LMU

Monitors
and
Blocking Synchronisation

Concurrent Stacks
and
Elimination

Gliederung

Monitors and Blocking Synchronisation

- Monitor Locks und Conditions
- Producer – Consumer Problem
- Readers – Writers Lock
- Reentrant Lock
- Monitore in Java

Concurrent Stacks and Elimination

- Unbounded Lock – free – Stack
- Elimination Backoff Stack

1
Monitors and Blocking Synchronisation

Monitore

- Zusammensetzung einer oder mehrerer Prozeduren, lokalen Daten und einer Warteschlange für ankommende Prozesse
- Zugang zu lokalen Variablen nur über Monitorprozeduren
- Gleichzeitiger Zugriff mehrerer Prozesse auf Monitor nicht möglich
- Zur Sicherung des Wechselseitigen Ausschlusses

Monitor Locks

```
public interface Lock {
    void lock();                                //unterbricht den aufrufenden Thread, bis er das Lock erhält
    void lockInterruptibly()
        throws InterruptedException;            //funtioniert wie lock() und wirft eine Exception falls der
                                                //Thread beim Warten unterbrochen wird
    boolean tryLock();                          //übernimmt den Lock, falls dieser frei ist und gibt einen Boolean
                                                //zurück, der aussagt, ob die Lockanforderung erfolgreich war.
                                                //Dies kann auch in Verbindung mit einem Timeout aufgerufen werden
    boolean tryLock(long time, Timeout unit);
    Condition newCondition();                   //erzeugt ein Object vom Typ Condition und gibt dieses zurück
    void unlock();                              //gibt das Lock wieder frei
}
```

- newCondition() erzeugt ein neues Condition Objekt

Conditions

```
public Interface Condition {
    void await() throws InterruptedException;
    boolean await(long time, TimeUnit unit) throws InterruptedException;
    boolean awaitUntil(Date deadline) throws InterruptedException;
    long awaitNanos(long nanosTimeout) throws InterruptedException;
    void awaitUninterruptibly();
    void signal();          //weckt einen wartenden Thread
    void signalAll();       //weckt alle wartenden Threads
}
```

- Condition Object: grundlegender Bestandteil von Monitoren

Aufgabe der Conditions

- Erfüllung spezieller Bedingungen notwendig, damit ein Thread ausgeführt werden kann
- Thread fordert Lock an
- Thread überprüft Bedingung
- Thread gibt Lock temporär frei, falls Bedingung nicht erfüllt ist:

→ await()

- anderer Thread erhält Lock und wird ausgeführt
- weckt einen oder mehrere schlafende Threads durch:

→ signal() weckt speziellen Thread

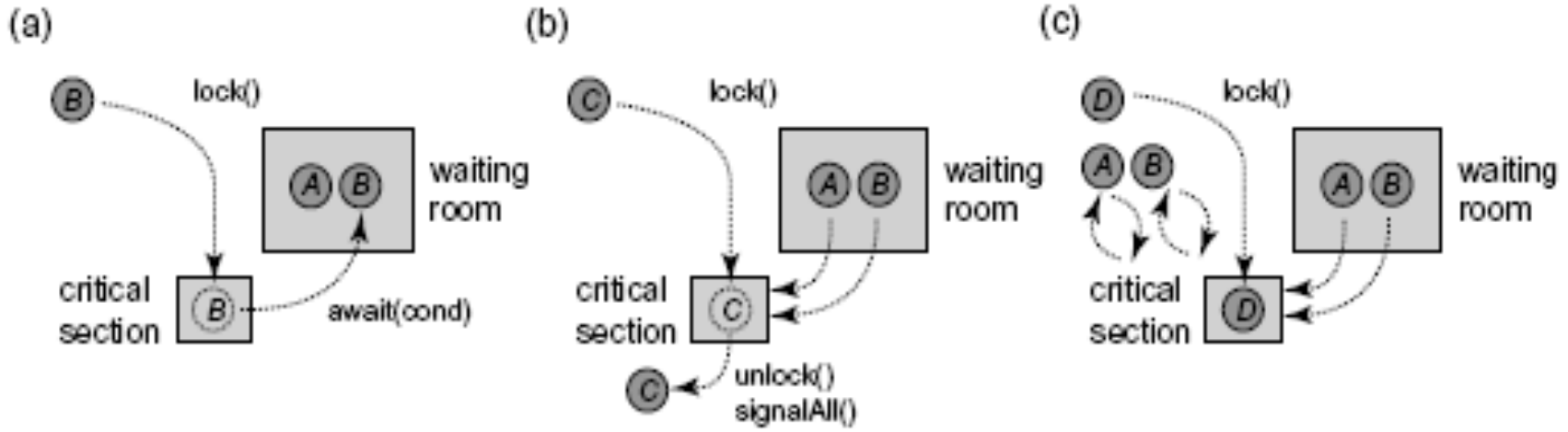
→ signalAll() weckt alle wartenden Threads

Aufgabe der Conditions

- Grundgerüst für die Verwendung von Conditions:

```
Condition condition = mutex.newCondition();
...
mutex.lock();
try {
    while(!property) {      //property ist die Bedingung, die erfüllt sein muss,
                            //damit der Thread ausgeführt werden kann
        condition.await();  //es wird auf die Erfüllung der Bedingung gewartet
    }
    catch (InterruptedException e) {
    ...                     //Application-abhängige Reaktion
    }
    ...                     //die Bedingung ist erfüllt, der Thread wird ausgeführt
}
```

Aufgabe der Conditions



a) Thread B fordert lock an, muss jedoch auf Bedingung warten
   Thread A wartet bereits
b) Thread C wird ausgeführt und weckt durch signalAll() alle wartenden Threads
c) D erhält lock vor A oder B → A und B befinden sich im Zustand spinning

Producer – Consumer Problem

- Operationen auf einer Queue
- Zwei Condition Objekte:
  - notEmpty: Bedingung für wartende Verbraucher
  - notFull: Bedingung für wartende Erzeuger

Producer – Consumer Problem

```
class LockedQueue<T> {
    final Lock lock = new ReentrantLock();
    final Condition notFull = lock.newCondition();
    final Condition notEmpty = lock.newCondition();
    final T[] items;
    int tail, head, count;

    publicLockedQueue(int capacity) {
        items = (T[]) new Object[100];
    }
```

```
public void enq(T x) {  //fügt Element  an
    lock.lock();
    try {
                        //Bedingung
        while (count == items.length) {
            notFull.await();
        }

        //Element wird angehängt

        ++count;
        notEmpty.signal();//Signal für notEmpty

    }
    finally {
        lock.unlock();
    }
}
```

```
    public void  T deq() {  //entfernt Element
        lock.lock();
        try {
            while (count == 0) {//Bedingung
                notEmpty.await();
            }

            //Element wird entfernt

            --count;
            notFull.signal();
            return x;
            finally {
                lock.unlock();
            }
        }
    }
}
```

Lost – Wakeup Problem

Problem:

- Wartende Threads erkennen nicht, dass ihre Bedingung schon erfüllt ist

Lösung:

- Einbau eines Timeouts
- Aufwecken aller Threads

Readers – Writers Locks

Zur Vermeidung von Deadlocks:

- Mehrere Leser dürfen gleichzeitig auf Objekt zugreifen
- Nur ein Schreiber darf zur gleichen Zeit auf ein Objekt zugreifen

2 Locks: ReadLock und WriteLock

- Solange WriteLock gehalten wird darf kein anderer Lock gehalten werden

Readers – Writers Lock

```
public class SimpleReadWriteLock implements ReadWriteLock {
    int readers;
    boolean writer;
    Lock lock;
    Condition condition;
    Lock readLock, writeLock;

    public SimpleReadWriteLock() {
        writer = false;
        readers = 0;
        lock = new ReentrantLock();
        readLock = new ReadLock();
        writeLock = new WriteLock();
        condition = lock.newCondition();
    }

    public Lock readLock(){
        return readLock;
    }

    public Lock writeLock() {
        return writeLock;
    }
}
```

Readers–Writers Lock

```
class ReadLock implements Lock {

    public void lock() {
        lock.lock();
        try {
            while (writer) {
                condition.await();
            }
            readers++;
        }
        finally {
            lock.unlock();
        }
    }

    public void unlock() {
        lock.lock();
        try {
            readers--;
            if (readers == 0) {
                condition.signalAll();
            }
        }
        finally {
            lock.unlock();
        }
    }
}
```

```
protected class WriteLock implements Lock {

    public void lock() {
        lock.lock();
        try {
            while (readers > 0) {
                condition.await();
            }
            writer = true;
        }
        finally {
            lock.unlock();
        }
    }

    public void unlock() {
        writer = false;
        condition.signalAll();
    }
}
```

Reentrant Lock

- Spezielles Lock
- Kann ohne Deadlock mehrmals vom gleichen Thread angefordert werden
- Sinnvoll, falls Methode andere Methode aufruft, welche das gleiche Lock anfordert
- ID des lockhaltenden Threads wird dafür gespeichert

Reentrant Lock

```
public class SimpleReentrantLock {
    Lock lock;
    Condition condition;
    int owner, holdCount;
    public SimpleReentrantLock() {
        lock = new SimpleLock();
        condition = lock.newCondition();
        owner = 0;
        holdCount = 0;
    }
    public void lock() {
        int me = ThreadID.get();
        lock.lock();
        if (owner == me) {
            holdCount++;
            return;
        }
        while (holdCount != 0) {
            condition.await();
        }
        owner = me;
        holdCount = 1;
    }
```

```
    public void unlock() {
        lock.lock();
        try {
            if (holdCount == 0 || owner != ThreadID.get()) {
                throw new IllegalMonitorStateException();
            }
            holdCount--;
            if (holdCount == 0) {
                condition.signal();
            }
        }
        finally {
            lock.unlock();
        }
    }

    public Condition newCondition() {
        throw new UnsupportedOperationException("Not supported yet.");
    }
    ...
}
```

Monitore in Java

- müssen nicht selbst implementiert werden
- durch „synchronized“ werden Methoden oder ganze Codeblöcke zu kritischen Abschnitten erklärt
- interner Monitor mit ReentrantLock wird erzeugt

  → für wechselseitigen Ausschluss wird gesorgt
- wait() und notify() bzw notifyAll() entsprechen await() und signal() bzw signalAll()

```
public synchronized void action() {
    while (!bedingung) {
        wait();
    }
    //Aktion wird ausgeführt
    notifyAll();
}
```

2
Concurrent Stacks and Elimination

Stack

Stack<T> Klasse:

- Stapel von Elementen des Typs T
- Unterstützung von push() und pop() - Methoden nach dem LIFO – Prinzip
- Nicht automatisch sequentiell

Stack

- Stack als Linked List aus Einträgen von Typ Node

```
public class Node{

    public T value;          //Inhalt des zu speichernden Elementes
    public Node next;        //Nachfolgerknoten

    public Node(T value){    //Konstruktor
        value = value;
        next = null;
    }
}
```

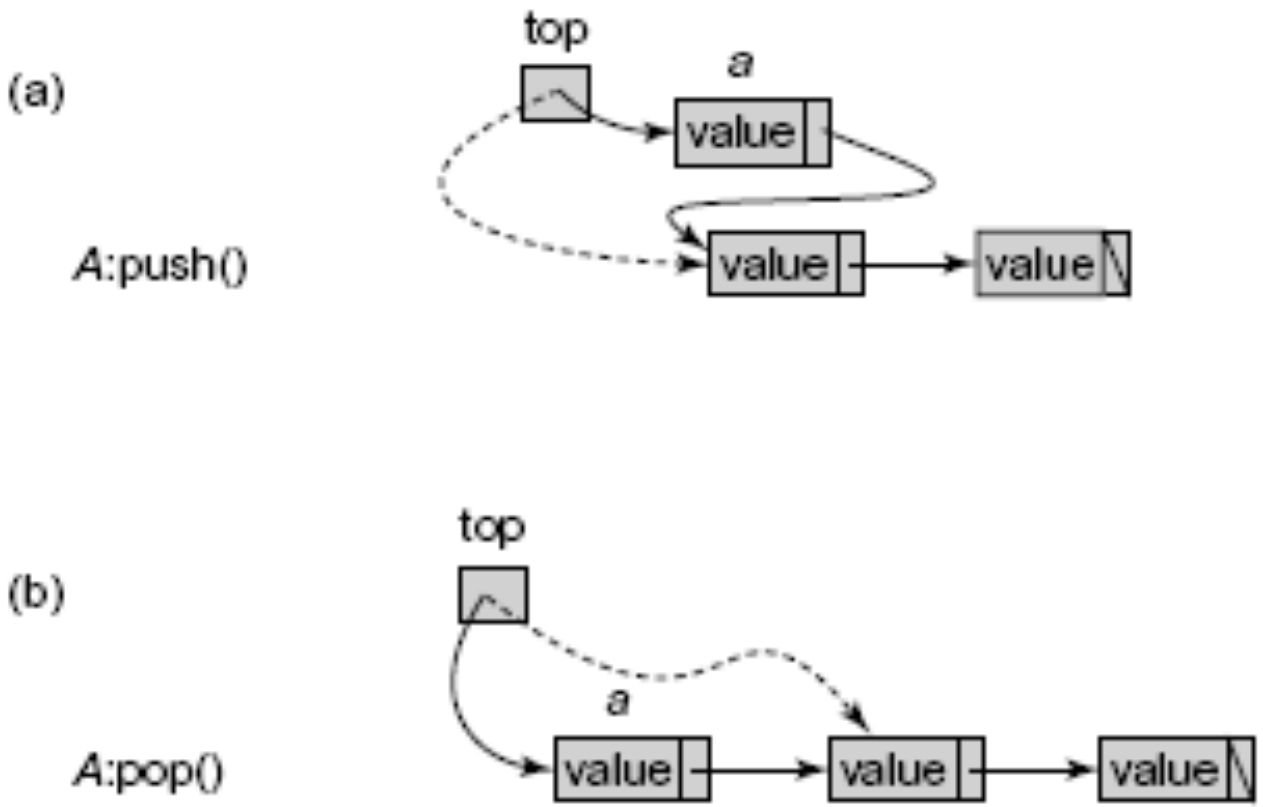
Unbounded Lock Free Stack

```
public class LockFreeStack<T>{

    AtomicReference<Node> top = new AtomicReference<Node>(null);
    static final int MIN_DELAY = ...;
    static final int MAX_DELAY = ...;

    Backoff backoff = new Backoff(MIN_DELAY, MAX_DELAY);

        protected boolean tryPush(Node node){
            Node oldTop = top.get();    // holt sich den alten Knoten
            node.next = oldTop;         // neuer Knoten setzt Zeiger auf alten ersten Knoten
            return(top.compareAndSet(oldTop, node)); //setzt Referenz von top auf neuen Knoten
        }

        public void push(T value){
            Node node = new Node(value);
            while(true){
                if(tryPush(node)){
                    return;
                }else{
                    backoff.backoff();
                }
            }
        }
```

Unbounded Lock Free Stack

```
    protected Node tryPop() throws EmptyException {

        Node oldTop = top.get();
        if(oldTop == null){
            throw new EmptyException();
        }
        Node newTop = oldTop.next;
        if(top.compareAndSet(oldTop, newTop)){
            return oldTop;
        }else{
            return null;
        }
    }
    public T pop() throws EmptyException{
        while(true){
        Node returnNode = tryPop();
        if(returnNode != null) {
            return returnNode.value;
        }else{
            backoff.backoff();
        }
    }
  }
}
```

Unbounded Lock Free Stack



push() und tryPush()

push(T value)

- Erstellung eines neuen Knotens und Einfügen in die Liste
- Erfolgreiches Einfügen mithilfe von tryPush
- Einfügen fehlgeschlagen: Backoff

tryPush(Node node)

- Versuch, den ersten Eintrag der Liste durch neuen Knoten zu ersetzen

→ Referenz neu: compareAndSet()

pop() und tryPop()

pop()

- Aufruf von tryPop
- Bei Erfolg: Ausgabe des entfernten Eintrages
- tryPop() fehlgeschlagen: Backoff

tryPop()

- Exception bei leerem Stack
- Versuch, den ersten Eintrag der Liste zu löschen
- Referenz auf den nachfolgenden Eintrag:

→ Ausgabe entfernten Eintrages

Elimination

- Gegenseitiges Aufheben von aufeinanderfolgenden push – und pop – Befehlen

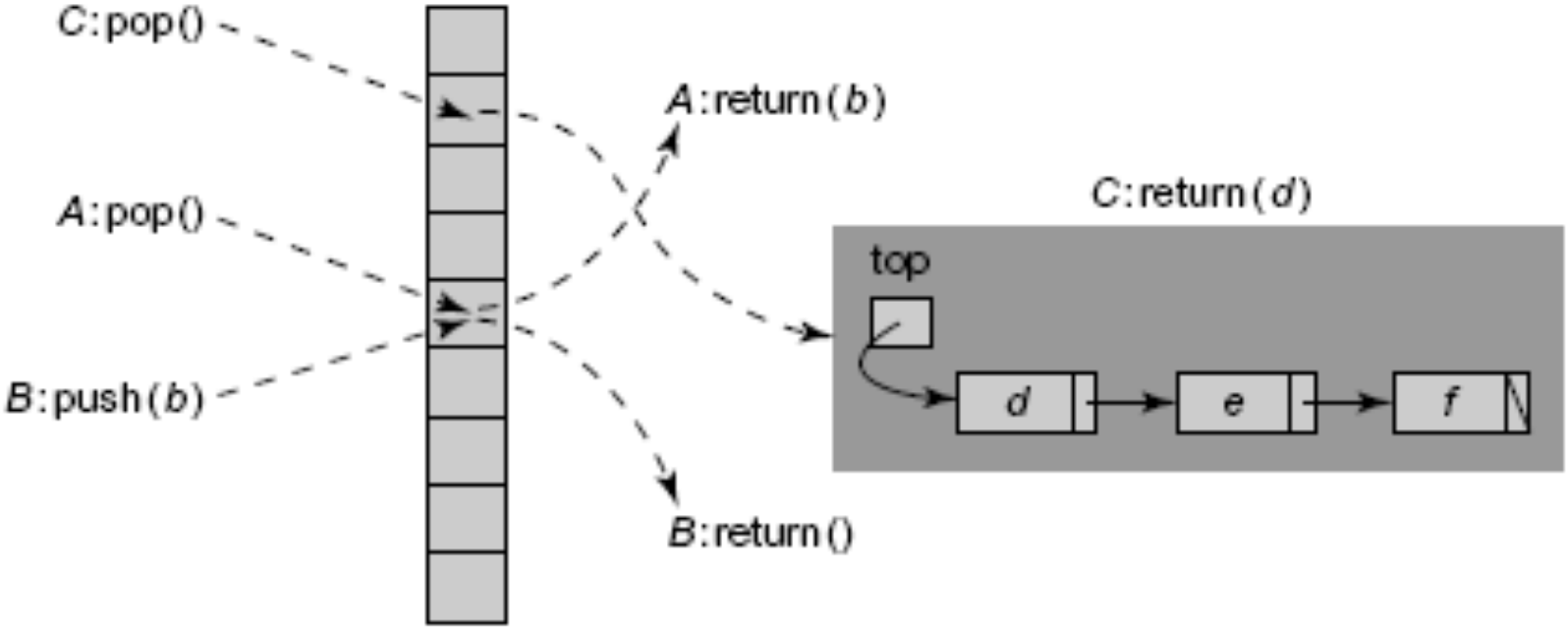
→ Zusammenlegen von Threads mit komplementären Aufrufen zu Paaren

Elimination Array

- Array, in dem Threads komplementäre Aufrufe suchen und eventuell finden
- push und pop Paar tauschen und geben Werte zurück ohne Stack zu verändern

  → Lock – Free Exchanger
- Bei keinem komplementären Threadaufruf → weitere Suche oder Stack betreten

Elimination Array



Elimination Array

```
public class EliminationArray<T>{
    private static final duration = ...;
    LockFreeExchanger<T>[] exchanger; //Anlegen mehrerer LockFreeExchanger
    Random random;

    public EliminationArray(int capacity){ //Array wird mit bestimmter Grösse initialisiert
        exchanger = (LockfreeExchanger<T>[]) new LockFreeExchanger[capacity];
        for(int i = 0; i < capacity; i++){
            exchanger[i] = new LockFreeExchanger<T>();
        }
        random = new Random();
    }

    public T visit(T value, int range) throws TimeoutException{
        int slot = random.nextInt(range);
        // Wenn passender komplementaerer Aufruf gefunden wurde --> Rückgabe des Wertes
        return (exchanger[slot].exchange(value, duration, TimeUnit.MILLISECONDS));
    }
}
```

Lock – Free Exchanger

- EliminationArray Einträge sind Lock – Free Exchanger Objekte
- Klasse, die den Austausch der Werte zweier komplementärer Threads durchführt
- Ankommender Thread sucht Exchanger und wartet auf Ankunft eines komplementären Threads → TimeoutException
- Drei Zustände: Empty, Waiting, Busy
- Bei Erfolg: Werte tauschen und zurückgegeben

Elimination Backoff Stack

- Kein Erfolg im Lock Free Stack: Eintritt ins Elimination Array

→Backoff wird durch Eintritt ins Elimination Array ersetzt

→Elimination Backoff Stack

Elimination Backoff Stack

- EliminationBackoffStack erbt von LockFreeStack
- Überschreiben von push(T value) und pop()
- Bei erfolglosem tryPop() oder tryPush(Node node) → anstatt Backoff: Eintritt ins Elimination Array

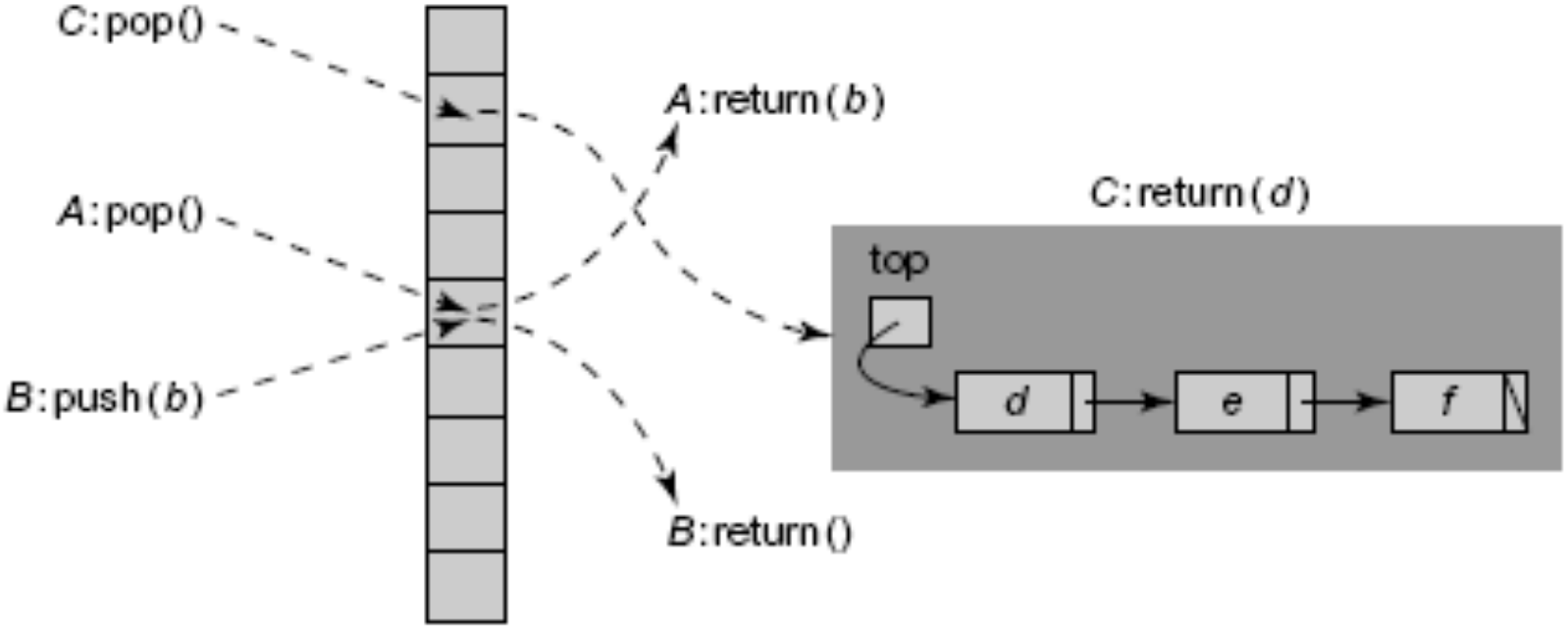
Elimination Backoff Stack

```
public class EliminationBackoffStack<T> extends LockFreeStack<T>{

    static final int capacity = ...;
    EliminationArray<T> eliminationArray = new EliminationArray<T>(capacity);

    static ThreadLocal<RangePolicy> policy = new ThreadLocal<RangePolicy>(){
        protected synchronized RangePolicy initialValue(){
            return new RangePolicy();
        }
    }

    public void push(T value){
        RangePolicy rangePolicy = policy.get();
        Node node = new Node(value);
        while(true){
            if(tryPush(node)){
                return;
            }else try{
                T otherValue = eliminationArray.visit(value, rangePolicy.getRange());
                if(otherValue == null){
                    rangePolicy.recordEliminationSuccess();
                    return; //exchanged with pop
                }
            }catch(TimeoutException ex){
                rangePolicy.recordEliminationTimeout();
            }
        }
    }
```

Elimination Backoff Stack

```
public void pop() throws EmptyException{
    RangePolicy rangePolicy = policy.get();
    while(true){
        Node returnNode = tryPop();
        if(returnNode != null){
            return returnNode.value;
        } else try{
            T otherValue = eliminationArray.visit(null, rangePolicy.getRange());
            if(otherValue != null){
                rangePolicy.recordEliminationSuccess();
                return otherValue;
            }
        } catch(TimeoutException ex){
            rangePolicy.recordEliminationTimeout();
        }
    }
}
```

Elimination Backoff Stack



Quellen

1. The art of multiprocessor programming,
   Maurice Herlihy & Nir Shavit
2. Skript Betriebsysteme WiSe 09/10,
   Prof. Dr. Claudia Linnhoff – Popien
   http://www.mobile.ifi.uni-muenchen.de/studium_lehre/verg_semester /ws0910/betriebssysteme/bsscript120202010.pdf