
  myLeaf.await();
}
```

- await() - Methode der Klasse TreeBarrier ruft die await() - Methode des Knotens auf, der von zugeordnetem Thread bearbeitet wird
- gut für NUMA – Architekturen

4. Combining Tree Barrier

- Probleme:

  - kann nicht gleichmäßige Notifications-time verursachen

  - Threads besuchen eine unberechenbare Sequenz von locations, und dieser Ansatz ist nicht gut für cacheless NUMA-Architekturen

  - übermäßige Kommunikation

Gliederung

- 1. Einführung
- 2. Simple Barrier
- 3. Sense-Reversing Barrier
- 4. Combining Tree Barrier
- **5. Static Tree Barrier**
- 6.Termination Detecting Barrier

5. Static Tree Barrier

- Unterschied zu der vorherigen Barriere :

  - Jeder Thread ist einem Knoten zugeordnet

  - Alle Knoten werden in einem Array gespeichert.

  - Die Node class ist verändert.

5. Static Tree Barrier

```
public class StaticTreeBarrier implements
Barrier {
  int radix;      // tree fan-in
  boolean sense;  // global sense
  Node[] node;    // array of nodes
  ThreadLocal<Boolean> threadSense;
                          //thread-local sense
  int nodes;      // used to build tree

  public StaticTreeBarrier(int size, int radix) {
    this.radix = radix;
    nodes = 0;
    this.node = new Node[size];
    int depth = 0;
    // compute tree depth
    while (size > 1) {
      depth++;
      size = size / radix;
    }
build(null, depth);
    sense = false;
    threadSense = new ThreadLocal<Boolean>()
{
      protected Boolean initialValue() { return !
sense; };
    };
  }

  public void await() {
    node[ThreadID.get()].await();
  }
```

5. Static Tree Barrier

```
//recursive tree constructor
    void build(Node parent, int depth) {
            if(depth == 0) {
            node[nodes++] = new Node(parent, 0);
        } else {
            Node myNode = new Node(parent, radix);
            node[nodes++] = myNode;
            for(int i=0; i<radix; i++) {
                    build(myNode, depth – 1);
            }
        }
    }
```

- Kleine Änderung am Konstruktor : alle Knoten werden im Array abgespeichert.

5. Static Tree Barrier

- Jeder Knoten enthält :
  - Die Anzahl seiner Kinder
  - Seinen Vaterknoten
  - Anzahl der noch nicht abgearbeiteten Kinder
- Die await() - Methode hat auch eine andere Arbeitsweise
- Ein Thread wartet erst bis alle Knoten unterhalb von ihm fertig sind um weiterzuarbeiten

5. Static Tree Barrier

```
class Node {
  final int children; // number of children
  final Node parent;
  AtomicInteger childCount; // number of children incomplete

  public Node(Node parent, int count) {
    this.children = count;
    this.childCount = new AtomicInteger(count);
    this.parent = parent;
  }

  public void await() {
    boolean mySense = threadSense.get();
    while (childCount.get() > 0) {};  // spin until children done
    childCount.set(children);         // prepare for next round
    if (parent != null) {         // not root?
      parent.childDone();           // indicate child subtree
                                    // completion
      while (sense != mySense) {}; // wait for global sense to
                                   // change
    } else {
      sense = !sense;    // am root: toggle global sense
    }
    threadSense.set(!mySense); // toggle sense
  }

  public void childDone() {
    childCount.getAndDecrement();
  }
}
```

5. Static Tree Barrier

- diese Barriere beseitigt die Probleme die wir bei den letzten 2 Barrieren kennengelernt haben
- Zur Beendigung der Barriere bei cache-kohärenten Multiprozessoren benötigt man log(n) Schritte den Baum hinauf

Gliederung

- 1. Einführung
- 2. Simple Barrier
- 3. Sense-Reversing Barrier
- 4. Combining Tree Barrier
- 5. Static Tree Barrier
- **6.Termination Detecting Barrier**

6. Termination Detecting Barrier

- Bisher wurde die Arbeit in Phasen aufgeteilt, wo jeder Thread eine Phase abarbeitet
- In diesem Algorithmus können Threads die Arbeit von anderen Threads "stehlen" und bearbeiten.
- Jeder Thread ist entweder aktiv oder inaktiv
- setActive() und isTerminated()

6. Termination Detecting Barrier

```
public class SimpleTDBarrier implements TDBarrier {
  AtomicInteger count;

  public SimpleTDBarrier(int n){
    this.count = new AtomicInteger(n);
  }

  public void setActive(boolean active) {
    if (active) {
      count.getAndDecrement();
    } else {
      count.getAndIncrement();
    }
  }

  public boolean isTerminated() {
    return count.get() == 0;
  }
}
```

6. Termination Detecting Barrier

- Bsp. modifizierte run() - Methode :
  - Am Anfang werden alle Threads als aktiv deklariert.
  - Sobald eine lokale Warteschlange erschöpft ist werden alle inaktiv.
  - Bevor ein Thread eine Aufgabe stiehlt testet er ob die Queue leer ist.

```
public void run() {
  int me = ThreadID.get();
  tdBarrier.setActive(true);
  Runnable task = queue[me].popBottom(); // attempt to
                                         //pop 1st item
  while (true) {
   while (task != null) {        // if there is an item
    task.run();                  // execute it and then
    task = queue[me].popBottom(); // pop the next item
   }
   tdBarrier.setActive(false);       // no work
 while (task == null) {          // steal an item
    int victim = random.nextInt(queue.length);
    if (!queue[victim].isEmpty()) {
     tdBarrier.setActive(true);        // tentatively active
     task = queue[victim].popTop();
     if (task == null) {
      tdBarrier.setActive(false);
     }
    }
    if (tdBarrier.isTerminated()) {
     return;
    }
   }
  }
```

6. Termination Detecting Barrier

- Eine korrekt benutzte TDBarrier muss zwei Eigenschaften erfüllen:
  - Sicherheit
  - Lebendigkeit

Quellen

- The art of multiprocessor programming, Maurice Herlihy & Nir Shavit, (Chapter 15 & 17)

Vielen Dank für ihre Aufmerksamkeit