

Parallel Programming

Prof. Dr. Uwe Kastens

Winter 2014 / 2015

Lecture Parallel Programming WS 2014/2015 / Slide 01

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.

Lecture Parallel Programming WS 2014/2015 / Slide 02

Objectives:

Understand the objectives

In the lecture:

Explanation of the objectives

Questions:

Do these objectives coincide with yours?

Contents

Week	Topic
1	1. Introduction
2	2. Properties of Parallel Programs
4	3. Monitors in General and in Java
5	4. Systematic Development of Monitors
6	5. Data Parallelism: Barriers
7	6. Data Parallelism: Loop Parallelization
11	7. Asynchronous Message Passing
12	8. Messages in Distributed Systems
14	9. Synchronous message Passing
	10. Conclusion

Lecture Parallel Programming WS 2014/2015 / Slide 03

Objectives:

Overview over the topics of the course

In the lecture:

Brief explanations of the topics

Questions:

- Which topics are you mostly interested in?
- Which are of less interest?
- Do you miss any topic?

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, interleaved execution	KMS KMS
address spaces, threads, process states monitor	KMS KMS
process, concurrency, parallelism, threads,	GP, KMS GP, KMS
synchronization, monitors in Java	GP, KMS
verification of properties of programs	Modellierung

Lecture Parallel Programming WS 2014/2015 / Slide 04

Objectives:

Sources for prerequisites

In the lecture:

- Explanations.
- The notions will be briefly repeated in the first chapter of this lecture - as introduced in GP and KMS

Suggested reading:

Relevant sections of lecture material of

- Grundlagen der Programmierung 1, 2,
- Konzepte und Methoden der Systemsoftware

Questions:

- Did you attend those lectures?
- Are you going to learn or to repeat those topics?

Organization of the course

Lecturer

Prof. Dr. Uwe Kastens:

Office hours: on appointment by email

Teaching Assistant:

- [Peter Pfahler](#)

Lecture

- V2 Mon 11:15 – 12:45, F1.110

Start date: Oct 13, 2014

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

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.

Lecture Parallel Programming WS 2014/2015 / Slide 05

Objectives:

Introduce the form of the material.

In the lecture:

- Explain the organization of the material.

Questions:

- Did you already explore the material?
- Did you place bookmarks into it?

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

Lecture Parallel Programming WS 2014/2015 / Slide 06

Objectives:

Reference to books

In the lecture:

Explain

- Andrews' book treats the concepts very thoroughly - deeper than we can do it in this lecture.
- The 3 books on Java present programming techniques very extensively with many elaborated examples; in some parts orientation is a bit missing.

Questions:

Are you going to dive into the matter along those books?

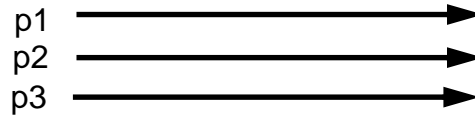
Fundamental notions (repeated): Parallel processes

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

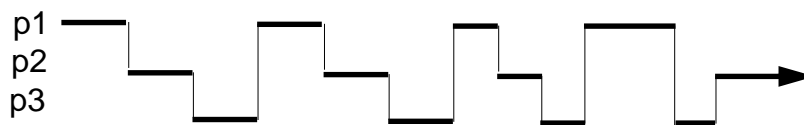
parallel processes:

several processes, which are executed simultaneously on several processors



interleaved processes:

several processes, which are executed piecewise alternately 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

Lecture Parallel Programming WS 2014/2015 / Slide 07

Objectives:

Repeat fundamental notions of processes

In the lecture:

- The notions are explained.
- Interleaved execution is also used as a model for describing properties of a system of processes.

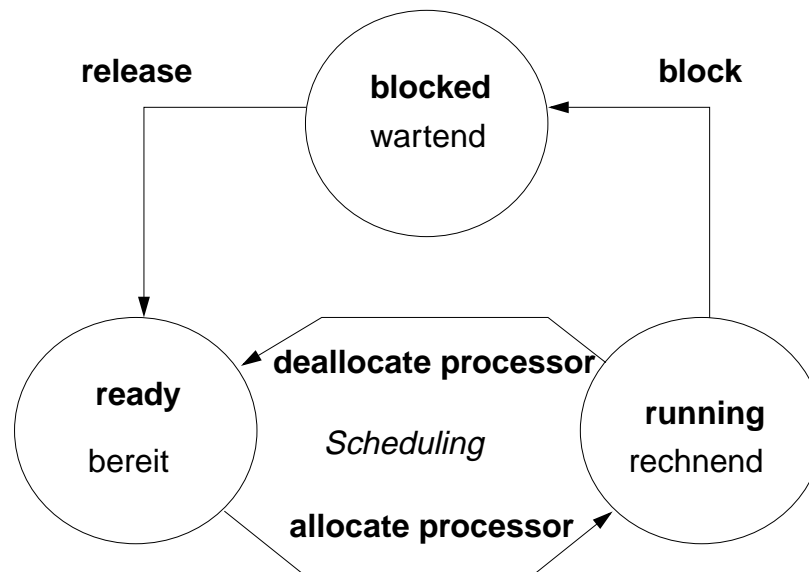
Suggested reading:

SWE-131

Questions:

- Which are the situations when a process switch is performed?

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.

Lecture Parallel Programming WS 2014/2015 / Slide 08

Objectives:

Understand process switching

In the lecture:

- Explain states and transitions.
- Role of the scheduler.

Questions:

- Give reasons and examples for state transitions.

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**.

Lecture Parallel Programming WS 2014/2015 / Slide 09

Objectives:

recognize different goals of parallelism

In the lecture:

Example are used to explain the classes of applications

Suggested reading:

SWE-132

Questions:

- Give further examples for the use of parallel processes, and point out their category.

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 ()          The interface requires to implement the method run
  {...}                      - the program part to be executed as a process.
  public MyTask(...) {...}   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`

Lecture Parallel Programming WS 2014/2015 / Slide 10

Objectives:

Understand declaration of process classes

In the lecture:

3 development steps:

- declare the class with its run method
- create a process object
- start the execution of the process object

If the user's class would need further object methods, they would be difficult to access. In that case one should better apply the second technique for process class declaration.

Suggested reading:

SWE-133

Questions:

- The class `Thread` has class methods and object methods. Which can be called from the run method in which way?

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 ()                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!

Lecture Parallel Programming WS 2014/2015 / Slide 11

Objectives:

Understand declaration of process classes

In the lecture:

3 development steps:

- declare the class with its run method
- create a process object
- start the execution of the process object

Compare to the variant with the interface `Runnable`.

Suggested reading:

SWE-134

Questions:

- The class `Thread` has class methods and object methods. Which can be called from the run method in which way?

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.

Lecture Parallel Programming WS 2014/2015 / Slide 12

Objectives:

Overview over the Thread methods

In the lecture:

- Explain the methods.
- Demonstrate the execution of the involved processes graphically.
- Point to examples.

Suggested reading:

SWE-137

Assignments:

Demonstrate the execution of the methods graphically.

Questions:

- Which method calls involve two processes, which only one?

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

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

Applet

Tue Mar 30 18:18:47 CEST 1999

Applet started.

```
class DigiClock extends Thread
{ public void run ()
  { while (running)
    { line.setText(new Date().toString());
      try { sleep (1000); } catch (Exception ex) {}
    }
  }
```

iterate until it is terminated from the outside

write the date

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**.

Lecture Parallel Programming WS 2014/2015 / Slide 13

Objectives:

A first complete example

In the lecture:

Explanation of

- the execution until termination from the outside.
- the `stopIt` method,
- the reason for the variant "subclass of `Thread`".

Demonstrate the applet Digital Clock Process

Suggested reading:

SWE-135

Assignments:

Install the example and modify it.

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.

Lecture Parallel Programming WS 2014/2015 / Slide 14

Objectives:

Start a process from an applet

In the lecture:

Explain how to start, suspend, resume, and terminate a process from an applet.

Suggested reading:

SWE-136

Assignments:

Modify the classes of this example such that `DigiClock` implements `Runnable` instead of being a subclass of `Thread`.

Questions:

- Explain why `DigiClock` extends `Thread` in the presented version.

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.

Methods:

Hoare Logic, Weakest Precondition, techniques for parallel programs

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.

Lecture Parallel Programming WS 2014/2015 / Slide 15a

Objectives:

Recognize the necessity for formal reasoning

In the lecture:

Explain the goals and the examples.

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 B₁:

```

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 B₂:

```

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**?

Lecture Parallel Programming WS 2014/2015 / Slide 15ab

Objectives:

How to proof parallel programs

In the lecture:

Check the correctness of proofs

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:

$\{\mathbf{true}\}$ characterizing all states, and $\{\mathbf{false}\}$ 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.

Lecture Parallel Programming WS 2014/2015 / Slide 15b

Objectives:

Recall the fundamental notions of Hoare logic

In the lecture:

The notions are explained. (see lecture material "Modellierung", slides Mod-4.51 to Mod-4.68)

Axioms and inference rules for sequential constructs

statement sequence

$$\frac{\begin{array}{l} \{P\} S_1 \quad \{Q\} \\ \{Q\} S_2 \quad \{R\} \end{array}}{\{P\} S_1; S_2 \quad \{R\}}$$

1

stronger precondition

$$\frac{\begin{array}{l} \{P\} \rightarrow \{R\} \\ \{R\} S \quad \{Q\} \end{array}}{\{P\} S \quad \{Q\}}$$

3

weaker postcondition

$$\frac{\begin{array}{l} \{P\} S \quad \{R\} \\ \{R\} \rightarrow \{Q\} \end{array}}{\{P\} S \quad \{Q\}}$$

4

assignment

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

2

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

multiple alternative (guarded command)

$$\frac{\begin{array}{l} P \wedge \neg(B_1 \vee \dots \vee B_n) \Rightarrow Q \\ \{P \wedge B_i\} S_i \quad \{Q\}, \quad 1 \leq i \leq n \end{array}}{\{P\} \text{ if } B_1 \rightarrow S_1 \quad \square \quad \dots \quad \square \quad B_n \rightarrow S_n \text{ fi } \{Q\}}$$

5

selecting iteration

$$\frac{\{INV \wedge B_i\} S_i \quad \{INV\}, \quad 1 \leq i \leq n}{\{INV\} \text{ do } B_1 \rightarrow S_1 \quad \square \quad \dots \quad \square \quad B_n \rightarrow S_n \text{ od } \{INV \wedge \neg(B_1 \vee \dots \vee B_n)\}}$$

6

no operation

$$\{P\} \text{ skip } \{P\}$$

7

Lecture Parallel Programming WS 2014/2015 / Slide 15c

Objectives:

Understand the inference rules

In the lecture:

The rules are explained:

- 1, 2, 3 are explained in "Modellierung" Mod-4.57 to Mod-4.60,
- guarded commands and iteration are generalized form of those explained in Mod-4.61 to Mod-4.66b,
- skip is clear.

Verification: algorithm computes gcd

precondition: $x, y \in \mathbb{N}$, 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 \wedge $x > 0 \wedge y > 0$ }

a := x; b := y;

{ INV: G is gcd of a and b \wedge $a > 0 \wedge b > 0$ }

do a \neq b ->

{ INV \wedge $a \neq b$ }

if a > b ->

{ G is gcd of a and b \wedge $a > 0 \wedge b > 0 \wedge a > b$ } \rightarrow

{ G is gcd of a-b and b \wedge $a-b > 0 \wedge b > 0$ }

a := a - b

{ INV }

[] a \leq b ->

{ G is gcd of a and b \wedge $a > 0 \wedge b > 0 \wedge b > a$ } \rightarrow

{ G is gcd of a and b-a \wedge $a > 0 \wedge b-a > 0$ }

b := b - a

{ INV }

fi { INV \wedge $a \neq b \wedge \neg(a > b \vee a \leq b) \rightarrow$ INV } „there is no 3rd case for the if -> INV“

{ INV }

od

{ INV \wedge $a = b$ } \rightarrow

{ a = G }

the loop terminates:

- a+b decreases monotonic
- a+b > 0 is invariant

Lecture Parallel Programming WS 2014/2015 / Slide 15d

Objectives:

Example for application of inference rules

In the lecture:

The verification steps are explained in "Modellierung" slide and PDF-file Mod-4.68

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** $\text{wp} (s, Q) = P$ of a statement s maps a predicate Q on a predicate P (wp is a **predicate transformer**).

$\text{wp} (s, Q) = P$ 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 $P \Rightarrow \text{wp} (s, Q)$ 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.

Lecture Parallel Programming WS 2014/2015 / Slide 15e

Objectives:

Understand the notion of weakest precondition

In the lecture:

The notion is explained using some examples.

Examples for weakest preconditions

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

Lecture Parallel Programming WS 2014/2015 / Slide 15f

Objectives:

Learn to find WPs

In the lecture:

The topics of the slide are explained:

- The formula.
- If $i \leq 0$, then the execution of $i := i + 1$ terminates with $i \leq 1$, while if $i > 0$ the execution of S cannot make $i \leq 1$.
- Execution of S always sets z to $\max(x, y)$.
- Execution of S beginning with $y \geq x$; sets z to y and execution of S beginning with $y < x$ sets z to x , which is $\neq y$.
- There is no start state for S such that it can set z less than y .
- Only if $x = y + 1$ holds when execution of S begins, it will set z to $y + 1$.
- Clear.

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 $\langle a, b \rangle$ and $\langle d, e \rangle$ (or $\langle b, c \rangle$ and $\langle e, f \rangle$) 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.

Lecture Parallel Programming WS 2014/2015 / Slide 17a

Objectives:

Motivation of the execution model

In the lecture:

- Explain the notion of atomic operations.
- Scheduling strategies are discussed later.
- Processes interfere via common, global variables.
- The desired result of a program execution may not depend on unjustified assumptions on the interleaving.
- Check all results the example may yield.

Questions:

- Which results may the example yield?
- Declare atomic statement sequences such that any interleaved execution yields the same result.

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.

Lecture Parallel Programming WS 2014/2015 / Slide 17b

Objectives:

Notion of atomic actions in the interleaving model

In the lecture:

- Explanation and examples for AMO.
- The example of PPJ-15 is varied.

Questions:

- Explain the AMO property using the terms "observable states" and "interleaved execution".

Atomic by AMO

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

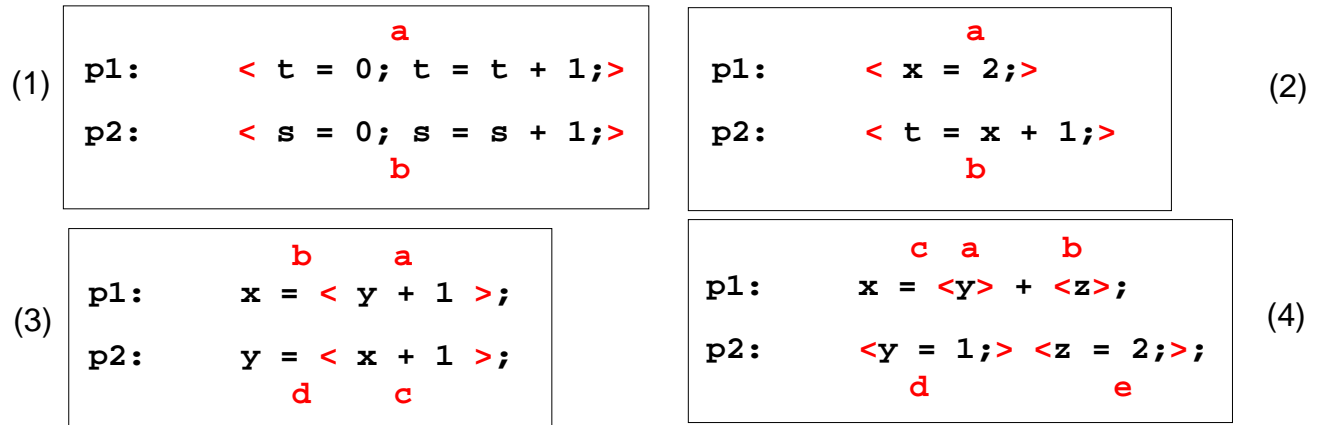
Check AMO property for nested constructs. Consider the most enclosing one to be atomic.

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

atomic AMO constructs $\langle \dots \rangle$:

$\langle t = \langle \langle x \rangle + \langle 1 \rangle \rangle; \rangle \langle x = \langle 1 \rangle; \rangle$

interleaving actions of two processes:



Lecture Parallel Programming WS 2014/2015 / Slide 17c

Objectives:

Understand: AMO constructs can be considered atomic

In the lecture:

The examples are explained using the definition of AMO.

Questions:

Which states can the processes in (1) to (4) reach depending on the execution order of the atomic actions?

Interference between processes

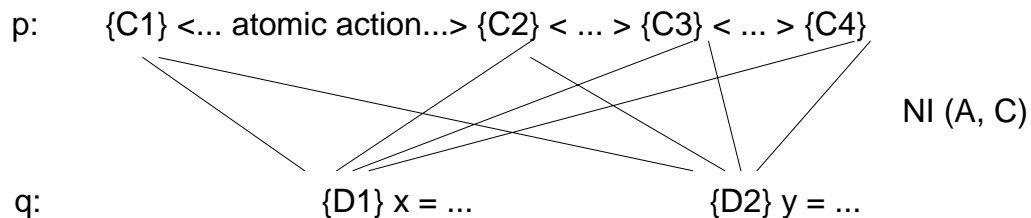
Critical assertions characterize **observable states** of a process p :

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:

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

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.

Lecture Parallel Programming WS 2014/2015 / Slide 17d

Objectives:

Interleaving and assertions on processes

In the lecture:

Explain

- NI,
- the role of $pre(A)$,
- the more possibilities for interleaving the more proofs of NI are needed,
- assertions that are globally true simplify the proofs,
- it is easier to prove weaker assertions.

Questions:

- Why can assertions on non-observable states be ignored?

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: $a == 1$
 $\{a == 1 \wedge d + 1 > 0\} a = d + 1 \{a == 1\}$ is not provable \Rightarrow interference
C C
2. C: $a > 0$
 $\{a > 0 \wedge d + 1 > 0\} a = d + 1 \{a > 0\}$ is provable \Rightarrow non-interference
3. C: $a == 1 \wedge d < 0$
 $\{a == 1 \wedge d < 0 \wedge d + 1 > 0\} a = d + 1 \{a == 1 \wedge d < 0\}$ is provable \Rightarrow non-interference
_____f_____

Lecture Parallel Programming WS 2014/2015 / Slide 17e

Objectives:

Understand interference checks

In the lecture:

The topics on the slide are explained using the example:

- Assertions are proven within their process and checked for non-interference;
- NI definition;
- 3 examples for interference check.

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.

$$\{x+1 = 1 \wedge y+1 = 1\} y:= y+1 \{x+1 = 1 \wedge y = 1\} \Rightarrow \{x+1 = 1\}$$

```
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}
```

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

$$\{y+1 = 1 \wedge x+1 = 1\} y:= x+1 \{y+1 = 1\}$$

is not correct

is not correct

Lecture Parallel Programming WS 2014/2015 / Slide 17f

Objectives:

Apply interference checks

In the lecture:

The interference checks of the examples are explained.

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**.

$$\frac{\{P \wedge B\} S \{Q\}}{\{P\} \langle \text{await } B \rightarrow S \rangle \{Q\}}$$

The statement for **concurrent processes**

`co S1 // ... // Sn oc`

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

Non-Interference is to be proven.

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

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

Lecture Parallel Programming WS 2014/2015 / Slide 17g

Objectives:

Understand the inference rules

In the lecture:

The two statements and their inference rules are explained.

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.

$p:: \dots x := e \dots$

$p:: \dots x := e \dots$

$q:: \dots s1 \{C\} s2 \dots$

$q:: \dots \langle s1 \{C\} s2 \rangle \dots$

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 $x:=e$

$p:: \dots x := e \dots$

$p:: \dots \langle \text{await not } C \text{ or } B \rightarrow x:=e \rangle \dots$
with $B = wp(x:=e, C)$

$q:: \dots s1 \{C\} s2 \dots$

$q:: \dots s1 \{C\} s2 \dots$

Lecture Parallel Programming WS 2014/2015 / Slide 17h

Objectives:

Techniques to reduce interference

In the lecture:

The techniques are explained using small examples.

- (4): Show that the await statement causes NI($x:=e, C$) to hold.

3. Monitors in general and in Java

Communication and synchronization of parallel processes

Communication between parallel processes: exchange of data by

- using a common, global variable, only in a programming model with **common storage**
- **messages** in programming model **distributed** or **common storage**
synchronous messages: sender waits for the receiver (languages: CSP, Occam, Ada, SR)
asynchronous messages: sender does not wait for the receiver (languages: SR)

Synchronization of parallel processes:

- **mutual exclusion (gegenseitiger Ausschluss):**
certain statement sequences (critical regions) may not be executed by several processes at the same time
- **condition synchronization (Bedingungssynchronisation):**
a process waits until a certain condition is satisfied by a different process

Language constructs for synchronization:

Semaphore, monitor, condition variable (programming model with common storage)
messages (see above)

Deadlock (Verklemmung):

Some processes are waiting cyclically for each other, and are thus blocked forever

Lecture Parallel Programming WS 2014/2015 / Slide 18

Objectives:

Fundamental notions for synchronization und communication

In the lecture:

Explain

- communication in common and in distributed storage,
- the difference of the two kinds of synchronization: mutual exclusion and condition synchronization,
- examples for them,
- language constructs for them.

Questions:

- Give examples where mutual exclusion or condition synchronization is needed.

Monitor - general concept

Monitor: high level synchronization concept introduced in [C.A.R. Hoare 1974, P. Brinch Hansen 1975]

Definition:

- A monitor is a **program module** for concurrent programming with **common storage**; it encapsulates data with its operations.
- A monitor has **entry procedures** (which operate on its data); they are **called by processes**; the monitor is **passive**.
- The monitor guarantees **mutual exclusion for calls of entry procedures**:
at most one process executes an entry procedure at any time.
- **Condition variables** are defined in the monitor and are used within entry procedures for **condition synchronization**.

Lecture Parallel Programming WS 2014/2015 / Slide 19a

Objectives:

Understand the fundamental concept of monitors

In the lecture:

Explain

- the properties of monitors,
- the 2 kinds of synchronization;
- condition variables are necessary for synchronization in monitors;
- examples for that

Questions:

- Are monitors usable without condition variables? for what applications?

Condition variables

A **condition variable** c is defined to have 2 operations to operate on it. They are executed by processes when executing a call of an entry procedure.

- **wait (c)** The executing process **leaves the monitor** and waits in a set associated to c , until it is released by a subsequent call $\text{signal}(c)$; then the process accesses the monitor again and continues.
- **signal (c):** The executing process releases **one arbitrary process** that waits for c .

Which of the two processes immediately continues its execution in the monitor depends on the variant of the signal semantics (see PPJ-22).

signal-and-continue:

The signal executing process continues its execution in the monitor.

A call $\text{signal}(c)$ has **no effect, if no process is waiting** for c .

Condition synchronization usually has the form

`if not B then wait (c);` or `while not B do wait (c);`

The **condition variable** c is used to synchronize on the **condition B**.

Note the difference between condition variables and semaphores:

Semaphores are counters. The effect of a call $V(s)$ on a semaphore is not lost if no process is waiting on s .

Lecture Parallel Programming WS 2014/2015 / Slide 19b

Objectives:

Understand condition variables

In the lecture:

Explain

- the 2 operations,
- distinction between B and c ,
- comparison with semaphores.

Questions:

- Why has the wait operation to release the monitor?

Example: bounded buffer

monitor Buffer

```
buf: Queue (k);
notFull, notEmpty: Condition;      2 condition variables: state of the buffer
```

entry put (d: Data)

```
do length(buf) = k -> wait (notFull); od;
enqueue (buf, d);
signal (notEmpty);
```

```
end;
```

entry get (var d: Data)

```
do length (buf) = 0 -> wait (notEmpty); od;
d := front (buf); dequeue (buf);
signal (notFull);
```

```
end;
```

```
end;
```

process Producer (i: 1..n) d: Data;

```
loop d := produce(); Buffer.put(d); end;
end;
```

process Consumer (i: 1..m) d: Data;

```
loop Buffer.get(d); consume(d); end;
end;
```

Lecture Parallel Programming WS 2014/2015 / Slide 20

Objectives:

Recall the monitor notion using a simple example

In the lecture:

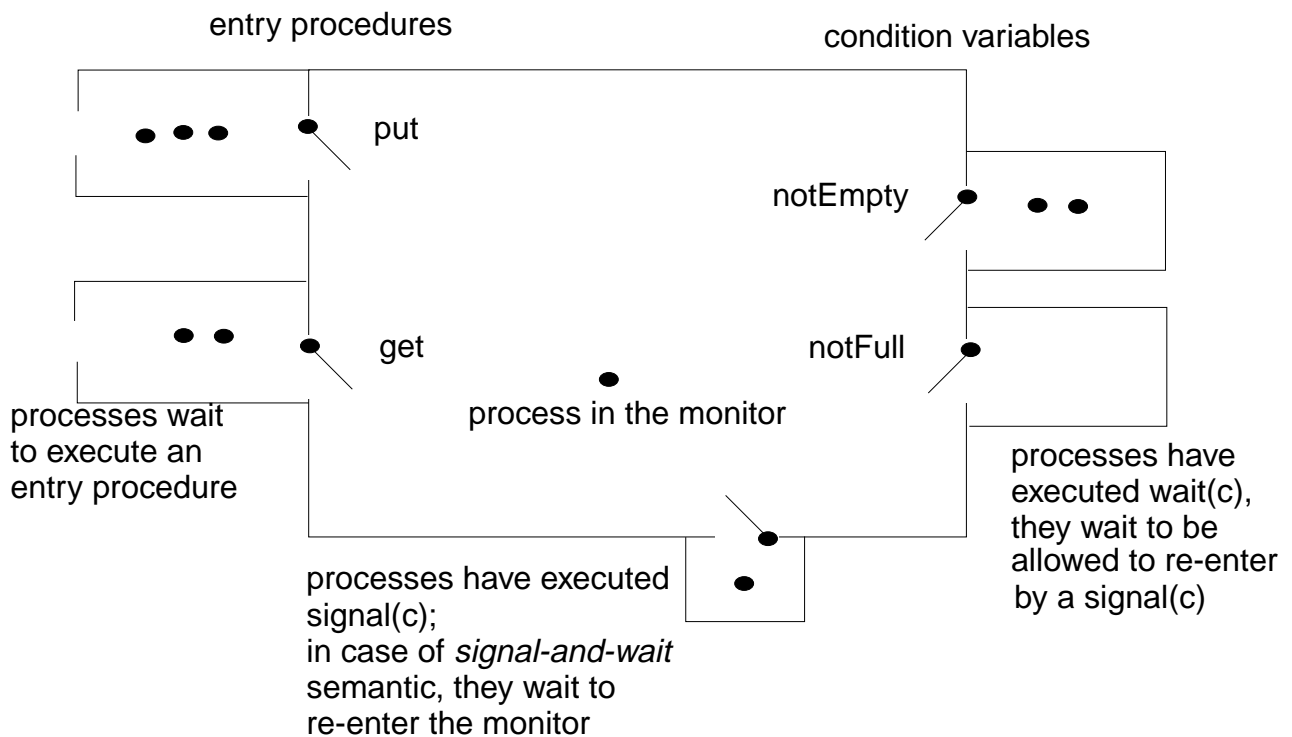
Explain

- 1 monitor, n producer processes, m consumer processes;
- monitor constructs: entry procedures, condition variable with wait and signal;
- usage of condition variables,
- notation: language SR, similar to Modula-2

Questions:

- What are the roles of the 2 condition variables?
- Explain the monitor using the notions of PPJ-19.

Synchronization in a monitor



Lecture Parallel Programming WS 2014/2015 / Slide 21

Objectives:

Visualization of monitor synchronization

In the lecture:

Explain

- waiting conditions using the example of PPJ-20;
- guaranteed: at most 1 process in the monitor;
- why waiting after a signal-operation

Questions:

- Explain the notions of PPJ-19 using this diagram.
- Can the example of a bounded buffer be implemented with only one condition variable? Explain.

Variants of signal-wait semantics

Processes compete for the monitor

- processes that are blocked by executing `wait(c)`,
- process that is in the monitor, may be executing `signal(c)`
- processes that wait to execute an entry procedure

signal-and-exit semantics:

The process that executes `signal` terminates the entry procedure call and leaves the monitor.

The released process enters the monitor **immediately** - without a state change in between

signal-and-wait semantics:

The process that executes `signal` leaves the monitor and waits to re-enter the monitor.

The released process enters the monitor **immediately** - without a state change in between

Variant **signal-and-urgent-wait**:

The process that has executed `signal` gets a higher priority than processes waiting for entry procedures

signal-and-continue semantics:

The process that executes `signal` continues execution in the monitor.

The released process has to wait until the monitor is free. The **state** that held at the

`signal` call may be changed meanwhile; the waiting condition has to be checked again:

```
do length(buf) = k -> wait(notFull); od;
```

Lecture Parallel Programming WS 2014/2015 / Slide 22

Objectives:

Understand the signal/wait semantics

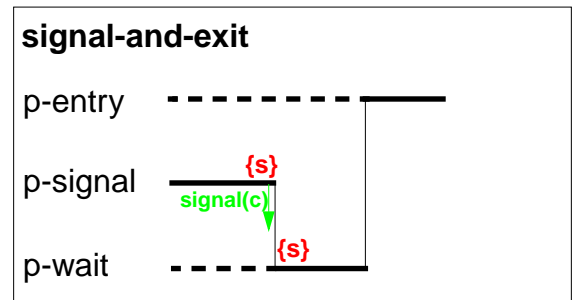
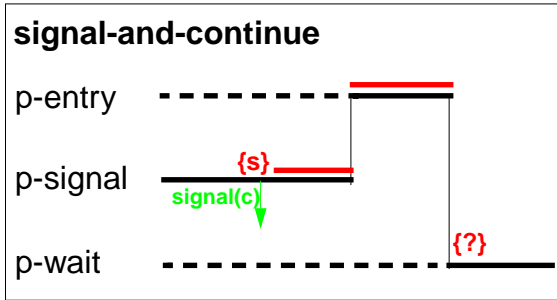
In the lecture:

Explain the notions using slide PPJ-21

Questions:

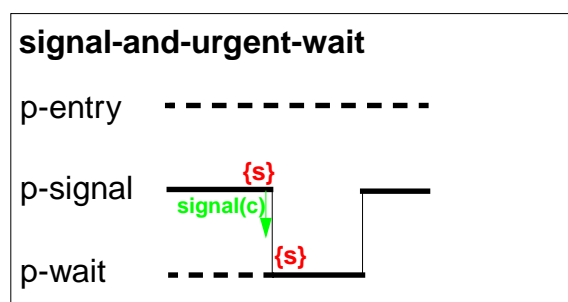
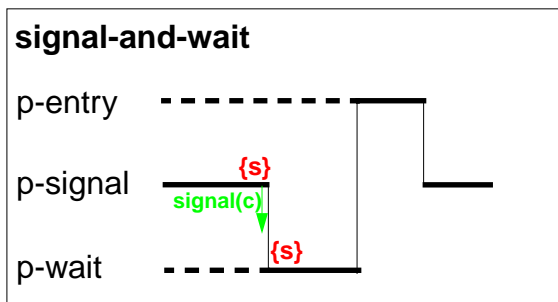
- Consider the example of PPJ-20 and assume signal-and-continue semantics. The wait conditions have to be checked in loops, although all signal calls are placed immediately before ends of entry procedures. Why?

Variants of signal-wait semantics: examples of execution



3 processes:
 p-entry waits to enter an entry procedure
 p-signal executes **signal(c)**
 p-wait has executed **wait(c)**

{s} state when **signal(c)** is executed
{s} may be modified here: **—**



© 2011 bei Prof. Dr. Uwe Kastens

Lecture Parallel Programming WS 2014/2015 / Slide 22a

Objectives:

Examples to understand the signal/wait semantics

In the lecture:

Explain the signal semantics of slide PPJ-22

Monitors in Java: mutual exclusion

Objects of any class can be used as **monitors**

Entry procedures:

Methods of a class, which implement critical operations on instance variables can be marked **synchronized**:

```
class Buffer
{   synchronized public void put (Data d) {...}
    synchronized public Data get () {...}
    ...
    private Queue buf;
}
```

If several processes **call synchronized methods** for the same object, they are executed under **mutual exclusion**.

They are synchronized by an internal synchronization variable of the object (lock).

Non-synchronized methods can be executed at any time concurrently.

There are also **synchronized class methods**: they are called under mutual exclusion with respect to the class.

synchronized blocks can be used to specify execution of a critical region with respect to an arbitrary object.

Lecture Parallel Programming WS 2014/2015 / Slide 23

Objectives:

Special properties of monitors in Java

In the lecture:

Explain

- objects being monitors;
- mutual exclusion for each object individually;
- synchronized methods are entry procedures;
- mutual exclusion only between calls of synchronized methods;

Questions:

Give examples for monitor methods that need *not* be executed under mutual exclusion.

Monitors in Java: condition synchronization

All processes that are blocked by `wait` are held in a single set;
condition variables can not be declared (there is only an implicit one)

Operations for condition synchronization:
 are to be called from inside **synchronized** methods:

- `wait()` **blocks** the executing process;
 releases the monitor object, and
 waits in the unique set of blocked processes of the object
- `notifyAll()` releases **all** processes that are blocked by `wait` for this object;
 they then compete for the monitor;
 the executing process continues in the monitor
 (signal-and-continue semantics).
- `notify()` releases **an arbitrary** one of the processes that are blocked by `wait`
 for this object;
 the executing process continues in the monitor
 (signal-and-continue semantics);
only usable if all processes wait for the same condition.

Always call `wait` in loops, because with **signal-and-continue** semantics
 after `notify`, `notifyAll` the **waiting condition may be changed**:

```
while (!Condition) try { wait(); } catch (InterruptedException e) {}
```

Lecture Parallel Programming WS 2014/2015 / Slide 24

Objectives:

Understand condition synchronization in Java

In the lecture:

Explain

- meaning of `wait`, `notifyAll`; and `notify`;
- more than one waiting condition;
- when to use `notify` or `notifyAll`;
- consequences of signal-and-continue semantics.

Questions:

- Construct a situation where a condition *C* holds before a call of `notifyAll`, but does not hold after the `wait` operation that is executed in the released process. Use interleaved execution to demonstrate the effects.

A Monitor class for bounded buffers

```

class Buffer
{
    private Queue buf;                // Queue of length n to store the elements
    public Buffer (int n) {buf = new Queue(n); }

    synchronized public void put (Object elem)
    {
        // a producer process tries to store an element
        while (buf.isFull())           // waits while the buffer is full
            try {wait();} catch (InterruptedException e) {}
        buf.enqueue (elem);           // changes the waiting condition of the get method
        notifyAll();                  // every blocked process checks its waiting condition
    }

    synchronized public Object get ()
    {
        // a consumer process tries to take an element
        while (buf.isEmpty())          // waits while the buffer is empty
            try {wait();} catch (InterruptedException e) {}
        Object elem = buf.first();
        buf.dequeue();                 // changes the waiting condition of the put method
        notifyAll();                   // every blocked process checks its waiting condition
        return elem;
    }
}

```

Lecture Parallel Programming WS 2014/2015 / Slide 25

Objectives:

Example for a monitor class in Java

In the lecture:

Explain

- changes of the waiting condition;
- why using `notifyAll`;
- the state transitions of `notifyAll` in the `get`-Operation;

Questions:

- In which states can a buffer be with respect to the two waiting conditions?
- What can one conclude if several processes are waiting?
- Explain in detail what happens if `notifyAll()` is executed when several processes are waiting.

Concurrency Utilities in Java 2

The **Java 2 platform** includes a **package of concurrency utilities**. These are classes which are designed to be used as building blocks in building concurrent classes or applications. ...

...

Locks - While locking is built into the Java language via the synchronized keyword, there are a number of **inconvenient limitations to built-in monitor locks**. The `java.util.concurrent.locks` package provides a high-performance lock implementation with **the same memory semantics as synchronization**, but which also supports specifying a timeout when attempting to acquire a lock, **multiple condition variables per lock**, non-lexically scoped locks, and support for interrupting threads which are waiting to acquire a lock.

<http://java.sun.com/j2se/1.5.0/docs/guide/concurrency/index.html>

<http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/locks/Condition.html>

Lecture Parallel Programming WS 2014/2015 / Slide 25j

Objectives:

Recognize improvements in Java 2 Concurrency Package

In the lecture:

The topics on the slide are explained.

Concurrency Utilities in Java 2 (example)

```

class BoundedBuffer {
    final Lock lock = new ReentrantLock();           explicit lock
    final Condition notFull = lock.newCondition();  condition variables
    final Condition notEmpty = lock.newCondition();

    final Object[] items = new Object[100];
    int putptr, takeptr, count;

    public void put (Object x) throws InterruptedException {
        lock.lock();                               explicit mutual exclusion
        try { while (count == items.length) notFull.await();    specific wait
            items[putptr] = x;
            if (++putptr == items.length) putptr = 0;
            ++count;
            notEmpty.signal();                     specific signal
        } finally { lock.unlock(); }               explicit mutual exclusion
    }

    public Object get () throws InterruptedException {
        lock.lock();                               explicit mutual exclusion
        try { while (count == 0) notEmpty.await();             specific wait
            Object x = items[takeptr];
            if (++takeptr == items.length) takeptr = 0;
            --count;
            notFull.signal();                       specific signal
        } finally { lock.unlock(); }               explicit mutual exclusion
    }
}

```

Lecture Parallel Programming WS 2014/2015 / Slide 25k

Objectives:

Recognize improvements in Java 2 Concurrency Package

In the lecture:

The topics on the slide are explained.

3. Systematic Development of monitors

Monitor invariant

A **monitor invariant (MI)** specifies **acceptable states of a monitor**

MI has to be true whenever a process may leave or (re-)enter the monitor:

- after the **initialization**,
- at the **beginning** and at the **end of each entry procedure**,
- before and after each call of **wait**,
- before and after each call of **signal** with **signal-and-wait** semantics (*),
- before each call of **signal** with **signal-and-exit** semantics (*).

Example of a monitor invariant for the bounded buffer:

MI: $0 \leq \text{buf.length}() \leq n$

The **monitor invariant has to be proven** for the program positions
after the initialization, at the end of entry procedures, before calls of wait (and signal if (*)).

One can **assume that the monitor invariant holds** at the other positions
at the beginning of entry procedures, after calls of wait (and signal if (*)).

Lecture Parallel Programming WS 2014/2015 / Slide 26

Objectives:

Understand monitor invariants

In the lecture:

Explain

- An invariant is a property to be guaranteed.
- MI for the example.

Suggested reading:

Andrews: 6.1, 6.2

Questions:

- Why can MI be assumed at the begin of entry procedures and after calls of wait?

Design steps using monitor invariant

1. Define the **monitor state**, and design the **entry procedures without synchronization**
e. g. bounded buffer: element count; entry procedures put and get
2. Specify a **monitor invariant**
e. g.: **MI**: $0 \leq \text{length}(\text{buf}) \leq N$
3. Insert **conditional waits**:
Consider every operation that may violate **MI**, e. g. `enqueue(buf)`;
find a condition **Cond** such that the operation may be executed safely if **Cond** **&&** **MI** holds,
e. g. `{ length(buf) < N && MI } enqueue(buf)`;
define one condition variable **c** for each condition **Cond**
insert a conditional wait in front of the operation:
`do !(length(buf) < N) -> wait(c); od`
Loop is necessary in case of **signal-and-continue** or the **may** in step 4!
4. **Insert notification of processes:**
after every state change that **may** make a waiting condition **Cond** true insert
`signal(c)` for the condition variable **c** of **Cond**
e. g. `dequeue(buf); signal(c)`;
Too many signal calls do not influence correctness - they only cause inefficiency.
5. **Eliminate unnecessary calls of signal** (see PPJ-28)
Caution: Missing signal calls may cause deadlocks!
Caution: **signal-and-continue** semantics lacks control of state changes

Lecture Parallel Programming WS 2014/2015 / Slide 27

Objectives:

Learn a design method

In the lecture:

Explain the single steps using the buffer example.

Questions:

- Explain step (5).

Bounded buffers

Derivation step 1: monitor **state** and **entry procedures**

```
monitor Buffer
  buf: Queue;                               // state: buf, length(buf)

  init buf = new Queue(n); end
  entry put (d: Data)                       // a producer process tries to store an element

    enqueue (buf, d);

  end;
  entry get (var d: Data)                   // a consumer process tries to take an element

    d := front(buf);
    dequeue(buf);

  end;
end;
```

Lecture Parallel Programming WS 2014/2015 / Slide 27a

Objectives:

Stepwise monitor design

In the lecture:

Explain step 1 for the buffer example

Bounded buffers

Derivation step 2: monitor invariant **MI**

```
monitor Buffer
  buf: Queue; // state: buf, length(buf)

  init buf = new Queue(n); end // MI: 0 <= length(buf) <= N
  entry put (d: Data) // a producer process tries to store an element

    enqueue (buf, d);

  end;
  entry get (var d: Data) // a consumer process tries to take an element

    d := front(buf);
    dequeue(buf);

  end;
end;
```

Lecture Parallel Programming WS 2014/2015 / Slide 27b

Objectives:

Stepwise monitor design

In the lecture:

Explain step 2 for the buffer example

Bounded buffers

Derivation step 3: insert **conditional waits**

```

monitor Buffer
  buf: Queue;                                // state: buf, length(buf)
  notFull, notEmpty: Condition;
  init buf = new Queue(n); end              // MI: 0 <= length(buf) <= N
  entry put (d: Data)                        // a producer process tries to store an element

    /* length(buf) < N && MI */
    enqueue (buf, d);

  end;
  entry get (var d: Data)                    // a consumer process tries to take an element

    /* length(buf) > 0 && MI */
    d := front(buf);
    dequeue(buf);

  end;
end;

```

Lecture Parallel Programming WS 2014/2015 / Slide 27c

Objectives:

Stepwise monitor design

In the lecture:

Explain step 3 for the buffer example.

Loop is needed for signal-and-continue and harmless for other semantics.

Bounded buffers

Derivation step 3: insert conditional waits

```

monitor Buffer
  buf: Queue; // state: buf, length(buf)
  notFull, notEmpty: Condition;
  init buf = new Queue(n); end // MI: 0 <= length(buf) <= N
  entry put (d: Data) // a producer process tries to store an element
    do length(buf) >= N -> wait(notFull); od;
    /* length(buf) < N && MI */
    enqueue (buf, d);

  end;

  entry get (var d: Data) // a consumer process tries to take an element
    do length(buf) <= 0 -> wait(notEmpty); od;
    /* length(buf) > 0 && MI */
    d := front(buf);
    dequeue(buf);

  end;
end;

```

Lecture Parallel Programming WS 2014/2015 / Slide 27ca

Objectives:

Stepwise monitor design

In the lecture:

Explain step 3 for the buffer example.

Loop is needed for signal-and-continue and harmless for other semantics.

Bounded buffers

Derivation step 4: insert notifications

```

monitor Buffer
  buf: Queue; // state: buf, length(buf)
  notFull, notEmpty: Condition;
  init buf = new Queue(n); end // MI: 0 <= length(buf) <= N
  entry put (d: Data) // a producer process tries to store an element
    do length(buf) >= N -> wait(notFull); od;
    /* length(buf) < N && MI */
    enqueue (buf, d);
    /* length(buf)>0 */
  end;
  entry get (var d: Data) // a consumer process tries to take an element
    do length(buf) <= 0 -> wait(notEmpty); od;
    /* length(buf) > 0 && MI */
    d := front(buf);
    dequeue(buf);
    /* length(buf)<N */
  end;
end;

```

Lecture Parallel Programming WS 2014/2015 / Slide 27d

Objectives:

Stepwise monitor design

In the lecture:

Explain step 4 for the buffer example.

Here the signal-calls are inserted at positions where the release-condition is guaranteed to hold - not only may hold. (So the loops around wait are in this case only needed if we have signal-and-continue semantics.)

Bounded buffers

Derivation step 4: insert notifications

```

monitor Buffer
  buf: Queue; // state: buf, length(buf)
  notFull, notEmpty: Condition;
  init buf = new Queue(n); end // MI: 0 <= length(buf) <= N
  entry put (d: Data) // a producer process tries to store an element
    do length(buf) >= N -> wait(notFull); od;
    /* length(buf) < N && MI */
    enqueue (buf, d);
    /* length(buf)>0 */ signal(notEmpty);
  end;
  entry get (var d: Data) // a consumer process tries to take an element
    do length(buf) <= 0 -> wait(notEmpty); od;
    /* length(buf) > 0 && MI */
    d := front(buf);
    dequeue(buf);
    /* length(buf)<N */ signal(notFull);
  end;
end;

```

Lecture Parallel Programming WS 2014/2015 / Slide 27da

Objectives:

Stepwise monitor design

In the lecture:

Explain step 4 for the buffer example.

Here the signal-calls are inserted at positions where the release-condition is guaranteed to hold - not only may hold. (So the loops around wait are in this case only needed if we have signal-and-continue semantics.)

Bounded buffers

Derivation step 5: eliminate unnecessary notifications

```

monitor Buffer
  buf: Queue; // state: buf, length(buf)
  notFull, notEmpty: Condition;
  init buf = new Queue(n); end // MI: 0 <= length(buf) <= N
  entry put (d: Data) // a producer process tries to store an element
    do length(buf) >= N -> wait(notFull); od;
    /* length(buf) < N && MI */
    enqueue (buf, d);
    if (length(buf) == 1) signal(notEmpty); // see PPJ-28
    // not correct under signal-and-continue
  end;
  entry get (var d: Data) // a consumer process tries to take an element
    do length(buf) <= 0 -> wait(notEmpty); od;
    /* length(buf) > 0 && MI */
    d := front(buf);
    dequeue(buf);
    if length(buf) == (N-1) -> signal(notFull); // see PPJ-28
    // not correct under signal-and-continue
  end;
end;

```

Lecture Parallel Programming WS 2014/2015 / Slide 27e

Objectives:

Stepwise monitor design

In the lecture:

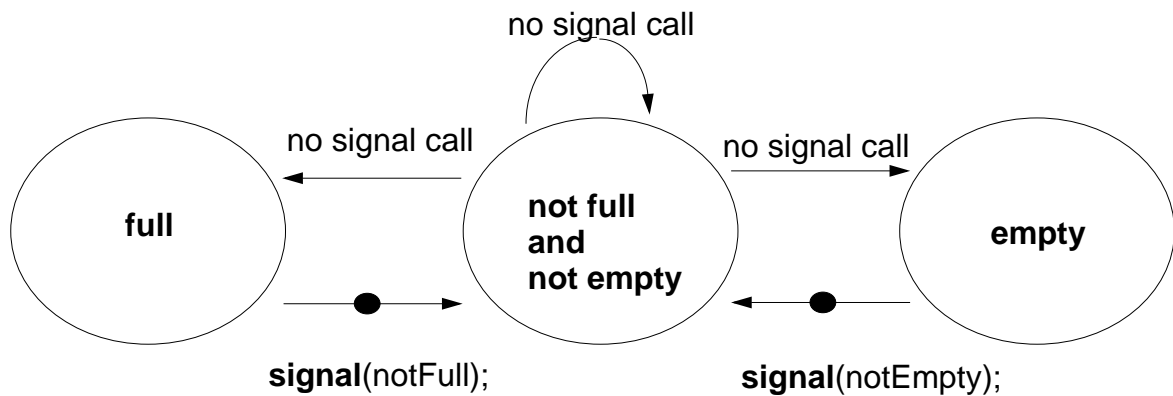
Explain step 5 for the buffer example

Relevant state changes

Processes need only be awakened when the state change is relevant:
when the waiting condition Cond changes from false to true,
 i.e. when a waiting process can be released.

These arguments do **not** apply for **signal-and-continue** semantics; there **Cond** may be changed between the signal call and the resume of the released process.

E. g. for the bounded buffer states w.r.t signalling are considered:



Lecture Parallel Programming WS 2014/2015 / Slide 28

Objectives:

Improve efficiency

In the lecture:

Explain

- state variables and waiting conditions;
- deadlock problem.

Suggested reading:

Lea: 4.3.2

Questions:

- What happens with processes that are awakened unnecessarily?

Pattern: Allocating counted resources

A **monitor** grants access to a set of $k \geq 1$ resources of the **same kind**.
Processes request n resources, $1 \leq n \leq k$, and return them after having used them.

Examples:

Lending bikes in groups ($n \geq 1$), allocating blocks of storage ($n \geq 1$),
 Taxicab provider ($n=1$), drive with a weight of $n \geq 1$ tons on a bridge

Monitor invariant	requestRes(1)	returnRes(1)
$0 \leq \text{avail}$ don't give a non-ex. resource	<code>if/do (!(1≤avail)) wait(av);</code> <code>avail--;</code>	<code>avail++; /* no wait! */</code> <code>signal(av);</code>
stronger invariant:		
$0 \leq \text{avail} \ \&\& \ 0 \leq \text{inUse}$... and don't take back more than have been given	<code>if/do (!(1≤avail)) wait(av);</code> <code>avail--; inUse++;</code> <code>signal(iu);</code>	<code>if/do (!(1≤inUse)) wait(iu);</code> <code>avail++; inUse--;</code> <code>signal(av);</code>
Monitor invariant	requestRes(n)	returnRes(n)
$0 \leq \text{avail}$ don't give a non-ex. resource	<code>do (!(n≤avail)) wait(av[n]);</code> <code>avail = avail - n;</code>	<code>avail = avail + n; /* no wait! */</code> <code>signal(av[1]); ... signal(av[avail]);</code>

The **identity** of the resources may be relevant: use a boolean array `avail[1] ... avail[k]`

Lecture Parallel Programming WS 2014/2015 / Slide 29

Objectives:

Allocation of equal resources

In the lecture:

Explain

- the task,
- the monitor invariant and the waiting conditions,
- variants of the pattern.

Questions:

- Elaborate the examples.
- Describe further examples.

Monitor for resource allocation

A **monitor** grants access to a set of $k \geq 1$ resources of the **same kind**.

Processes request n resources, $1 \leq n \leq k$, and return them after having used them.

Assumption: Process does not return more than it has received \Rightarrow simpler invariant:

```
class Resources
{ private int avail;                                // invariant: avail >= 0

  public Resources (int k) { avail = k; }

  synchronized public void getElems (int n)        // request n elements
  { while (avail < n)                               // negated waiting condition
    try { wait(); } catch (InterruptedException e) {}
    avail -= n;
  }

  synchronized public void putElems (int n)        // return n elements
  { avail += n;                                     // waiting is not needed because of assumption
    notifyAll();                                   // notify() would be wrong!
  }
}
```

Lecture Parallel Programming WS 2014/2015 / Slide 30

Objectives:

Java monitor for resource allocation

In the lecture:

Explain

- the program structure,
- the consequence of the assumption.

Questions:

- Why do we need `notifyAll()`?

Processes and main program for resource monitor

```
import java.util.Random;

class Client extends Thread
{ private Resources mon; private Random rand;
  private int ident, rounds, maximum;

  public Client (Resources m, int id, int rd, int max)
  { mon = m; ident = id; rounds = rd; maximum = max;
    rand = new Random(); // a number generator determines how many
  } // elements are requested in each round,

  public void run () // and when they are returned
  { while (rounds > 0)
    { int m = Math.abs(rand.nextInt()) % maximum + 1;
      mon.getElems (m);
      try { sleep (Math.abs(rand.nextInt()) % 1000 + 1); }
          catch (InterruptedException e) {}
      mon.putElems (m);
      rounds--;
    }
  }
}

public class TestResource
{ public static void main (String[] args)
  { int avail = 20;
    Resources mon = new Resources (avail);
    for (int i=0; i<5; i++)
      new Client (mon, i, 4, avail).start();
  }
}
```

Lecture Parallel Programming WS 2014/2015 / Slide 31

Objectives:

Use the monitor class of PPJ-30

In the lecture:

Explain the classes

Assignments:

Implement the program, add control output, and test it.

Readers-Writers problem (Step 1)

A monitor grants reading and writing access to a data base:
readers shared, writers exclusive.

```
monitor ReadersWriters
  nr: int; // number readers
  nw: int; // number writers
  init nr=0; nw=0; end
```

```
entry requestRead()
```

```
  nr++;
```

```
end;
```

```
entry releaseRead()
```

```
  nr--;
```

```
end;
```

```
entry requestWrite()
```

```
  nw++;
```

```
end;
```

```
entry releaseWrite()
```

```
  nw--;
```

```
end;
```

```
end;
```

Lecture Parallel Programming WS 2014/2015 / Slide 32a

Objectives:

Understand synchronization of readers and writers

In the lecture:

Explain

- important class of synchronization: shared reading and exclusive writing,
- the readers/writers problem,
- the monitor invariant,
- the design steps,
- different overlapping waiting conditions,
- consequences: several signals in releaseWrite.

Assignments:

- Implement the monitor.
- Implement processes for readers and writers. Delay the processes using `sleep` with random numbers as parameters. Produce output using the observer module.
- To avoid starvation of writers apply the following strategy: New readers have to wait until no writer is waiting. Introduce a new counter for that purpose. What do you observe?

Questions:

The following problem is similar - but symmetric: Control bi-directional traffic over a bridge that has only one lane. Explain the design!

Readers-Writers problem (Step 2)

A monitor grants reading and writing access to a data base:
readers shared, writers exclusive.

```
monitor ReadersWriters
  nr: int; // number readers
  nw: int; // number writers
  init nr=0; nw=0; end
```

```
entry requestRead()
```

```
  nr++;
```

```
end;
```

```
entry releaseRead()
```

```
  nr--;
```

```
end;
```

Monitor invariant RW:

```
(nr == 0 || nw == 0) && nw <= 1
```

```
entry requestWrite()
```

```
  nw++;
```

```
end;
```

```
entry releaseWrite()
```

```
  nw--;
```

```
end;
```

```
end;
```

Lecture Parallel Programming WS 2014/2015 / Slide 32b

Objectives:

Understand synchronization of readers and writers

In the lecture:

Explain

- important class of synchronization: shared reading and exclusive writing,
- the readers/writers problem,
- the monitor invariant,
- the design steps,
- different overlapping waiting conditions,
- consequences: several signals in releaseWrite.

Assignments:

- Implement the monitor.
- Implement processes for readers and writers. Delay the processes using `sleep` with random numbers as parameters. Produce output using the observer module.
- To avoid starvation of writers apply the following strategy: New readers have to wait until no writer is waiting. Introduce a new counter for that purpose. What do you observe?

Questions:

The following problem is similar - but symmetric: Control bi-directional traffic over a bridge that has only one lane. Explain the design!

Readers-Writers problem (Step3)

A monitor grants reading and writing access to a data base:
readers shared, writers exclusive.

```
monitor ReadersWriters
  nr: int; // number readers
  nw: int; // number writers
  init nr=0; nw=0; end

  entry requestRead()
    do !(nw==0)
      -> wait(okToRead);
    od;
    { nw==0 && RW }
    nr++;
    { RW }
  end;

  entry releaseRead()
    { RW && nr>0} nr--;

end;
```

Monitor invariant RW:

$$(nr == 0 \parallel nw == 0) \&\& nw \leq 1$$

```
entry requestWrite()
  do !(nr==0 && nw<1)
    -> wait(okToWrite);
  od;
  { nr==0 && nw<1 && RW }
  nw++;
  { RW }
end;

entry releaseWrite()
  { RW && nw==1} nw--;

end;
end;
```

Lecture Parallel Programming WS 2014/2015 / Slide 32c

Objectives:

Understand synchronization of readers and writers

In the lecture:

Explain

- important class of synchronization: shared reading and exclusive writing,
- the readers/writers problem,
- the monitor invariant,
- the design steps,
- different overlapping waiting conditions,
- consequences: several signals in releaseWrite.

Assignments:

- Implement the monitor.
- Implement processes for readers and writers. Delay the processes using `sleep` with random numbers as parameters. Produce output using the observer module.
- To avoid starvation of writers apply the following strategy: New readers have to wait until no writer is waiting. Introduce a new counter for that purpose. What do you observe?

Questions:

The following problem is similar - but symmetric: Control bi-directional traffic over a bridge that has only one lane. Explain the design!

Readers-Writers problem (Step 4)

A monitor grants reading and writing access to a data base:
readers shared, writers exclusive.

```
monitor ReadersWriters
  nr: int; // number readers
  nw: int; // number writers
  init nr=0; nw=0; end

entry requestRead()
  do !(nw==0)
    -> wait(okToRead);
  od;
  { nw==0 && RW }
  nr++;
  { RW }
end;

entry releaseRead()
  { RW && nr>0 } nr--;
  { RW && nr>=0 }
  { maybe nr==0 }

  signal(okToWrite);
end;
```

Monitor invariant RW:

$$(nr == 0 \parallel nw == 0) \ \&\& \ nw \leq 1$$

```
entry requestWrite()
  do !(nr==0 && nw<1)
    -> wait(okToWrite);
  od;
  { nr==0 && nw<1 && RW }
  nw++;
  { RW }
end;

entry releaseWrite()
  { RW && nw==1 } nw--;
  { nr==0 && nw==0 }
  signal(okToWrite);
  signal_all(okToRead);
end;
end;
```

Lecture Parallel Programming WS 2014/2015 / Slide 32d

Objectives:

Understand synchronization of readers and writers

In the lecture:

Explain

- important class of synchronization: shared reading and exclusive writing,
- the readers/writers problem,
- the monitor invariant,
- the design steps,
- different overlapping waiting conditions,
- consequences: several signals in releaseWrite.

Assignments:

- Implement the monitor.
- Implement processes for readers and writers. Delay the processes using `sleep` with random numbers as parameters. Produce output using the observer module.
- To avoid starvation of writers apply the following strategy: New readers have to wait until no writer is waiting. Introduce a new counter for that purpose. What do you observe?

Questions:

The following problem is similar - but symmetric: Control bi-directional traffic over a bridge that has only one lane. Explain the design!

Readers-Writers problem (Step 5)

A monitor grants reading and writing access to a data base:
readers shared, writers exclusive.

```
monitor ReadersWriters
  nr: int; // number readers
  nw: int; // number writers
  init nr=0; nw=0; end

entry requestRead()
  do !(nw==0)
    -> wait(okToRead);
  od;
  { nw==0 && RW }
  nr++;
  { RW }
end;

entry releaseRead()
  { RW && nr>0 } nr--;
  { RW && nr>=0 }
  { may be nr==0 }
  if nr==0
    -> signal(okToWrite);
end;
```

Monitor invariant RW:

$$(nr == 0 \parallel nw == 0) \ \&\& \ nw \leq 1$$

```
entry requestWrite()
  do !(nr==0 && nw<1)
    -> wait(okToWrite);
  od;
  { nr==0 && nw<1 && RW }
  nw++;
  { RW }
end;

entry releaseWrite()
  { RW && nw==1 } nw--;
  { nr==0 && nw==0 }
  signal(okToWrite);
  signal_all(okToRead);
end;
end;
```

Lecture Parallel Programming WS 2014/2015 / Slide 32e

Objectives:

Understand synchronization of readers and writers

In the lecture:

Explain

- important class of synchronization: shared reading and exclusive writing,
- the readers/writers problem,
- the monitor invariant,
- the design steps,
- different overlapping waiting conditions,
- consequences: several signals in releaseWrite.

Assignments:

- Implement the monitor.
- Implement processes for readers and writers. Delay the processes using `sleep` with random numbers as parameters. Produce output using the observer module.
- To avoid starvation of writers apply the following strategy: New readers have to wait until no writer is waiting. Introduce a new counter for that purpose. What do you observe?

Questions:

The following problem is similar - but symmetric: Control bi-directional traffic over a bridge that has only one lane. Explain the design!

Readers/writers monitor in Java

```

class ReaderWriter
{ private int nr = 0, nw = 0;
    // monitor invariant RW: (nr == 0 || nw == 0) && nw <= 1
    synchronized public void requestRead ()
    { while (nw > 0) // negated waiting condition
        try { wait(); } catch (InterruptedException e) {}
        nr++;
    }
    synchronized public void releaseRead ()
    { nr--;
        if (nr == 0) notify (); // awaken one writer is sufficient
    }

    synchronized public void requestWrite ()
    { while (nr > 0 || nw > 0) // negated waiting condition
        try { wait(); } catch (InterruptedException e) {}
        nw++;
    }
    synchronized public void releaseWrite ()
    { nw--;
        notifyAll (); // notify 1 writer and all readers would be sufficient!
    }
}

```

Lecture Parallel Programming WS 2014/2015 / Slide 33

Objectives:

Readers/writers monitor in Java

In the lecture:

Explain the methods.

Assignments:

Use the monitor in a complete program as described for PPJ-32.

Questions:

- How would you program the monitor if you could use condition variables? Write it in the notation of slide PPJ-20.

Method: rendezvous of processes

Processes pass through a **sequence of states** and **interact** with each other.
A monitor coordinates the **rendezvous in the required order**.

Design method:

Specify states by counters;

characterize **allowed states by invariants** over counters;

derive waiting conditions of monitor operations from the invariants;

substitute counters by binary variables.

Example: Sleeping Barber:

In a sleepy village close to Paderborn a barber is sleeping while waiting for customers to enter his shop. When a customer arrives and finds the barber sleeping, he awakens him, sits in the barber's chair, and sleeps while he gets his hair cut. If the barber is busy when a customer arrives, the customer sleeps in one of the other chairs. After finishing the haircut, the barber gets paid, lets the customer exit, and awakens a waiting customer, if any.

2 kinds of processes: barber (1 instance), customer (many instances)

2 rendezvous: haircut and customer leaves

The task is also an example for the Client/Server pattern.

Lecture Parallel Programming WS 2014/2015 / Slide 34

Objectives:

Overview over the method.

In the lecture:

Explain the steps of the method and the example.

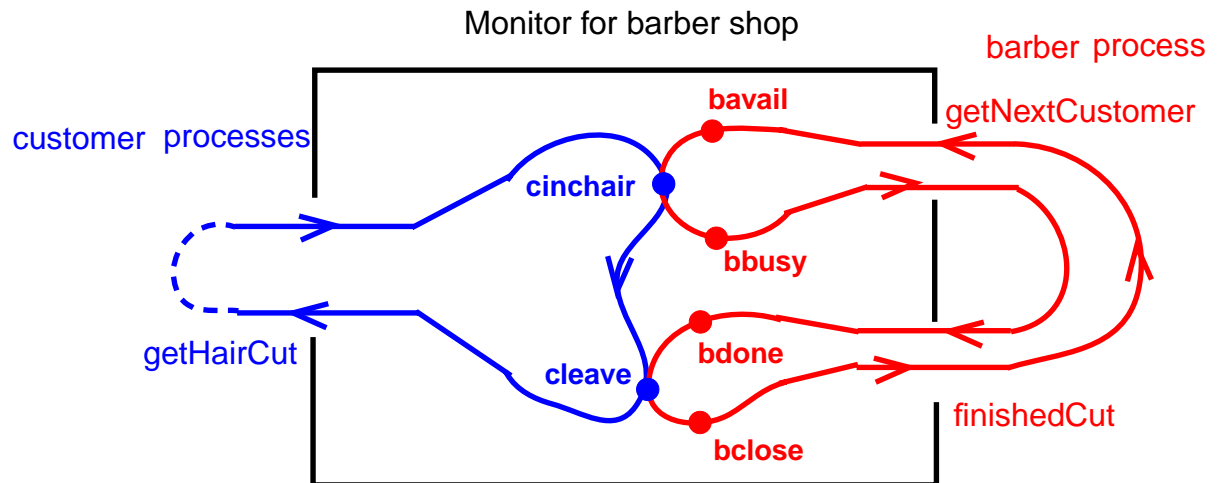
Assignments:

Solve the task "Roller Coaster (Achterbahn)" correspondingly.

Questions:

- Describe similar tasks.

Monitor design for the Sleeping Barber problem (step 1)



Counters represent states, incremented in entry procedures:

entry proc `getHairCut`:

```
cinchair++;
cleave++;
```

entry proc `getNextCustomer`:

```
bavail++;
bbusy++;
```

entry proc `finishedCut`:

```
bdone++;
bclose++;
```

Lecture Parallel Programming WS 2014/2015 / Slide 35

Objectives:

Characterize rendezvous by counters

In the lecture:

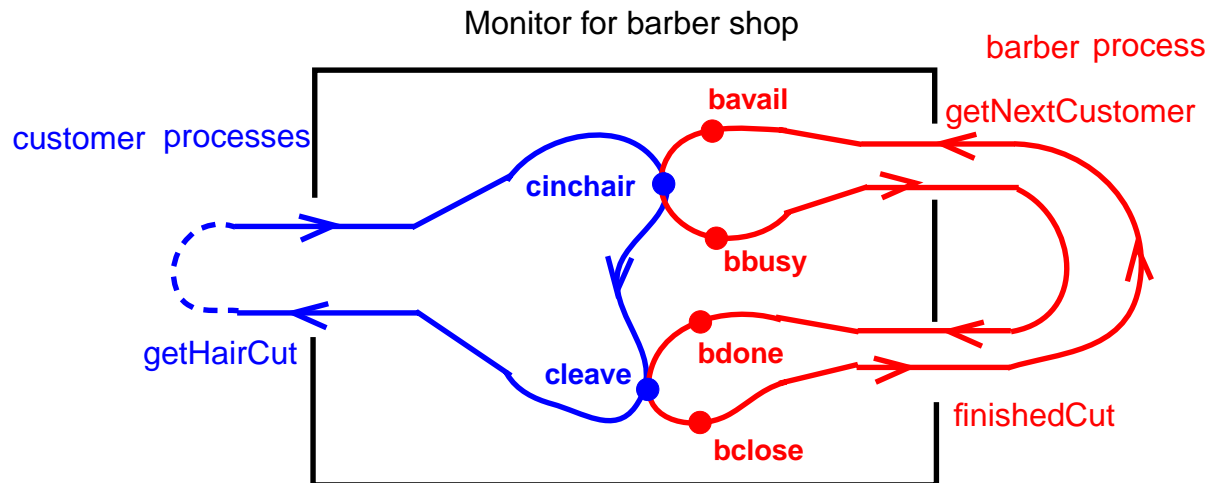
Explain

- the role of the counters,

Questions:

- How are the values of the counters related?

Monitor invariant for the Sleeping Barber problem (step 2)



Invariants over counters:

C1: $\text{cinchair} \geq \text{cleave}$ and
 $\text{bavail} \geq \text{bbusy} \geq \text{bdone} \geq \text{bclose}$

C2: $\text{bavail} \geq \text{cinchair} \geq \text{bbusy}$

C3: $\text{bdone} \geq \text{cleave} \geq \text{bclose}$

Monitor invariant: BARBER: C1 and C2 and C3

Lecture Parallel Programming WS 2014/2015 / Slide 35a

Objectives:

Monitor invariant over counters

In the lecture:

Explain

- the meaning of the inequalities

Questions:

- What must the processes do to guarantee *C2*?
- What must the processes do to guarantee *C1*?

Waiting conditions for the Sleeping Barber problem (step 3)

Monitor invariant: BARBER: C1 and C2 and C3:

C1: `cinchair >= cleave` and
`bavail >= bbusy >= bdone >= bclose`

guaranteed by execution order

C2: `bavail >= cinchair >= bbusy`

leads to 2 waiting conditions

C3: `bdone >= cleave >= bclose`

leads to 2 waiting conditions

entry proc `getHairCut`:

```
do not (bavail > cinchair) -> wait (b); done;
cinchair++;
do not (bdone > cleave) -> wait (o); done;
cleave++;
```

entry proc `getNextCustomer`:

```
bavail++;
do not (cinchair > bbusy) -> wait (c); done;
bbusy++;
```

entry proc `finishedCut`:

```
bdone++;
do not (cleave > bclose) -> wait (e); done;
bclose++;
```

Lecture Parallel Programming WS 2014/2015 / Slide 36

Objectives:

First phase of monitor design

In the lecture:

- Explain the waiting conditions.

Questions:

- Why need some incrementations a waiting condition, and others don't?

Substitute counters (step 3a)

new binary variables:

barber = **bavail** - **cinchair**

chair = **cinchair** - **bbusy**

open = **bdone** - **cleave**

exit = **cleave** - **bclose**

value ranges: {0, 1}

Old invariants:

C2: **bavail** >= **cinchair** >= **bbusy**

C3: **bdone** >= **cleave** >= **bclose**

New invariants:

C2: **barber** >= 0 && **chair** >= 0

C3: **open** >= 0 && **exit** >= 0

increment operations and conditions are substituted:

entry proc **getHairCut**:

do not (**barber** > 0) -> wait (**b**); done;

barber--; **chair++**;

do not (**open** > 0) -> wait (**o**); done;

open--; **exit++**;

entry proc **getNextCustomer**:

barber++;

do not (**chair** > 0) -> wait (**c**); done;

chair--;

entry proc **finishedCut**:

open++;

do not (**exit** > 0) -> wait (**e**); done;

exit--;

Lecture Parallel Programming WS 2014/2015 / Slide 37

Objectives:

Understand substitution of variables

In the lecture:

- Show substitution in comparison to PPJ-36.
- All state variables have the value range {0, 1}.

Questions:

- Explain how the general condition variables are used.

Signal operations for the Sleeping Barber problem (step 4)

new binary variables:

barber = **bavail** - **cinchair**

chair = **cinchair** - **bbusy**

open = **bdone** - **cleave**

exit = **cleave** - **bclose**

value ranges: {0, 1}

Old invariants:

C2: **bavail** >= **cinchair** >= **bbusy**

C3: **bdone** >= **cleave** >= **bclose**

New invariants:

C2: **barber** >= 0 && **chair** >= 0

C3: **open** >= 0 && **exit** >= 0

insert call `signal(x)` call where a condition of `x` may become true:

entry proc `getHairCut`:

do not (**barber** > 0) -> wait (**b**); done;

barber--; **chair++**; **signal(c)**;

do not (**open** > 0) -> wait (**o**); done;

open--; **exit++**; **signal(e)**;

entry proc `getNextCustomer`:

barber++; **signal(b)**;

do not (**chair** > 0) -> wait (**c**); done;

chair--;

entry proc `finishedCut`:

open++; **signal(o)**;

do not (**exit** > 0) -> wait (**e**); done;

exit--;

Lecture Parallel Programming WS 2014/2015 / Slide 37a

Objectives:

Understand substitution of variables

In the lecture:

- Explain how to use general condition variables for the implementation of the monitor.

Assignments:

- Implement the monitor in Java according to this plan and test it.

Questions:

- Explain insertion of the awoken operations.

5. Data Parallelism: Barriers

Many processes execute the **same operations at the same time on different data**; usually on elements of **regular data structures**: arrays, sequences, matrices, lists.

Data parallelism as an **architectural model of parallel computers**:

vector machines, e. g. Cray

SIMD machines (Single Instruction Multiple Data), e. g. Connection Machine, MasPar

GPUs (Graphical Processing Units); massively parallel processors on graphic cards

Data parallelism as a **programming model for parallel computers**:

- computations on **arrays in nested loops**
- analyze **data dependences** of computations, **transform** and **parallelize** loops
- iterative **computations in rounds**, synchronize with **Barriers**
- **systolic computations**: 2 phases are iterated: compute - shift data to neighbour processes

Applications mainly in **technical, scientific computing**, e. g.

- fluid mechanics
- image processing
- solving differential equations
- finite element method in design systems

Lecture Parallel Programming WS 2014/2015 / Slide 38

Objectives:

Overview over notions of data parallelism

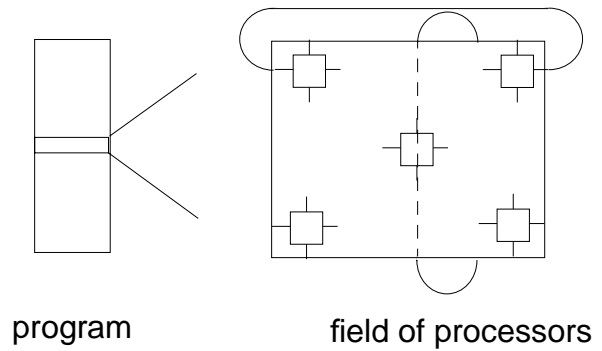
In the lecture:

Explain the notions

Data parallelism as an architectural model

SIMD machine: Single Instruction Multiple Data

- very many processors, **massively parallel**
e. g. 32 x 64 processor field
- **local memory** for each processor
- same instructions in **lock step**
- fast communication in **lock step**
- fixed topology, usually a **grid**
- machine types e. g. Connection Machine, MasPar



Lecture Parallel Programming WS 2014/2015 / Slide 39

Objectives:

Architecture of a SIMD computer

In the lecture:

Explanation of the properties

Data parallelism as a programming model

- regular data structures (arrays, lists) are mapped onto a field of processors
- processes execute the same program on individual data in lock step
- communication with neighbours in the same direction in lock step

simple example matrix addition:

$$\boxed{C} = \boxed{A} + \boxed{B}$$

sequential:

```
for (i = 0; i < N; i++)
  for (j = 0; j < M; j++)
    c[i,j] = a [i,j] + b[i,j];
```

```
distribute A, B
c = a + b
collect C
```

1 step!

- these can be parallelized directly, since there are no **data dependences**
- **data mapping** is trivial: array element [i,j] on process [i,j]
- **communication** is not needed
- no **algorithmic idea** is needed

Lecture Parallel Programming WS 2014/2015 / Slide 40

Objectives:

idea of loop parallelization

In the lecture:

- explain the example,
- show the reasons for the simplicity of the parallelization

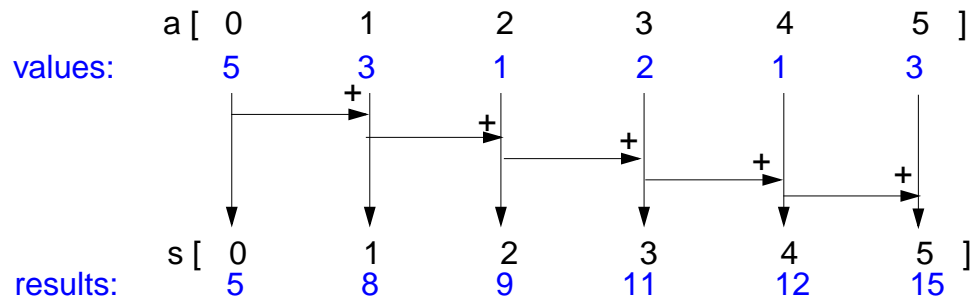
Questions:

- Give examples for array operations that can be parallelized with similar ease.

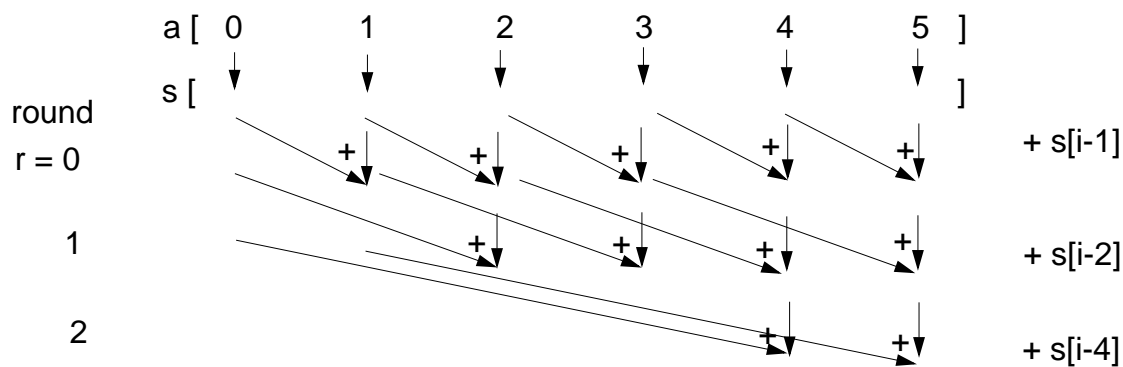
Example prefix sums

input: sequence a of numbers;
output: sequence s of sums of the prefixes of a

$$s[i] = \sum_{j=0}^i a[j]$$



parallel algorithmic idea:



Lecture Parallel Programming WS 2014/2015 / Slide 41

Objectives:

Understand the parallel computation of prefix sums

In the lecture:

Explain

- the task,
- the algorithmic idea,
- how to exploit associativity,
- computations in rounds,
- duplication of distance

Questions:

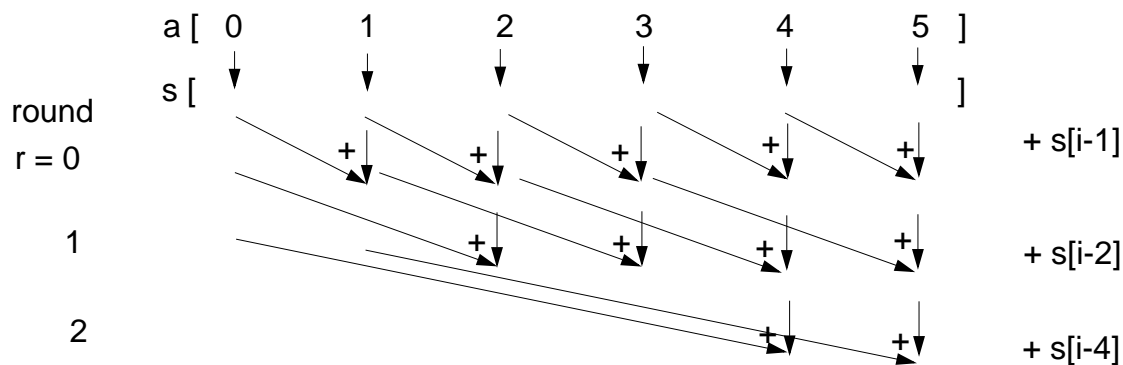
- What is the formula for the number of steps in the sequential and in the parallel case?

Example prefix sums (2)

input: sequence a of numbers;
output: sequence s of sums of the prefixes of a

$$s[i] = \sum_{j=0}^i a[j]$$

parallel algorithmic idea:



Proof for process $p = 0 \dots n - 1$

Invariant SUM: $s[p] = a[p-d+1] + \dots + a[p]$ with $d = 1, 2, \dots, m \leq n$ distance before next round

Induction begin: $d = 1$; $s[p] = a[p]$ holds by initialization

induction step: computation $s[p] = s[p - d] + a[p-2d+1] + \dots + a[p-d] + a[p-d+1] + \dots + a[p]$

substitution of $2d$ by d implies SUM

Lecture Parallel Programming WS 2014/2015 / Slide 41a

Objectives:

Proof the parallel computation of prefix sums

In the lecture:

Explain

- the proof

Prefix sums: applied methods

- **computational scheme reduction:**
all array elements are comprised using a reduction operation (here: addition)
- **iterative computation in rounds:**
in each round all processes perform a computation step
- **duplication of distance:**
data is exchanged in each round with a neighbour at twice the distance as in the previous round
- **barrier synchronization:**
processes may not enter the next round, before all processes have finished the previous one

Lecture Parallel Programming WS 2014/2015 / Slide 42

Objectives:

Point out the methods

In the lecture:

- Explain the methods for the prefix sums.
- Point out other applications of these methods.

Barriers

Several processes meet at a common point of synchronization

Rule: All processes must have reached the barrier (for the j-th time), before one of them leaves it (for the j-th time).

Applications:

- iterative computations, where iteration j uses results of iteration j-1
- separation of computational phases

Scheme:

```
public void run ()
{ do { computeNewValues (i);
      b.barrier();
    }
  while (!converged);
}
```

Implementation techniques for barriers:

- central controller: monitor or coordination process
- worker processes coordinated as a tree
- worker processes symmetrically coordinated (butterfly barrier, dissemination barrier)

Lecture Parallel Programming WS 2014/2015 / Slide 43

Objectives:

Understand the concept of barriers

In the lecture:

Explain

- the barrier rule,
- the relation to the prefix sums,
- applications.

Barrier implemented by a monitor

Monitor stops a given number of processes and releases them together:

```
class BarrierMonitor
{ private int processes // number of processes to be synchronized
    arrived = 0; // number of processes arrived at the barrier

    public BarrierMonitor (int procs)
    { processes = procs; }

    synchronized public barrier ()
    { arrived++;
      if (arrived < processes)
        try { wait(); } catch (InterruptedException e) {}
        // exception destroys barrier behaviour

      else
        { arrived = 0; // reset arrival count
          notifyAll(); // release the other processes
        }
    }
}
```

Lecture Parallel Programming WS 2014/2015 / Slide 44

Objectives:

Understand the monitor implementation

In the lecture:

Explain

- the implementation,
- why waiting in a loop is not necessary.

Questions:

- Why does this central solution cause a bottleneck?

Distributed tree barrier

Barrier synchronization of the worker processes organized as a **binary tree**.
Bottleneck of central synchronization is avoided.

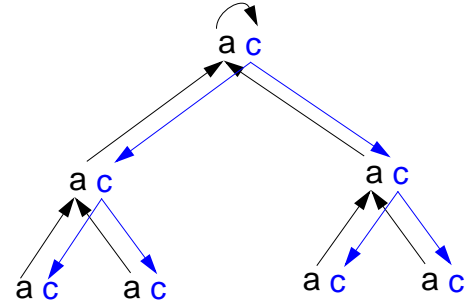
2 synchronization variables (flags) at each node:

arrived: all processes in a subtree have arrived,
is propagated upward

continue: all processes in a subtree may continue,
is propagated downward

disadvantage:

different code is needed for root, inner nodes, and for leafs



Lecture Parallel Programming WS 2014/2015 / Slide 45

Objectives:

Understand the tree barrier

In the lecture:

Explain

- the principle of 2 phases,
- the advantage of the distributed solution,

2 Rules for Synchronization Using Flags

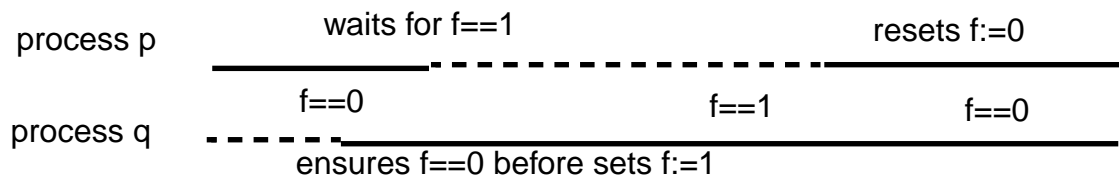
Flag for synchronization between exactly 2 processes

One process waits until the flag is set.
The other process sets the flag.

May be implemented by a monitor in Java.

Flag rules: 1. The process that waits for a flag resets it.
2. A flag that is set may not be set again before being reset.

Consequence: no state change will be lost.



Lecture Parallel Programming WS 2014/2015 / Slide 45a

Objectives:

Understand flag synchronization

In the lecture:

Explain

- the general flag rules.

Assignments:

- Design a Java class for flag synchronization between 2 processes. Ensure that the flag rules are obeyed.

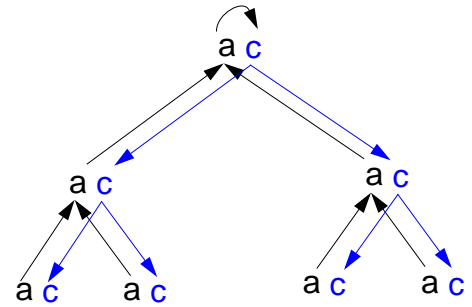
Distributed tree barrier implementation

2 synchronization variables (flags) at each node:

arrived: all processes in a subtree have arrived propagated upward

continue: all processes in a subtree may continue propagated downward

initially all flags are reset



code for an **inner** node:

```
execute this.task();
wait for left.arrived; reset left.arrived;
wait for right.arrived; reset right.arrived;
set this.arrived;
wait for this.continue; reset this.continue;
set left.continue;
set right.continue;
```

leaf

x

root

x

x

x

x

x

x

x

Lecture Parallel Programming WS 2014/2015 / Slide 45b

Objectives:

Understand the tree barrier

In the lecture:

Explain

- the different code for the 3 kinds of nodes,

Assignments:

- Write the code for the 3 kinds of nodes using objects of the flag class.

Symmetric, distributed barrier (dissemination)

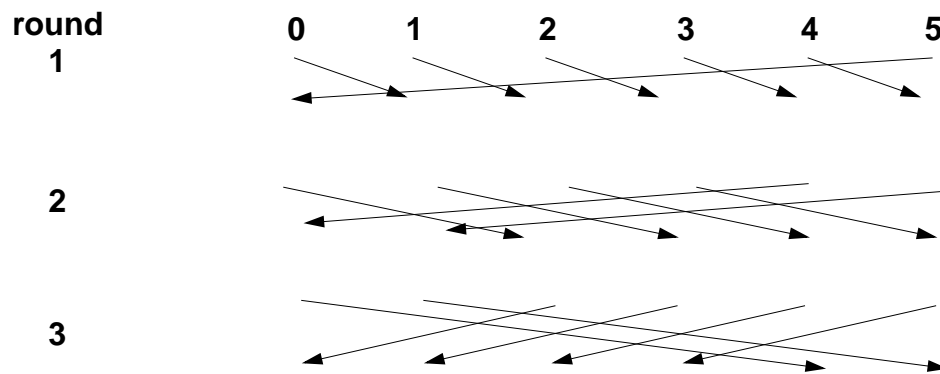
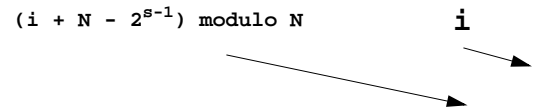
Processes **synchronize pairwise** in rounds with **doubled distances**.

N processes are synchronized after r rounds if $N \leq 2^r$

In round s

process i indicates its arrival and then waits

for the arrival of process $(i + N - 2^{s-1}) \bmod N$:



After r rounds each process is synchronized with each other. Proof idea: For each process i each other process occurs in a tree of processes which have synchronized (in)directly with i .

Lecture Parallel Programming WS 2014/2015 / Slide 46

Objectives:

Understand the dissemination barrier

In the lecture:

- Symmetric code for arbitrary many processes.
- Arc i to j in the diagram means j waits for arrival of i .
- show the synchronization for pairs.
- No cyclic waiting, because the arrival is indicated first, then the partner is waited for.
- After the last round all processes are synchronized, because for all processes p a binary tree exists s.t. p is its root, all processes are in that tree, the arcs are waiting pairs from the diagram forming pathes from the leaves to the root..

Questions:

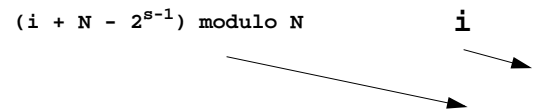
- Write the synchronization code.
- Show one of the binary trees.

Symmetric, distributed barrier: implementation

In round s

process i indicates its arrival and

waits for the arrival of process $(i + N - 2^{s-1}) \bmod N$



Code for each process:

```
execute this.task();

// synchronize:
s = 0;
while (N > 2s)
    s++;
    wait for f==0; set f=1;
    partner=p[(i + N - 2s-1) modulo N];
    wait partner.f; reset partner.f=0
```

Lecture Parallel Programming WS 2014/2015 / Slide 46a

Objectives:

Understand the dissemination barrier

In the lecture:

- Processes have to wait before they set AND before they reset the flag.
- Symmetric code for arbitrary many processes.

Questions:

- Write the synchronization code.
- Show one of the binary trees.

Prefix sums with barriers

```

class PrefixSum extends Thread
{ private int procNo;                // number of process
  private BarrierMonitor bm;         // barrier object

  public PrefixSum (int p, BarrierMonitor b)
  { procno = p; bm = b; }

  public void run ()
  { int addIt, dist = 1;              // distance
                                        // global arrays a and s
    s[procNo] = a[procNo];           // initialize result array
    bm.barrier();

    // invariant SUM: s[procNo] == a[procNo-dist+1]+...+a[procNo]
    while (dist < N)
    { if (procNo - dist >= 0)
      addIt = s[procNo - dist];      // value before overwritten
      bm.barrier();
      if (procNo - dist >= 0)
        s[procNo] += addIt;
      bm.barrier();
      dist = dist * 2;               // doubled distance
    } } }

```

Lecture Parallel Programming WS 2014/2015 / Slide 47

Objectives:

Examples for synchronization points

In the lecture:

Explain

- the invariant,
- the access of `s[procNo]`,
- the reasons for the 3 synchronization points.

Questions:

- Explain the reasons for the 3 synchronization points.

Prefix sums in a synchronous parallel programming model

Notation in Modula-2* with synchronous (and asynchronous) loops for parallel machines

```

VAR a, s, t: ARRAY [0..N-1] OF INTEGER;
VAR dist: CARDINAL;
BEGIN
  ...
  FORALL i: [0..N-1] IN SYNC                parallel loop in lock step
    s[i] := a[i];
  END;

  dist := 1;

  WHILE dist < N                            parallel loop in lock step
    FORALL i: [0..N-1] IN SYNC
      IF (i-dist) >= 0 THEN
        t[i] := s[i - dist];                implicit barrier
        s[i] := s[i] + t[i];                for each statement
      END
    END;
    dist := dist * 2;
  END
END

```

Lecture Parallel Programming WS 2014/2015 / Slide 48

Objectives:

Implicit barriers

In the lecture:

- Explain the language constructs.
- If expressions were evaluated in lock step, too, the array `t` could be omitted.
- The MasPar SIMD machine would be programmed similarly.

Questions:

- Explain the execution if values were not saved in `t[i]`.

Finding list ends: data parallel approach

input: int array link stores lists; link[i] contains the index of the successor or nil

output: int array last; last[i] contains the index of the last element of list link[i]

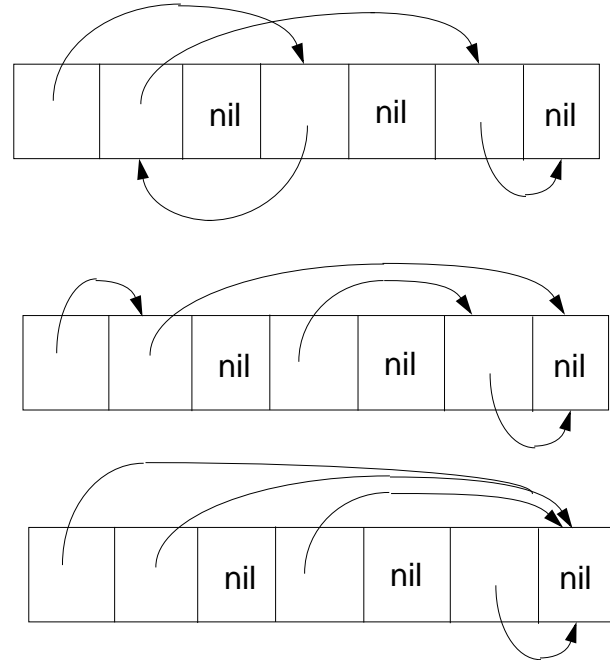
method: **worker process** i computes $last[i] = last[last[i]]$ in $\log N$ rounds

```

int d = 1;
last[i] = link[i];
barrier

while (d < N)
{
  int newlast = nil;
  if ( last[i] != nil &&
      last[last[i]] != nil)
    newlast = last[last[i]];
  barrier
  if (newlast != nil)
    last[i] = newlast;
  barrier
  d = 2*d;
}

```



last[i] points to the end of those lists which are not longer than d

Lecture Parallel Programming WS 2014/2015 / Slide 49

Objectives:

Data parallelism not only for arrays!

In the lecture:

Explain

- parallel scanning of lists,
- doubling distances for lists,
- $last[last[i]]$,
- that it is only useful if the ends of many long lists are searched.

Questions:

- Which role plays the distance d here?

5.2 / 6. Data Parallelism: Loop Parallelization

Regular loops on orthogonal data structures - parallelized for **data parallel** processors

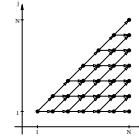
Development steps (automated by compilers):

- **nested loops** operating on **arrays**, sequential execution of iteration space

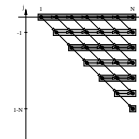
```

DECLARE B[0..N,0..N+1]
FOR I := 1 .. N
  FOR J := 1 .. I
    B[I,J] :=
      B[I-1,J]+B[I-1,J-1]
  END FOR
END FOR
  
```

- analyze **data dependences**
data-flow: definition and use of array elements

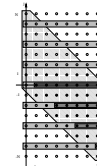


- **transform loops**
keep data dependences forward in time



- **parallelize inner loop(s)**
map to field or vector of processors

- **map arrays to processors**
such that many accesses are local,
transform index spaces



Lecture Parallel Programming WS 2014/2015 / Slide 50

Objectives:

Overview

In the lecture:

Explain

- Application area: scientific computations
- goals: execute inner loops in parallel with efficient data access
- transformation steps

Iteration space of loop nests

Iteration space of a loop nest of depth n :

- **n -dimensional space of integral points** (polytope)
- each point (i_1, \dots, i_n) represents an execution of the innermost loop body
- loop bounds are in general not known before run-time
- iteration need not have orthogonal borders
- iteration is elaborated sequentially

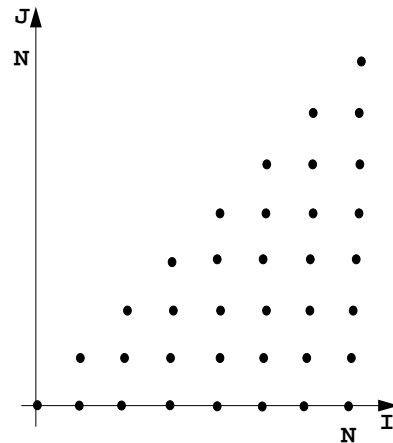
example:
computation of Pascal's triangle

```

DECLARE B[-1..N,-1..N]

FOR I := 0 .. N
  FOR J := 0 .. I
    B[I,J] :=
      B[I-1,J]+B[I-1,J-1]
  END FOR
END FOR

```



Lecture Parallel Programming WS 2014/2015 / Slide 51

Objectives:

Understand the notion of iteration space

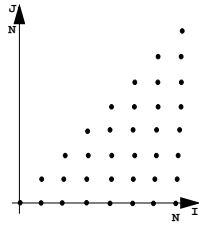
In the lecture:

- Explain the iteration space of the example.
- Show the order of elaboration of the iteration space.
- If the step size is greater than 1 the iteration space has gaps - the polytope is not convex.

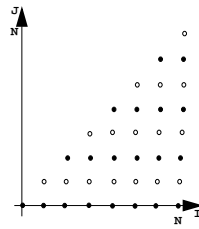
Questions:

- Draw an iteration space that has step size 3 in one dimension.

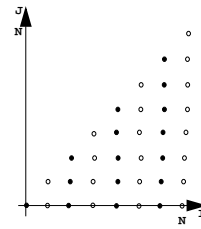
Examples for Iteration spaces of loop nests



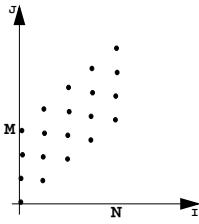
```
FOR I := 0 .. N
  FOR J := 0 .. I
```



```
FOR I := 0 .. N
  FOR J := 0..I BY 2
```

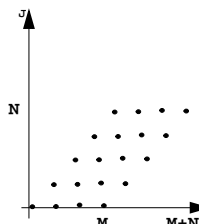


```
FOR I := 0..N BY 2
  FOR J := 0 .. I
```



```
FOR I := 0 .. N
  FOR J := I..I+M
```

$M = 3, N = 4$



```
FOR I := 0 .. M+N
  FOR J := max(0, I-M)..
    min(I, N)
```

Lecture Parallel Programming WS 2014/2015 / Slide 51a

Objectives:

Relate loop nests to iteration spaces

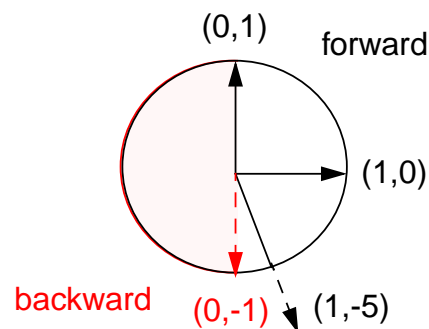
In the lecture:

- Explain the iteration spaces of the examples

Data Dependences in Iteration Spaces

Data dependence from iteration point i_1 to i_2 :

- Iteration i_1 computes a value that is used in iteration i_2 (flow dependence)
- relative **dependence vector**
 $\mathbf{d} = \mathbf{i}_2 - \mathbf{i}_1 = (i_{2_1} - i_{1_1}, \dots, i_{2_n} - i_{1_n})$
 holds for all iteration points except at the border
- Flow-dependences can **not be directed against the execution order**, can not point backward in time: each dependence vector must be **lexicographically positive**, i. e. $\mathbf{d} = (0, \dots, 0, d_i, \dots)$, $d_i > 0$

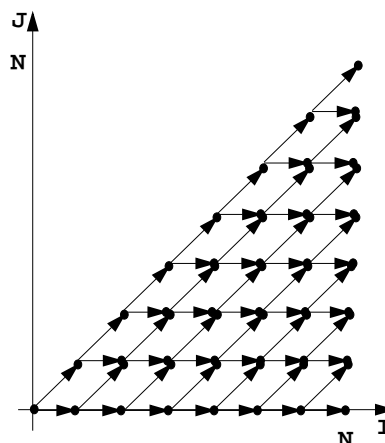


Example:

Computation of Pascal's triangle

```

DECLARE B[-1..N,-1..N]
FOR I := 0 .. N
  FOR J := 0 .. I
    B[I,J] :=
      B[I-1,J]+B[I-1,J-1]
  END FOR
END FOR
  
```



Lecture Parallel Programming WS 2014/2015 / Slide 52

Objectives:

Understand dependences in loops

In the lecture:

Explain:

- Vector representation of dependences,
- examples,
- admissible directions graphically

Questions:

- Show different dependence vectors and array accesses in a loop body which cause dependences of given vectors.

Loop Transformation

The **iteration space** of a loop nest is transformed to **new coordinates**. Goals:

- **execute innermost loop(s) in parallel**
- improve **locality** of data accesses;
in space: use storage of executing processor,
in time: reuse values stored in cache
- **systolic** computation and communication scheme

Data dependences must **point forward in time**, i.e. **lexicographically positive** and **not within parallel dimensions**

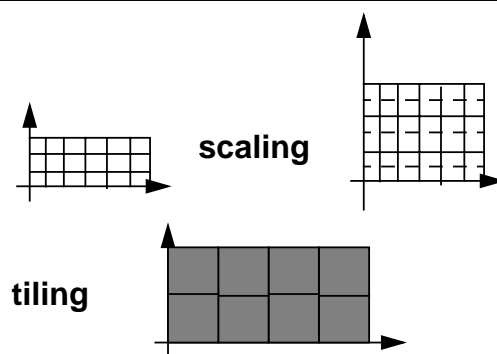
linear basic transformations:

- **Skewing**: add iteration count of an outer loop to that of an inner one
- **Reversal**: flip execution order for one dimension
- **Permutation**: exchange two loops of the loop nest

SRP transformations (next slides)

non-linear transformations, e. g.

- **Scaling**: stretch the iteration space in one dimension, causes gaps
- **Tiling**: introduce **additional inner loops** that **cover tiles** of fixed size



Lecture Parallel Programming WS 2014/2015 / Slide 53

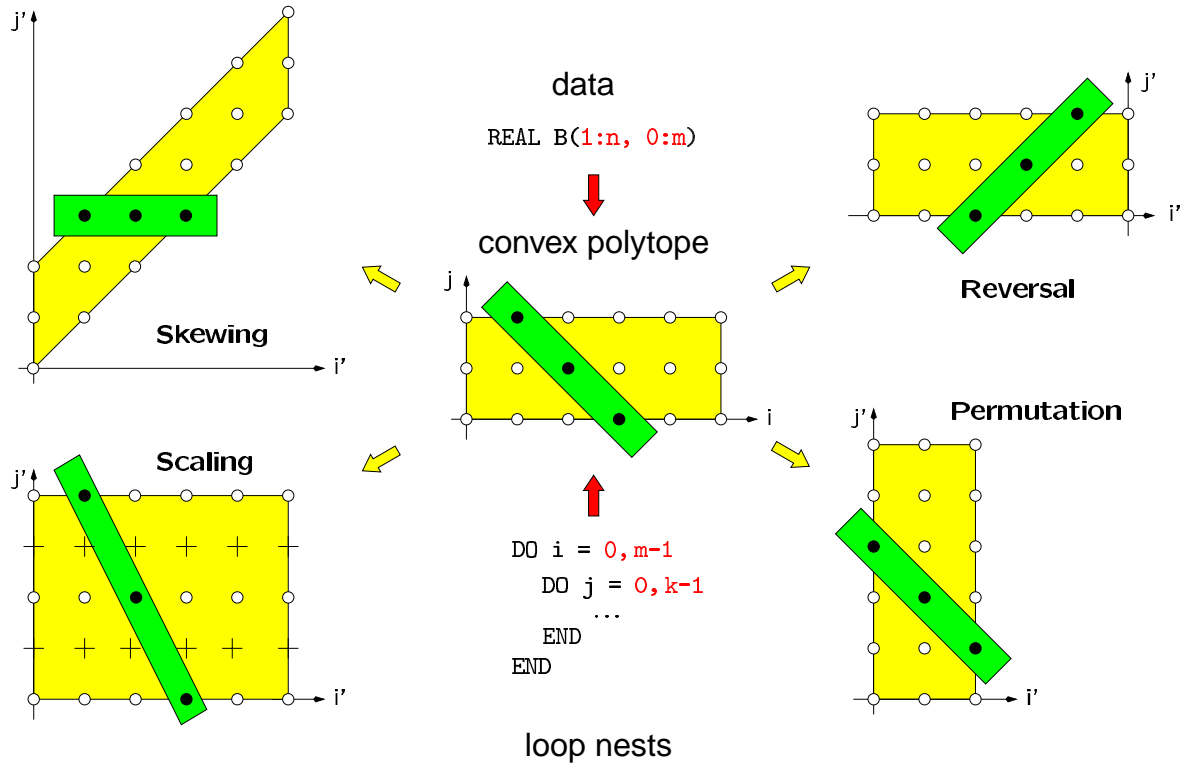
Objectives:

Overview

In the lecture:

- Explain the goals.
- Show admissible directions of dependences.
- Show diagrams for the transformations.

Transformations of



Lecture Parallel Programming WS 2014/2015 / Slide 54

Objectives:

Visualize the transformations

In the lecture:

- Give concrete loops for the diagrams.
- Show how the dependence vectors are transformed.
- Skewing and scaling do not change the order of execution; hence, they are always applicable.

Questions:

- Give dependence vectors for each transformation, which are still valid after the transformation.

Transformations defined by matrices

Transformation matrices: systematic transformation, check dependence vectors

$$\text{Reversal} \quad \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} i \\ -j \end{pmatrix} = \begin{pmatrix} i' \\ j' \end{pmatrix}$$

$$\text{Skewing} \quad \begin{pmatrix} 1 & 0 \\ f & 1 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} i \\ f*i+j \end{pmatrix} = \begin{pmatrix} i' \\ j' \end{pmatrix}$$

$$\text{Permutation} \quad \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} j \\ i \end{pmatrix} = \begin{pmatrix} i' \\ j' \end{pmatrix}$$

Lecture Parallel Programming WS 2014/2015 / Slide 55

Objectives:

Understand the matrix representation

In the lecture:

- Explain the principle.
- Map concrete iteration points.
- Map dependence vectors.
- Show combinations of transformations.

Questions:

- Give more examples for skewing transformations.

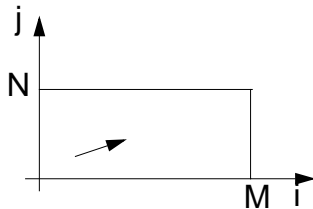
Reversal

Iteration count of one loop is negated, that dimension is enumerated backward

general transformation matrix

$$\begin{pmatrix} 1 & & & & & \\ & \dots & & & & \\ & & 1 & & & 0 \\ & & & -1 & & \\ 0 & & & & 1 & \dots \\ & & & & & & 1 \end{pmatrix}$$

```
for i = 0 to M
  for j = 0 to N
    ...
```



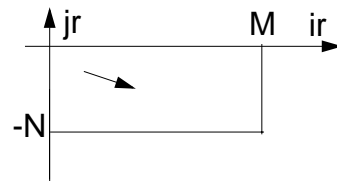
2-dimensional:

$$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} i \\ -j \end{pmatrix} = \begin{pmatrix} ir \\ jr \end{pmatrix}$$

old new

```
for ir = 0 to M
  for jr = -N to 0
    ...
```

original
transformed



Lecture Parallel Programming WS 2014/2015 / Slide 55a

Objectives:

Understand reversal transformation

In the lecture:

- Explain the effect of reversal transformation.
- Explain the notation of the transformation matrix.
- There may be no dependences in the direction of the reversed loop - they would point backward after the transformation.

Questions:

- Show an example where reversal enables loop fusion.

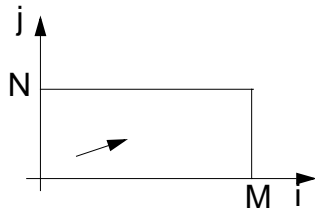
Skewing

The **iteration count** of an outer loop is **added to the count of an inner loop**;
iteration space is shifted; **execution order** of iteration points **remains unchanged**

general transformation matrix:

$$\begin{pmatrix} 1 & & & & & \\ & \dots & & & & 0 \\ & & 1 & & & \\ & f & 1 & & & \\ & & & 1 & & \dots \\ & 0 & & & & 1 \end{pmatrix}$$

```
for i = 0 to M
  for j = 0 to N
    ...
```



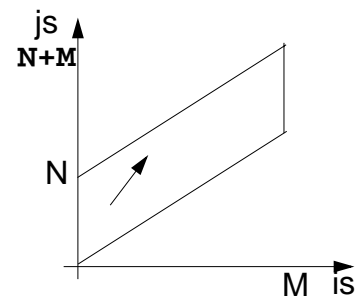
original

2-dimensional:

$$\begin{matrix} & & & \text{loop variables} \\ & & & \text{old} & & \text{new} \\ \begin{pmatrix} 1 & 0 \\ f & 1 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} i \\ f*i+j \end{pmatrix} = \begin{pmatrix} is \\ js \end{pmatrix} \end{matrix}$$

```
for is = 0 to M
  for js = f*is to N+f*is
    ...
```

transformed



Lecture Parallel Programming WS 2014/2015 / Slide 55b

Objectives:

Understand skewing transformation

In the lecture:

- Explain the effect of a skewing transformation.
- Skewing is always applicable.
- Skewing can enable loop permutation

Questions:

- Show an example where skewing enables loop permutation.

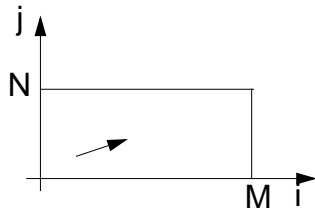
Permutation

Two loops of the loop nest are interchanged; the iteration space is flipped; the **execution order** of iteration points **changes**; new dependence vectors must be legal.

general transformation matrix:

$$\begin{pmatrix} 1 & & & & \\ & 0 & 1 & & \\ & & 1 & & \\ & 1 & & 0 & \\ & & & & \dots \\ & 0 & & & & 1 \end{pmatrix}$$

```
for i = 0 to M
  for j = 0 to N
    ...
```



original

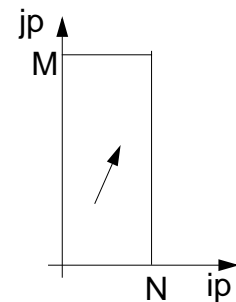
2-dimensional:

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} j \\ i \end{pmatrix} = \begin{pmatrix} ip \\ jp \end{pmatrix}$$

old new

```
for ip = 0 to N
  for jp = 0 to M
    ...
```

transformed



Lecture Parallel Programming WS 2014/2015 / Slide 55c

Objectives:

Understand loop permutation

In the lecture:

- Explain the effect of loop permutation.
- Show effect on dependence vectors.
- Permutation often yields a parallelizable innermost loop.

Questions:

- Show an example where permutation yields a parallelizable innermost loop.

Use of Transformation Matrices

- Transformation matrix T defines **new iteration counts** in terms of the old ones: $T * i = i'$

e. g. Reversal
$$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} i \\ -j \end{pmatrix} = \begin{pmatrix} i' \\ j' \end{pmatrix}$$

- Transformation matrix T transforms old **dependence vectors** into new ones: $T * d = d'$

e. g.
$$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} * \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

- inverse Transformation matrix T^{-1} defines **old iteration counts** in terms of new ones, for transformation of index expressions in the loop body: $T^{-1} * i' = i$

e. g.
$$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} * \begin{pmatrix} i' \\ j' \end{pmatrix} = \begin{pmatrix} i' \\ -j' \end{pmatrix} = \begin{pmatrix} i \\ j \end{pmatrix}$$

- concatenation of transformations** first T_1 then T_2 : $T_2 * T_1 = T$

e. g.
$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} * \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$$

Lecture Parallel Programming WS 2014/2015 / Slide 56

Objectives:

Learn to Use the matrices

In the lecture:

- Explain the 4 uses with examples.
- Transform a loop completely.

Questions:

- Why do the dependence vectors change under a transformation, although the dependence between array elements remains unchanged?

Inequalities Describe Loop Bounds

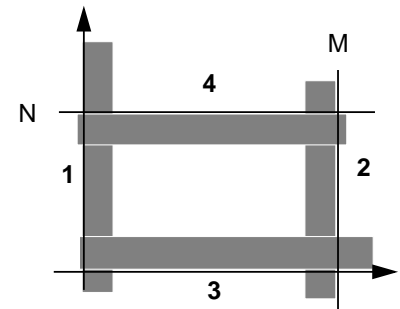
The bounds of a loop nest are described by a **set of linear inequalities**.
Each **inequality separates the space** in „inside and outside of the iteration space“:

$$\mathbf{B} * \mathbf{i} \leq \mathbf{c}$$

$$\begin{pmatrix} -1 & 0 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} \leq \begin{pmatrix} 0 \\ M \\ 0 \\ N \end{pmatrix}$$

example 1

- 1 $-i \leq 0$
- 2 $i \leq M$
- 3 $-j \leq 0$
- 4 $j \leq N$

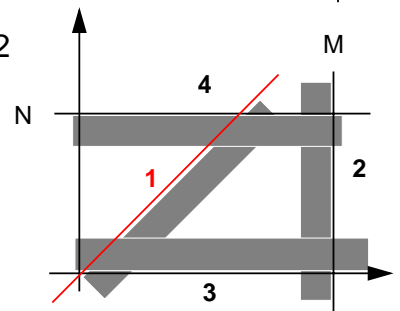


$$\begin{pmatrix} -1 & 1 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} \leq \begin{pmatrix} 0 \\ M \\ 0 \\ N \end{pmatrix}$$

example 2

- 1 $-i + j \leq 0$
- 2 $i \leq M$
- 3 $-j \leq 0$
- 4 $j \leq N$

transformed



positive factors represent **upper** bounds
negative factors represent **lower** bounds

$$1, 4: j \leq \min(i, N)$$

$$3: 0 \leq j$$

$$1+3: 0 \leq i$$

$$2: i \leq M$$

Lecture Parallel Programming WS 2014/2015 / Slide 56a

Objectives:

Understand representation of bounds

In the lecture:

- Explain matrix notation.
- Explain graphic interpretation.
- There can be arbitrary many inequalities.

Questions:

- Give the representations of other iteration spaces.

Transformation of Loop Bounds

The inverse of a transformation matrix T^{-1} transforms a set of inequalities: $B * T^{-1} i' \leq c$

$$\begin{array}{cc} \text{skewing} & \text{inverse} \\ \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} & \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix} \end{array} \quad B \quad T^{-1} \quad B * T^{-1}$$

$$\begin{pmatrix} -1 & 0 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{pmatrix} * \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix} = \begin{pmatrix} -1 & 0 \\ 1 & 0 \\ 1 & -1 \\ -1 & 1 \end{pmatrix}$$

example 1
new bounds:

$$B * T^{-1} \quad i' \quad c$$

$$\begin{pmatrix} -1 & 0 \\ 1 & 0 \\ 1 & -1 \\ -1 & 1 \end{pmatrix} * \begin{pmatrix} i' \\ j' \end{pmatrix} \leq \begin{pmatrix} 0 \\ M \\ 0 \\ N \end{pmatrix}$$

- 1 $-i' \leq 0$
- 2 $i' \leq M$
- 3 $i' - j' \leq 0$
- 4 $-i' + j' \leq N$

Lecture Parallel Programming WS 2014/2015 / Slide 56b

Objectives:

Understand the transformation of bounds

In the lecture:

- Explain how the inequalities are transformed

Questions:

- Compute further transformations of bounds.

Example for Transformation and Parallelization of a Loop

```
for i = 0 to N
  for j = 0 to M
    a[i, j] = (a[i, j-1] + a[i-1, j]) / 2;
```

Parallelize the above loop.

1. Draw the iteration space.
2. Compute the dependence vectors and draw examples of them into the iteration space. Why can the inner loop not be executed in parallel?
3. Apply a skewing transformation and draw the iteration space.
4. Apply a permutation transformation and draw the iteration space. Explain why the inner loop now can be executed in parallel.
5. Compute the matrix of the composed transformation and use it to transform the dependence vectors.
6. Compute the inverse of the transformation matrix and use it to transform the index expressions.
7. Specify the loop bounds by inequalities and transform them by the inverse of the transformation matrix.
8. Write the complete loops with new loop variables i_p and j_p and new loop bounds.

Lecture Parallel Programming WS 2014/2015 / Slide 56c

Objectives:

Exercise the method for an example

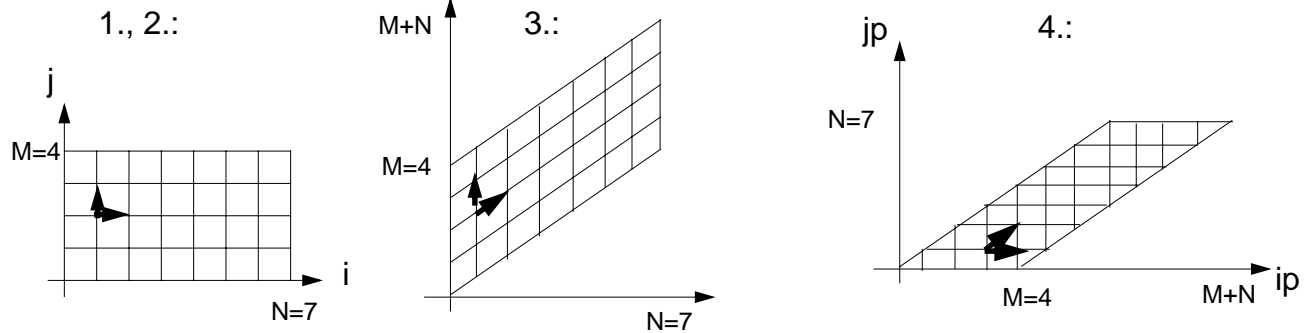
In the lecture:

- Explain the steps of the transformation.
- Solution on C-5.22

Questions:

- Are there other transformations that lead to a parallel inner loop?

Solution of the Transformation and Parallelization Example



5.:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

6.: Inverse

$$\begin{pmatrix} 0 & 1 \\ 1 & -1 \end{pmatrix}$$

7. Bounds:

	B	c	$B * T^{-1}$		
orig.:	$\begin{pmatrix} -1 & 0 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 0 \\ N \\ 0 \\ M \end{pmatrix}$	$\begin{pmatrix} 0 & -1 \\ 0 & 1 \\ -1 & 1 \\ 1 & -1 \end{pmatrix}$	1 $-jp \leq 0$	1, 3 $\Rightarrow 0 \leq ip$
				2 $jp \leq N$	2, 4 $\Rightarrow ip \leq M+N$
				3 $-ip+jp \leq 0$	1, 4 $\Rightarrow \max(0, ip-M) \leq jp$
				4 $ip - jp \leq M$	2, 3 $\Rightarrow jp \leq \min(ip, N)$

8. for $ip = 0$ to $M+N$
 for $jp = \max(0, ip-M)$ to $\min(ip, N)$
 $a[jp, ip-jp] = (a[jp, ip-jp-1] + a[jp-1, ip-jp]) / 2;$

Lecture Parallel Programming WS 2014/2015 / Slide 56d

Objectives:

Solution for C-60

In the lecture:

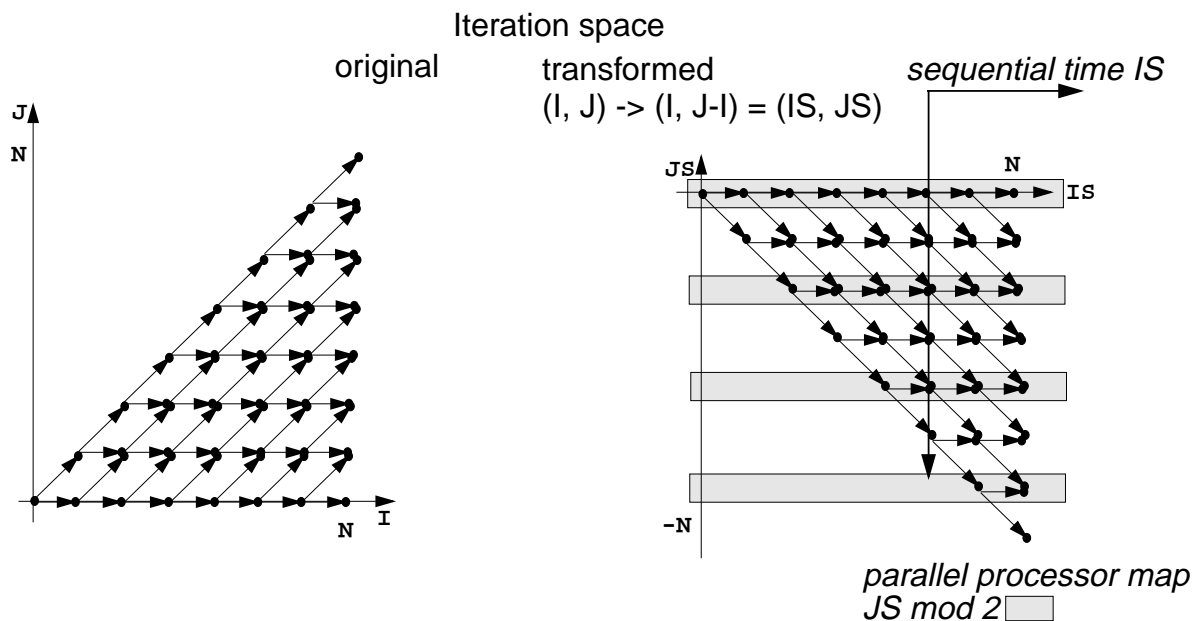
Explain

- the bounds of the iteration spaces,
- the dependence vectors,
- the transformation matrix and its inverse,
- the conditions for being parallelizable,
- the transformation of the index expressions
- the transformation of the loop bounds.

Questions:

- Describe the transformation steps.

Transformation and Parallelization



```

DECLARE B[-1..N,-1..N]
FOR I := 0 .. N
  FOR J := 0 .. I
    B[I,J] :=
      B[I-1,J]+B[I-1,J-1]
  END FOR
END FOR

```

```

DECLARE B[-1..N,-1..N]
FOR IS := 0.. N
  FOR JS := -IS .. 0
    B[IS,JS+IS] :=
      B[IS-1,JS+IS]+B[IS-1,JS-1+IS]
  END FOR
END FOR

```

Lecture Parallel Programming WS 2014/2015 / Slide 57

Objectives:

Example for parallelization

In the lecture:

- Explain skewing transformation: $f = -1$
- Inner loop in parallel.
- Explain the time and processor mapping.
- $\bmod 2$ folds the arbitrary large loop dimension on a fixed number of 2 processors.

Questions:

- Give the matrix of this transformation.
- Use it to compute the dependence vectors, the index expressions, and the loop bounds.

Data Mapping

Goal:

Distribute array elements over processors, such that as many **accesses as possible are local**.

Index space of an array:

n-dimensional space of integral index points (polytope)

- **same properties as iteration space**
- same mathematical model
- same **transformations** are applicable (Skewing, Reversal, Permutation, ...)
- **no restrictions** by data dependences

Lecture Parallel Programming WS 2014/2015 / Slide 58

Objectives:

Reuse model of iteration spaces

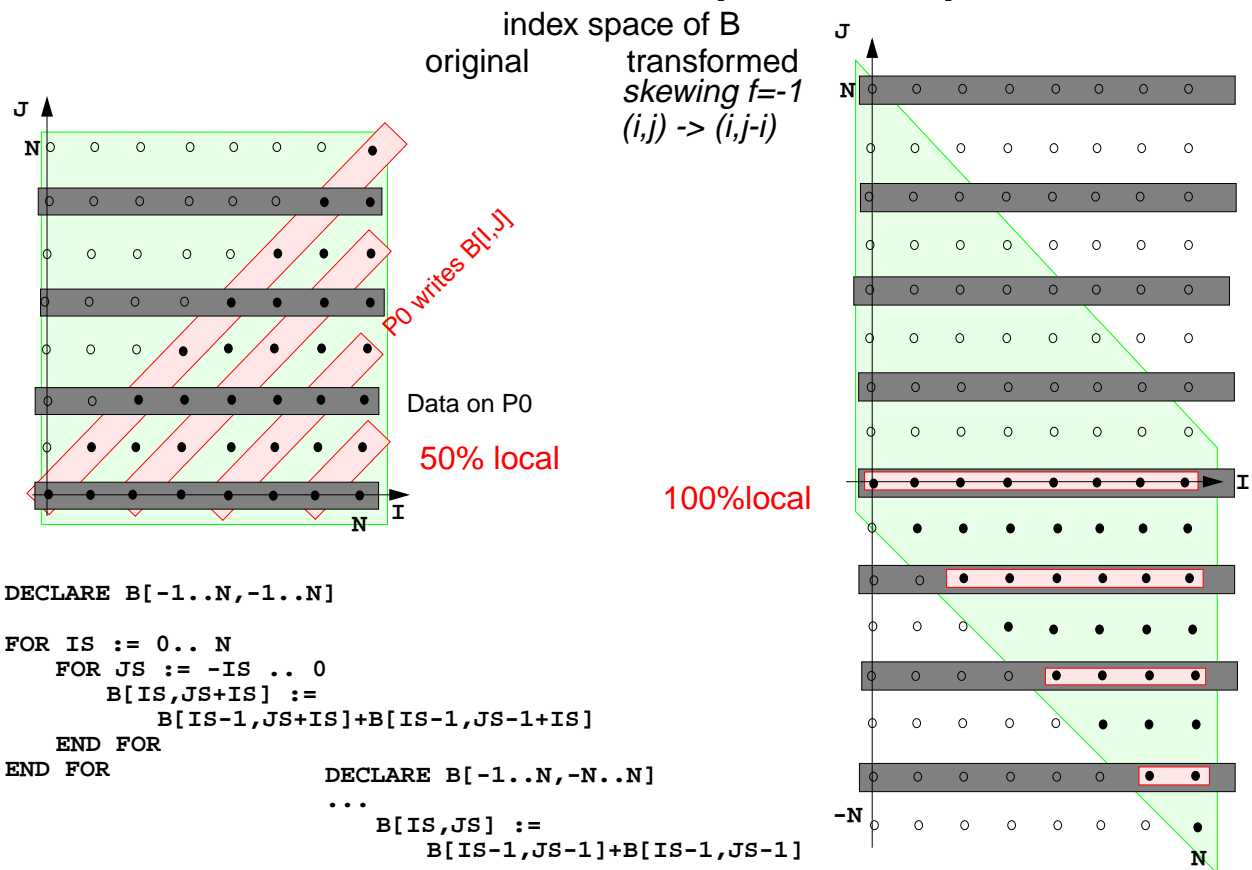
In the lecture:

Explain, using examples of index spaces

Questions:

- Draw an index space for each of the 3 transformations.

Data distribution for parallel loops



Lecture Parallel Programming WS 2014/2015 / Slide 59

Objectives:

The gain of an index transformation

In the lecture:

Explain

- local and non-local accesses,
- the index transformation,
- the gain of locality,
- unused memory because of skewing.

Questions:

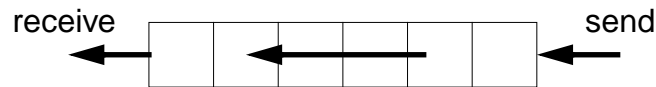
- How do you compute the index transformation using a transformation matrix?

7. Asynchronous Message Passing

Processes send and receive messages via channels

Message: value of a composed data type or object of a class

Channel: **queue** of arbitrary length, containing messages



operations on a channel:

- **send (b):** adds the message b to the end of the queue of the channel; does **not block** the executing process (in contrast to synchronous send)
- **receive():** yields the oldest message and deletes it from the channel; **blocks** the executing process as long as the channel is empty.
- **empty():** yields true, if the channel is empty; the result is **not necessarily up-to-date**.

send and **receive** are executed under mutual exclusion.

Lecture Parallel Programming WS 2014/2015 / Slide 60

Objectives:

Understand channels

In the lecture:

Explain:

- non-blocking send requires a channel;
- non-blocking send is the important difference between asynchronous and synchronous message passing;
- how to use results of empty();
- for tight synchronization of processes several channels are needed.

Questions:

- Why does a channel need a queue?
- Why may the result of empty() be not upto date?

Channels implemented in Java

```

public class Channel
{
    // implementation of a channel using a queue of messages
    private Queue msgQueue;

    public Channel ()
    { msgQueue = new Queue (); }

    public synchronized void send (Object msg)
    { msgQueue.enqueue (msg); notify(); } // wake a receiving process

    public synchronized Object receive ()
    {
        while (msgQueue.empty())
            try { wait(); } catch (InterruptedException e) {}
        Object result = msgQueue.front(); // the queue is not empty
        msgQueue.dequeue();
        return result;
    }

    public boolean empty ()
    { return msgQueue.empty (); }
}

```

All waiting processes wait for the same condition => notify() is sufficient.

After a notify-call a new receive-call may have stolen the only message => wait loop is needed

Lecture Parallel Programming WS 2014/2015 / Slide 61

Objectives:

Understand the channel implementation

In the lecture:

- explain the mutual exclusion;
- explain why the result of need not be up to date - even if Channel.empty would be synchronized;
- argue why notify() is sufficient, but a wait loop is needed.

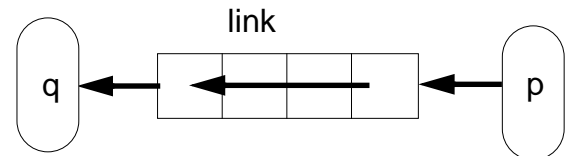
Questions:

- Where do you know this synchronization pattern from?

Processes and channels

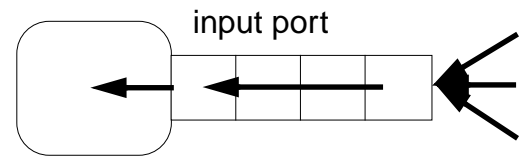
link:

one sender is connected to **one receiver**;
e. g. processes form chains of transformation steps (pipeline)



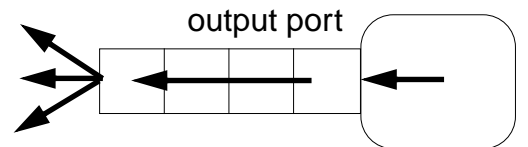
input port of a process:

many senders - one receiver;
channel belongs to the receiving process;
e. g. a server process receives tasks from several client processes



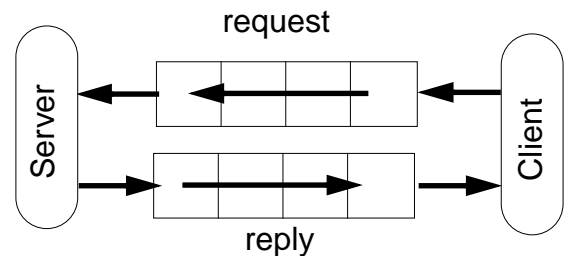
output port of a process:

one sender - many receivers;
channel belongs to the sending process;
e. g. a process distributes tasks to many servers (unusual structure)



pair of request and reply channels;

one process requests - the others replies;
tight synchronization,
e. g. between client and server



Lecture Parallel Programming WS 2014/2015 / Slide 62

Objectives:

Identify channel structures

In the lecture:

Explain applications of the structures

Termination conditions

When system of processes terminates the following **conditions** must hold:

1. **All channels are empty.**
2. **No processes are blocked on a receive operation.**
3. **All processes are terminated.**

Otherwise the **system state is not well-defined**, e.g. task is not completed, some operations are pending.

Problem:

In general, the processes **do not know the global system state**.

Lecture Parallel Programming WS 2014/2015 / Slide 62a

Objectives:



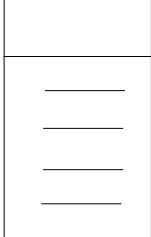
Final clean-up

In the lecture:

The conditions are explained.

Message structures

A **message object** may have arbitrary structure suitable for the **particular purpose**:

	empty	synchronization only
	kind	different kinds of messages, without data e. g. signal different kinds of events
	kind argument vector	different kinds of messages with data e. g. number and or identities of resources special case: a channel on which the sender expects a reply

Operations on messages:

constructors

setKind (k), getKind ()

setArg (ind, val), getArg (ind), getArgList ()

Lecture Parallel Programming WS 2014/2015 / Slide 63

Objectives:

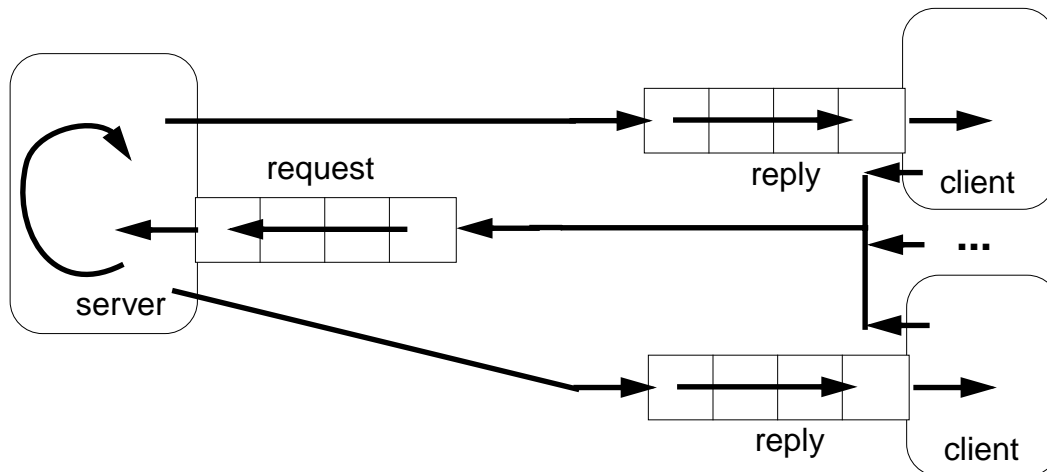
Message structures for different purposes

In the lecture:

Explain the use of different message structures

Client / server: basic channel structure

One server process responds to requests of several client processes



request channel:

input port of the server

reply channel:

one for each client (input port),
may be sent to the server included in the request message

Application: server distributes data or work packages on requests

Lecture Parallel Programming WS 2014/2015 / Slide 64

Objectives:

Understand the channel structure

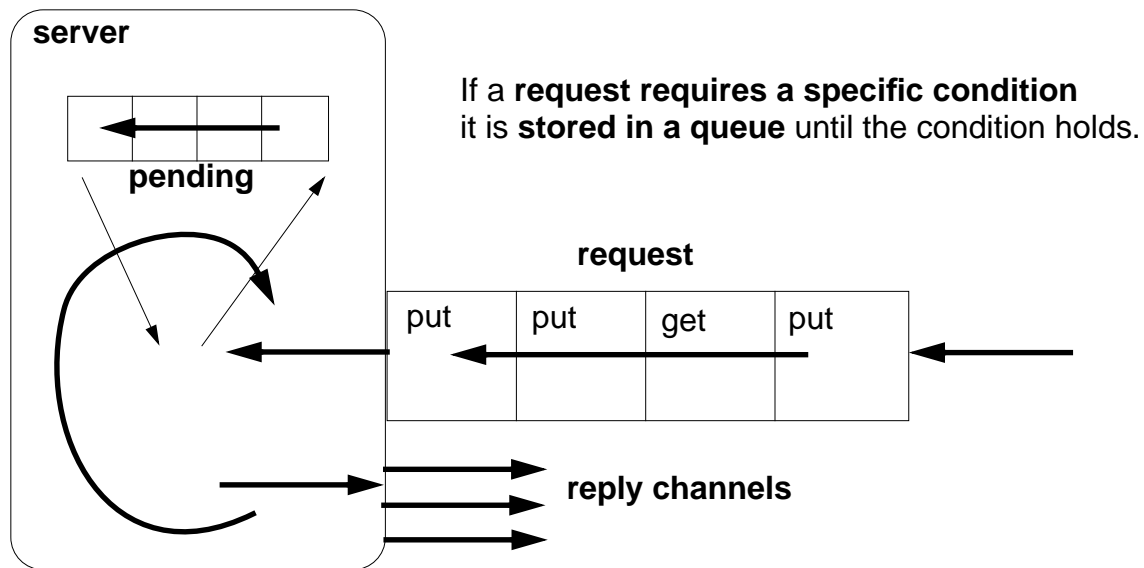
In the lecture:

Explain how

- the channels are used,
- channels are communicated,
- such a system is terminated: stop sending requests; let first the server and then the clients empty their channels.

Server processes: different kinds of operations

Different requests (operations) are represented by different kinds of messages.



The server processes the requests **strictly sequentially**; thus, it is guaranteed that critical sections are **not executed interleaved**.

Termination: terminate clients, empty channel, empty queue.

Lecture Parallel Programming WS 2014/2015 / Slide 65

Objectives:

Understand the structure of a server process

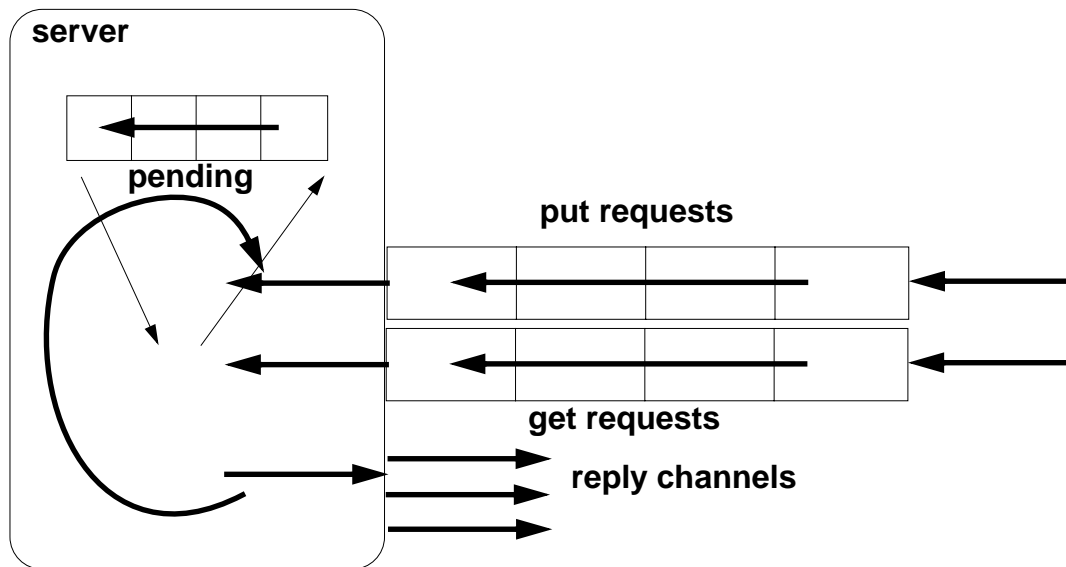
In the lecture:

- Explain the loop for execution of operations.
- Explain why requests are stored.
- Explain why operations are executed under mutual exclusion.

Questions:

- Design a server that implements a counting semaphore, which can be used to synchronize many processes.
- How can the monitors of PPJ-19 and following, be transformed into such a server?

Different kinds of operations on different channels



Server must not block on an empty input port while another port may be non-empty:

```
while (running) {
    if (!putPort.empty()) { msg = putPort.receive(); ... }
    if (!getPort.empty()) { msg = getPort.receive(); ... }
    if (!pending.empty()) { msg = pending.dequeue(); ... }
}
```

Lecture Parallel Programming WS 2014/2015 / Slide 66

Objectives:

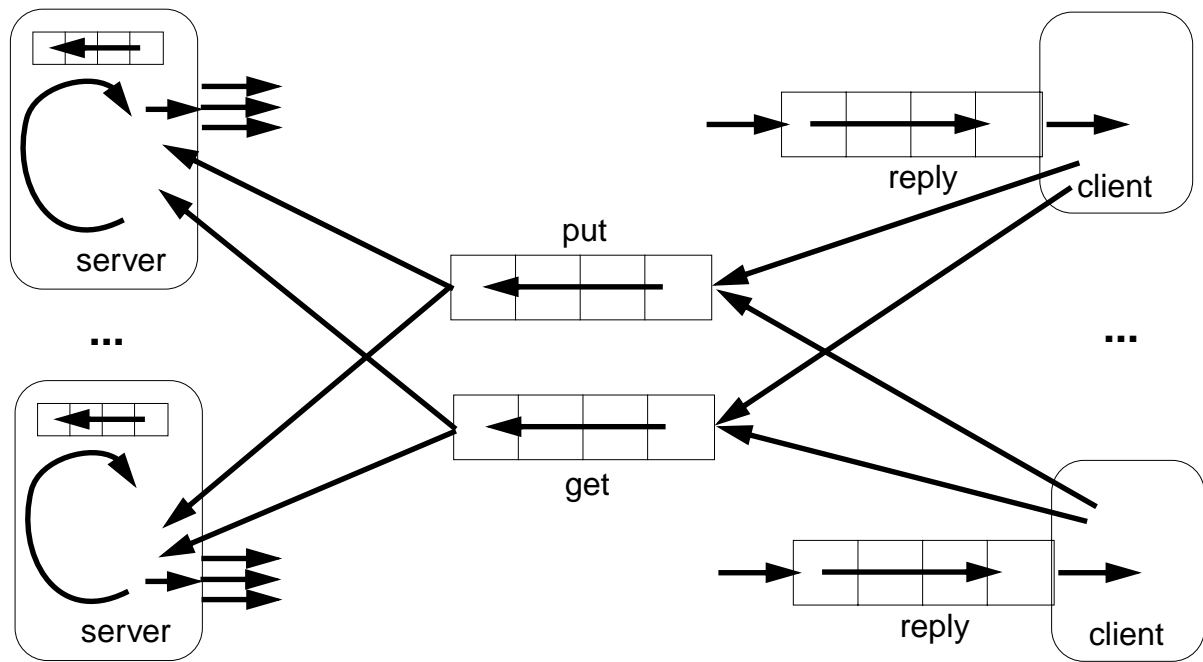
Compare to the one-channel structure

In the lecture:

Explain how channels are checked.

Several servers

Several server processes, several client processes, several request channels



Termination: empty request channels, empty queues, empty reply channels

Caution: a receive on a channel may block a server!

Lecture Parallel Programming WS 2014/2015 / Slide 67

Objectives:

Multi server structure

In the lecture:

- Parallelism is increased by several servers.
- Messages contain their reply channels.
- Explain termination.

Receive without blocking

If several processes receive from a channel `ch`, then the check

```
if (!ch.empty()) msg = ch.receive();
```

may block.

That is not acceptable when several channels have to be checked in turn.

Hence, a new non-blocking channel method is introduced:

```
public class Channel
{
    ...
    public synchronized Object receiveMsgOrNull ()
    {
        if (msgQueue.empty()) return null;
        Object result = msgQueue.front();
        msgQueue.dequeue();
        return result;
    }
}
```

Checking several channels:

```
while (msg == null)
{
    if ((msg = ch1.receiveMsgOrNull()) == null)
        if ((msg = ch2.receiveMsgOrNull()) == null)
            Thread.sleep (500);
}
```

Lecture Parallel Programming WS 2014/2015 / Slide 68

Objectives:

Avoid receive on empty channel

In the lecture:

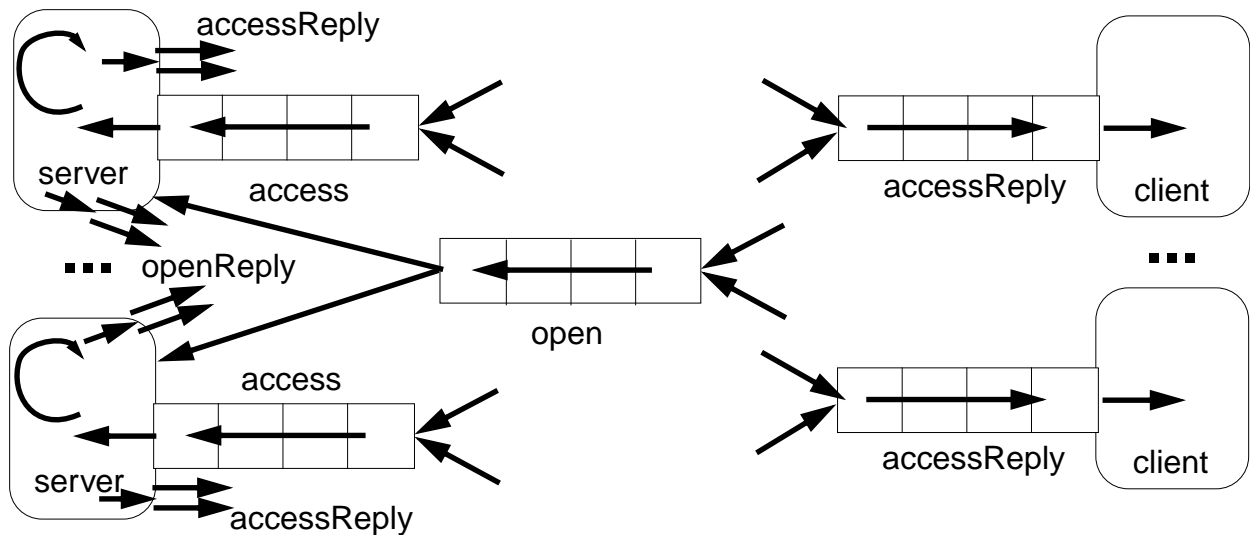
Explain:

- Multi servers check common channels.
- A false result of `empty()` may not be up to date when the `receive()` is executed.
- Hence, an atomic operation is needed.

Conversation sequences between client and server

Example for an **application pattern** is „file servers“:

- **several equivalent servers** respond to requests of **several clients**
- a client sends an **opening request** on a **channel common** for all servers (**open**)
- one server commits to the task; it then leads a conversation with the client according to a **specific protocol**, e. g.
(**open openReply**) ((**read readReply**) | (**write writeReply**))* (**close closeReply**)
- **reply channels** are contained in the **open** and **openReply** messages.



© 2015 bei Prof. Dr. Uwe Kastens

Lecture Parallel Programming WS 2014/2015 / Slide 69

Objectives:

Typical client/server paradigm

In the lecture:

- Explain the channel structure.
- The server sends its reply channel to the client, too.
- Explain the central server loop.

Active monitor (server) vs. passive monitor

active monitor

active process

request - reply via channels

kinds of messages and/or
different channels

requests are handled
sequentially

queue of pending requests
replies are delayed

may cooperate on the
same request channels

1. program structure

2. client communication

3. server operations

4. mutual exclusion

5. delayed service

6. multiple servers

passive monitor

passive program module

calls of entry procedures

entry procedures

guaranteed for entry procedure
calls

client processes are blocked
condition variables, wait - signal

multiple monitors are not related

Lecture Parallel Programming WS 2014/2015 / Slide 70

Objectives:

Compare monitor structures

In the lecture:

Explain the differences

8. Messages in Distributed Systems

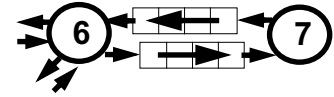
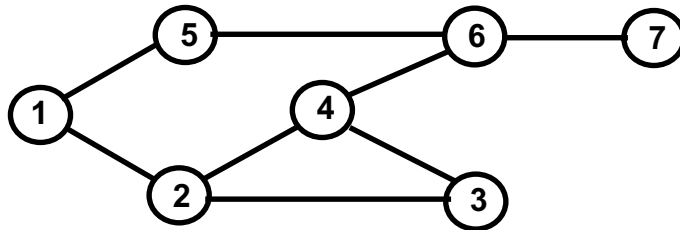
Distributed processes: Broadcast in a net of processors

Net: bi-directional graph, connected, irregular structure;

node: a process

edge: a pair of links (channels) which connect two nodes in both directions

A node knows only its direct neighbours and the links to and from each neighbour:



Broadcast:

A message is sent from an initiator node such that it reaches every node in the net.
Finally all channels have to be empty.

Problems:

- graph may have cycles
- nodes do not know the graph beyond their neighbours

Lecture Parallel Programming WS 2014/2015 / Slide 71

Objectives:

Understand the task

In the lecture:

Explain

- the task,
- why the limited knowledge is a problem,
- why it is non-trivial to empty the channels.

Broadcast method

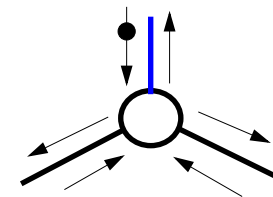
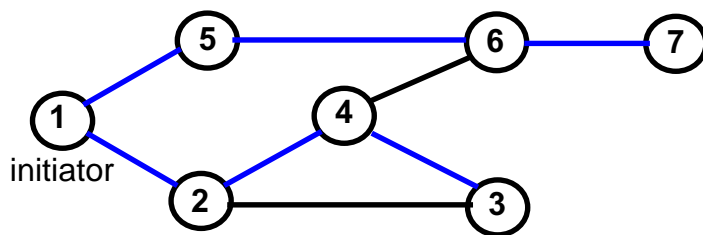
Method (for all nodes but the initiator node):

1. The node waits for a message on its incoming links.
2. After having **received the first message** it sends a **copy to all of its n neighbours** - including to the sender of the first message
3. The node then receives **n-1 redundant messages** from the remaining neighbours

All nodes are finally reached because of (2).

All channels are finally empty because of (3).

The connection to the sender of the first message is considered to be an edge of a **spanning tree** of the graph. That information may be used to simplify subsequent broadcasts.



total number of messages: $2 * |\text{edges}|$

Lecture Parallel Programming WS 2014/2015 / Slide 72

Objectives:

Understand the broadcast method

In the lecture:

Explain

- the method,
- that a node knows only one of its spanning tree edges.

Questions:

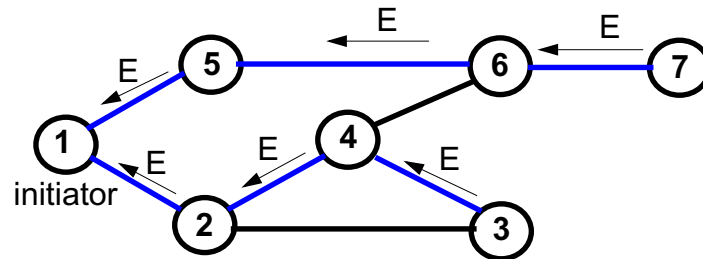
- Explain why a copy is send to the sender of the first message, too.

Probe and echo in a net

Task: An initiator requests combined **information from all nodes** in the graph (**probe**).
The information is **combined** on its way through the net (**echo**);
e. g. sum of certain values local to each node, topology of the graph, some global state.

Method (roughly):

- distribute the probes like a broadcast,
- let the first reception determine a spanning tree,
- return the echoes on the spanning tree edges.



Lecture Parallel Programming WS 2014/2015 / Slide 73

Objectives:

Understand the probe/echo task

In the lecture:

Explain

- the task and the method,
- that the nodes do not know their outgoing spanning tree edges.

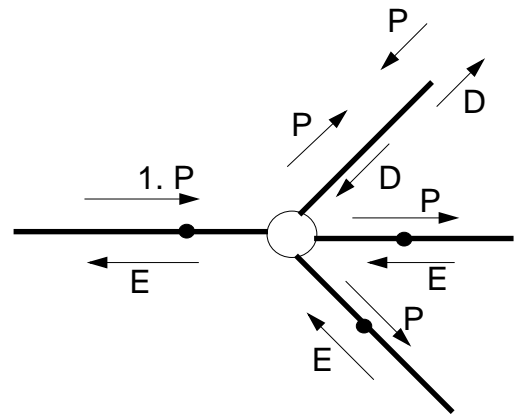
Questions:

- How can a node distinguish outgoing spanning tree edges from other edges?

Probe and echo: detailed operations

Operations of each node (except the initiator):

- The node has **n neighbours** with an **incoming and outgoing link to each of them**.
- After having **received the first probe from neighbour s** , send a **probe to all neighbours except to s** , i. e. **$n - 1$ probes**.
- Each further **incoming probe** is replied with a **dummy** message.
- Wait until **$n - 1$ dummies and echoes** have arrived.
- Then combine the echoes and **send it to s** .



2 messages are sent on each **spanning tree edge**.

4 messages are sent on each **other edge**.

Lecture Parallel Programming WS 2014/2015 / Slide 74

Objectives:

Understand the operations

In the lecture:

- Process does not know which of the outgoing edges belong to the spanning tree.
- Further probes arrive on non-spanning-tree edges.
- They are replied by dummies.

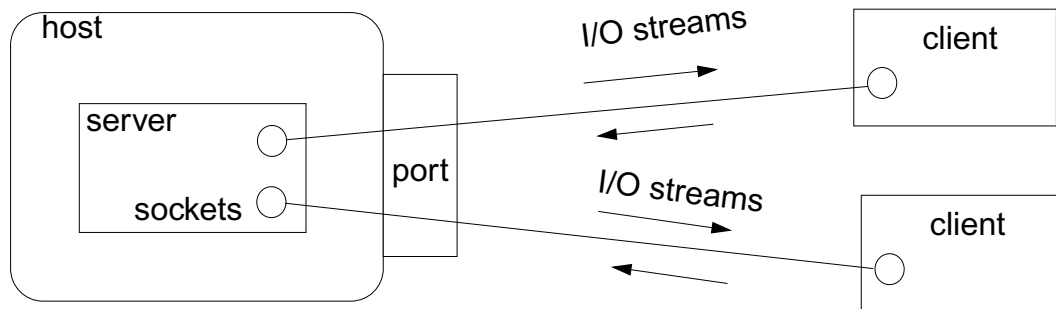
Questions:

- How can the method be simplified if probe and echo is to be executed several times?

Connections via ports and sockets

Port:

- an **abstract connection point** of a computer; numerically encoded
- a **server process** is determined to **respond to a certain port**, e. g. port 13: date and time
- client processes on other machines may send requests via **machine name and port number**



Socket:

- Abstraction of **network software** for communication via ports.
- Sockets are created from **machine address and port number**.
- **Several sockets** on one port may serve several clients.
- **I/O streams** can be setup on a socket.

Lecture Parallel Programming WS 2014/2015 / Slide 75

Objectives:

Understand ports and sockets

In the lecture:

Explain it.

Sockets and I/O-streams

Get a machine address:

```
InetAddress  addr1 = InetAddress.getByName ("java.sun.com"),
              addr2 = InetAddress.getByName ("206.26.48.100"),
              addr3 = InetAddress.getLocalHost();
```

Client side: create a socket that connects to the server machine:

```
Socket myServer = new Socket (addr2, port);
```

Setup I/O-streams on the socket:

```
BufferedReader in =
    new BufferedReader
        (new InputStreamReader (myServer.getInputStream()));

PrintWriter out =
    new PrintWriter (myServer.getOutputStream(), true);
```

Server side: create a specific socket, accept incoming connections:

```
ServerSocket listener = new ServerSocket (port);
...
Socket client = listener.accept(); ... client.close();
```

Lecture Parallel Programming WS 2014/2015 / Slide 76

Objectives:

Using sockets

In the lecture:

Explain how to

- get machine addresses,
- create sockets and streams,
- accept clients and create processes for them.

Worker paradigm

A task is decomposed dynamically in a **bag of subtasks**.
 A set of **worker processes** of the same kind
solve subtasks of the bag and may **create new ones**.

Speedup if the processes are executed
 in parallel on different processors.

Applications: dynamically **decomposable** tasks, e.g.

- solving **combinatorial problems** with methods like Branch & Bound, Divide & Conquer, Backtracking
- image processing

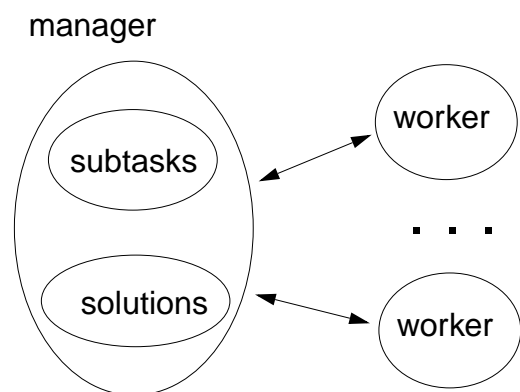
general process structure:

manager process

manages the subtasks to be solved and
 combines the solutions of the subtasks

worker process

solves one subtask after another,
 creates new subtasks, and
 provides solutions of subtasks.



Lecture Parallel Programming WS 2014/2015 / Slide 77

Objectives:

A paradigm for a class of algorithms

In the lecture:

- Remind the algorithmic methods, and
- their parallelization.

Questions:

- Give examples for combinatorial problems.

Branch and Bound

Algorithmic method for the solution of **combinatorial problems** (e. g. traveling salesperson)

tree structured solution space is searched for a best solution

General scheme of operations:

- **partial solution S is extended** to S_1, S_2, \dots (e. g. add an edge to a path)
- is a partial solution **valid**? (e. g. is the added node reached the first time?)
- is S a **complete** solution? (e. g. are all nodes reached)
- **MinCost (S) = C**: each solution that can be created from S has at least cost C (e. g. sum of the costs of the edges of S)
- **Bound**: costs of the best solution so far.

Data structures: a queue sorted according to MinCost; a bound variable

sequential algorithm:

iterate until the queue is empty:
 remove the first element and extend it
 check the thus created new elements
 a new solution and a better bound may be found
 update the queue

Lecture Parallel Programming WS 2014/2015 / Slide 78

Objectives:

Remember the B&B method

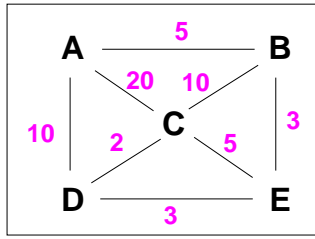
In the lecture:

Explain the general scheme using Traveling Salesperson as an example

Questions:

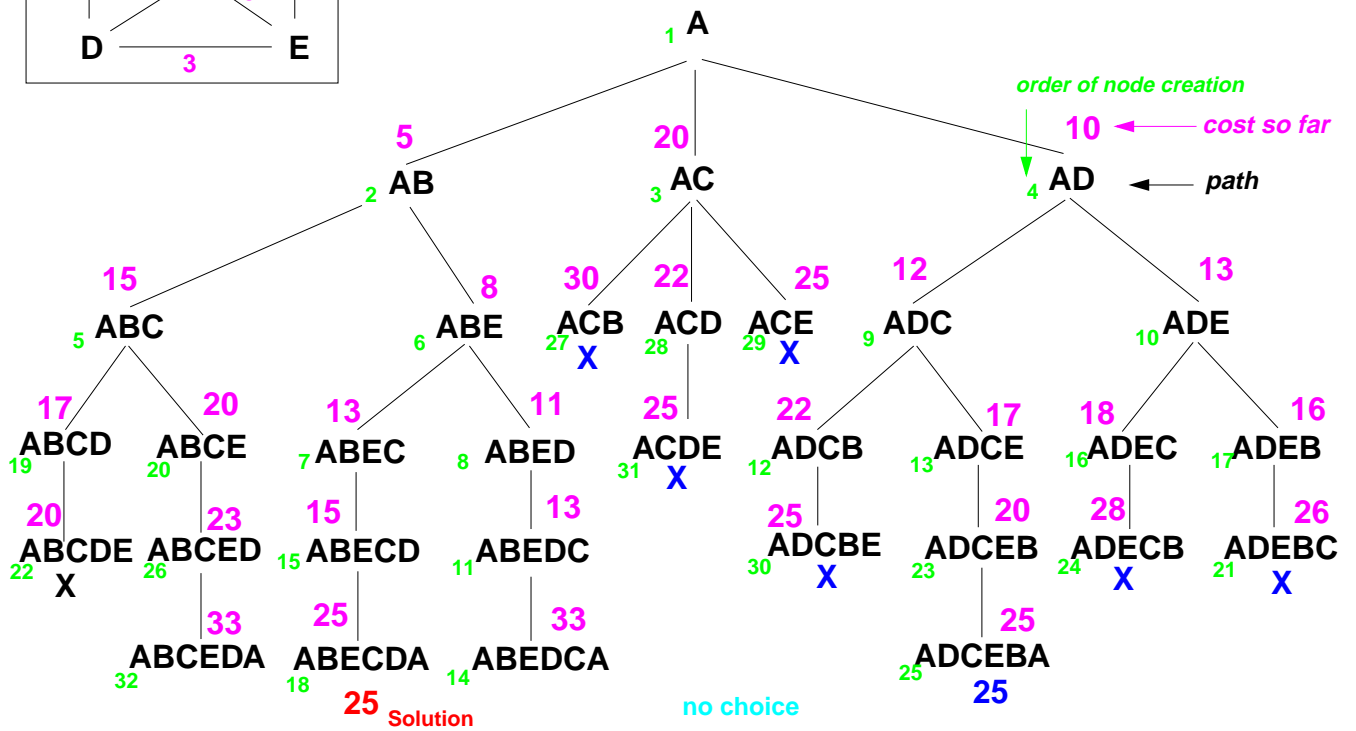
- Explain the general scheme using the backpack problem as an example.

B&B example: Travelling sales person



Connection graph

Solution space



© 2014 bei Prof. Dr. Uwe Kästens

Lecture Parallel Programming WS 2014/2015 / Slide 78a

Objectives:

Reminder for TSP computation

In the lecture:

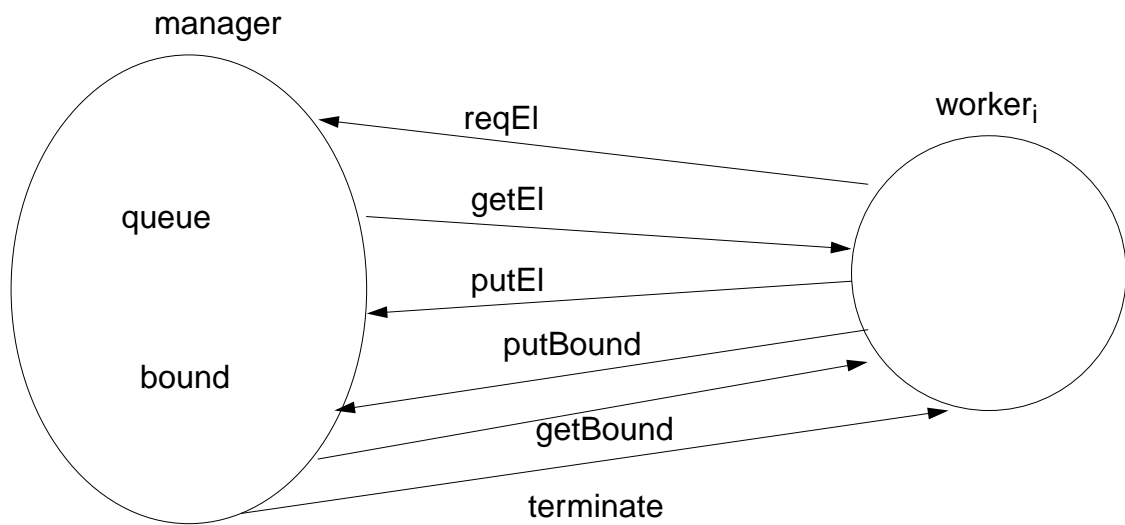
The Branch-and-Bound method is explained.

Get the animated slides.

Parallel Branch & Bound (central)

A **central manager process** holds the queue and the bound variable

Each **worker process** extends an element, checks it, computes its costs, and a new bound



Protocol: reqEI (getEI [getBound] (putEI | putBound)* reqEI)* terminate
for a single Worker

Lecture Parallel Programming WS 2014/2015 / Slide 79

Objectives:

Understand the central organization

In the lecture:

Explain

- the interface,
- the protocol.

Derive them from the general scheme.

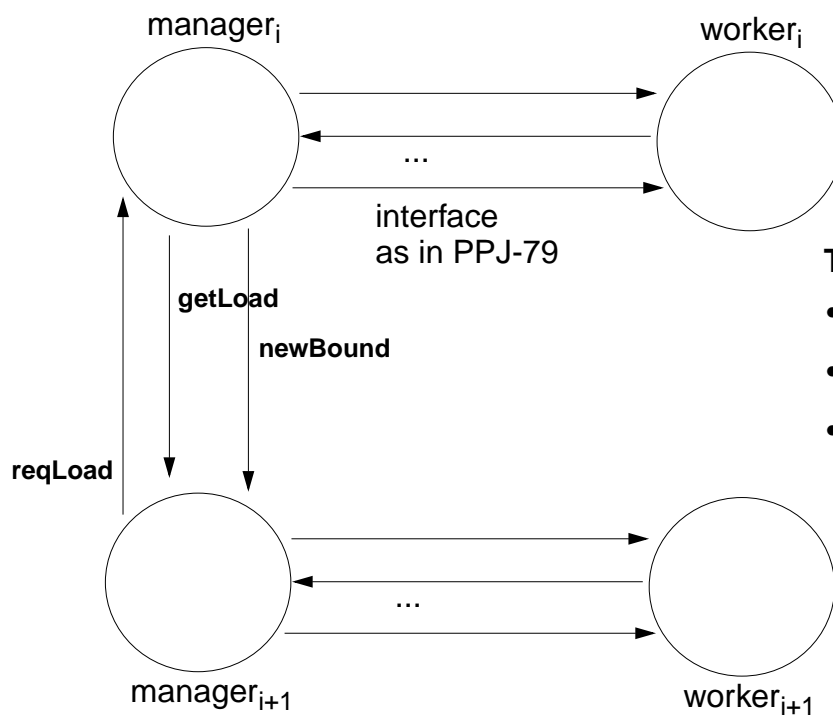
Questions:

- Describe how the execution begins and how it terminates.

Parallel Branch & Bound (distributed)

Several **manager processes cooperate** - one for each worker process.

The work load is balanced between neighbours, e. g. organized in a ring



Termination condition:

- all workers are inactive,
- no manager has another task
- all task channels are empty

Lecture Parallel Programming WS 2014/2015 / Slide 80

Objectives:

Understand the distributed configuration

In the lecture:

Explain

- the interface between manager processes,
- the load balancing task,
- the problem of termination,
- the advantages compared to the central configuration.

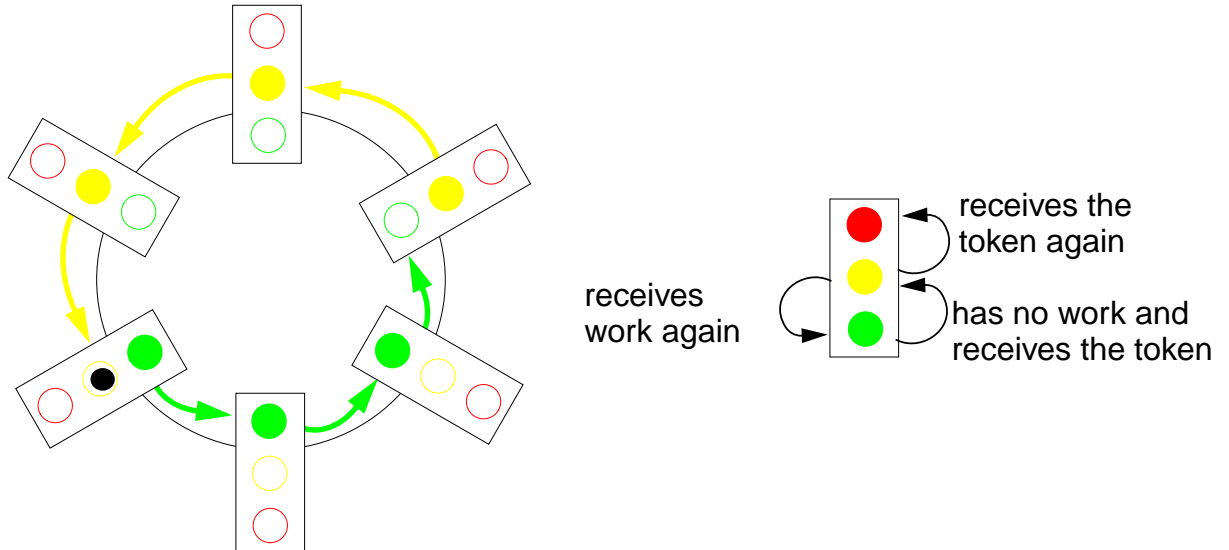
Questions:

- Compare the central and the distributed configuration.

Termination in a ring

Task: Determine a **global state of processes** that communicate in a **directed ring**, and inform all processes, e. g. „all processes are inactive“.

Idea: A token rotates through the ring and marks the processes (**yellow**) that have reached the state in question (inactive).
At the end of the marked sequence the mark may be reset again.
When the token reaches the end of the marked sequence, the state holds globally



Lecture Parallel Programming WS 2014/2015 / Slide 81

Objectives:

Understand a technique for distributed termination

In the lecture:

Explain the

- problem, and
- the solution technique,
- use animated slides.

Get the animated slides.

Questions:

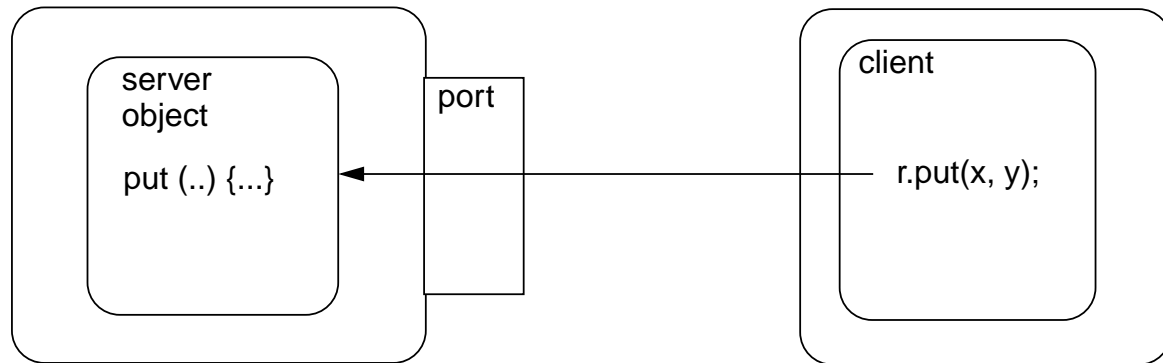
- In which situations is the token passed forward?
- How is guaranteed that all channels are empty when all processes are terminated?

Method calls for objects on remote machines (RMI)

Remote Method Invocation (RMI): Call of a method for an object that is on a remote machine

In Java RMI is available via the library `java.rmi`.

Comparable techniques: CORBA with IDL, Microsoft DCOM with COM



Tasks:

- **identify objects** across machine borders (object management, naming service)
- **interface** for remote accesses and executable proxies for the remote objects (skeleton, stub)
- **method call**, parameter and result are transferred (object serialization)

Lecture Parallel Programming WS 2014/2015 / Slide 82

Objectives:

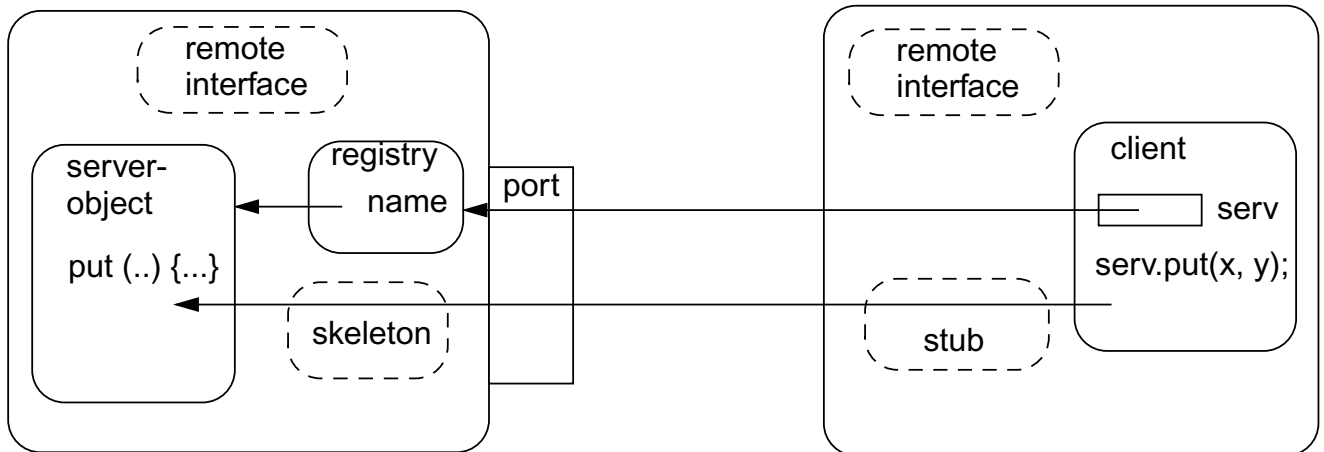
Understand the RMI task

In the lecture:

Explain

- identification of object references,
- representation of objects on I/O streams,
- transfer of objects.

RMI in Java



remote interface: special requirements for interface methods

registry: system process for the machine and for a port;
establishes relations between names and object references

server skeleton: proxy of the server for remote accesses to server objects,
performs I/O transfer on the server side,

client stub: proxy of the server, performs I/O transfer on the client side

Lecture Parallel Programming WS 2014/2015 / Slide 83

Objectives:

Overview over the components

In the lecture:

Explain

- Registry is a stand-alone process.
- Registry can map many objects.
- Skeleton and Stub are generated.

RMI development steps

Example: make a `Hashtable` available as a server object

1. Define a remote interface:

```
public interface RemoteMap extends java.rmi.Remote
{ public Object get (Object key) throws RemoteException; ...}
```

2. Develop an adapter class to adapt the server class to a remote interface:

```
public class RemoteMapAdapter extends UnicastRemoteObject
    implements RemoteMap
{ public RemoteMapAdapter (Hashtable a) { adaptee = a; }
  public Object get (Object key) throws RemoteException
  { return adaptee.get (key); }
  ...
}
```

3. Server main program creates the server object and enters it into the registry:

```
Hashtable adaptee = new Hashtable();
RemoteMapAdapter adapter = new RemoteMapAdapter (adaptee);
Naming.rebind (registeredObjectName, adapter);
```

4. Generate the skeleton and stub from the adapted server class;
copy the client stub on to the client machine:

```
rmic RemoteMapAdapter
```

Lecture Parallel Programming WS 2014/2015 / Slide 84

Objectives:

A work plan

In the lecture:

Explain the steps

RMI development steps (continued)

5. Client identifies the server object on a target machine and calls methods:

```
Registry remoteRegistry = LocateRegistry.getRegistry (hostName);  
RemoteMap serv = (RemoteMap) remoteRegistry.lookup (remObjectName);  
v = serv.get (key);
```

6. Start a registry on the server machine:

```
rmiregistry [port] &  
Default Port is 1099
```

7. Start some servers on the server machine.
8. Start some clients on client machines.

Lecture Parallel Programming WS 2014/2015 / Slide 85

Objectives:

Work plan (continued)

In the lecture:

Explain the steps.

Objects as parameters of RMI calls

Parameters and results of RMI calls are transferred via I/O streams.

That is straight-forward for values of **basic types** and **strings**.

For objects in general:

The values of their variables are transferred,
on the receiver side a new object is created from those values.

The class of such objects has to implement the interface **Serializable**:

```
import java.io.Serializable;

class SIPair implements java.io.Serializable
{   private String s;
    private int i;

    public SIPair (String a, int b) { s = a; i = b; }
    public String toString () { return s + "-" + i; }
}
```

Lecture Parallel Programming WS 2014/2015 / Slide 86

Objectives:

Transfer of objects

In the lecture:

Explain it

9. Synchronous message passing

Processes communicate and synchronize directly, space is provided for **only one message** (instead of a channel).

Operations:

- **send (b):** **blocks** until the partner process is ready to receive the message
- **receive (v):** blocks until the partner process is ready to send a message.

When both sender and receiver processes are ready for the communication, the message is transferred, like an assignment $v := b$;

A send-receive-pair is both **data transfer and synchronization point**

Origin: Communicating Sequential Processes (CSP) [C.A.R. Hoare, CACM 21, 8, 1978]



Lecture Parallel Programming WS 2014/2015 / Slide 87

Objectives:

Notions of synchronous message passing

In the lecture:

- Explain the operations.
- Compare with asynchronous messages.

Questions:

- Compare the notions of synchronous and asynchronous messages.

Notations for synchronous message passing

Notation in CSP und Occam:

p: ... **q ! ex ...** **send** the value of the expression **ex** to process **q**

q: ... **p ? v ...** **receive** a value from process **p** and assign it to variable **v**

multiple ports and **composed messages** may be used:

p: ... **q ! Port1 (a1,...,an) ...**

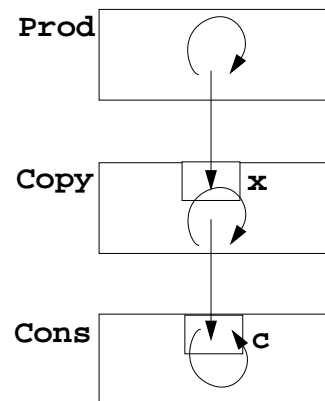
q: ... **p ? Port1 (v1,...,vn) ...**

Example: copy data from a producer to a consumer:

```
Prod:  var p: int;
       do true -> p :=...; Copy ! p od

Copy:  var x: int;
       do true -> Prod ? x; Cons ! x od

Cons:  var c: int;
       do true -> Copy ? c; ... od
```



Lecture Parallel Programming WS 2014/2015 / Slide 88

Objectives:

Notations of synchronous message passing

In the lecture:

- Explain the notations.
- Synchronization without a reply channel.
- Example: copy process.

Selective wait

Guarded command: (invented by E. W. Dijkstra)

a branch may be taken, if a **condition** is true and a **communication** is enabled (**guard**)

```
if Condition1; p ! x -> Statement1
[] Condition2; q ? y -> Statement2
[] Condition3; r ? z -> Statement3
fi
```

A communication statement in a guard yields

true, if the partner process is ready to communicate

false, if the partner process is terminated,

open otherwise (process is not ready, not terminated)

Execution of a guarded command depends on the guards:

- If **some guards are true**, one of them is chosen, the communication and the branch statement are executed.
- If **all guards are false** the guarded command is completed without executing anything.
- **Otherwise** the process is blocked until one of the above cases holds.

Notation of an indexed selection:

```
if (i: 1..n) Condition; p[i] ? v -> Statements fi
```

Lecture Parallel Programming WS 2014/2015 / Slide 89

Objectives:

Understand guards

In the lecture:

- Guarded commands are needed to check whether a message is available without blocking the process.
- Explain the 3 states of a guard.
- Conditions are evaluated only once.

Questions:

- Compare selective wait with the operations empty and receive-if-not-empty of asynchronous messages.

Guarded loops

A **guarded loop** repeats the execution of its guarded command **until all guards yield false**:

```
do
  Condition1; p ! x-> Statement1
[] Condition2; r ? z-> Statement2
od
```

Example: bounded buffer:

```
process Buffer
  do
    cnt < N; Prod ? buf[rear] -> cnt++; rear := rear % N + 1;
  [] cnt > 0; Cons ! buf[front] -> cnt--; front := front % N + 1;
  od
end

process Prod
  var p:=0: int;
  do p<42; Buffer ! p -> p:=p+1;
  od
end

process Cons
  var c: int;
  do Buffer ? c -> print c;
  od
end
```

Lecture Parallel Programming WS 2014/2015 / Slide 90

Objectives:

Understand guarded loops

In the lecture:

Explain

- the example,
- mutual exclusion: process with synchronization points,
- condition synchronization: condition in a guard.

Prefix sums computed with synchronous messages

Synchronous communication provides both **transfer of data and synchronization**.

Necessary synchronization only (cf. synchronous barriers, PPJ-48)

```

const N := 6; var a [0:N-1] : int;

process Worker (i := 0 to N-1)           a process for each element
  var d := 1, sum, new: int

  sum := a[i];

                                     {Invariant SUM: sum = a[i-d+1] + ... + a[i]}
do d < N-1 ->
  if (i+d) < N -> Worker(i+d) ! sum fi   shift old value to the right
  if (i-d) >= 0 -> Worker(i-d) ? new; sum := sum + new fi
                                     get new value from the left
                                     double the distance
  d := 2*d
od
end                                     {SUM and d >= N-1}

```

Why can deadlocks not occur?

Lecture Parallel Programming WS 2014/2015 / Slide 91

Objectives:

See an application of synchronous messages

In the lecture:

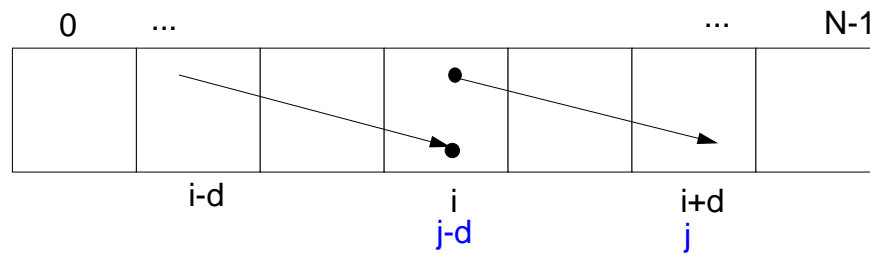
- Explain the communication graphically.
- Compare with asynchronous messages

Questions:

- Why are programs based on synchronous messages more compact and less redundant than those with asynchronous messages?

No deadlocks in synchronous prefix sums

synchronization pattern



- ! and ? operations occur always **in pairs**:

if $i+d < N$ and $i \geq 0$ process i executes `Worker(i+d)!sum`
 let $j = i+d$, i.e. $j-d = i \geq 0$, hence process j executes `Worker(j-d)?new`

- There is always a process that does **not send but receives**:

Choose i such that $i < N$ and $i+d \geq N$, then process i only receives:
 Prove by induction.

- As **no process first receives and then sends**, there is **no deadlock**

Lecture Parallel Programming WS 2014/2015 / Slide 92

Objectives:

Deadlock proof for PPJ-91

In the lecture:

Explain

- why absence of deadlocks is crucial,
- the steps of the proof.

Client/Server scheme with synchronous messages

Technique:

for each **kind of operation** that the server offers, a communication via **2 ports**:

- `oprReq` for transfer of the parameters
- `oprRepl` for transfer of the reply

Scheme of the **client processes**:

```

process Client (I := 1 to N)
  ...
  Server ! oprReq (myArgs)
  Server ? oprRepl (myRes)
  ...
end

```

Scheme of the **server process**:

```

process Server ()
  ...
  do (c: 1..N) ConditionOpr1; Client[c] ? oprReq(oprArgs)
    -> process the request ...
    Client[c] ! oprRepl(oprResults)
  [] correspondingly for other operations ...
  od
end

```

Lecture Parallel Programming WS 2014/2015 / Slide 93

Objectives:

Understand the scheme

In the lecture:

Explain the communication structure

Questions:

- Describe a server for resource allocation in this scheme.

Synchronous Client/Server: variants and comparison

Synchronous servers have the
same characteristics as asynchronous servers,
i. e. active monitors (PPJ-70).

Variants of synchronous servers:

1. Extension to **multiple instances of servers**:
use **guarded command loops** to check
whether a communication is enabled
2. If an operation can **not be executed immediately**,
it has to be delayed, and
its arguments have to be stored in a pending queue
3. The **reply port can be omitted** if
 - there is no result returned, and
 - the request is never delayed
4. Special case: resource allocation with request and release.
5. **Conversation sequences** are executed in the part „process the request“.
Conversation protocols are implemented by a
sequence of send, receive, and guarded commands.

Lecture Parallel Programming WS 2014/2015 / Slide 94

Objectives:

Understand the variants

In the lecture:

Explain

- how pending requests are handled,
- when a channel can be omitted,
- how conversation sequences are handled,

Compare to active monitors.

Synchronous messages in Occam

Occam:

- concurrent programming language, based on **CSP**
- initially developed in 1983 at INMOS Ltd. as native language for **INMOS Transputer** systems
- a program is a nested structure of parallel processes (**PAR**), sequential code blocks (**SEQ**), guarded commands (**ALT**), synchronous send (!) and receive (?) operations, procedures, imperative statement forms;
- communication via **1:1 channels**
- fundamental data types, arrays, records
- extended 2006 to **Occam-pi**, University of Kent, GB
pi-calculus (Milner et. al, 1999):
formal process calculus where names of channels can be communicated via channels
Kent Retargetable occam Compiler (**KRoC**)
(open source)

```
CHAN OF INT chn:
PAR
```

```
SEQ
```

```
INT a:
```

```
a := 42
```

```
chn ! a
```

```
SEQ
```

```
INT b:
```

```
chn ? b
```

```
b := b + 1
```

Lecture Parallel Programming WS 2014/2015 / Slide 94a

Objectives:

A brief introduction to Occam

In the lecture:

- Occam: CSP-based language, standard language of Inmos Transputers
- parallel processes are program constructs (PAR)
- ? and !: send and receive as in CSP
- ALT: guarded command; (! not allowed in a guard)
- channels are here 1:1-links between processes for synchronous message passing
- indexed processes: PAR i=1 FOR n ...
- very restricted data types
- program structure by indentation

Bounded Buffer in Occam

```

CHAN OF Data in, out:
  PAR
    SEQ -- process buffer
      Queue (k) buf:
      Data d:
      WHILE TRUE
        ALT
          in ? d & length(buf) < k
            enqueue(buf, d)
          out ! front(buf) & length(buf) > 0
            ! not allowed in a guard
            dequeue(buf)

```

```

SEQ
  -- only one producer process
  Data d:
  WHILE TRUE
    SEQ
      d = produce ()
      in ! d

```

```

SEQ
  -- only one consumer process
  Data d:
  WHILE TRUE
    SEQ
      out ? d
      consume (d)

```

Lecture Parallel Programming WS 2014/2015 / Slide 94aa

Objectives:

Bounded buffer in Occam

In the lecture:

Explain

- program structure: 3 processes
- ALT: guarded command; (! not allowed in a guard)
- ? and !: send and receive as in CSP
- 2 channels between producer, consumer, and buffer

Synchronous rendezvous in Ada

Ada:

- **general purpose** programming language dedicated for **embedded systems**
- 1979: Jean Ichbiah at CII-Honeywell-Bull (Paris) wins a **competition** of language proposals initiated by the **US DoD**
- **Ada 83 reference manual**
- **Ada 95 ISO Standard**, including oo constructs
- **Ada 2005**, extensions
- **concurrency notions**: processes (**task**, **task type**), shared data, synchronous communication (**rendezvous**), entry operations pass data in both directions, guarded commands (**select**, **accept**)

```

task type Producer;

task body Producer is
  d: Data;
begin
  loop
    d := produce ();
    Buffer.Put (d);
  end loop;
end Producer;

task type Consumer;

task body Consumer is
  d: Data;
begin
  loop
    Buffer.Get (d);
    consume (d);
  end loop;
end Consumer;

```

Lecture Parallel Programming WS 2014/2015 / Slide 94b

Objectives:

Brief introduction to Ada

In the lecture:

Explain

- Ada: general purpose language, in particular suitable for embedded systems
- processes are defined as tasks; task types for several processes of the same type
- communicate synchronously by rendezvous: bi-directional communication operation
- parameters may be passed in either direction (call-by-value-and-result)

Ada: Synchronous rendezvous

```

task type Buffer is      -- interface
  entry Put (d: in Data); -- input port
  entry Get (d: out Data); -- output port
end Buffer;

task body Buffer is
  buf: Queue (k);
  d: Data;
begin
  loop
    select                -- guarded command
      when length(buf) < k =>
        accept Put (d: in Data) do
          enqueue(buf, d);
        end Put;
      or
      when length(buf) > 0 =>
        accept Get (d: out Data) do
          d := front(buf);
          dequeue(buf);
        end Get;
    end select;
  end loop;
end Buffer;

```

```

task type Producer;

task body Producer is
  d: Data;
begin
  loop
    d := produce ();
    Buffer.Put (d);
  end loop;
end Producer;

task type Consumer;

task body Consumer is
  d: Data;
begin
  loop
    Buffer.Get (d);
    consume (d);
  end loop;
end Consumer;

```

Lecture Parallel Programming WS 2014/2015 / Slide 94ba

Objectives:

Bounded buffer using Ada rendezvous

In the lecture:

Explain

- task declares communication interface: entries
- entries are called by other tasks
- parameters may be passed in either direction (call-by-value-and-result)
- each entry has several accept-statements (communication operation) in the task body
- select is a guarded command
- one-sided anonymous: the task does not know who calls its entry

10. Concurrent and functional programming

Overview

1. Pure **functional programs do not have side-effects**:
operands of an operation and arguments of a call
can be **evaluated in any order**, in particular **concurrently**
2. **Recursive task decomposition** can be parallelized according to the
paradigm **bag of subtasks**
3. **Lazy evaluation** of lists leads to **programs that transform streams**, can be
parallelized according the **pipelining** paradigma
4. **Dataflow languages** and dataflow machines support **stream programming**
5. **Concurrency notions in functional languages**:
Message passing in Erlang
Actors in Scala

Lecture Parallel Programming WS 2014/2015 / Slide 94c

Objectives:

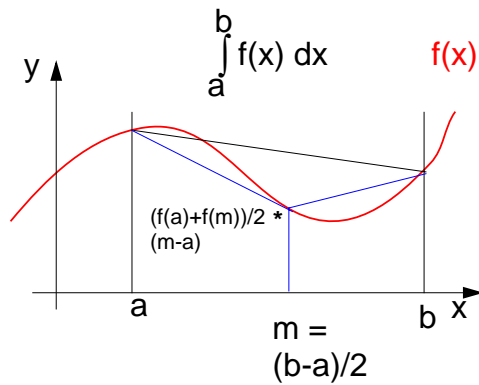
Understand close relation between FP and concurrency

In the lecture:

Explain

- the items.

Recursive adaptive quadrature computation



```

fun quad (f, l, r, area, eps) =
  let m = (r-l)/2 and
      fl = f(l) and
      fm = f(m) and
      fr = f(r) and
      larea = (fl+fm)*(m-l)/2 and
      rarea = (fm+fr)*(r-m)/2 and
  in
    if abs(larea+rarea-area)>eps
    then
      let

```

Compute an **approximation of the integral** over $f(x)$ between a and b .

Recursively **refine the interval** into two subintervals until the sum of the **areas of the two trapezoids** differs less than eps from the **area of the big trapezoid**.

See [G. Andrews: Foundations of Multithreaded, Parallel, and Distributed Programming, Addison Wesley, 2000, pp. 17-19]

```

      lar = quad(f,l,m,larea,eps) and
      rar = quad(f,m,r,rarea,eps)
    in (lar+rar)
    end
  else area
  end
end

initial call:

quad (f,a,b,(f(a)+f(b))/2*(b-a),0.001)

```

Lecture Parallel Programming WS 2014/2015 / Slide 94d

Objectives:

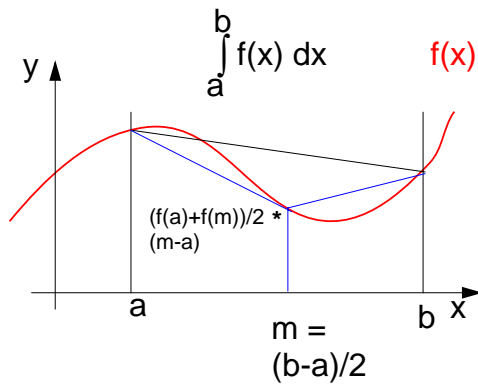
Understand the recursive quadrature computation

In the lecture:

Explain

- the task,
- the approximation idea,
- the functional notation

Recursive adaptive quadrature computation



Compute an **approximation of the integral** over $f(x)$ between a and b .

Recursively **refine the interval** into two subintervals until the sum of the **areas of the two trapezoids** differs less than ϵ from the **area of the big trapezoid**.

Fork two concurrent processes.

See [G. Andrews: Foundations of Multithreaded, Parallel, and Distributed Programming, Addison Wesley, 2000, pp. 17-19]

```

fun quad (f, l, r, area, eps) =
  let m = (r-l)/2 and
      fl = f(l) and
      fm = f(m) and
      fr = f(r) and
      larea = (fl+fm)*(m-l)/2 and
      rarea = (fm+fr)*(r-m)/2 and
  in
    if abs(larea+rarea-area)>eps
    then
      let
        co
        lar = quad(f,l,m,larea,eps) and
        //
        rar = quad(f,m,r,rarea,eps)
        oc
      in (lar+rar)
      end
    else area
  end

initial call:

quad (f,a,b,(f(a)+f(b))/2*(b-a),0.001)

```

Lecture Parallel Programming WS 2014/2015 / Slide 94e

Objectives:

Parallelized refinement

In the lecture:

Explain

- the dynamics of the refinement

Streams in functional programming

Linear lists are fundamental data structures in functional programming, e.g. in **SML**:

```
datatype 'a list = nil | :: of 'a * 'a list
```

Eager evaluation: all elements of a list are to be computed, before any can be accessed.

Lazy evaluation only those elements of a list are computed which are going to be accessed.

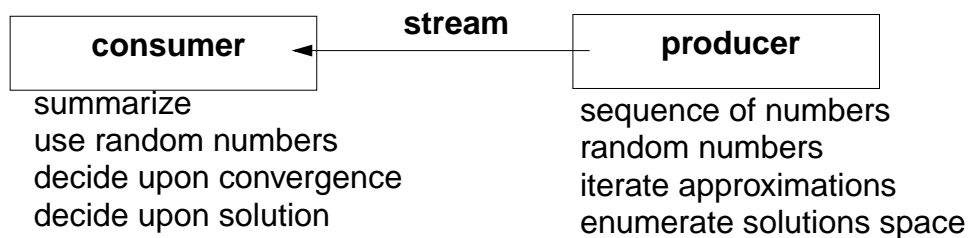
That can be achieved by replacing the (pointer to) the tail of the list by a parameterless **function which computes the tail of the sequence when needed**:

```
datatype 'a seq= Nil | Cons of 'a * (unit->'a seq)
```

Lazy lists are called **streams**.

Streams establish a useful **programming paradigm**:

Programming the **creation** of a stream can be **separated** from programming its **use**.



Functions on streams can be understood as communicating concurrent processes.

Lecture Parallel Programming WS 2014/2015 / Slide 94f

Objectives:

Understand streams in functional programming

In the lecture:

Explain

- the notion of streams,
- the programming paradigm

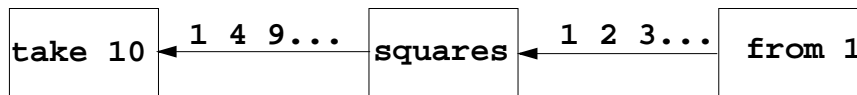
Examples for stream functions (1)

produce a stream of numbers: `int -> int seq`
`fun from k = Cons (k, fn()=> from (k+1));`

consume the first n elements into a list: `'a seq * int -> 'a list`
`fun take (xq, 0) = []`
`| take (Nil, n) = raise Empty`
`| take (Cons(x, xf), n) = x :: take (xf (), n - 1);`

transform a stream of numbers: `int seq -> int seq`
`fun squares Nil = Nil`
`| squares (Cons (x, xf)) = Cons (x * x, fn() => squares (xf()));`

`take (squares (from 1), 10);`



Lecture Parallel Programming WS 2014/2015 / Slide 94g

Objectives:

Understand simple stream functions

In the lecture:

Explain

- structure of stream functions,
- classify: producer, consumer, transformer,

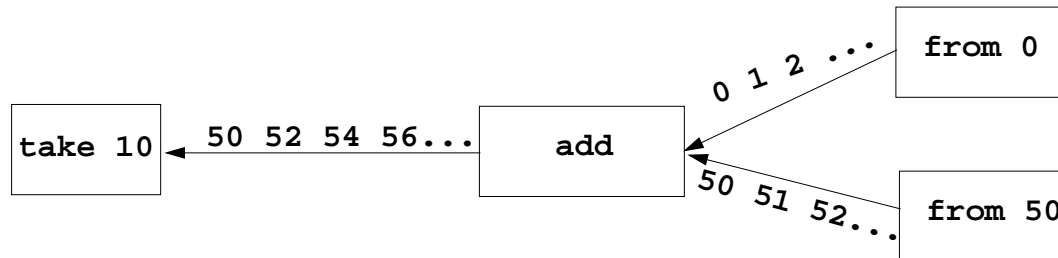
Examples for stream functions (2)

add the numbers of two streams: `int seq * int seq -> int seq`

```

fun add (Cons(x, xf),Cons(y, yf)) =
  Cons (x+y, fn() => add (xf(), yf()))
|   add _ = Nil;

```



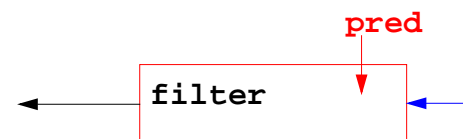
Filter-Schema:

`('a -> bool) -> 'a seq -> 'a seq`

```

fun filter pred Nil = Nil
|   filter pred (Cons(x,xf)) =
  if pred x then Cons (x, fn()=> filter pred (xf()))
  else filter pred (xf());

```



Lecture Parallel Programming WS 2014/2015 / Slide 94h

Objectives:

Combine streams

In the lecture:

Explain the examples

Recursive stream composition

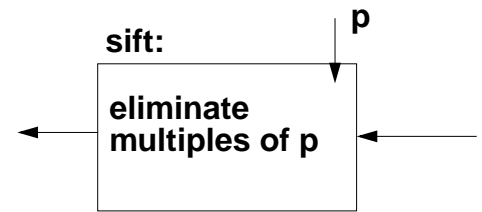
```

fun sift p =
  filter (fn n => n mod p <> 0);

fun sieve (Cons(p,nf)) =
  Cons (p, fn() => sieve (sift p (nf())));

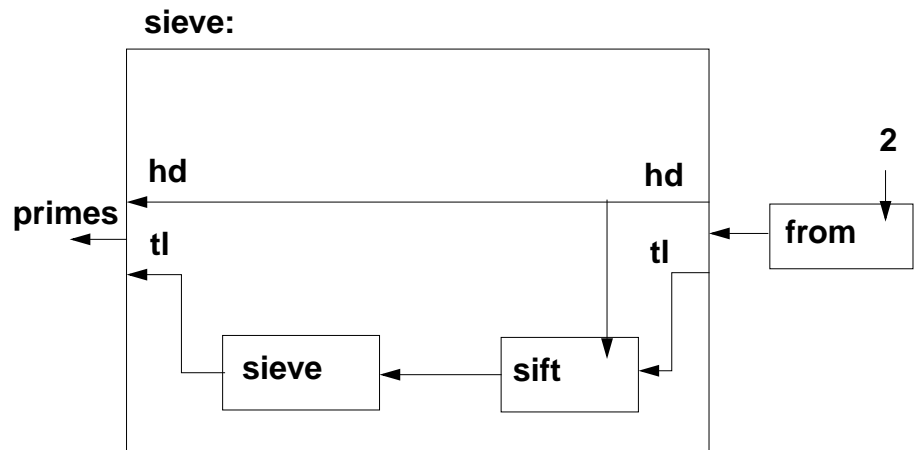
val primes = sieve (from 2);
take (primes, 25);

```



Compute prime numbers:

Sieve of Eratosthenes



All recursively constructed sift-sieve-pairs can execute concurrently!

Lecture Parallel Programming WS 2014/2015 / Slide 94i

Objectives:

Understand recursive composition of streams

In the lecture:

The recursion is explained

Sieve of Eratosthenes in CSP

A pipeline of filters:

L processes are created, each sends a **stream of numbers** to its successor.

The **first number p** received is a prime. It is **used to filter** the following numbers.

Finally, each process holds a prime in p.

```

process Sieve[1]
  for [1 = 3 to n by 2]
    Sieve[2] ! i # pass odd numbers to Sieve[2]

process Sieve[i = 2 to L]
  int p, next
  Sieve[i-1] ? p          # p is a prime
  do Sieve[i-1] ? next -># receive next candidate
    if (next mod p) != 0 ->
      Sieve[i+1] ! next # pass it on
    fi
  od

```

[G. Andrews: Foundations of Multithreaded, Parallel, and Distributed Programming, Addison Wesley, 2000, pp. 326-328]

Lecture Parallel Programming WS 2014/2015 / Slide 94j

Objectives:

Pipeline of processes

In the lecture:

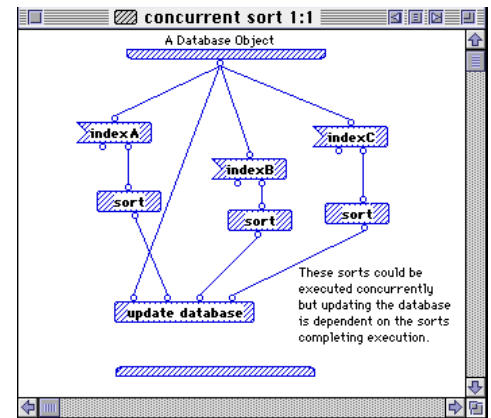
The communication is explained and compared to the stream functions.

Dataflow languages

Textual languages:

Lucid: stream computations by equations, no side effects; 1976, Wadge, Ashcroft

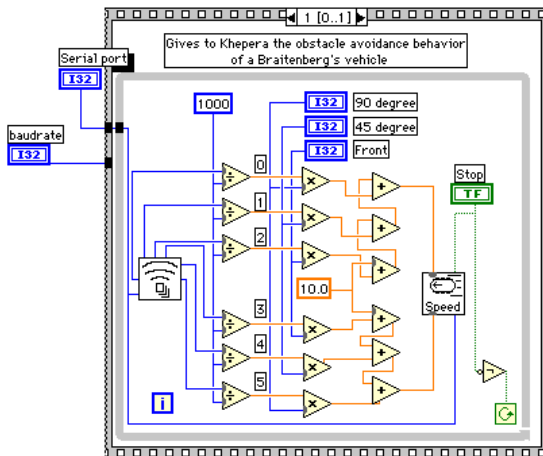
SISAL: (Streams and Iteration in a Single Assignment Language), no side-effects, fine-grained parallelization by compiler, 1983



Visual languages:

Prograph (Acadia University 1983):
dataflow and object-oriented

LabVIEW (National Instruments, 1986):
Nodes represent stream processing functions connected by wires, concurrent execution triggered by available input. Strong support of interfaces to instrumentation hardware.



Lecture Parallel Programming WS 2014/2015 / Slide 94k

Objectives:

Pointers to dataflow languages

In the lecture:

General information on these languages is given.

Language Erlang

Erlang developed 1986 by Joe Armstrong, et.al at **Ericsson**

- multi-paradigm: **functional** and **concurrent**
- initial application area: **telecommunication**
requirements: distributed, fault-tolerant, soft-real-time, non-stopping software
- **processes** communicate via **asynchronous message** passing
- **single-assignment** variables, **no shared memory** between processes

Explanations and examples taken from

[J. Armstrong, R. Virding, C. Wikström, M. Williams: Concurrent Programming in ERLANG, Second Edition, Ericsson Telecommunications Systems Laboratories, Prentice Hall, 1996]

<http://www.erlang.org>

Lecture Parallel Programming WS 2014/2015 / Slide 94I

Objectives:

Characteristics of Erlang

In the lecture:

Explain background

Basic communication constructs

process creation:

```
Pid = spawn(Module, FunctionName, ArgumentList)
```

asynchronous message send:

```
Pid ! Message
```

The operands are expressions which yield a process id and a message.

selective receive:

```
receive
  Pattern1 [when Guard1] ->
    Actions1 ;
  Pattern2 [when Guard2] ->
    Actions2 ;
  ...
end
```

Searches the process' **mailbox** for a **message that matches a pattern**, and receives it.
Can not block on an unexpected message!

Initial example

A module that creates counter processes:

```
-module(counter).
-export([start/0,loop/1]).

start() ->
  spawn(counter, loop, [0]).

loop(Val) ->
  receive
    increment ->
      loop(Val + 1)
  end.

clients send increment messages to it
```

Lecture Parallel Programming WS 2014/2015 / Slide 94m

Objectives:

Understand send and selective receive constructs

In the lecture:

Explain the constructs

Complete example: Counter

Interface functions are called by client processes.

They send 3 kinds of messages.

`self()` gives the client's pid, to reply to it.

The counter process identifies itself in the reply.

The receive is iterated (tail-recursion).

Unexpected messages are removed

```
-module(counter).
-export([start/0,loop/1,increment/1,value/1,stop/1]).

%% First the interface functions.
start() -> spawn(counter, loop, [0]).

increment(Counter) -> Counter ! increment.

value(Counter) ->
    Counter ! {self(),value},
    receive {Counter,value} -> Value
end.

stop(Counter) -> Counter ! stop.

%% The counter loop.
loop(Val) ->
    receive increment ->    loop(Val + 1);
           {From,value} -> From ! {self(),Val},
           loop(Val);

           stop ->         true;
           Other ->        loop(Val)
end.
```

Lecture Parallel Programming WS 2014/2015 / Slide 94n

Objectives:

Understand the constructs in context

In the lecture:

Explain the constructs

Example: Allocation server (interface)

A server maintains two lists of **free and allocated resources**. Clients call a function **allocate** to request a resource and a function **free** to return that resource.

The two lists of **free and allocated resources** are initialized.

register associates the pid to a name.

The calls of **allocate** and **free** are transformed into **different kinds of messages**. Thus, implementation details are not disclosed to clients.

```
-module(allocator).
-export([start/1,server/2,allocate/0,free/1]).

start(Resources) ->
    Pid = spawn(allocator, server,
                [Resources,[]]),
    register(resource_alloc, Pid).

% The interface functions.

allocate() -> request(alloc).

free(Resource) -> request({free,Resource}).

request(Request) ->
    resource_alloc ! {self(),Request},
    receive {resource_alloc,Reply} -> Reply
end.
```

Lecture Parallel Programming WS 2014/2015 / Slide 94o

Objectives:

Understand a non-trivial server implementation

In the lecture:

Explain the techniques

Example: Allocation server (implementation)

The function `server` receives the two kinds of messages and transforms them into calls of `s_allocate` and `s_free`.

`s_allocate` returns **yes** and the resource or **no**, and **updates** the two lists in the recursive `server` call.

`s_free`: `member` checks whether the returned resource `R` is in the free list, returns **ok** and updates the lists,

or it returns **error**.

The `server` call loops.

```
server(Free, Allocated) ->
  receive
    {From, alloc} ->
      s_allocate(Free, Allocated, From);
    {From, {free, R}} ->
      s_free(Free, Allocated, From, R)
  end.

s_allocate([R|Free], Allocated, From) ->
  From ! {resource_alloc, {yes, R}},
  server(Free, [{R, From}|Allocated]);
s_allocate([], Allocated, From) ->
  From ! {resource_alloc, no},
  server([], Allocated).

s_free(Free, Allocated, From, R) ->
  case member({R, From}, Allocated) of
    true -> From ! {resource_alloc, ok},
      server([R|Free],
              delete({R, From},
                    Allocated));
    false -> From ! {resource_alloc, error},
      server(Free, Allocated)
  end.
```

Lecture Parallel Programming WS 2014/2015 / Slide 94p

Objectives:

Understand a non-trivial server implementation

In the lecture:

Explain the techniques

Scala: object-oriented and functional language

Scala: Object-oriented language (like Java, more compact notation), augmented by functional constructs (as in SML); object-oriented execution model (Java)

functional constructs:

- nested functions, higher order functions, currying, case constructs based on pattern matching
- functions on lists, streams,... provided in a big language library
- parametric polymorphism; restricted local type inference

object-oriented constructs:

- classes define all types (types are consequently oo - including basic types), subtyping, restrictable type parameters, case classes
- object-oriented mixins (traits)

general:

- static typing, parametric polymorphism and subtyping polymorphism
- very compact functional notation
- complex language, and quite complex language description
- compilable and executable together with Java classes
- since 2003, author: Martin Odersky, www.scala.org, docs.scala-lang.org

Lecture Parallel Programming WS 2014/2015 / Slide 94q

Objectives:

Overview over properties of Scala

In the lecture:

Brief explanations are given

Actors in Scala (1)

An **actor** is a lightweight process:

- **actor** { **body** } creates a process that executes **body**
- **asynchronous** message passing
- **send**: **p ! msg** puts **msg** into **p**'s mailbox
- **receive** operation searches the mailbox for the first message that matches one of the case patterns (as in **Erlang**)
- **case x** is a catch-all pattern

Example: orders and cancellations

```
val orderMngr = actor {
  while (true)
    receive {
      case Order(sender, item) =>
        val o =
          handleOrder(sender,item)
        sender ! Ack(o)
      case Cancel(sender, o) =>
        if (o.pending) {
          cancelOrder(o)
          sender ! Ack(o)
        } else sender ! NoAck
      case x => junk += x
    }
}

val customer = actor {
  orderMngr ! Order(self, myItem)
  receive {
    case Ack(o) => ...
  }
}
```

[P. Haller, M. Odersky: Actors That Unify Threads and Events; in A.L. Murphy and J. Vitek (Eds.): COORDINATION 2007, LNCS 4467, pp. 171–190, 2007. © Springer-Verlag Berlin Heidelberg 2007]

Lecture Parallel Programming WS 2014/2015 / Slide 94r

Objectives:

Understand actors in Scala

In the lecture:

Explain

- the constructs using the example,
- asynchronous message passing via unordered mailbox
- sender is explicitly transmitted for reply: common pattern

Actors in Scala (2)

Constructs used to simplify replying:

- The sender of a received message is stored in `sender`
- `reply(msg)` sends `msg` to `sender`
- `a !? msg` sends `msg` to `a`, waits for a reply, and returns it.

Example: orders and cancellations

```
val orderMgr = actor {
  while (true)
    receive {
      case Order(item) =>
        val o =
          handleOrder(sender,item)
          reply(Ack(o))
      case Cancel(o) =>
        if (o.pending) {
          cancelOrder(o)
          reply(Ack(o))
        } else reply(NoAck)
      case x => junk += x
    }
}

val customer = actor {
  orderMgr !? Order(myItem)
  match {
    case Ack(o) => ...
  }
}
```

[P. Haller, M. Odersky: Actors That Unify Threads and Events; in A.L. Murphy and J. Vitek (Eds.): COORDINATION 2007, LNCS 4467, pp. 171–190, 2007. © Springer-Verlag Berlin Heidelberg 2007]

Lecture Parallel Programming WS 2014/2015 / Slide 94s

Objectives:

Scala provides simplifying notations

In the lecture:

Explain the simplifying constructs

11. Check your knowledge (1)

Introduction

1. Explain the notions: sequential, parallel, interleaved, concurrent execution of processes.
2. How are Threads created in Java (3 steps)?

Properties of Parallel Programs

3. Explain axioms and inference rules in Hoare Logic.
4. What does the weakest precondition $wp(s, Q) = P$ mean?
5. Explain the notions: atomic action, at-most-once property.
6. How is interference between processes defined?
7. How is non-interference between processes proven?
8. Explain techniques to avoid interference between processes.

Monitors

9. Explain how the two kinds of synchronization are used in monitors.
10. Explain the semantics of condition variables and the variants thereof.
11. Which are the 3 reasons why a process may wait for a monitor?
12. How do you implement several conditions with a single condition variable?

Lecture Parallel Programming WS 2014/2015 / Slide 95

Objectives:

Understand and repeat the material

In the lecture:

- Answer some of the questions.

Check your knowledge (2)

13. Signal-and-continue requires loops to check waiting-conditions. Why?
14. Explain the properties of monitors in Java.
15. When can notify be used instead of notifyAll?
16. Where does a monitor invariant hold? Where has it to be proven?
17. Explain how monitors are systematically developed in 5 steps.
18. Formulate a monitor invariant for the readers/writers scheme?
19. Explain the development steps for the method „Rendezvous of processes“.
20. How are waiting conditions and release operations inserted when using the method of counting variables?

Barriers

21. Explain duplication of distance at the example prefix sums.
22. Explain the barrier rule; explain the flag rules.
23. Describe the tree barrier.
24. Describe the symmetric dissemination barrier.

Lecture Parallel Programming WS 2014/2015 / Slide 95a

Objectives:

Understand and repeat the material

In the lecture:

- Answer some of the questions.

Check your knowledge (3)

Data parallelism

25. Explain how list ends are found in parallel.
26. Show iteration spaces for given loops and vice versa.
27. Explain which dependence vectors may occur in sequential (parallel) loops.
28. Explain the SRP transformations.
29. How are the transformation matrices used?
30. Which transformations can be used to parallelize the inner loop if the dependence vectors are $(0,1)$ and $(1,0)$?
31. How are bounds of nested loops described formally?

Asynchronous messages

32. Explain the notion of a channel and its operations.
33. Explain typical channel structures.
34. Explain channel structures for the client/server paradigm.
35. What problem occurs if server processes receive each from several channels?
36. Explain the notion of conversation sequences.

Lecture Parallel Programming WS 2014/2015 / Slide 96

Objectives:

Understand and repeat the material

In the lecture:

- Answer some of the questions.

Check your knowledge (4)

37. Which operations does a node execute when it is part of a broadcast in a net?

38. Which operations does a node execute when it is part of a probe-and-echo?

39. How many messages are sent in a probe-and-echo scheme?

Messages in distributed systems

40. Explain the worker paradigm.

41. Describe the process interface for distributed branch-and-bound.

42. Explain the technique for termination in a ring.

Synchronous messages

43. Compare the fundamental notions of synchronous and asynchronous messages.

44. Explain the constructs for selective wait with synchronous messages.

45. Why are programs based on synchronous messages more compact and less redundant than those with asynchronous messages?

46. Describe a server for resource allocation according to the scheme for synchronous messages.

Lecture Parallel Programming WS 2014/2015 / Slide 97

Objectives:

Understand and repeat the material

In the lecture:

- Answer some of the questions.

Check your knowledge (5)

Concurrent and functional programming

47. Explain why paradigms in functional and concurrent programming match well.
48. What are benefits of stream programming?
49. Compare implementations of the Sieve of Eratosthenes using streams or CSP.
50. Explain concurrency in Erlang, in particular selective receive.
51. Explain the characteristics of Scala, in particular its Actors.

Lecture Parallel Programming WS 2014/2015 / Slide 98

Objectives:

Understand and repeat the material

In the lecture:

- Answer some of the questions.