Samy Ateia – Philip Czech

Futures, Scheduling and Work Distribution

Proseminar Nebenläufige Programmierung

Sommersemester 2010

Um was geht es?

- Programme Aufteilen und parallel ausführen
- Effizienz und Parallelität analysieren
- Thread Management

Themen

- Parallel Programming
- Analyzing Parallelism
- Realistic Multiprocessor Scheduling
- Work Distribution
- Work-Stealing DEQueues
- Work Balancing

Parallele Programme

- Typische parallele Probleme
  - Server Anfragen; Producer & Consumer; Brute-Force-Suche
- Nicht so offensichtlich parallele Probleme
  - Matrizenmultiplikation

Matrizenmultiplikation

$$c_{ij} = \sum_{k=0}^{N-1} a_{ik} * b_{kj}$$

$C_{00} = 1*2 + 3*4 = 14$ $C_{01} = 2*0 + 3*2 = 6$

$C_{10} = 1*1 + 1*4 = 5$ $C_{11} = 1*0 + 1*2 = 2$

	B:	1	0
A:	C:	4	2
2	3	14	6
1	1	5	2

Matrizenmultiplikation

```
class MMThread {
  double[][] a, b, c;
  int n;
  public MMThread(double[][] a, double[][]  b) {
    n = a.length;
    this.a = a;
    this.b = b;
    this.c = new double[n][n];
  }
  void multiply() {
    Worker[][] worker = new Worker[n][n];

    for (int row = 0; row < n; row++)
      for (int col = 0; col < n; col++)
        worker[row][col] = new Worker(row,col);

    for (int row = 0; row < n; row++)
      for (int col = 0; col < n; col++)
        worker[row][col].start();

    for (int row = 0; row < n; row++)
      for (int col = 0; col < n; col++)
          worker[row][col].join();
  }}
```

```
  class Worker extends Thread {
    int row, col;
    Worker(int row, int col) {
      this.row = row; this.col = col;
    }
    public void run() {
      double dotProduct = 0.0;
      for (int i = 0; i < n; i++) {
        dotProduct += a[row][i] * b[i][col];
      }
      c[row][col] = dotProduct;
    }
  }
}
```

Matrizenmultiplikation

- Probleme?
  - Was passiert bei 1000 X 1000 Matrizen?

  ⟶ Extremer Verwaltungs-Overhead

- Lösung:
  - Thread-Pools

    -Threads werden wiederverwendet

Futures und Thread-Pools in Java

- Interface: java.util.concurrent.ExecutorService

```
Future<?> future = executor.submit(Runnable task);
future.get(); //nur Warten
```

```
Future<T> future = submit(Callable<T> task);
T value = future.get(); //Warten auf  ein Ergebnis
```

- Achtung Parallele Ausführung ohne Gewähr!

Matrizenaddition

- Zur Vereinfachung A(nxn): n = 2i (i Є N)
- C = A + B

$$\begin{pmatrix} C_{00} & C_{00} \\ C_{10} & C_{10} \end{pmatrix} = \begin{pmatrix} A_{00} + B_{00} & B_{01} + A_{01} \\ A_{10} + B_{10} & A_{11} + B_{11} \end{pmatrix}$$

Matrizenaddition

```
private static class Matrix {
    int dim;
    double[][] data;
    int rowDisplace;
    int colDisplace;
    Matrix(int d) {
      dim = d;
      rowDisplace = colDisplace = 0;
      data = new double[d][d];
}
    Matrix(double[][] matrix, int x, int y, int d) {
      data = matrix;
      rowDisplace = x;
      colDisplace = y;
      dim  = d;
}
    double get(int row, int col) {
      return data[row+rowDisplace][col+colDisplace];
}
    void set(int row, int col, double value) {
      data[row+rowDisplace][col+colDisplace] = value;
}
    int getDim() {
      return dim;
}
```

```
Matrix[][] split() {
      Matrix[][] result = new Matrix[2][2];
      int newDim = dim / 2;
      result[0][0] = new Matrix(data,
rowDisplace, colDisplace, newDim);
      result[0][1] = new Matrix(data,
rowDisplace, colDisplace + newDim, newDim);
      result[1][0] = new Matrix(data, rowDisplace
+ newDim, colDisplace, newDim);
      result[1][1] = new Matrix(data, rowDisplace
+ newDim, colDisplace + newDim, newDim);
      return result;
}
}
```

```
public class MatrixTask {
  static ExecutorService exec = Executors.newCachedThreadPool();
...
static Matrix add(Matrix a, Matrix b) throws InterruptedException, ExecutionException {
    int n = a.getDim();
    Matrix c = new Matrix(n);
    Future<?> future = exec.submit(new AddTask(a, b, c));
    future.get();
    return c;
  }
static class AddTask implements Runnable {
    Matrix a, b, c;
    public AddTask(Matrix a, Matrix b, Matrix c) {
      this.a = a; this.b = b; this.c = c;
    }
    public void run() {
        int n = a.getDim();
        if (n == 1) {c.set(0, 0, a.get(0,0) + b.get(0,0));} else {
          Matrix[][] aa = a.split(), bb = b.split(), cc = c.split();
          Future<?>[][] future = (Future<?>[][]) new Future[2][2];
          for (int i = 0; i < 2; i++)
            for (int j = 0; j < 2; j++)
              future[i][j] = exec.submit(new AddTask(aa[i][j], bb[i][j], cc[i][j]));
          for (int i = 0; i < 2; i++)
            for (int j = 0; j < 2; j++)
              future[i][j].get();
}}}}
```

Fibonacci-Folge

$$F(n) = \begin{cases} 1 \text{ if } n = 0 \text{ or } 1 \\ F(n-1) + F(n-2) \text{ if } n > 1 \end{cases}$$

F(4) =

F(3) = + F(2)=

F(2)= + F(1) F(1) + F(0)

F(1) + F(0)

Paralleles Potential da, aber auch Abhängigkeiten!

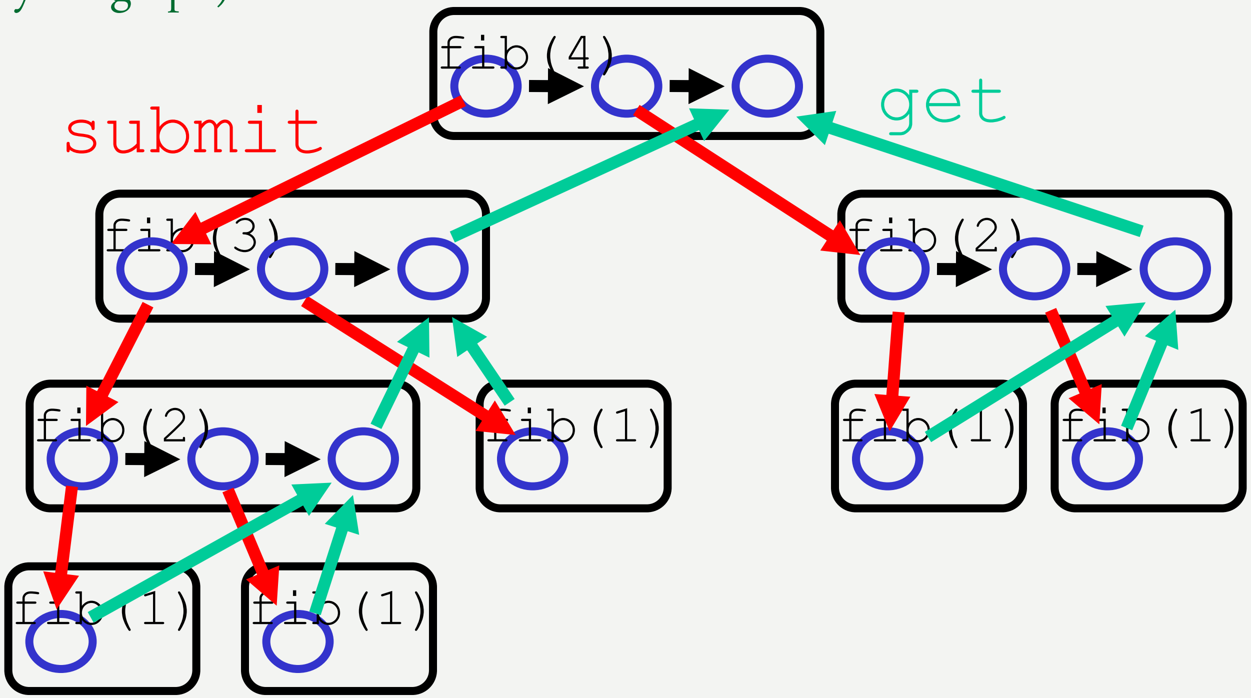
Multithreaded Fibonacci

```
•class FibTask implements Callable<Integer> {

  static ExecutorService exec = Executors.newCachedThreadPool();
  int arg;
  public FibTask(int n) {
    arg = n;
}
  public Integer call() {
      if (arg > 2) {
        Future<Integer> left = exec.submit(new FibTask(arg-1));
        Future<Integer> right = exec.submit(new FibTask(arg-2));
        return left.get() + right.get();
      } else {
        Return 1;
}}}
```

Dynamisches Verhalten analysieren

- DAG (directed acyclic graph)



Dynamisches Verhalten analysieren

- Tp = Sequentielle Anzahl von Rechenschritten auf p Prozessoren
- T1 = Rechenschritte auf einem Prozessor
  - Tp >= T1/P
- T∞ = Rechenzeit auf ∞ Prozessoren
  - T1/Tp = speedup auf p Prozessoren
  - T1/T∞ = max speedup, Parallelität einer Berechnung

Dynamisches Verhalten analysieren

- Matrizenaddition
  - Besteht rekursiv aus 4 Additionen + split
- Ap(n)

  $A_1(n) = 4A_1(n/2) + \Theta(1) = \Theta(n^2)$

  $A_\infty(n) = A_\infty(n/2) + \Theta(1) = \Theta(\log n)$
- Parallelität

  $A_1(n)/A_\infty(n) = \Theta(n^2)/\Theta(\log n)$

  n=1000; 10^7 / ~10 = 10^6

Dynamisches Verhalten analysieren

- Achtung: Hohe Parallelität != Hohe Performanz
- Warum?

Realistic Multiprocessor Scheduling

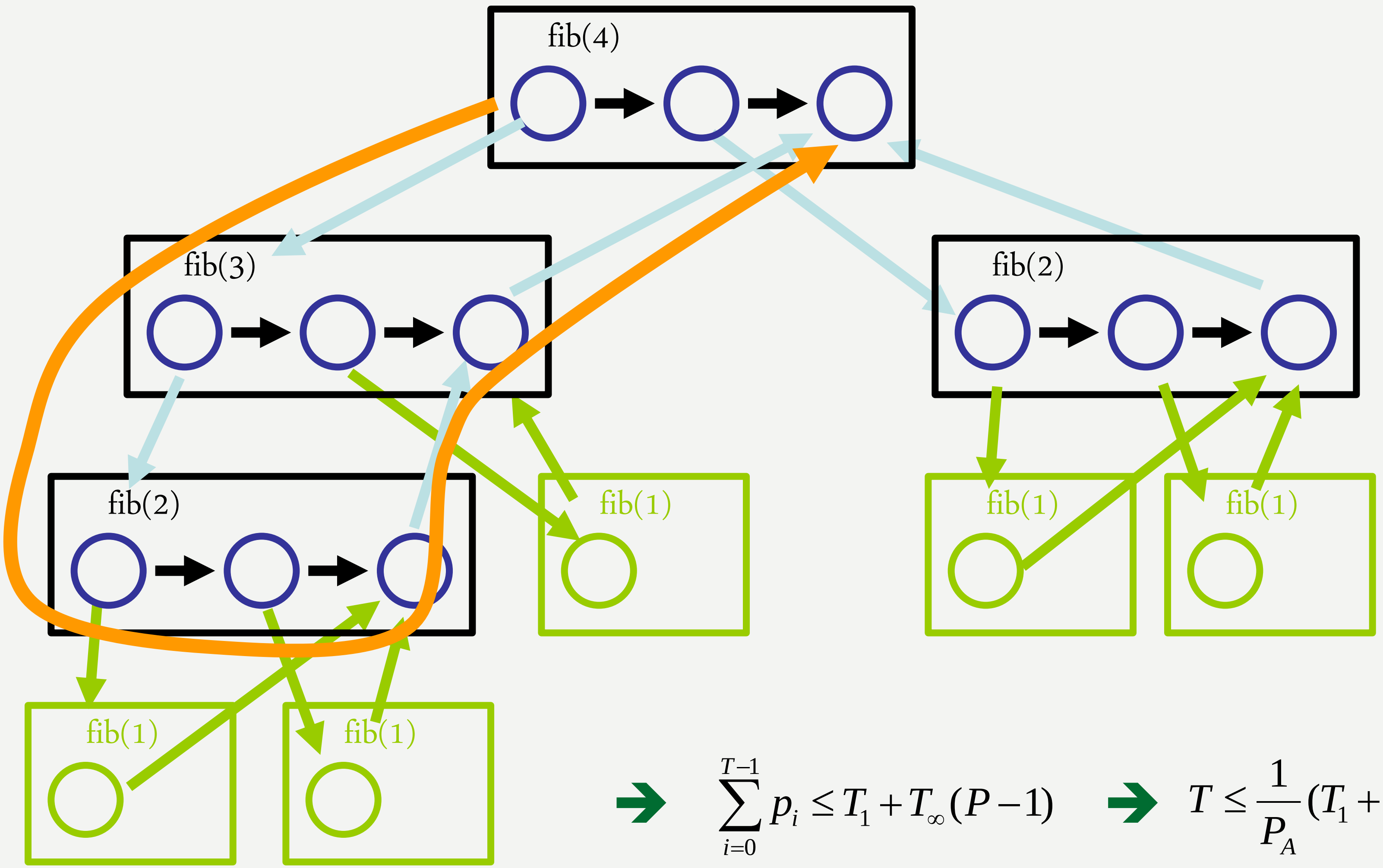
- Internes 3-Level-Model: User Level / Scheduler / Kernel Level
- kein Zugriff auf Kernel Level

  > Kein P Speedup möglich, aber:

  > Pa Speedup: $P_A = \frac{1}{T}\sum_{i=0}^{T-1} p_i$ $T = \frac{1}{P_A}\sum_{i=0}^{T-1} p_i$

- Theorem zur maximale Steplänge(T) in einem Multithread Programm:

$$\frac{T_1}{P_A} + \frac{T_\infty (P-1)}{P_A}$$



$$\rightarrow \sum_{i=0}^{T-1} p_i \leq T_1 + T_\infty (P-1) \quad \rightarrow \quad T \leq \frac{1}{P_A}(T_1 + T_\infty (P-1))$$

Work Distribution

- Work Dealing: Beschäftigte Threads geben Arbeit an unbeschäftigte ab - ineffektiv

- Konsequenz: Work-Stealing – Threads ohne Tasks holen sich Arbeit von anderen

- Jeder Thread besitzt einen Pool an Tasks die in Form einer DEQueue

(Double-Ended-Queue) gespeichert werden.

Diese stellt zum Zugriff die vier Methoden Push/Pop-Top/Bottom() zur Verfügung.

```java
public class WorkStealingThread {
    DEQueue[] queue;
    int me;
    Random random;
    public WorkStealingThread(DEQueue[] myQueue) {
        queue = myQueue;
        random = new Random();
    }
    public void run() {
        int me = ThreadID.get();
        Runnable task = queue[me].popBottom();
        while(true) {
            while (task != null) {
                task.run();
                task = queue[me].popBottom();
            }
            while (task == null) {
                Thread.yield();
                int victim = random.nextInt(queue.length);
                if (!queue[victim].isEmpty()) {
                    task = queue[victim].popTop();
                }
            }
        }
    }
}
```

Bounded Work-Stealing DEQueue

```
public class BDEQueue {
    Runnable[] tasks;
    volatile int bottom;
    AtomicStampedReference<Integer> top;
    public BDEQueue(int capacity) {
        tasks = new Runnable[capacity];
        top = new AtomicStampedReference<Integer>(0, 0);
        bottom = 0;
    }
    public void pushBottom(Runnable r) {
        tasks[bottom] = r;
        bottom++;
    }
    boolean isEmpty() {
        return (top.getReference() < bottom);
    }
}
```

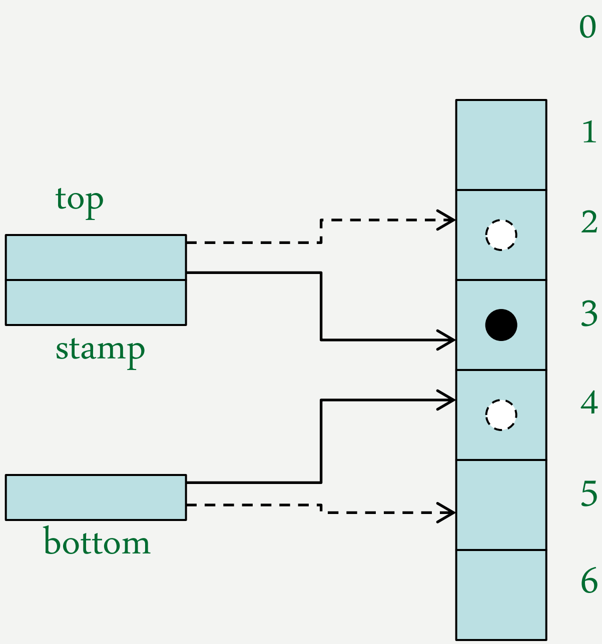
Bounded Work-Stealing DEQueue

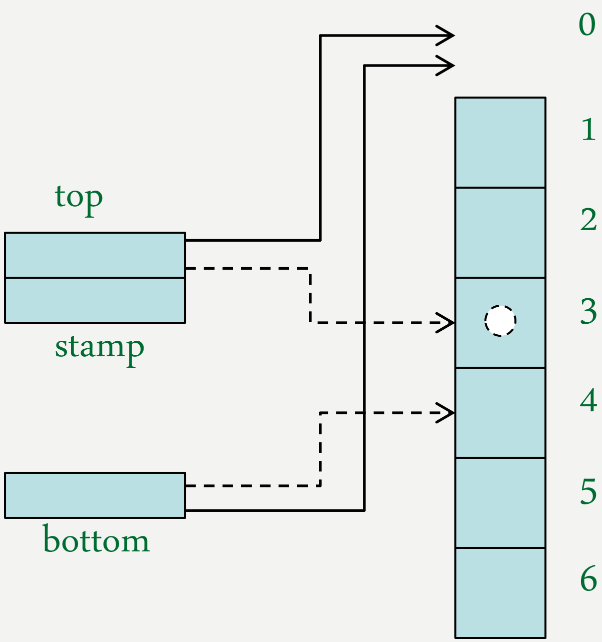
```java
public Runnable popTop() {
    int[] stamp = new int[1];
    int oldTop = top.get(stamp), newTop = oldTop + 1;
    int oldStamp = stamp[0], newStamp = oldStamp + 1;
    if (bottom <= oldTop)
        return null;
    Runnable r = tasks[oldTop];
    if (top.compareAndSet(oldTop, newTop, oldStamp, newStamp))
        return r;
    return null;
}
```

Bounded Work-Stealing DEQueue

```java
public Runnable popBottom() {
    if (bottom == 0)
        return null;
    bottom --;
    Runnable r = tasks[bottom];
    int[] stamp = new int[1]
    int oldTop = top.get(stamp), newTop = 0;
    int oldStamp = stamp[0], newStamp = oldStamp + 1;
    if (bottom > oldTop)
        return r;
    if (bottom == oldTop) {
        bottom = 0;
        if (top.compareAndSet(oldTop, newTop, oldStamp, newStamp))
            return r;
    }
    top.set(newTop, newStamp);
    return null;
}
```

Bounded Work-Stealing DEQueue





Bounded Work-Stealing DEQueue

- push/pop – Methoden um Tasks aus einer DEQueue zu holen oder hinzuzufügen
- yield – Methode um Prozessoren effektiv an arbeitende Threads zu verteilen
- stamps um ABA Problem zu lösen und Mehrfachzugriffe zu vermeiden
- compareAndSet wird erst aufgerufen wenn eine DEQueue sehr klein wird um Prozessorlast zu sparen
- bottom Variable wird volatile deklariert, damit Arbeit suchende Threads gleich informiert werden wenn ein anderer Thread leer ist (da bottom nicht synchronized/atomic ist)

➔ Problem: Nicht für Threads geeignet die zufällig sehr viel oder wenig Arbeit haben
➔ Lösung: Unbounded Work-Stealing DEQueues

```java
class CircularArray {
    private int logCapacity;
    private Runnable[] currentTasks;
    CircularArray(int myLogCapacity) {
        logCapacity = myLogCapacity;
        currentTasks = new Runnable[1 << logCapacity];
    }
    int capacity() {
        return 1 << logCapacity;
    }
    Runnable get(int i) {
        return currentTasks[i % capacity()];
    }
    void put(int i, Runnable task) {
        currentTasks[i % capacity()] = task;
    }
    CircularArray resize(int bottom, int top) {
        CircularArray newTasks = new CircularArray(logCapacity+1);
        for (int i = top; i < bottom; i++) {
            newTasks.put(i, get(i));
        }
        return newTasks;
    }
}
```

```java
public class UnboundedDEQueue {
    private final static int LOG_CAPACITY = 4;
    private volatile CircularArray tasks;
    volatile int bottom;
    AtomicReference<Integer> top;
    public UnboundedDEQueue(int LOG_CAPACITY) {
        tasks = new CircularArray(LOG_CAPACITY);
        top = new AtomicReference<Integer>(0);
        bottom = 0;
    }
    boolean isEmpty() {
        int localTop = top.get();
        int localBottom = bottom;
        return (localBottom <= localTop);
    }

    public void pushBottom(Runnable r) {
        int oldBottom = bottom;
        int oldTop = top.get();
        CircularArray currentTasks = tasks;
        int size = oldBottom - oldTop;
        if (size >= currentTasks.capacity() - 1) {
            currentTasks = currentTasks.resize(oldBottom, oldTop);
            tasks = currentTasks;
        }
        tasks.put(oldBottom, r);
        bottom = oldBottom + 1;
    }
```

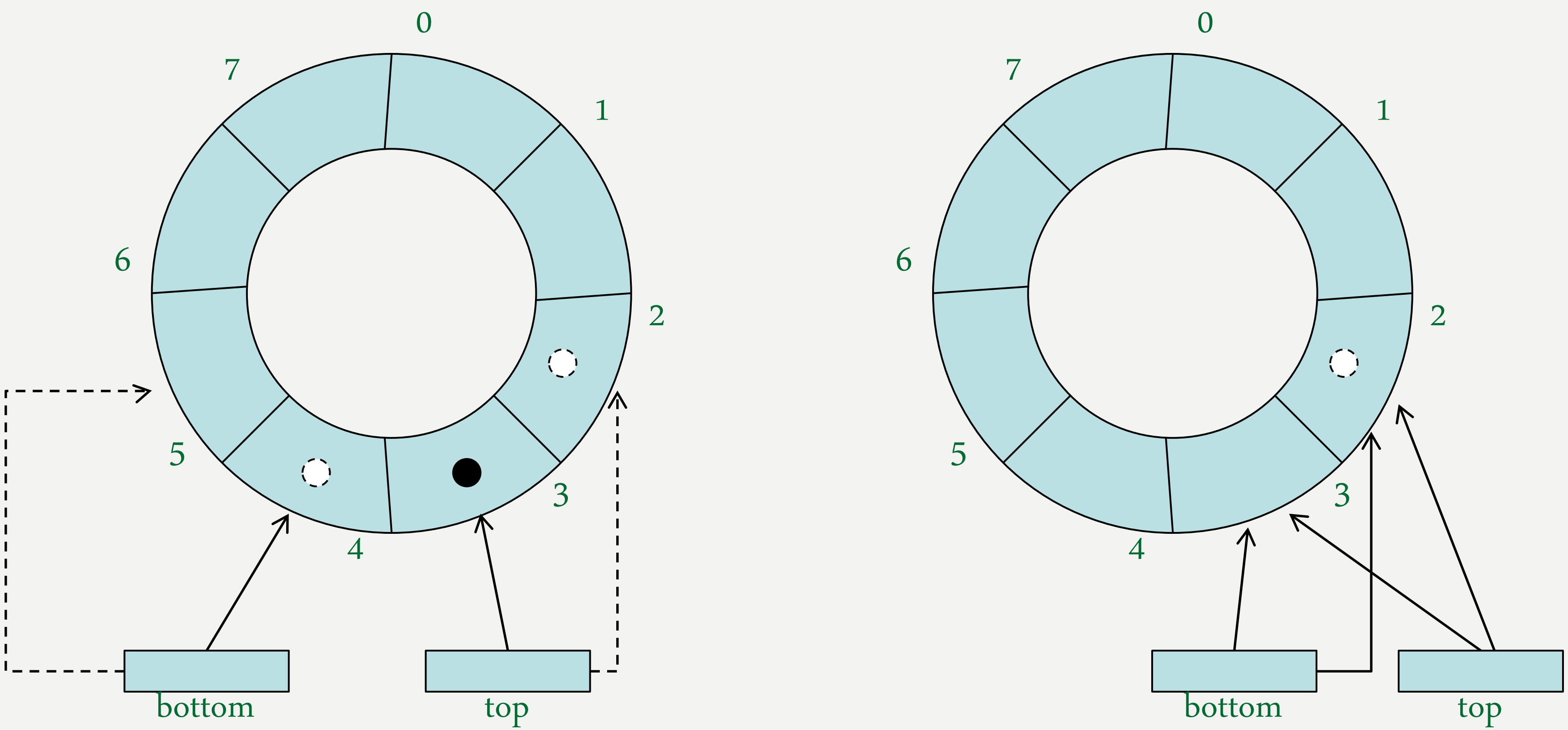
Unbounded Work-Stealing DEQueue

```
public Runnable popTop() {
    int oldTop = top.get();
    int newTop = oldTop + 1;
    int oldBottom = bottom;
    CircularArray currentTasks = tasks;
    int size = oldBottom - oldTop;
    if (size <= 0) return null;
    Runnable r = tasks.get(oldTop);
    if (top.compareAndSet(oldTop, newTop))
        return r;
    return null;
}
```

Unbounded Work-Stealing DEQueue

```java
public Runnable popBottom() {
    CircularArray currentTasks = tasks;
    bottom--;
    int oldTop = top.get();
    int newTop = oldTop + 1;
    int size = bottom - oldTop;
    if (size < 0) {
        bottom = oldTop;
        return null;
    }
    Runnable r = tasks.get(bottom);
    if (size > 0)
        return r;
    if (!top.compareAndSet(oldTop, newTop))
        r = null;
    bottom = oldTop + 1;
    return r;
}
```

Unbounded Work-Stealing DEQueue



Unbounded Work-Stealing DEQueue

- Kreis Array um einen Reset von bottom und top auf 0 zu vermeiden und um Grösse dynamisch erweitern zu können

- Kreis Array ermöglicht weiterhin, dass top nur inkrementiert wird und verzichtet deshalb auf eine AtomicStampedReference

- pushBottom Methode vergrössert das Array dynamisch
- beim Kopieren in ein neues Array müssen top und bottom nie angepasst werden, da dass Array immer mit Index mod Kapazität angelegt wird

Work Balancing

- Alternativer Ansatz zum Verteilen von Tasks aufs Threads:
  > Threads gleichen regelmäßig ihren Workload mit Nachbar Threads aus

- Um Prozessorlast zu vermeiden initialisieren nur weniger ausgelastete Threads den Ausgleich
- Die Wahrscheinlichkeit zum Ausgleich ist entsprechend invers-proportional zur Taskmenge
- Vorteil: Mehrere Tasks werden gleichzeitig in einen anderen geschoben

```java
public class WorkSharingThread {
    Queue[] queue;
    Random random;
    private static final int THRESHOLD = ...;
    public WorkSharingThread(Queue[] myQueue) {
        queue = myQueue;
        random = new Random();
    }
    public void run() {
        int me = ThreadID.get();
        while(true) {
        Runnable task = queue[me].deq();
        if (task != null) task.run();
        int size = queue[me].size();
        if (random.nextInt(size+1) == size) {
            int victim = random.nextInt(queue.length);
            int min = (victim <= me) ? victim : me;
            int max = (victim <= me) ? me : victim;
            synchronized (queue[min]) {
                synchronized (queue[max]) {
                    balance(queue[min], queue[max]);
                }
            }
        }
    }
```

Work Balancing

```
private void balance(Queue q0, Queue q1) {
    Queue qMin = (q0.size() < q1.size()) ? q0 : q1;
    Queue qMax = (q0.size() < q1.size()) ? q1 : q0;
    int diff = qMax.size() - qMin.size();
    if (diff > THRESHOLD)
        while (qMax.size()  qMin.size())
            qMin.enq(qMax.deq());
}
}
```

Quellen:

- M. Herlihy and N. Shavit. **The art of multiprocessor programming. 2008, Morgan Kaufmann Publishers.**
- http://www.elsevierdirect.com/companion.jsp?ISBN=9780123705914