CARL VON OSSIETZKY universität OLDENBURG

Sommersemester 2020

Objektorientierte Modellierung und Programmierung

Dr. Christian Schönberg

CARL VON OSSIETZKY universität OLDENBURG

Java-Threads und Parallele Programmierung II

CARL VON OSSIETZKY universität OLDENBURG

Lernziele

- Warten und Benachrichtigen
- Locks
- Semaphoren

CARL VON OSSIETZKY universität OLDENBURG

Warten und Benachrichtigen

CARL VON OSSIETZKY universität OLDENBURG

`wait()` und `notify()`

- Methoden, die es einem Thread erlauben, so lange zu blockieren, bis eine bestimmte Bedingung erfüllt ist
  - Thread 1 benötigt Bedingung:
    ```
    while (!Bedingung) { obj.wait(); }
    ```
  - Thread 2 erfüllt die Bedingung:
    ```
    obj.notify();
    ```
- Aufruf nur innerhalb eines **`synchronized`**-Blocks, der auf **`obj`** synchronisiert
- **`obj.notify()`** setzt **nicht** die Ausführung von Thread 1 fort, sondern signalisiert dem Scheduler, dass *einer der* auf **`obj`** wartenden Threads *demnächst* aufgeweckt werden *sollte*

CARL VON OSSIETZKY universität OLDENBURG

wait() und notify(): Beispiel

```
public class BlockingList<T> {

    public static final int CAPACITY = 100;

    private List<T> list = new ArrayList<>(CAPACITY);

    public boolean add(T elem) {
        synchronized(list) {
            while (list.size() == CAPACITY) {
                try {
                    list.wait();
                } catch (InterruptedException e) {
                    return false;
                }
            }
            list.add(elem);
            return true;
        }
    }
```

CARL VON OSSIETZKY universität OLDENBURG

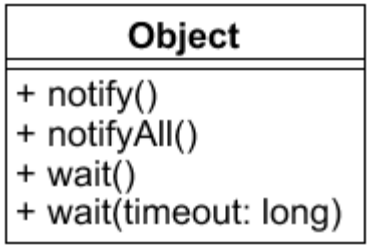
`wait()` und `notify()`: Beispiel

```
    public boolean remove(T elem) {
        synchronized(list) {
            if (list.remove(elem)) {
                list.notify(); // trigger waiting 'add'
                return true;
            }
            return false;
        }
    }

}
```

CARL VON OSSIETZKY universität OLDENBURG

`Object` Methoden



CARL VON OSSIETZKY universität OLDENBURG

`notify()` und `notifyAll()`

- **`obj.notify()`** weckt einen beliebigen auf **`obj`** wartenden Thread
  - Verwendung, wenn nur ein Thread sinnvoll weiterarbeiten kann
  - Beispiel: BlockingList
    - nach dem Löschen von *einem* Element kann *genau eine* blockierende **`add()`** Methode erfolgreich ausgeführt werden
- **`obj.notifyAll()`** weckt alle auf **`obj`** wartenden Threads
  - Verwendung, wenn alle Threads sinnvoll weiterarbeiten können
  - Beispiel: Lese-/Schreibzugriff
    - ein Thread *schreibt* Daten
    - danach können diese Daten von allen anderen Threads *gelesen* werden

CARL VON OSSIETZKY universität OLDENBURG

Producer/Consumer

- Ein Producer erzeugt stetig neue Objekte und schreibt sie in einen größenbeschränkten Puffer
- Ein Consumer entnimmt dem Puffer stetig Objekte und verarbeitet sie
- Puffer als größenbeschränkte Queue
  - Producer muss warten, wenn der Puffer voll ist
  - Consumer muss warten, wenn der Puffer leer ist

CARL VON OSSIETZKY universität OLDENBURG

Producer/Consumer: Beispiel

```
public class BlockingQueue<T> {
    public static final int CAPACITY = 100;
    private Queue<T> queue = new LinkedList<T>();

    public synchronized void enqueue(T element) throws InterruptedException {
        while (queue.size() == CAPACITY) {
            wait();
        }
        queue.add(element);
        notifyAll();
    }

    public synchronized T dequeue() throws InterruptedException {
        while (queue.isEmpty()) {
            wait();
        }
        T item = queue.remove();
        notifyAll();
        return item;
    } }
```

CARL VON OSSIETZKY universität OLDENBURG

Producer/Consumer: Beispiel

```
class Producer implements Runnable {
    private BlockingQueue<Integer> queue;

    public Producer(BlockingQueue<Integer> queue) {
        this.queue = queue;
    }

    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            try {
                queue.enqueue(i);
            } catch (InterruptedException e) {
                System.err.println("Producer was interrupted!");
                break;
            }
        }
    }
}
```

CARL VON OSSIETZKY universität OLDENBURG

Producer/Consumer: Beispiel

```
class Consumer implements Runnable {
    private BlockingQueue<Integer> queue;

    public Consumer(BlockingQueue<Integer> queue) {
        this.queue = queue;
    }

    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            try {
                int x = queue.dequeue();
                System.out.println(x);
            } catch (InterruptedException e) {
                System.err.println("Consumer was interrupted!");
                break;
            }
        }
    }
}
```

CARL VON OSSIETZKY universität OLDENBURG

Producer/Consumer: Beispiel

```
public static void main(String[] args) {
    BlockingQueue<Integer> queue = new BlockingQueue<>();
    Thread p1 = new Thread(new Producer(queue));
    Thread p2 = new Thread(new Producer(queue));
    Thread p3 = new Thread(new Producer(queue));
    Thread c1 = new Thread(new Consumer(queue));
    Thread c2 = new Thread(new Consumer(queue));
    Thread c3 = new Thread(new Consumer(queue));
    p1.start();
    p2.start();
    p3.start();
    c1.start();
    c2.start();
    c3.start();
}
```

CARL VON OSSIETZKY universität OLDENBURG

Weitere Konflikte

CARL VON OSSIETZKY universität OLDENBURG

Deadlock

- Thread 1 hat Ressource 1 reserviert und wartet auf Ressource 2
- Thread 2 hat Ressource 2 reserviert und wartet auf Ressource 1
  → Deadlock
- Deadlock führt zu einem Programmstillstand
  → neben dem gegenseitigen Überschreiben von Daten einer der häufigsten Programmierfehler bei Parallelität

CARL VON OSSIETZKY universität OLDENBURG

Deadlock: Beispiel

```
Object a = new Object(); Object b = new Object();
new Thread() {
    @Override
    public void run() {
        synchronized(a) {
            System.out.println("t1: lock on a");
            try { sleep(1000); } catch (InterruptedException e) { }
            synchronized(b) {
                System.out.println("t1: lock on b");
            } } }
}.start();
new Thread() {
    @Override
    public void run() {
        synchronized(b) {
            System.out.println("t2: lock on b");
            synchronized(a) {
                System.out.println("t2: lock on a");
            } } }
}.start();
```

CARL VON OSSIETZKY universität OLDENBURG

Lifelock

- Ähnlich wie ein Deadlock, aber
  - alle Threads sind aktiv
  - es passiert nur nichts sinnvolles
  - deutlich seltener
- Beispiel:
  - zwei Menschen begegnen sich auf dem Flur und versuchen sich auszuweichen
  - beide weichen in die eine Richtung aus
  - beide weichen in die andere Richtung aus
  - beide weichen in die eine Richtung aus
  - beide weichen in die andere Richtung aus
  - …

CARL VON OSSIETZKY universität OLDENBURG

Starvation

- Starvation bedeutet, dass ein Prozess nie vom Scheduler aufgerufen wird und somit niemals zum Zuge kommt
  - ein Prozess mit geringer Priorität wird ständig von Prozessen mit höherer Priorität ausgebootet
  - einem Prozess wird ständig Zugriff auf Ressourcen verweigert, weil die Freigabe fehlerhaft ist
    - Deadlock
    - **`notify()`** statt **`notifyAll()`**
    - kein **`notify()`**
    - **`notify()`** an der falschen Stelle
    - **`notify()`** wird aufgerufen, bevor im anderen Thread **`wait()`** aufgerufen wird

Blockierung

- Eine (absichtliche oder versehentliche) Blockierung passiert, wenn innerhalb einer Klasse Methoden als **`synchronized`** deklariert sind, und gleichzeitig von außerhalb auf eine Instanz der Klasse synchronisiert wird

```
public class SynchronizedList<T> {

    private List<T> list = new ArrayList<>();

    public synchronized void add(T elem) {
        list.add(elem);
    }
}
```

```
// equivalent
public void add(T elem) {
    synchronized(this) {
        list.add(elem);
    }
}
```

Deadlock

```
SynchronizedList<Integer> intList =
        new SynchronizedList<>();
new Thread(() -> {
    synchronized(intList) {
        while (intList.size() == 0) ;
    }
}).start();
try {
    Thread.sleep(1000);
} catch (InterruptedException e) { }
intList.add(1);
```

Blockierung (2)

- Blockierungen treten oft bei Datenstrukturen oder Java-Bibliotheken auf, deren Entwickler wenig Erfahrung mit Parallelität haben
- Lösung: Synchronisation auf ein dediziertes lock-Objekt

```
public class SynchronizedList<T> {

    private List<T> list = new ArrayList<>();
    private Object lock = new Object();

    public void add(T elem) {
        synchronized(lock) {
            list.add(elem);
        }
    }
}
```

```
SynchronizedList<Integer> intList =
        new SynchronizedList<>();
new Thread(() -> {
    synchronized(intList) {
        while (intList.size() == 0) ;
    }
}).start();
try {
    Thread.sleep(1000);
} catch (InterruptedException e) { }
intList.add(1);
```

CARL VON OSSIETZKY universität OLDENBURG

Kritische Abschnitte: Locks

CARL VON OSSIETZKY universität OLDENBURG

Locks

- Seit Java 1.5
- Flexibler als **`synchronized/wait/notify`**
- Interface mit verschiedenen Implementierungen
- Generelles Vorgehen:

```
Lock l = …
l.lock();
try {
    // kritischer Abschnitt
} finally {
    l.unlock();
}
```

- **`unlock()`** immer im **`finally`**-Block, damit die Ressourcen auch bei einer Exception freigegeben werden

CARL VON OSSIETZKY universität OLDENBURG

ReentrantLock

- Standard Lock-Implementierung
- Viele nützliche Methoden
  - `getOwner()`
  - `isLocked()`
  - `tryLock()`
  - `getQueueLength()`
- Optionale Fairness

CARL VON OSSIETZKY universität OLDENBURG

ReadWriteLock

- Erlaubt mehreren Threads lesenden Zugriff auf einen kritischen Abschnitt, solange kein anderer Thread schreibt
- Erlaubt einem Thread schreibenden Zugriff auf einen kritischen Abschnitt, solange kein anderer Thread liest
- Hält dazu zwei verschiedene Locks, eine für den Lesezugriff und eine für den Schreibzugriff
- Zugriff auf die Locks über
  - `Lock readLock()`
  - `Lock writeLock()`

CARL VON OSSIETZKY universität OLDENBURG

Condition

- Ersetzt **wait/notify**
- Gebunden an ein Lock
  - erzeugt durch die Lock-Methode **newCondition()**
- **Condition.await() → Object.wait()**
  - **Condition.awaitUntil(Date deadline)**
  - **Condition.awaitNanos(long timeout)**
- **Condition.signal() → Object.notify()**
- **Condition.signalAll() → Object.notifyAll()**

CARL VON OSSIETZKY universität OLDENBURG

Condition: Beispiel

```
public class LockingQueue<T> {
    public static final int CAPACITY = 5;

    private Lock lock = new ReentrantLock();
    private Condition notFull = lock.newCondition();
    private Condition notEmpty = lock.newCondition();

    private Queue<T> queue = new LinkedList<T>();

    public void enqueue(T element) throws InterruptedException {
        lock.lock();
        try {
            while (queue.size() == CAPACITY) {
                notFull.await();
            }
            queue.add(element);
            notEmpty.signal();
        } finally {
            lock.unlock();
        }
    }
```

CARL VON OSSIETZKY universität OLDENBURG

Condition: Beispiel

```
    public T dequeue() throws InterruptedException {
        lock.lock();
        try {
            while (queue.isEmpty()) {
                notEmpty.await();
            }
            T item = queue.remove();
            notFull.signal();
            return item;
        } finally {
            lock.unlock();
        }
    }
}
```

CARL VON OSSIETZKY universität OLDENBURG

Kritische Abschnitte: Semaphoren

Semaphore

- Eine Semaphore verwaltet Zugang zu einem kritischen Abschnitt und stellt dazu zwei atomare (nicht unterbrechbare) Operationen zur Verfügung
  - **`Semaphore(permits: int)`**: Erzeugt ein neues Semaphore-Objekt, das **`permits`** Prozessen gleichzeitig Zugriff auf einen kritischen Abschnitt gewährt (**`permits`** = 1 → vergleichbar mit Locks)
  - **`acquire()`**: Bittet um Erlaubnis (**`permit`**), auf einen kritischen Abschnitt zugreifen zu dürfen
    - setzt die Anzahl der verfügbaren **`permits`** um eins herab
    - blockiert, solange keine **`permits`** verfügbar sind
  - **`release()`**: Gibt nach Verlassen des kritischen Abschnitts ein **`permit`** zurück
    - setzt die Anzahl der verfügbaren **`permits`** um eins hoch
- Sie kann nicht nur einem Prozess sondern einer festgelegten Anzahl von Prozessen gleichzeitig Zugang gewähren (**`synchronized/wait/notify/Lock`** immer nur einem)

Semaphore: Beispiel

```
public class MoneyRunner implements Runnable {

    private static Semaphore semaphore = new Semaphore(1);
    …

    @Override
    public void run() {
        try {
            semaphore.acquire();
            int balance = account.getBalance();
            int total = balance + amount;
            System.out.println(balance + " + " + amount
                    + " = " + total);
            account.setBalance(total);
        } catch (InterruptedException e) {
            System.err.println("Could not deposit money!");
        } finally {
            semaphore.release();
        }
    }
…
```

kritischer Abschnitt

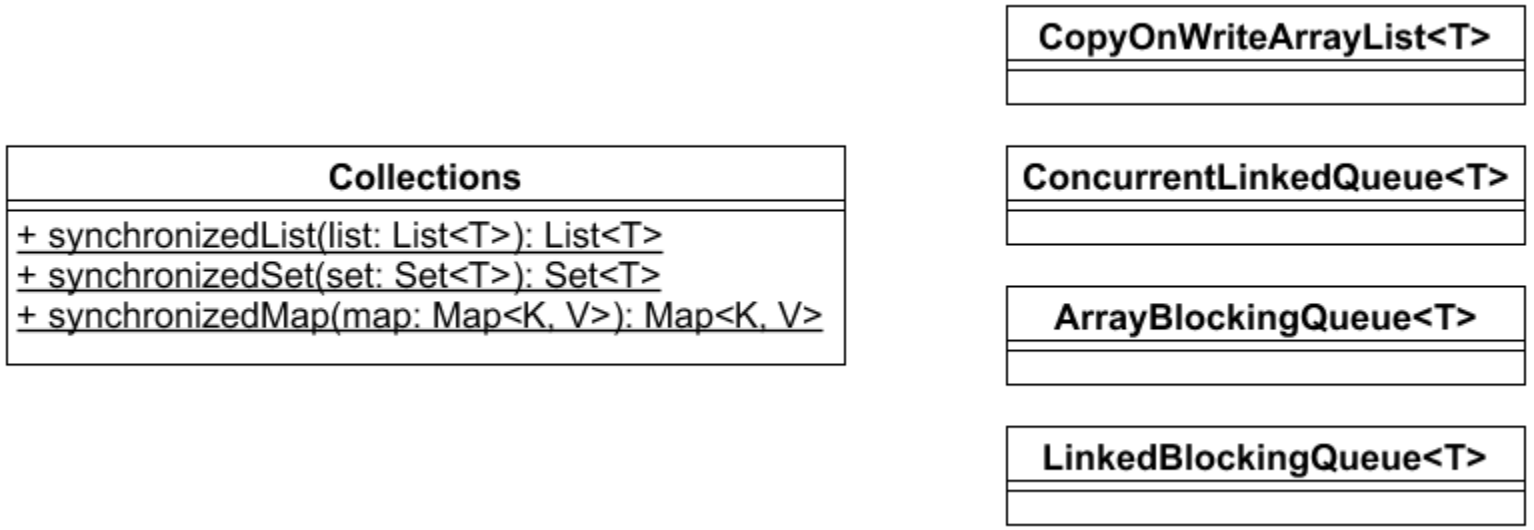
Semaphore: Beispiel

```
public static final int[] AMOUNTS = new int[] { 5, 10, 15, 20 };

public static void main(String[] args) {
    Account acc = new Account();
    Thread[] threads = new Thread[AMOUNTS.length];
    for (int i = 0; i < AMOUNTS.length; i++) {
        threads[i] = new Thread(new MoneyRunner(acc, AMOUNTS[i]));
    }
    for (int i = 0; i < AMOUNTS.length; i++) {
        threads[i].start();
    }
    try {
        for (int i = 0; i < AMOUNTS.length; i++) {
            threads[i].join();
        }
    } catch (InterruptedException e) {
        System.err.println("Thread was interrupted!");
    }
    System.out.println(acc.getBalance());
}
```

```
0 + 10 = 10
10 + 20 = 30
30 + 15 = 45
45 + 5 = 50
50
```

Thread-sichere Datenstrukturen



CARL VON OSSIETZKY universität OLDENBURG

Lernziele

- Warten und Benachrichtigen
- Locks
- Semaphoren