Ankündigungen 27.11.2008

- Werbung Teleseminar : Vorbesprechung Freitag 19.12.2008 13:00 in A5 C116
- Feedback-Bögen
- Terminplanung:
  - Donnerstag, 27.11.2008, 15:30: Nebenläufiges Programmieren in Java
  - Freitag 28.11.2008, **9:00**: Überraschung (Abschluss der Freitagsvorlesung)
  - Freitag 28.11.2008, 10:15: Übungsblatt 10 (Nebenläufiges Programmieren in Java)
  - Donnerstag, 4.12.2008: Abschlussvorlesung
  - Freitag, 5.12.2008, 10:15: Fragestunde
- Prüfungstermine:
  - Freitag, 19.12.2008, 14:00 Uhr in A5 B144
  - Basiskurs schreibt 66 Minuten (3 Anmeldungen)
  - Rest schreibt 100 Minuten (31 Anmeldungen)

Übersicht

- Einführung: Kritische Abschnitte
- Hardwaregestützte Mechanismen
- Betriebssystemgestützter Mechanismus: Semaphore
- Sprachgestützter Mechanismus: Monitore
- **Realisierungsbeispiele**
  - **Nebenläufigkeit und Synchronisation in Java**

Realisierungsbeispiel

- Java-Threads als Realisierungsbeispiel für sprachgestützte Prozessinteraktion
  - Erzeugung von Java-Threads
  - Thread-Synchronisation mit Monitoren
  - Komplexe Synchronisation mit `wait()` und `notify()`
  - Synchronisation mit expliziten Sperren und Conditions aus `java.util.concurrent.locks`
- Grundlage:
  - Stefan Middendorf, Reiner Singer, Jörn Heid: Java - Programmierhandbuch und Referenz für die Java2-Plattform, Standard Edition, dpunkt-Verlag, 3. Auflage 2002. Online: `http://www.dpunkt.de/java/`
  - Maurice Herlihy, Nir Shavit: The Art of Multiprocessor Programming. Morgan Kaufmann, 2008. Online-Materal: `http://books.elsevier.com/companions/9780123705914`

Java-Threads

- Java bietet User-Level-Threads als Teil seiner Standard-Klassenbibliothek an
- Threads können erzeugt werden als Unterklasse von `Thread` und durch Überschreiben der Methode `run()`
- Beispiel:

```
class DemoThread extends Thread {
   public void run() {
     for(int i = 0; i < 10; i++) {
       try {  sleep(5000); }
       catch(InterruptedException e) { ... }
       System.out.println("Demo-Thread");
     }
   }
}
```

Starten von Threads

- Um einen Thread zu starten muss man
  - ein Objekt der Thread-Klasse erzeugen und
  - die Methode `start()` des Threads aufrufen
    - Dabei wird die überschriebene Methode `run()` gestartet
- Scheduling von Threads ist nichtdeterministisch und preemptiv
  - Threads können beliebig unterbrochen und fortgesetzt werden
- Beispiel:

```
public class ThreadTest {
  public static void main(String args[]) {
    DemoThread demoThread;
    demoThread = new DemoThread();
    demoThread.start();
  }
}
```

Komplexeres Beispiel (1/2)

```
class TextThread extends Thread {
    String text;

    public TextThread(String text) {
      this.text = text;
    }
    public void run() {
      for(int i = 0; i < 10; i++) {
        try {
          sleep((int)(Math.random()*1000));
        }
        catch(InterruptedException e) {
        }
       System.out.println(text);
      }
    }
  }
```

Komplexeres Beispiel (2/2)

```
public class TextThreadDemo {

  public static void main(String args[]) {
    TextThread java, espresso, capuccino;

    java = new TextThread("Java");
    espresso = new TextThread("Espresso");
    capuccino = new TextThread("Cappuccino");
    java.start();
    espresso.start();
    capuccino.start();
  }

}
```

- Demo ...

Definition über Anonyme Innere Klassen

- Threads sind oft Einwegklassen (Wegwerfklassen); werden ein Mal definiert und benutzt
  - Benötigen keinen Namen
- Sparsamere Definition:

```
...
Thread thread = new Thread(new Runnable() {
        public void run() {
          System.out.println("Hello World!");
        }
      });
    }
thread.start();
...
```

Warten auf Threads

- Mittels Methode `start()` wird Thread gestartet
- Mittels Methode `join()` kann man auf das Ende eines Threads warten
- Beispiel:

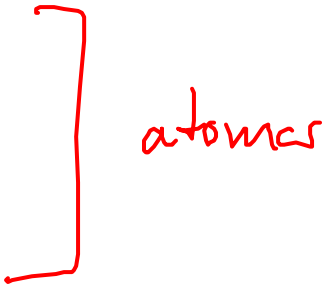
```
...
Thread[] thread = new Thread[8];
for (int i = 0; i < thread.length; i++) {
  final String message = "Hello world from thread " + i;
  thread[i] = new Thread(new Runnable() {
    public void run() { System.out.println(message); }
  });
}
for (int i = 0; i < thread.length; i++) {
  thread[i].start();
}
for (int i = 0; i < thread.length; i++) {
  thread[i].join();
}
System.out.println("done!");
...
```

Synchronisation: Monitore

- In Java wurde das Monitorkonzept zur Synchronisation von Threads umgesetzt
  - Ein Objekt wird zum Monitor, wenn eine seiner Methoden als `synchronized` deklariert wurde
- Beispiel: globales Sperrflag

```
public class Sperre {
  static int Sperrflag = 1;
  public Sperre() { } // Konstruktor

  synchronized public int Lock {
    int tmp = Sperrflag;
    Sperrflag = 0;
    return tmp;
  }
}
```



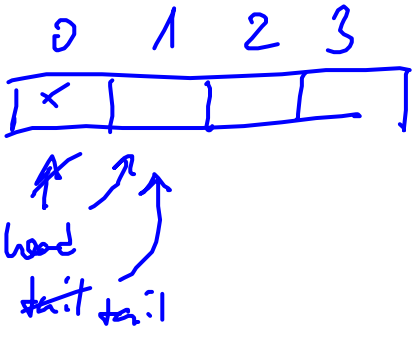
`wait()` und `notify()`

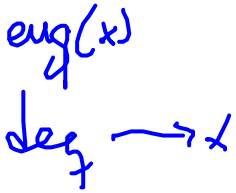
- Java bietet jedem Objekt die Methoden `wait()` und `notify()` an
  - Sie dürfen nur in Methoden aufgerufen werden, die `synchronized` sind (d.h. zu einem Monitor gehören)
  - `wait()` überführt den aktuellen Thread in den blockiert-Zustand und gibt den Monitor frei
  - `notify()` deblockiert einen mit `wait()` blockierten Thread
  - es gibt auch noch `notifyAll()`
- Beispiel: Erzeuger/Verbraucher-Problem in Java
  - Zwei Threads befüllen und entleeren einen gemeinsamen Puffer
- Standardmäßig nur eine namenlose “default” Condition Variable pro Monitor

Erzeuger/Verbraucher (mit Fehlern)

```
public class IncorrectStringQueue {
  final static int QSIZE = 100; // arbitrary size
  int head = 0;                 // next item to dequeue
  int tail = 0;                 // next empty slot
  String[] items = new String[QSIZE];

  public String deq() {
    return items[(head++) % QSIZE];
  }
  public void enq(String x) {
    items[(tail++) % QSIZE] = x;
  }
}
```





Erzeuger/Verbraucher (Variante 2)

```
public class IncorrectStringQueue {
  final static int QSIZE = 100; // arbitrary size
  int head = 0;                 // next item to dequeue
  int tail = 0;                 // next empty slot
  String[] items = new String[QSIZE];

  public synchronized String deq() {
    return items[(head++) % QSIZE];
  }
  public synchronized void enq(String x) {
    items[(tail++) % QSIZE] = x;
  }
}
```

Erzeuger/Verbraucher (Variante 3)

```
public class IncorrectStringQueue {
  final static int QSIZE = 100; // arbitrary size
  int head = 0;                 // next item to dequeue
  int tail = 0;                 // next empty slot
  String[] items = new String[QSIZE];

  public synchronized String deq() {
    while (head == tail) { wait(); }
    return items[(head++) % QSIZE];
  }
  public synchronized void enq(String x) {
    items[(tail++) % QSIZE] = x;
    notify();
  }
}
```

ist der Puffer leer?
Ja → wait

Erzeuger/Verbraucher (Variante 4)

```
public class IncorrectStringQueue {
  final static int QSIZE = 100; // arbitrary size
  int head = 0;                 // next item to dequeue
  int tail = 0;                 // next empty slot
  String[] items = new String[QSIZE];

  public synchronized String deq() {
    while (head == tail) { wait(); }
    return items[(head++) % QSIZE];
    notify();
  }
  public synchronized void enq(String x) {
    while (head == (tail - 1) % QSIZE) { wait(); }
    items[(tail++) % QSIZE] = x;
    notify();
  }
}
```

ist Puffer voll?

Explizite Sperren

- Java bietet im Package `java.util.concurrent.locks` explizite Sperren und Condition-Variablen
- Machen das eingebaute Monitorkonzept flexibler

Monitore und Sperren

- Monitore in Java haben eine implizite Sperre
- Man kann Monitore auch mit expliziten Sperren bauen
  - Klasse `Lock` (bzw. `ReentrantLock`)

Method Summary	
void	`lock()` Acquires the lock.
void	`lockInterruptibly()` Acquires the lock unless the current thread is `interrupted`.
`Condition`	`newCondition()` Returns a new `Condition` instance that is bound to this `Lock` instance.
boolean	`tryLock()` Acquires the lock only if it is free at the time of invocation.
boolean	`tryLock(long time, TimeUnit unit)` Acquires the lock if it is free within the given waiting time and the current thread has not been `interrupted`.
void	`unlock()` Releases the lock.

Monitor Ein-/Austritt

- Erzeugen einer Sperre: `Lock l = new Lock();`
- Erfolgreiches Locking (`l.lock();`) ist wie ein `Enter_Monitor`
- Rückgabe des Locks (`l.unlock();`) ist wie `Exit_Monitor`
- Rückgabe des Locks darf nie vergessen werden. Sinnvolles Programmiermuster:

```
Lock l = ...;
l.lock();
try {
  // access the resource protected by this lock
}
finally {
  l.unlock();
}
```

Condition Variablen

- Mit `newCondition()` kann man sich zu einem Lock eine Condition Variable besorgen
- Interface `Condition`:

Method Summary	
void	`await()` Causes the current thread to wait until it is signalled or `interrupted`.
boolean	`await(long time, TimeUnit unit)` Causes the current thread to wait until it is signalled or interrupted, or the specified waiting time elapses.
long	`awaitNanos(long nanosTimeout)` Causes the current thread to wait until it is signalled or interrupted, or the specified waiting time elapses.
void	`awaitUninterruptibly()` Causes the current thread to wait until it is signalled.
boolean	`awaitUntil(Date deadline)` Causes the current thread to wait until it is signalled or interrupted, or the specified deadline elapses.
void	`signal()` Wakes up one waiting thread.
void	`signalAll()` Wakes up all waiting threads.

Erzeuger/Verbraucher mit Locks

```
class Queue<T> {
  final Lock lock = new ReentrantLock();
  final Condition notFull  = lock.newCondition();
  final Condition notEmpty = lock.newCondition();
  final T[] items;
  int tail, head, count;
  public Queue(int capacity) {
    items = (T[])new Object[capacity];
  }
  public void enq(T x) throws InterruptedException {
    lock.lock();
    try {
      while (count == items.length) notFull.await();
      items[tail] = x;
      if (++tail == items.length) tail = 0;
      ++count;
      notEmpty.signal();
    } finally {
      lock.unlock();
    }
  }
```

```
public T deq() throws InterruptedException
   {
    lock.lock();
    try {
      while (count == 0)
        notEmpty.await();
      T x = items[head];
      if (++head == items.length)
        head = 0;
      --count;
      notFull.signal();
      return x;
    } finally {
      lock.unlock();
    }
 }
```

Zusammenfassung

- Java bietet eine mächtige Thread-Abstraktion
- Implementiert Monitore als Synchronisationsmechanismus
  - Mit oder ohne expliziten Locks
- Viele schöne Übungsaufgaben möglich, bei denen man Nebenläufigkeit selbst ausprobieren kann

Ausblick

- Gliederung der Vorlesung:
  1. Einführung und Formalia
  2. Auf was baut die Systemsoftware auf?
     Hardware-Grundlagen
  3. Was wollen wir eigentlich haben?
     Laufzeitunterstützung aus Anwendersicht
  4. Verwaltung von Speicher: Virtueller Speicher
  5. Verwaltung von Rechenzeit: Virtuelle Prozessoren (Threads)
  6. Synchronisation paralleler Aktivitäten auf dem Rechner
  7. Implementierungsaspekte