Parallel Programming

Prof. Dr. Uwe Kastens

Winter 2014 / 2015

Objectives

The participants are taught to understand and to apply

- **fundamental concepts** and **high-level paradigms** of parallel programs,
- **systematic methods** for developing parallel programs,
- **techniques** typical for parallel programming in Java;
- English language in a lecture.

Exercises:

- The exercises will be tightly integrated with the lectures.
- Small teams will solve given assignments practically supported by a lecturer.
- Homework assignments will be solved by those teams.

Contents

Prerequisites

Topic	Course that teaches it
practical experience in programming Java	Grundlagen der Programmierung 1, 2
foundations in parallel programming	Grundlagen der Programmierung 2, Konzepte und Methoden der Systemsoftware (KMS)
process, concurrency, parallelism,	KMS
interleaved execution	KMS
address spaces, threads, process states	KMS
monitor	KMS
process, concurrency, parallelism,	GP, KMS
threads,	GP, KMS
synchronization, monitors in Java	GP, KMS
verfication of properties of programs	Modellierung

Organization of the course

Lecturer

Prof. Dr. Uwe Kastens:

Office hours: on appointment by email

Teaching Assistant:

- Peter Pfahler

Tutorials

- Grp 1 Mon 09.30 - 11.00 Even Weeks, F2.211 / F1 pool, Start Oct. 27
- Grp 2 Fri 11.00 - 12.30 Odd Weeks, F2.211 / F1 pool, Start Oct. 24

Schedule

Tutorial	Group 1, Mon 09:30	Group 2, Fri 11:00
1	Oct 27	Oct 24
2	Nov 10	Nov 07
3	Nov 24	Nov 21
4	Dec 08	Dec 05
5	Jan 05	Dec 19
6	Jan 19	Jan 16
7	Feb 02	Jan 30

Lecture

- V2 Mon 11:15 - 12:45, F1.110

Start date: Oct 13, 2014

Examination

Oral examinations of 20 to 30 min duration. For students of the Computer Science Masters Program the examination is part of a module examination, see Registering for Examinations
In general the examination is held in English. As an alternative, the candidates may choose to give a short presentation in English at the begin of the exam; then the remainder of the exam is held in German. In this case the candidate has to ask via email for a topic of that presentation latest a week before the exam.

Literature

Course material „**Parallel Programming**“
http://ag-kastens.upb.de/lehre/material/ppje

Course material „Grundlagen der Programmierung“ (in German)
Course material „**Software-Entwicklung I + II**“ WS, SS 1998/1999:(in German)
http://ag-kastens.upb.de/lehre/material/swei
Course material „**Konzepte und Methoden der Systemsoftware**“ (in German)
Course material „**Modellierung**“ (in German)
http://ag-kastens.upb.de/lehre/material/model

Gregory R. Andrews: **Concurrent Programming**, Addison-Wesley, 1991

Gregory R. Andrews: **Foundations of multithreaded, parallel, and distributed programming,** Addison-Wesley, 2000

David Gries: **The Science of Programming**, Springer-Verlag, 1981

Scott Oaks, Henry Wong: **Java Threads**, 2nd ed., O'Reilly, 1999

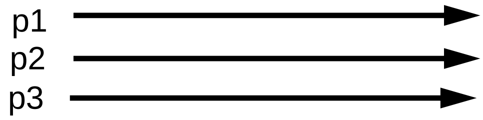
Jim Farley: **Java Distributed Computing**, O'Reilly, 1998

Doug Lea: **Concurrent Programming in Java**, Addison-Wesley, 2nd Ed., 2000

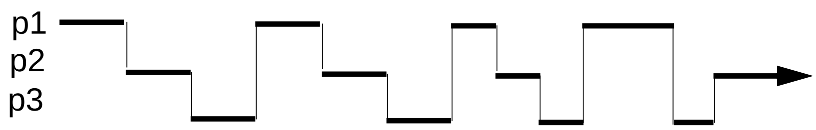
Fundamental notions (repeated): Parallel processes

process:
Execution of a sequential part of a program in its storage (address space).
Variable state: contents of the storage and the position of execution

parallel processes:
several processes, which are executed simultaneously on several processors



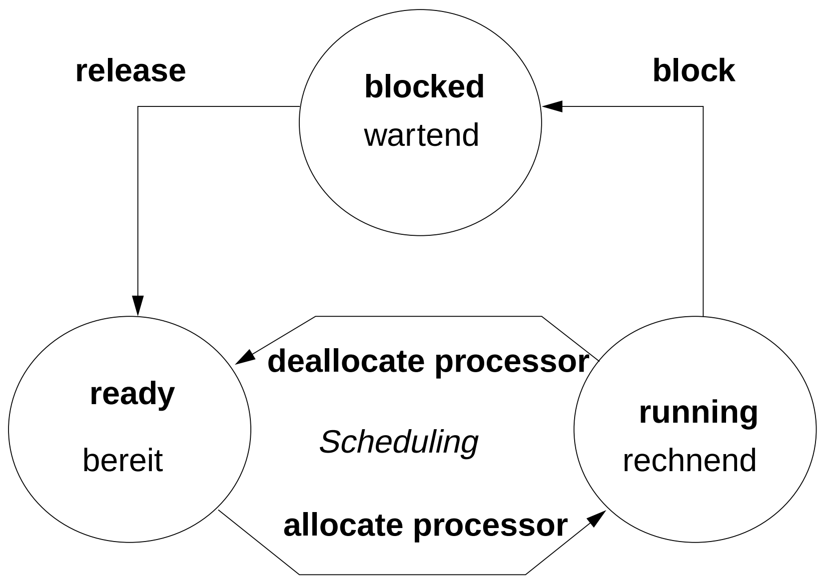
interleaved processes:
several processes, which are executed piecewise alternatingly on a single processor
processes are switched by a common process manager or by the processes themselves.



interleaved execution can simulate parallel execution;
frequent process switching gives the illusion that all process execute steadily.

concurrent processes:
processes, that can be executed in parallel or interleaved

Fundamental notions (repeated): States and transitions of processes



see KMS 2-17, 2-18

Threads (lightweight processes, Leichtgewichtsprozesse):
Processes, that are executed in parallel or interleaved in one common address space;
process switching is easy and fast.

Applications of parallel processes

- **Event-based user interfaces**:
  Events are propagated by a specific process of the system.
  Time consuming computations should be implemented by concurrent processes,
  to avoid blocking of the user interface.
- **Simulation** of real processes:
  e. g. production in a factory
- **Animation**:
  visualization of processes, algorithms; games
- **Control** of machines in **Real-Time:**
  processes in the computer control external facilities,
  e. g. factory robots, airplane control
- **Speed-up of execution** by parallel computation:
  several processes cooperate on a common task,
  e. g. parallel sorting of huge sets of data

The application classes follow **different objectives**.

Create threads in Java - technique: implement `Runnable`

Processes, threads in Java:
concurrently executed in the **common address space** of the program (or applet),
objects of class **`Thread`** with certain properties

Technique 1: A **user's class implements the interface `Runnable`**:

```
class MyTask implements Runnable
{ ...
    public void run ()
    {...}
    public MyTask(...) {...}
}
```

The interface requires to implement the method **`run`** - the program part to be executed as a process.

The constructor method.

The process is created as an **object of the predefined class `Thread`**:

```
Thread aTask = new Thread (new MyTask (...));
```

The following call starts the process:

```
aTask.start();
```

The new process starts executing in parallel with the initiating one.

This technique (implement the interface **`Runnable`**) should be used if

- the **new process need not be influenced** any further;
  i. e. it performs its task (method **`run`**) and then terminates, or
- the **user's class is to be defined as a subclass** of a class different from **`Thread`**

Create threads in Java - technique: subclass of Thread

Technique 2:
The user's class is defined as a **subclass of the predefined class Thread**:

```
class DigiClock extends Thread
{ ...
    public void run ()
    {...}
    DigiClock (...) {...}
}
```

- `public void run ()` — Overrides the **Thread** method **run.**
- `{...}` — The program part to be executed as a process.
- `DigiClock (...) {...}` — The constructor method.

The process is created as an **object of the user's class** (it is a **Thread** object as well):

```
Thread clock = new DigiClock (...);
```

The following call starts the process:

`clock.start();` The new process starts executing in parallel with the initiating one.

This technique (subclass of **Thread**) should be used if
the new process **needs to be further influenced**; hence,
further methods of the user's class are to be defined and called from outside the class,
e. g. to interrupt the process or to terminate it.
The class can not have another superclass!

Important methods of the class Thread

`public void run ();`
is to be overridden with a method that contains the code to be executed as a process

`public void start ();`
starts the execution of the process

`public void suspend ();`
(deprecated, deadlock-prone),
suspends the indicated process temporarily: e. g. `clock.suspend();`

`public void resume ();`
(deprecated), resumes the indicated process: `clock.resume();`

`public void join () throws InterruptedException;`
the calling process waits until the indicated process has terminated

```
try { auftrag.join(); } catch (Exception e){}
```

`public static void sleep (long millisec) throws InterruptedException;`
the calling process waits at least for the given time span (in milliseconds), e. g.

```
try { Thread.sleep (1000); } catch (Exception e){}
```

`public final void stop () throws SecurityException;`
not to be used! May terminate the process in an inconsistent state.

Example: Digital clock as a process in an applet (1)

The process displays the **current date and time** every second as a formatted text.


```
class DigiClock extends Thread
{  public void run ()
   {  while (running)                        iterate until it is terminated from the outside
      {  line.setText(new Date().toString());                                write the date
         try { sleep (1000); } catch (Exception ex) {}                                pause
      }
   }
                                        Method, that terminates the process from the outside:
   public void stopIt () { running = false; }
   private volatile boolean running = true;                                 state variable

   public DigiClock (Label t) {line = t;}                        label to be used for the text
   private Label line;
}
```

Technique **process as a subclass of `Thread`**, because it

- **is to be terminated** by a call of **`stopIt`**,
- **is to be interrupted** by calls of further **`Thread`** methods,
- other **super classes are not needed**.

Example: Digital clock as a process in an applet (2)

The process is created in the **`init`** method of the subclass of **`Applet`**:

```
public class DigiApp extends Applet
{  public void init ()
   {  Label clockText = new Label ("--------------------------------");
      add (clockText);

      clock = new DigiClock (clockText);                               create process
      clock.start();                                                    start process
   }

   public void start ()   { /* see below */ }                          resume process
   public void stop ()    { /* see below */ }                         suspend process
   public void destroy () { clock.stopIt(); }                       terminate process

   private DigiClock clock;
}
```

Processes, which are started in an applet

- may be suspended, while the applet is invisible (**`stop`**, **`start`**);
  better use synchronization or control variables instead of **`suspend`**, **`resume`**
- are to be terminated (**`stopIt`**), when the applet is deallocated (**`destroy`**).

Otherwise they bind resources, although they are not visible.

2. Properties of Parallel Programs

Goals:

- **formal reasoning** about parallel programs
- **proof properties** of parallel programs
- **develop** parallel programs such that certain **properties can be proven**

Example A:

```
x := 0; y := 0
co    x := x + 1 //
      y := y + 1
oc
z := x + y
```

Branches of **co-oc** are executed in parallel.

Proof that **z = 2** holds at the end.

Example B:

```
x := 0; y := 0
co    x := y+ 1 //
      y := x+ 1
oc
z := x + y
```

Show that **z = 2** can not be proven.

Methods:

Hoare Logic, Weakest Precondition, techniques for parallel programs

Proofs of parallel programs

Example A:

```
x := 0; y := 0 {x=0 ∧ y=0}
co
{x+1=1}x := x + 1{x=1} //
{y+1=1}y := y + 1{y=1}
oc
{x=1 ∧ y=1}→ {x+y=2}
z := x + y {z=2}
```

Example B$_1$:

```
x := 0; y := 0 {x=0 ∧ y=0}
co
{y+1=1}x := y + 1{x=1} //
{x+1=1}y := x + 1{y=1}
oc
{x=1 ∧ y=1}→ {x+y=2}
z := x + y {z=2}
```

Check each proof for correctness!

Explain!

Example B$_2$:

```
x := 0; y := 0 {x≥0 ∧ y≥0}
co
{y+1>0}x := y + 1{x>0} //
{x+1>0}y := x + 1{y>0}
oc
{x>0 ∧ y>0}→ {x+y≥2}
z := x + y {z≥2}
```

Does an **assignment of process p** interfere with an **assertion of process q**?

Hoare Logic: a brief reminder

Formal calculus for **proving properties of algorithms or programs** [C. A. R. Hoare, 1969]

Predicates (assertions) are stated for program positions:

```
{P} S1 {Q} S2 {R}
```

A predicate, like **Q**, characterizes the **set of states** that any execution of the program can achieve at that position. The predicates are expressions over variables of the program.

Each triple **{P} S {Q}** describes an effect of the execution of **S**. **P** is called a precondition, **Q** a postcondition of **S**.

The triple **{P} S {Q}** is correct, if the following holds:
If the execution of **S** is begun in a state of **P** and **if it terminates**, the the final state is in **Q** (partial correctness).

Two special assertions are:
{true} characterizing all states, and **{false}** characterizing no state.

Proofs of program properties are constructed using **axioms** and **inference rules** which describe the effects of each kind of statement, and define how proof steps can be correctly combined.

Axioms and inference rules for sequential constructs

statement sequence

1

$$\frac{\{P\}\ S_1\ \{Q\} \qquad \{Q\}\ S_2\ \{R\}}{\{P\}\ S_1; S_2\ \{R\}}$$

assignment

2

$$\{ P_{[x/e]} \}\ x := e\ \{P\}$$

$P_{[x/e]}$ means: P with all free occurrences of x substituted by e

stronger precondition

3

$$\frac{\{P\} \rightarrow \{R\} \qquad \{R\}\ S\ \{Q\}}{\{P\}\ S\ \{Q\}}$$

weaker postcondition

4

$$\frac{\{P\}\ S\ \{R\} \qquad \{R\} \rightarrow \{Q\}}{\{P\}\ S\ \{Q\}}$$

multiple alternative (guarded command)

5

$$\frac{P \wedge \neg(B_1 \vee ... \vee B_n) \Rightarrow Q \qquad \{P \wedge B_i\}\ S_i\ \{Q\},\quad 1 \le i \le n}{\{P\}\ \textbf{if}\ B_1 \rightarrow S_1\ []\ ...\ []\ B_n \rightarrow S_n\ \textbf{fi}\ \{Q\}}$$

selecting iteration

6

$$\frac{\{INV \wedge B_i\}\ S_i\ \{INV\},\quad 1 \le i \le n}{\{INV\}\ \textbf{do}\ B_1 \rightarrow S_1\ []\ ...\ []\ B_n \rightarrow S_n\ \textbf{od}\ \{INV \wedge \neg(B_1 \vee ... \vee B_n)\}}$$

no operation

7

$$\{P\}\ \textbf{skip}\ \{P\}$$

Verification: algorithm computes gcd

precondition: x, y ∈ ℕ , i. e. x > 0, y > 0; let G be greatest common divisor of x and y
postcondition: a = G
algorithm with { assertions over variables }:

{ G is gcd of x and y ∧ x>0 ∧ y>0 }
a := x; b := y;
{ INV: G is gcd of a and b ∧ a>0 ∧ b>0 }
do a ≠ b ->
  { INV ∧ a ≠ b }
  if a > b ->
    { G is gcd of a and b ∧ a>0 ∧ b>0 ∧ a>b } →
    { G is gcd of a-b and b ∧ a-b>0 ∧ b>0 }
    a := a - b
    { INV }
  [] a <= b ->
    { G is gcd of a and b ∧ a>0 ∧ b>0 ∧ b>a } →
    { G is gcd of a and b-a ∧ a>0 ∧ b-a>0 }
    b := b - a
    { INV }
  fi { INV ∧ a ≠ b ∧ ¬(a>b ∨ a ≤ b) → INV} „there is no 3rd case for the if -> INV“
  { INV }
od
{ INV ∧ a = b } →
{ a = G }

the loop terminates:
- a+b decreases monotonic
- a+b > 0 is invariant

Weakest precondition

A similar calculus as Hoare Logic is based on the notion of weakest preconditions [Dijkstra, 1976; Gries 1981]:

Program positions are also annotated by assertions that characterize program states.

The **weakest precondition** `wp (S, Q) = P` of a statement **S** maps a predicate **Q** on a predicate **P** (wp is a **predicate transformer**).
`wp (S, Q) = P` characterizes **the largest set of states** such that if the execution of **S** is begun in any state of **P**, then the execution is **guaranteed to terminate** in a state of **Q**
(**total correctness**).

If **P** ⇒ `wp (S, Q)` then `{P} S {Q}` holds in Hoare Logic.

This concept is a more goal oriented proof method compared to Hoare Logic.
We need weakest precondition only in the definition of „non-interference“ in proof for parallel programs.

Examples for weakest preconditions

1. P = wp (statement, Q)
2. i ≤ 0 = wp (**i := i + 1**, i ≤ 1)
3. true = wp (**if x >= y then z := x else z := y**, z = max (x, y))
4. (y ≥ x) = wp (**if x >= y then z := x else z := y**, z = y)
5. false = wp (**if x >= y then z := x else z := y**, z = y-1)
6. (x = y+1) = wp (**if x >= y then z := x else z := y**, z = y+1)
7. wp (S, true) = the set of all states such that the execution of S begun in one of them is guaranteed to terminate

Interleaving - used as an abstract execution model

Processes that are not blocked may be switched **at arbitrary points** in time.
A **scheduling strategy** reduces that freedom of the scheduler.

An example shows how different results are exhibited by switching processes differently.
Two processes operate on a common variable **account**:

```
                 account = 50;
                    a               b               c
Process1:  t1 = account;  t1 = t1 + 10;  account = t1;

Process2:  t2 = account;  t2 = t2 - 5;   account = t2;
                    d               e               f
```

Assume that the assignments *a* - *f* are atomic. Try any interleaved execution order of the two processes on a single processor. Check what the value of **account** is in each case.

Assume the sequences of statements *<a,b>* and *<d, e>* (or *<b, c>* and *<e, f>)* are atomic and check the results of any interleaved execution order.

We get the **same variety of results**, because there are **no global variables** in *b* or *e*
The coarser execution model is sufficient.

Atomic actions

Atomic action: A sequence of (one or more) operations, the internal states of which can not be observed because it has one of the following properties:

- it is a **non-interruptable machine instruction**,
- it has the **AMO** property, or
- **Synchronization** prohibits, that the action is interleaved with those of other processes, i. e. explicitly atomic.

At-most-once property (AMO):

The construct has **at most one** point where an other process can interact:

- **Expression E:**
  E has at most one variable v, that is written by a different process, and
  v occurs only once in E.
- **Assignment x := E:**
  E is AMO and x is not read by a different process, or
  x may be read by a different process, but E does not contain any global variable.
- **Statement sequence S:**
  one statement in S is AMO and all other statements in S do not have any global variable.

Atomic by AMO

Interleaving analysis is **simpler**, if **atomic decomposition is coarser**.

Check AMO property for nested constructs. Consider the most enclosing one to be atomic.

Examples: assume `x = 0; y = 0; z = 0;` to be global

atomic AMO constructs < ... >:

```
< t = < < x > + < 1 > >; > < x = < 1 >; >
```

interleaving actions of two processes:

(1)
```
                   a
p1:       < t = 0; t = t + 1;>

p2:       < s = 0; s = s + 1;>
                   b
```

(2)
```
               a
p1:       < x = 2;>

p2:       < t = x + 1;>
               b
```

(3)
```
             b     a
p1:       x = < y + 1 >;

p2:       y = < x + 1 >;
             d     c
```

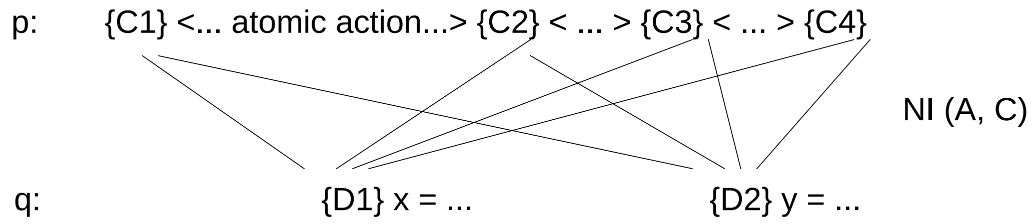
(4)
```
            c  a      b
p1:       x = <y> + <z>;

p2:       <y = 1;> <z = 2;>;
            d          e
```

Interference between processes

Critical assertions characterize **observable states** of a process p:
Let **{P} S {Q}** be the statement sequence of process p with its pre- and postcondition.
Then Q is critical.
Let T be a statement in S that is not part of an atomic statement and **R** its postcondition;
then **C = wp (T, R)** is critical.

For every critical assertion of the proof of p, it has to be proven that
non-interference NI (A, C) holds for each **assignment A** of every other process q:



non-interference NI (A, C) holds between
assignment A: {D} x = e in q having precondition D in a proof of q and
assertion C on p, if the following can be proven in programming logic:

$$\{ C \wedge D \} \quad A \quad \{ C \}$$

i. e. **the execution of A does not interfere with C (can not change C)**,
provided that the precondition D allows to execute A in a state where C holds.

Example: Interference between an assertion and an assignment

Consider processes p and q with **assertions at observable states**.

Consider a single critical **assertion C in p** and a single **assignment A in q**:

```
p:    ...<...> {C} <...>...

q:    ...<...> {d+1 > 0} a = d + 1; {Q} <...>...
                         A
```

Does A interfere with C? Depends on C:

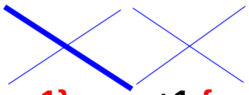
1. C: a == 1
`{a == 1 ∧ d + 1 > 0} a = d + 1 {a == 1}` is not provable ⇒ interference
(C ... C)

2. C: a > 0
`{a > 0 ∧ d + 1 > 0} a = d + 1 {a > 0}` is provable ⇒ non-interference

3. C: a==1 ∧ d<0
`{a==1 ∧ d<0 ∧ d+1>0} a = d + 1 {a==1 ∧ d<0}` is provable ⇒ non-interference
(d<0 ∧ d+1>0 is f)

Non-interference checks

x := 0; y := 0;
{ x = 0 ∧ y = 0 }
co {x+1 = 1} x := x+1 {x=1} //



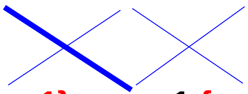
{y+1 = 1} y:= y+1 {y=1}
oc
{ x = 1 ∧ y = 1} => {x+y = 2}
z := x+y
{z = 2}

`NI(a, C)` holds for all 4 cases, e.g.

{ x+1 = 1 ∧ y+1 = 1} y:= y+1 {x+1 = 1 ∧ y = 1} =>
C
{x+1 = 1}
C

---

x := 0; y := 0;
{ x = 0 ∧ y = 0 }
co {y+1 = 1} x := y+1 {x=1} //



{x+1 = 1} y:= x+1 {y=1}
oc
{ x = 1 ∧ y = 1} => {x+y = 2}
z := x+y
{z = 2}

`NI(`y:= x+1, y+1 = 1`)` does not hold:

{ y+1 = 1 ∧ x+1 = 1} y:= x+1 {y+1 = 1}
is not correct

is not correct

Two inference rules for concurrent execution

The statement for **condition synchronization**

```
<await B -> S>
```

causes the executing process to be blocked until the condition **B** is true; then **S** is executed. The whole statement is executed as an atomic action; hence **B** holds at the begin of **S**.

{P ∧ B} S {Q}

{P} `<await B -> S >` {Q}

The statement for **concurrent processes**

co S_1 // ... // S_n oc

executes the statements S_i concurrently. It terminates when all S_i have terminated.

Non-Interference is to be proven.

{P_i} S_i {Q_i}, $1 \le i \le n$, are **interference-free theorems**

{$P_1 \wedge ... \wedge P_n$} co S_1 // ... // S_n oc {$Q_1 \wedge ... \wedge Q_n$}

Avoiding interference

1. **disjoint variables**:
   Two concurrent processes **p** and **q** are interference-free if the set of variables **p** writes to is disjoint from the set of variables **q** reads from and vice versa.

2. **weakened assertions**:
   The assertions in the proofs of concurrent processes can in some cases be made interference-free by weakening them.

3. **atomic action**:
   A non-interference-free assertion **C** can be hidden in an atomic action.

```
p:: ... x := e ...            p:: ... x := e ...

q:: ... S1 {C} S2 ...         q:: ...<S1 {C} S2> ...
```

4. **condition synchronization**:
   A synchronization condition can make an interfering assignment interference-free.

   S2 can not be executed in this state — or — C holds after **x:=e**

```
p:: ... x := e ...            p:: ..<await not C or B -> x:=e> ...
                                     with B = wp (x:=e, C)

q:: ... S1 {C} S2 ...         q:: ... S1 {C} S2 ...
```