Thread

2 manieren

- extends Thread
  - override run method

```
public class Print1 extends Thread {
   public void run() {
      for (int i = 0; i < 10; i++) {
         System.out.println("print1");
         try { sleep(10); }
         catch (InterruptedException e) {}
      }
}
```

- implements Runnable
  - override run method

```
public class Print2 implements Runnable {
   public void run() {
      for (int i = 0; i < 10; i++) {
         System.out.println("print2");
         try { Thread.currentThread().sleep(20); }
         catch (InterruptedException e) {}
      }
}
```

THRijswijk

Thread

```
public class TThread {
   public static void main (String[] args) {
      Thread t1=new Print1();
      Thread t2=new Thread(new Print2());
      t1.start();
      t2.start();
   }
}
```

```
print1
print2
print1
print1
print2
print1
print1
print2
print1
print1
print2
print1
print1
print2
print2
print2
print2
print2
```

THRijswijk

Thread

```
public class PrintN implements Runnable {
   public PrintNrunnable(int nn) {
      n=nn;
   }
   public void run() {
      for (int i = 0; i < 10; i++) {
         System.out.println("print"+n);
         try {
            Thread.currentThread().sleep(n*10);
         } catch (InterruptedException e) {}
      }
   }
   private int n;
}
```

Alternatieve implementatie

```
public class TThread {
   public static void main (String[] args) {
      Thread t1=new Thread(new PrintN(1));
      Thread t2=new Thread(new PrintN(2));
      t1.start();
      t2.start();
   }
}
```

THRijswijk

Thread

prioriteiten

- void **setPriority**(int newPriority)
- int getPriority()
- Thread.MAX_PRIORITY (=10)
  Thread.MIN_PRIORITY(=1)

Thread scheduling is preemptive. Maar er wordt **niet** gegarandeerd dat de ready thread met de hoogste prio altijd running is!

THRijswijk

Thread

Synchronisatie

- Java gebruikt monitors.
- Elk object heeft een lock.
- Object locked/unlocked door:
  - methode modifier **synchronized**.
  - block synchronisatie.
- Synchronized methoden hebben mutual exclusive toegang tot het object.
- ”Gewone” methoden hebben concurrent toegang tot object.
- Lock werkt re-entrant. Als een synchronized method een andere synchronised method van hetzelfde object aanroept ontstaan geen problemen!

THRijswijk

synchronized methods

```
public class Point {
   private int x=0;
   private int y=0;
//  ...
   public synchronized void stepNorthEast() {
      x++;
      y++;
   }
   public synchronized boolean isNorthEast() {
      return x==y;
   }
}
```

THRijswijk

TPoint

```
public class TPoint {
   private Point p;
   private class T1 implements Runnable {
      public void run() {
         for (int i = 0; i < 100000000; i++)
            p.stepNorthEast();
      }
   }
   private class T2 implements Runnable {
      public void run() {
         for (int i = 0; i < 100000000; i++)
            if (!p.isNorthEast())
               System.out.println("Probleem!");
      }
   }
   private void doTest() {
      p=new Point();
      Thread t1=new Thread(new T1());
      Thread t2=new Thread(new T2());
      t1.start(); t2.start();
   }
   public static void main(String args[]) {
      TPoint t=new TPoint(); t.doTest();
   }
}
```

THRijswijk

TPoint

Testresultaten

- Zonder synchronize:
  - executietijd ongeveer 10 sec
  - ongeveer 300 problemen!
- Met synchronize:
  - executietijd ongeveer 250 sec!
  - geen problemen.
- Testmachine
  - Pentium III 500MHz
  - 128 MByte RAM
  - Win98
  - Java 2 SE version 1.4.1

THRijswijk

synchronized blocks

- Je kunt ook een **deel** van een method mutual exclusive maken met een synchronized block.
- Het object dat “locked/unlocked” wordt geef je als parameter mee.
- Dit kan elk object zijn!
  - meer mogelijkheden dan monitor.
  - minder overzichtelijk. Object kan **overal** “locked/unlocked” worden!

```
public class Point {
//  ...
   public double timeConsumingCalculation() {
      int x1; int y1;
      synchronized (this) {
         x1=x;  int y1=y;
      }
//  ... use x1 and y1 to calculate res
      return res;
   }
}
```

THRijswijk

static

synchronisatie

- **static** fields worden gedeeld door **alle** objecten van een class.
- Mutual exclusive betekent in dit geval dat **alle** objecten "locked/ unlocked" moeten worden.
- Omdat een class in java ook een object (van de class Class) is, heeft ook elke class een lock.
- Deze class-lock wordt gebruikt bij **static synchronized** methods.
- Je kunt deze class-lock ook in een synchronised block gebruiken

```
synchronized (obj.getClass()) {
   // mutual exclusive voor alle objecten van
   // dezelfde class als obj
}
```

THRijswijk

conditionele synchronisatie

wait en notify

- De class Object (stamvader) heeft de volgende methoden:
  - public void **wait**();
  - public void **notify**();
  - public void **notifyAll**();
- Deze methoden mogen alleen aangeroepen worden als de current thread het object heeft ge-"locked". Anders wordt de exception IllegalMonitorStateException gegooid.

THRijswijk

conditionele synchronisatie

wait en notify

- Thread die **wait** aanroept wordt "waiting" en "unlocked" de lock.
- **wait** in een geneste monitor unlocked alleen de binnenste lock.
- **notify** maakt 1 waiting thread wakker (welke is onbepaald!)
- Wakker gemaakte thread moet nog wachten op de lock!
- **notifyAll** maakt alle waiting threads wakker.
- Als er geen "waiting" threads zijn hebben notify en notifyAll geen effect!
- Een "waiting" thread kan ook interrupted worden: gooit InterruptedException
- Er zijn **geen** expliciete condition variables.

THRijswijk

Voorbeeld

```
public class Teller {
   private int teller=0;
   public synchronized void verwerkTienTallen()
         throws InterruptedException {
      while (teller<10)
         wait();
      teller-=10;
      System.out.println(
         "teller is verlaagd en is nu "+teller);
   }
   public synchronized void verhoog(int n) {
      teller+=n;
      notify();
   }
}
```

Vergelijk met POSIX oplossing uit les 3.

THRijswijk

Voorbeeld

```
public class TTeller {
   private Teller tel;
   private class T1 implements Runnable {
      public void run() {
         while (true) {
            try {
               tel.verwerkTienTallen();
            } catch (InterruptedException e) { }
   } } }
   private class T2 implements Runnable {
      public void run() {
         BufferedReader din=
            new BufferedReader(
               new InputStreamReader(System.in));
         System.out.print("Geef waarden voor n: ");
         while (true) {
            int n=0;
            try {
               n=new Integer(din.readLine()).intValue();
            } catch (IOException e) { }
            tel.verhoog(n);
   } } }
   private void doTest() {
      tel=new Teller();
      Thread t1=new Thread(new Thread1());
      Thread t2=new Thread(new Thread2());
      t1.start();   t2.start();
   }
}
```

THRijswijk

Voorbeeld

```
Geef waarden voor n: 2
3
4
5
teller is verlaagd en is nu 4
6
teller is verlaagd en is nu 0
123
teller is verlaagd en is nu 113
teller is verlaagd en is nu 103
teller is verlaagd en is nu 93
teller is verlaagd en is nu 83
teller is verlaagd en is nu 73
teller is verlaagd en is nu 63
teller is verlaagd en is nu 53
teller is verlaagd en is nu 43
teller is verlaagd en is nu 33
teller is verlaagd en is nu 23
teller is verlaagd en is nu 13
teller is verlaagd en is nu 3
8
teller is verlaagd en is nu 1
```

THRijswijk

Readers en Writers

Voorbeeld

- Meerdere concurrent readers toegestaan
- Writer heeft alleenrecht nodig

```
ReadersWritersLock rwLock=
         new ReadersWritersLock()
```

Readers:

```
rwLock.startRead();
//
// read ...
//
rwLock.stopRead();
```

Writers:

```
rwLock.startWrite();
//
// write ...
//
rwLock.stopWrite();
```

THRijswijk

Oplossing 1

```
public class ReadersWritersLock {
   private int readers=0;
   private int waitingWriters=0;
   private boolean writing=false;
   public synchronized void startWrite()
         throws InterruptedException {
      while (readers>0 || writing) {
         waitingWriters++; wait(); waitingWriters--;
      }
      writing=true;
   }
   public synchronized void stopWrite() {
      writing=false; notifyAll();
   }
   public synchronized void startRead()
         throws InterruptedException {
      while (writing || waitingWriters>0) wait();
      readers++;
   }
   public synchronized void stopRead() {
      readers--;
      if (readers==0) notifyAll();
   }
}
```

THRijswijk

Oplossing 1

Nadeel

- Inefficient omdat altijd alle threads wakker gemaakt worden!
- Eigenlijk wil je 2 conditionele variabelen gebruiken:
  - okToRead:
    - readers wachten hierop (in startRead) als een writer actief is. Of als er wachtende writers zijn.
    - alle readers worden wakker gemaakt in stopWrite als er geen wachtende writers meer zijn.
  - okToWrite:
    - writers wachten hierop (in startWrite) als andere readers actief zijn of als een andere writer actief is.
    - 1 writer wordt wakker gemaakt in:
      - stopWrite als er wachtende writers zijn.
      - stopRead als er geen actieve readers meer zijn en als er wachtende writers zijn.

THRijswijk

Oplossing 2

```
public class ReadersWritersLock {
   private class ConditionVariable {
      public boolean wantToSleep=false;
   }
   private int readers=0;
   private int waitingReaders=0;
   private int waitingWriters=0;
   private boolean writing=false;
   private ConditionVariable okToRead=
      new ConditionVariable();
   private ConditionVariable okToWrite=
      new ConditionVariable();
   public void startWrite()
         throws InterruptedException { /* ... */ }
   public void stopWrite() { /* ... */ }
   public void startRead()
         throws InterruptedException { /* ... */ }
   public void stopRead() { /* ... */ }
}
```


Oplossing 2

```
public void startWrite() throws
      InterruptedException {
   synchronized (okToWrite) {
      synchronized (this) {
         if (writing || readers>0) {
            waitingWriters++;
            okToWrite.wantToSleep=true;
         } else  {
            writing=true;
            okToWrite.wantToSleep=false;
      } }
      if (okToWrite.wantToSleep)
         okToWrite.wait();
} }
public void stopWrite() {
   synchronized (okToRead) {
      synchronized (okToWrite) {
         synchronized (this) {
            if (waitingWriters>0) {
               waitingWriters--;
               okToWrite.notify();
            } else {
               writing=false;
               readers=waitingReaders;
               waitingReaders=0;
               okToRead.notifyAll();
} } } }
```


Oplossing 2

```
public void startRead()
      throws InterruptedException {
   synchronized (okToRead) {
      synchronized (this) {
         if (writing || waitingWriters>0) {
            waitingReaders++;
            okToRead.wantToSleep=true;
         } else {
            readers++;
            okToRead.wantToSleep=false;
      } }
      if (okToRead.wantToSleep)
         okToRead.wait();
} }
public void stopRead() {
   synchronized (okToWrite) {
      synchronized (this) {
         readers--;
         if (readers==0 && waitingWriters>0) {
            waitingWriters--;
            writing=true;
            okToWrite.notify(); // wakeup one writer
} } } }
```

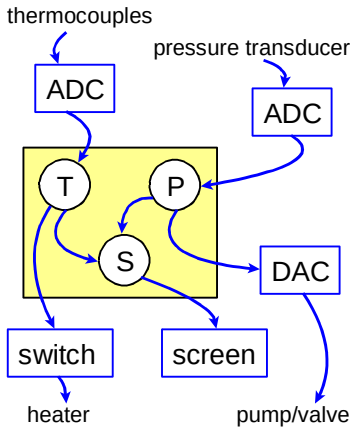
// Important for all methods to use the same order
// of locking otherwise **deadlock** will occur

Readers en Writers

Voorbeeld

- Posix heeft standaard **pthread_rwlock**
  - pthread_rwlock_init
  - pthread_rwlock_destroy
  - pthread_rwlock_rdlock
  - pthread_rwlock_wrlock
  - pthread_rwlock_unlock
- Zie blz. 63 van het QNX boek.
- Java neemt waarschijnlijk ook dergelijk classes op in versie 1.5
  http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html

Voorbeeld



Sequentieel

```
#include <stdio.h>
#include <stdlib.h>

double readTemp(void);
void writeSwitch(int i);
double readPres(void);
void writeDAC(double d);
int tempConvert(double temp);
double presConvert(double pres);

int main(void) {
   double temp, pres, dac;
   int switch_;
   while (1) {
      temp=readTemp();
      switch_=tempConvert(temp);
      writeSwitch(switch_);
      pres=readPres();
      dac=presConvert(pres);
      writeDAC(dac);
      printf("%4.1lf, %4.1lf, %d, %5.1lf\n",
         temp, pres, switch_, dac);
   }
   return EXIT_SUCCESS;
}
```


Sequentieel

problemen

- Sample rate van temperatuur en druk is gelijk.
  - kan wel wat aan worden gedaan met tellers maar: Wat doe je als pressureConcert langer duurt dan gewenste sample rate van temperatuur?
- Als readTemperature niet werkt (blijft pollen) dan loopt ook de drukregeling vast.

De temperatuurregeling en de drukregeling zijn twee afzonderlijke “processen”. Maar in het sequentiële programma zitten ze verweven!

Processen

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void) {
   pid_t pid;
   pid=fork();
   if (pid==-1) {
      perror("fork");
      return EXIT_FAILURE;
   }
   if (pid==0) { // this is the child
      double temp;
      int switch_;
      while (1) {
         temp=readTemp();
         printf("temperature = %4.1lf, ", temp);
         switch_=tempConvert(temp);
         printf("switch = %d\n", switch_);
         writeSwitch(switch_);
         sleep(3);
      }
   }
```

Zie volgende sheet

Processen

```
   else { // this is the parent
      double pres, dac;
      while (1) {
         pres=readPres();
         printf("pressure = %4.1lf", pres);
         dac=presConvert(pres);
         printf(", DAC = %5.1lf\n", dac);
         writeDAC(dac);
         sleep(1);
      }
      wait(0);
   }
   return EXIT_SUCCESS;
}
```

```
temperature =  1.0, switch = 0
pressure = 18.4, DAC =  -8.4
pressure = 18.0, DAC =  -8.0
pressure = 17.6, DAC =  -7.6
temperature =  1.5, switch = 0
pressure = 17.2, DAC =  -7.2
pressure = 16.8, DAC =  -6.8
pressure = 16.4, DAC =  -6.4
```

Threads

```
void* tempThread(void* p) {
   double temp;
   int switch_;
   while (1) {
      temp=readTemp();
      printf("temperature = %4.1lf, ", temp);
      switch_=tempConvert(temp);
      printf("switch = %d\n", switch_);
      writeSwitch(switch_);
      sleep(3);
   }
   return NULL;
}
void* presThread(void* p) {
   double pres, dac;
   while (1) {
      pres=readPres();
      printf("pressure = %4.1lf", pres);
      dac=presConvert(pres);
      printf(", DAC = %5.1lf\n", dac);
      writeDAC(dac);
      sleep(1);
   }
   return NULL;
}
```

Zie volgende sheet

THRijswijk

108

Threads

```
#include <pthread.h>

void check(int error) {
   if (error!=0) {
      fprintf(stderr, "Error: %s\n", strerror(error));
      exit(EXIT_FAILURE);
   }
}

int main(void) {
   pthread_t p1, p2;
   check( pthread_create(&p1, NULL,
      tempThread, NULL) );
   check( pthread_create(&p2, NULL,
      presThread, NULL) );
   check( pthread_join(p1, NULL) );
   check( pthread_join(p2, NULL) );
   return EXIT_SUCCESS;
}
```

```
pressure = 14.4, DAC =  -4.4
pressure = 14.0temperature =  3.0, , DAC =  -4.0
switch = 0
pressure = 13.6, DAC =  -3.6
```

THRijswijk

109

Synchronize

```
#include <pthread.h>

pthread_mutex_t mutex =
   PTHREAD_MUTEX_INITIALIZER;

void* tempThread(void* p) {
   double temp;
   int switch_;
   while (1) {
      temp=readTemp();
      check( pthread_mutex_lock(&mutex) );
      printf("temperature = %4.1lf, ", temp);
      switch_=tempConvert(temp);
      printf("switch = %d\n", switch_);
      check( pthread_mutex_unlock(&mutex) );
      writeSwitch(switch_);
      sleep(3);
   }
   return NULL;
}
```

THRijswijk

110

Threads

```
// file: TempThread.java
public class TempThread extends Thread {
   private double temp;
   private boolean switch_;
   public TempThread() {
      temp=0.0;
   }
   private void readTemp() { /* ... */ }
   private void writeSwitch() { /* ... */ }
   private void tempConvert() { /* ... */ }
   public void run() {
      while (true) {
         readTemp();
         System.out.print("temperature = " +
            temp + ", ");
         tempConvert();
         System.out.println("switch = "+switch_);
         writeSwitch();
         try {
            sleep(3000);
         } catch (InterruptedException e) {}
      }
   }
}
```

Zie volgende sheet

THRijswijk

111

Threads

```
// file: TwoThreadsDemo.java
public class TwoThreadsDemo {
   public static void main (String[] args) {
      Thread t1=new TempThread();
      Thread t2=new PresThread();
      t1.start();
      t2.start();
   }
}
```

```
pressure = 14.4, DAC = -4.4
pressure = 14.0temperature = 3.0, , DAC = -4.0
switch = false
pressure = 13.6, DAC = -3.6
pressure = 13.2, DAC = -3.2
temperature = 3.5, switch = false
pressure = 12.8, DAC = -2.8
pressure = 12.4, DAC = -2.4
pressure = 12.0, DAC = -2.0
temperature = 4.0, switch = false
pressure = 11.6, DAC = -1.4
```

THRijswijk

112

Synchronize

```
// file: TempThread.java
public class TempThread extends Thread {
   private double temp;
   private boolean switch_;
   public TempThread() { temp=0.0; }
   private void readTemp() { /* ... */ }
   private void writeSwitch() { /* ... */ }
   private void tempConvert() { /* ... */ }
   public void run() {
      while (true) {
         readTemp();
         synchronized(System.out) {
            System.out.print("temperature = " +
               nf.format(temp) + ", ");
            tempConvert();
            System.out.println("switch = " +
               switch_);
         }
         writeSwitch();
         try {
            sleep(3000);
         } catch (InterruptedException e) {}
// ...
```

THRijswijk

113