Parallel Programming

Prof. Dr. Uwe Kastens

Winter 2014 / 2015

Objectives

The participants are taught to understand and to apply

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

Exercises:

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

Contents

Prerequisites

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

Organization of the course

Lecturer

Prof. Dr. Uwe Kastens:

Office hours: on appointment by email

Teaching Assistant:

- Peter Pfahler

Tutorials

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

Schedule

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

Lecture

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

Start date: Oct 13, 2014

Examination

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

Literature

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

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

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

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

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

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

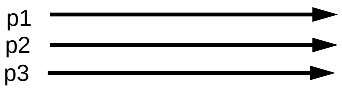
Jim Farley: **Java Distributed Computing**, O'Reilly, 1998

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

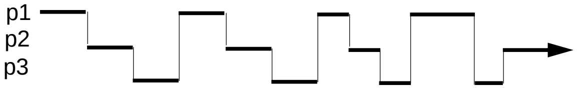
Fundamental notions (repeated): Parallel processes

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

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



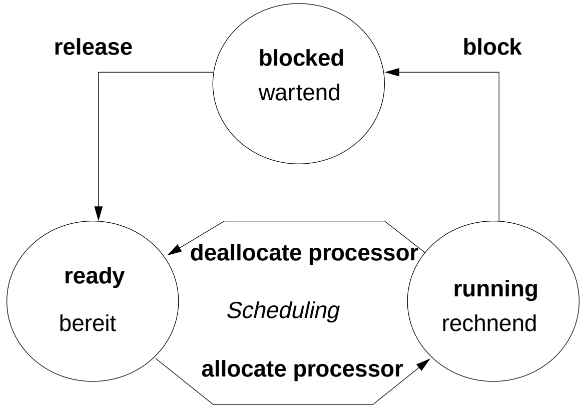
interleaved processes:
several processes, which are executed piecewise alternatingly on a single processor
processes are switched by a common process manager or by the processes themselves.



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

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

Fundamental notions (repeated): States and transitions of processes



see KMS 2-17, 2-18

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

Applications of parallel processes

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

The application classes follow **different objectives**.

Create threads in Java - technique: implement `Runnable`

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

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

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

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!

Important methods of the class `Thread`

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

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

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

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

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

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

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

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

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

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

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


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

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

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

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

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

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

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

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

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

   private DigiClock clock;
}
```

Processes, which are started in an applet

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

Otherwise they bind resources, although they are not visible.

2. Properties of Parallel Programs

Goals:

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

Example A:

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

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

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

Example B:

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

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

Methods:

Hoare Logic, Weakest Precondition, techniques for parallel programs

Proofs of parallel programs

Example A:

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

Example B_1:

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

Check each proof for correctness!

Explain!

Example B_2:

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

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

Hoare Logic: a brief reminder

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

Predicates (assertions) are stated for program positions:

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

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

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

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

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

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

Axioms and inference rules for sequential constructs

statement sequence (1)

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

assignment (2)

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

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

stronger precondition (3)

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

weaker postcondition (4)

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

multiple alternative (guarded command) (5)

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

selecting iteration (6)

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

no operation (7)

{P} **skip** {P}

Verification: algorithm computes gcd

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

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

the loop terminates:

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

Weakest precondition

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

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

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

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

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

Examples for weakest preconditions

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

Interleaving - used as an abstract execution model

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

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

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

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

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

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

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

Atomic actions

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

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

At-most-once property (AMO):

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

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

Atomic by AMO

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

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

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

atomic AMO constructs < ... >:

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

interleaving actions of two processes:

(1)
```
                a
p1:   < t = 0; t = t + 1;>
p2:   < s = 0; s = s + 1;>
                b
```

(2)
```
            a
p1:   < x = 2;>
p2:   < t = x + 1;>
            b
```

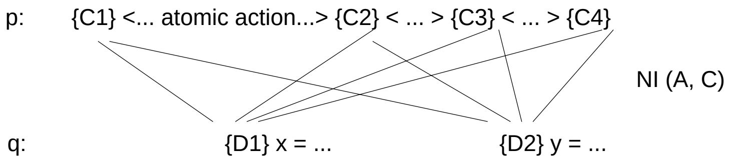
(3)
```
          b    a
p1:   x = < y + 1 >;
p2:   y = < x + 1 >;
          d    c
```

(4)
```
       c  a      b
p1:   x = <y> + <z>;
p2:   <y = 1;> <z = 2;>;
       d         e
```

Interference between processes

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

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



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

Example: Interference between an assertion and an assignment

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

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

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

Does A interfere with C? Depends on C:

1. **C: a == 1**
   {a == 1 ∧ d + 1 > 0} a = d + 1 {a == 1} is not provable ⇒ interference
   (C ... C)
2. **C: a > 0**
   {a > 0 ∧ d + 1 > 0} a = d + 1 {a > 0} is provable ⇒ non-interference
3. **C: a==1 ∧ d<0**
   {a==1 ∧ d<0 ∧ d+1>0} a = d + 1 {a==1 ∧ d<0} is provable ⇒ non-interference
   (a==1 ∧ d<0 ∧ d+1>0 is f)

Non-interference checks

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

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

{ x+1 = 1 ∧ y+1 = 1} y:= y+1 {x+1 = 1 ∧ y = 1} => {x+1 = 1}
(C ... C)

---

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

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

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

is not correct

Two inference rules for concurrent execution

The statement for **condition synchronization**

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

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

$$\frac{\{P \wedge B\}\ S\ \{Q\}}{\{P\}\ \texttt{<await B -> S >}\ \{Q\}}$$

The statement for **concurrent processes**

$\texttt{co}\ S_1\ \texttt{//}\ \ldots\ \texttt{//}\ S_n\ \texttt{oc}$

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

Non-Interference is to be proven.

$$\frac{\{P_i\}\ S_i\ \{Q_i\},\ 1 \le i \le n, \text{ are \textbf{interference-free theorems}}}{\{P_1 \wedge \ldots \wedge P_n\}\ \texttt{co}\ S_1\ \texttt{//}\ \ldots\ \texttt{//}\ S_n\ \texttt{oc}\ \{Q_1 \wedge \ldots \wedge Q_n\}}$$

Avoiding interference

1. **disjoint variables**:
   Two concurrent processes **p** and **q** are interference-free if the set of variables **p** writes to is disjoint from the set of variables **q** reads from and vice versa.
2. **weakened assertions**:
   The assertions in the proofs of concurrent processes can in some cases be made interference-free by weakening them.
3. **atomic action**:
   A non-interference-free assertion `C` can be hidden in an atomic action.

```
p:: ... x := e ...          p:: ... x := e ...
q:: ... S1 {C} S2 ...       q:: ...<S1 {C} S2> ...
```

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

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

```
p:: ... x := e ...          p:: ..<await not C or B -> x:=e> ...
                                  with B = wp (x:=e, C)
q:: ... S1 {C} S2 ...       q:: ... S1 {C} S2 ...
```


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

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**.

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 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 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**.

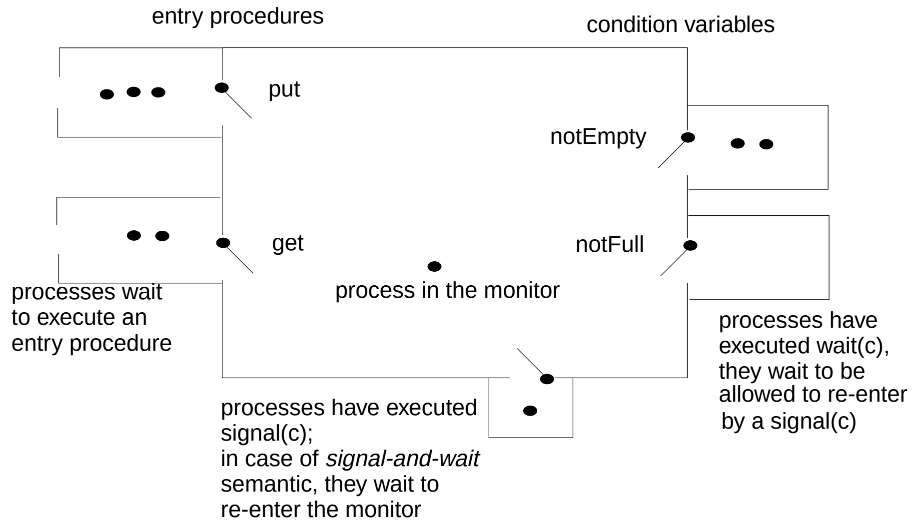
Note the difference between condition variables and semaphores:
Semaphores are counters. The effect of a call V(s) on a semaphore is not lost if no process is waiting on s.

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;
```


Synchronization in a monitor



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

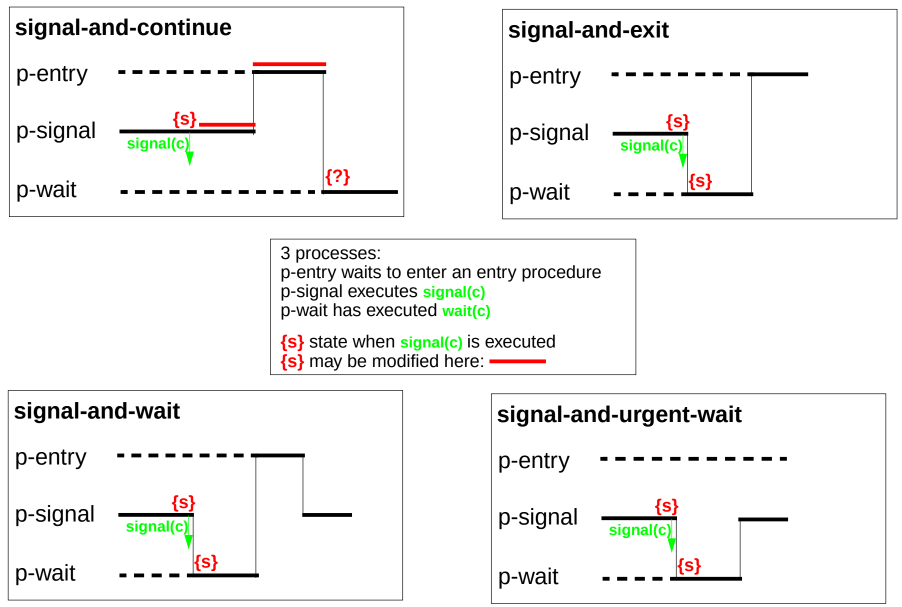
signal-and-exit semantics:
The process that executes **`signal`** terminates the entry procedure call and leaves the monitor.
The released process enters the monitor **immediately** - without a state change in between

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;
```


Variants of signal-wait semantics: examples of execution



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.

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) {}
```


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;
   }
}
```


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`

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
             return x;
      } finally { lock.unlock();}                 explicit mutual exclusion
   }
}
```


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 <= buf.length() <= 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 (*)).

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 <= length(buf) <= 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

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;
```

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;
```


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;
```


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;
```


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;
```

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;
```


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;
```

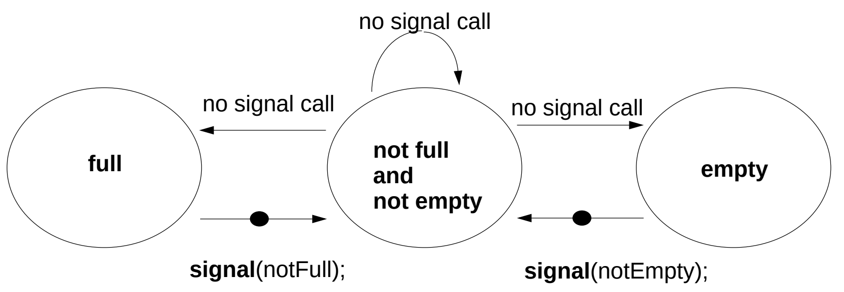

Relevant state changes

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:



Pattern: Allocating counted resources

A **monitor** grants access to a set of k ≥ 1 resources of the **same kind**.
Processes request n resources, 1≤ n ≤ k, and return them after having used them.
Examples:
Lending bikes in groups (n ≥ 1), allocating blocks of storage (n ≥ 1),
Taxicab provider (n=1), drive with a weight of n ≥ 1 tons on a bridge

Monitor invariant	**requestRes(1)**	**returnRes(1)**
0 ≤ avail don't give a non-ex. resource	if/do (!(1≤avail)) wait(av); avail--;	avail++; /* no wait! */ signal(av);
stronger invariant:		

0 ≤ avail && 0 ≤ inUse
... and don't take back more than have been given
	if/do (!(1≤avail)) wait(av);
	avail--; inUse++;
	signal(iu);
		if/do (!(1≤inUse)) wait(iu);
		avail++; inUse--;
		signal(av);

Monitor invariant	**requestRes(n)**	**returnRes(n)**
0 ≤ avail don't give a non-ex. resource	do (!(n≤avail)) wait(av[n]); avail = avail - n;	avail = avail + n; /* no wait! */ signal(av[1]); ... signal(av[avail]);

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

Monitor for resource allocation

A **monitor** grants access to a set of k >= 1 resources of the **same kind**.
Processes request n resources, 1<=n<=k, and return them after having used them.

Assumption: Process does not return more than it has received => 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!
   }
}
```


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();
  }
}
```


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;
```


Readers-Writers problem (Step 2)

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

Monitor invariant RW:

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

```
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;
```

Readers-Writers problem (Step3)

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

Monitor invariant RW:

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

```
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;

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;
```


Readers-Writers problem (Step 4)

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

Monitor invariant RW:

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

```
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}

   signal(okToWrite);
end;

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;
```


Readers-Writers problem (Step 5)

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

Monitor invariant RW:

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

```
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;

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;
```


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!
   }
}
```

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:

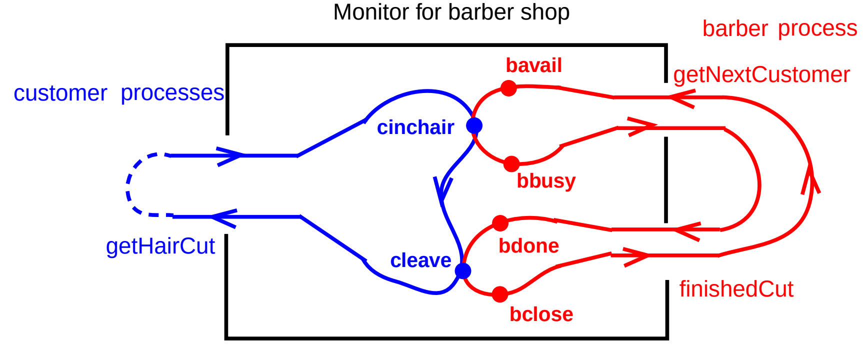
In a sleepy village close to Paderborn a barber is sleeping while waiting for customers to enter his shop. When a customer arrives and finds the barber sleeping, he awakens him, sits in the barber's chair, and sleeps while he gets his hair cut. If the barber is busy when a customer arrives, the customer 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.

Monitor design for the Sleeping Barber problem (step 1)



Counters represent states, incremented in entry procedures:

entry proc getHairCut:

cinchair++;
cleave++;

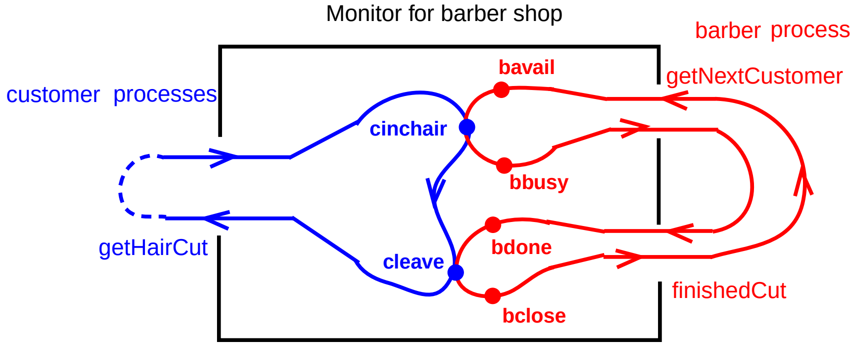
entry proc getNextCustomer:

bavail++;
bbusy++;

entry proc finishedCut:

bdone++;
bclose++;

Monitor invariant for the Sleeping Barber problem (step 2)



Invariants over counters:

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

C2: bavail >= cinchair >= bbusy

C3: bdone >= cleave >= bclose

Monitor invariant: BARBER: C1 and C2 and C3

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++;

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--;
```


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--;
```


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

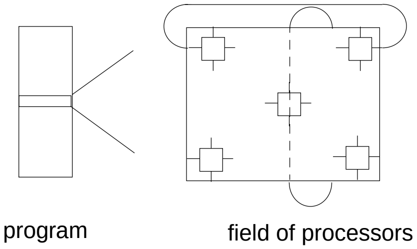
Applications mainly in **technical, scientific computing**, e. g.

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

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



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:

| C | = | A | + | 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
```

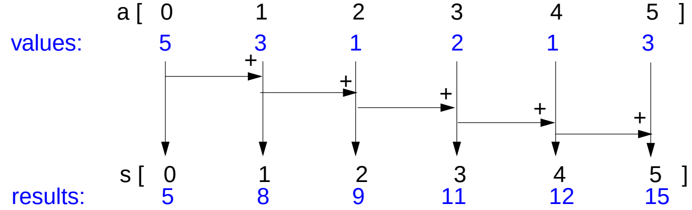
1 step!

- 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

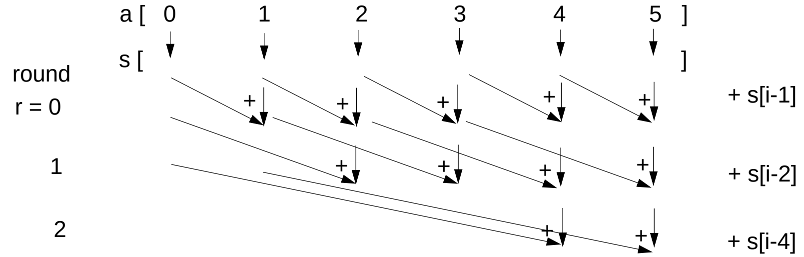
Example prefix sums

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:

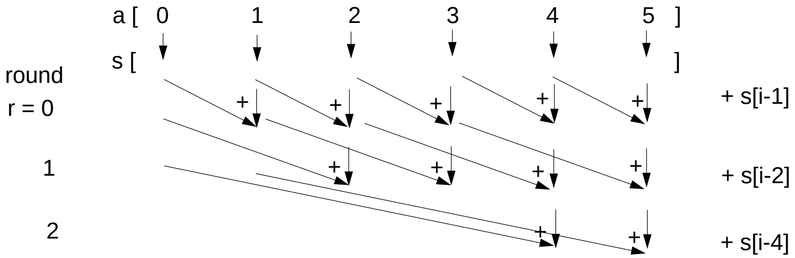


Example prefix sums (2)

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 .. n - 1

Invariant SUM: s[p] = a[p-d+1] + ... + a[p] with d = 1, 2, ..., m <= n distance before next round

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

induction step: computation s[p] = s[p - d] + s[p]
a[p-2d+1] + ... + a[p-d] + a[p-d+1] + ... + a[p]

substitution of 2d by d implies SUM

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

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)

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
} } }
```


Distributed tree barrier

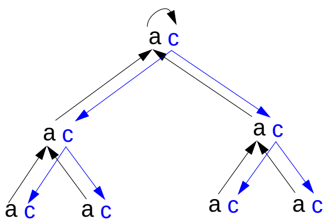
Barrier synchronization of the worker processes organized as a **binary tree**.
Bottleneck of central synchronization is avoided.

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



2 Rules for Synchronization Using Flags

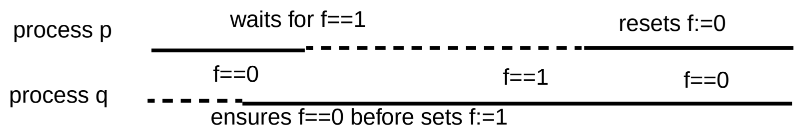
Flag for synchronization between exactly 2 processes

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.



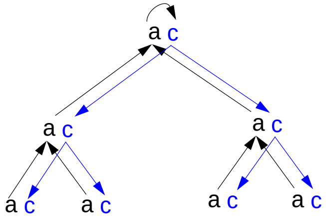
Distributed tree barrier implementation

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:

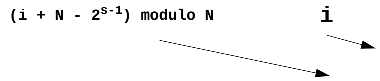
```
                                                  leaf     root
execute this.task();                              x        x
wait for left.arrived; reset left.arrived;                 x
wait for right.arrived; reset right.arrived;               x
set this.arrived;                                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

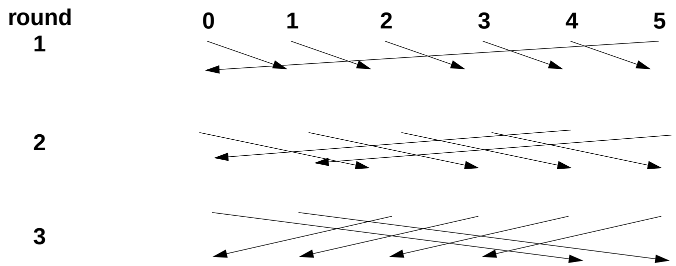

Symmetric, distributed barrier (dissemination)

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

N processes are synchronized after r rounds if N <= 2^r

In round s
process i indicates its arrival and then waits
for the arrival of process (i + N - 2^{s-1}) modulo N:

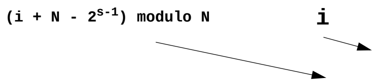




After r rounds each process is synchronized with each other. Proof idea: For each process i each other process occurs in a tree of processes which have synchronized (in)directly with i.

Symmetric, distributed barrier: implementation

In round s
process i indicates its arrival and
waits for the arrival of process (i + N - 2^{s-1}) modulo N



Code for each process:

```
execute this.task();

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


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
} } }
```

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
```

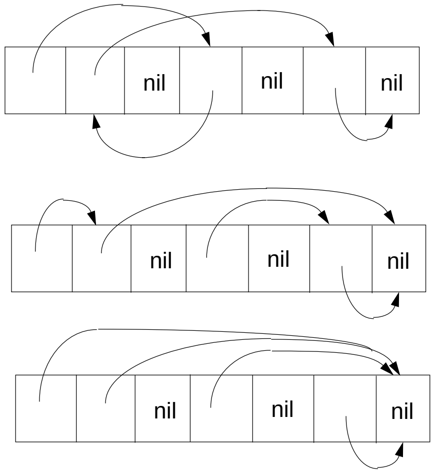

Finding list ends: data parallel approach

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

5.2 / 6. Data Parallelism: Loop Parallelization

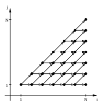
Regular loops on orthogonal data structures - parallelized for **data parallel** processors

Development steps (automated by compilers):

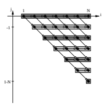
- **nested loops** operating on **arrays**, sequential execution of iteration space

```
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

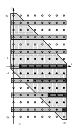


- **transform loops** keep data dependences forward in time



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

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



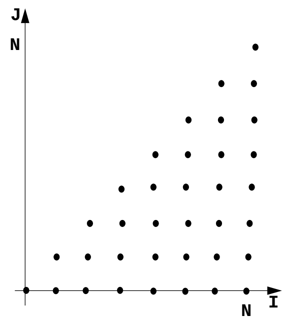
Iteration space of loop nests

Iteration space of a loop nest of depth n:

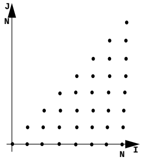
- **n-dimensional space of integral points** (polytope)
- each point (i_1, ..., i_n) represents an execution of the innermost loop body
- loop bounds are in general not known before run-time
- iteration need not have orthogonal borders
- iteration is elaborated sequentially

example:
computation of Pascal's triangle

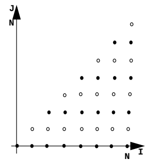
```
DECLARE B[-1..N,-1..N]

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



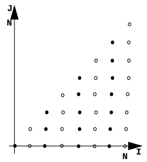
Examples for Iteration spaces of loop nests



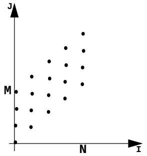

```
FOR I := 0 .. N
    FOR J := 0 .. I
```



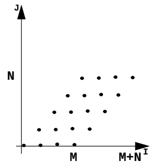

```
FOR I := 0 .. N
    FOR J := 0..I BY 2
```




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




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

M = 3, N = 4
```

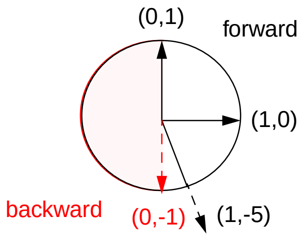



```
FOR I := 0 .. M+N
    FOR J := max(0, I-M)..
             min (I, N)
```


Data Dependences in Iteration Spaces

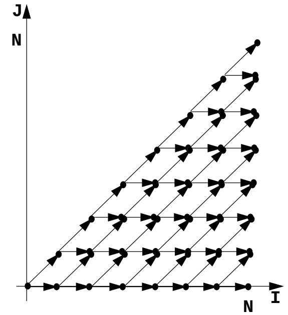
Data dependence from iteration point i1 to i2:

- Iteration **i1** computes a value that is used in iteration **i2** (flow dependence)
- relative **dependence vector**
  d = **i2 - i1** = ($i2_1$ - $i1_1$, ..., $i2_n$ - $i1_n$)
  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. **d** = (0, ..., 0, d_i, ...), 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
```



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

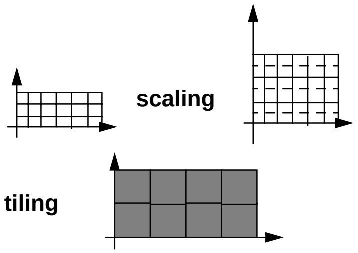
linear basic transformations:

- **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

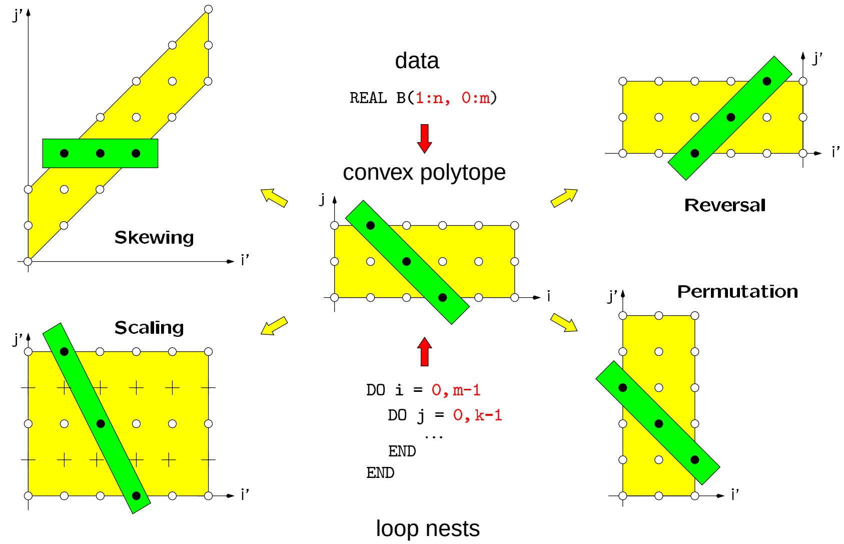
SRP transformations (next slides)

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



Transformations of



Transformations defined by matrices

Transformation matrices: systematic transformation, check dependence vectors

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

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

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

Reversal

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

general transformation matrix

$$\begin{pmatrix} 1 & & & & & 0 \\ & \ddots & & & & \\ & & 1 & & & \\ & & & \mathbf{-1} & & \\ & & & & 1 & \\ 0 & & & & & \ddots \; 1 \end{pmatrix}$$

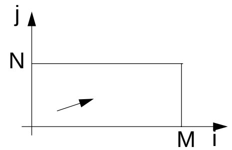
2-dimensional:

loop variables: old, new

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

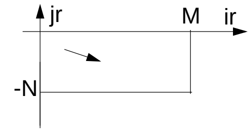
```
for i = 0 to M
   for j = 0 to N
      ...
```

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



original

transformed



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:

$$\begin{pmatrix} 1 & & & & & 0 \\ & \ddots & & & & \\ & & 1 & & & \\ & & \mathbf{f} & 1 & & \\ & & & & 1 & \\ 0 & & & & & \ddots \; 1 \end{pmatrix}$$

2-dimensional:

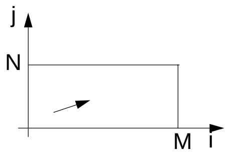
loop variables: old, new

$$\begin{pmatrix} 1 & 0 \\ f & 1 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} i \\ f*i+j \end{pmatrix} = \begin{pmatrix} is \\ js \end{pmatrix}$$

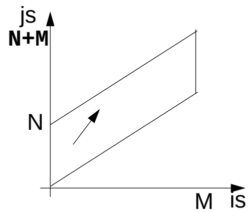
```
for i = 0 to M
   for j = 0 to N
      ...
```

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

original



transformed



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:

$$\begin{pmatrix} 1 & & & & & 0 \\ & \mathbf{0} & \mathbf{1} & & & \\ & & 1 & & & \\ & \mathbf{1} & \mathbf{0} & & & \\ 0 & & & 1 & \ddots & 1 \end{pmatrix}$$

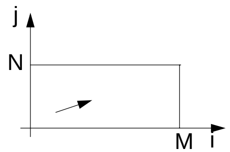
2-dimensional:

loop variables: old, new

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

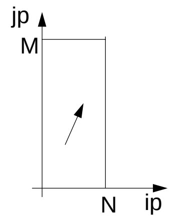
```
for i = 0 to M
   for j = 0 to N
      ...
```

original




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

transformed



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

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

Inequalities Describe Loop Bounds

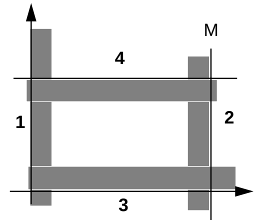
The bounds of a loop nest are described by a **set of linear inequalities**.
Each **inequality separates the space** in „inside and outside of the iteration space":

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

example 1

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

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

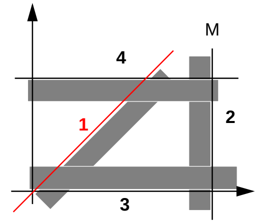


example 2

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

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

transformed



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

1, 4: $j \le \min(i, N)$ **1+ 3:** $0 \le i$
3: $0 \le j$ **2:** $i \le M$

Transformation of Loop Bounds

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

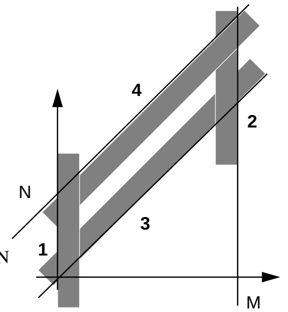
skewing $\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$ inverse $\begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix}$

$$\underset{B}{\begin{pmatrix} -1 & 0 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{pmatrix}} * \underset{T^{-1}}{\begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix}} = \underset{B * T^{-1}}{\begin{pmatrix} -1 & 0 \\ 1 & 0 \\ 1 & -1 \\ -1 & 1 \end{pmatrix}}$$

example 1
new bounds:

$$\underset{B * T^{-1}}{\begin{pmatrix} -1 & 0 \\ 1 & 0 \\ 1 & -1 \\ -1 & 1 \end{pmatrix}} * \underset{i'}{\begin{pmatrix} i' \\ j' \end{pmatrix}} \le \underset{c}{\begin{pmatrix} 0 \\ M \\ 0 \\ N \end{pmatrix}}$$

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



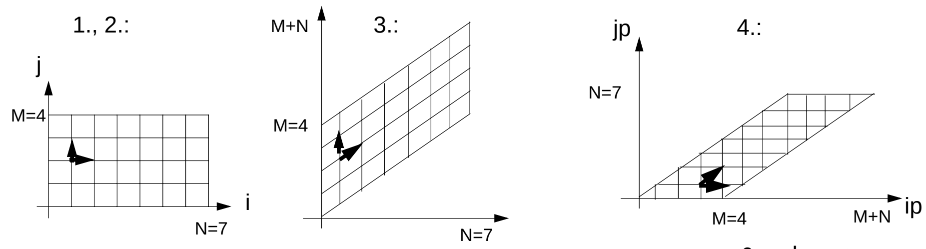
Example for Transformation and Parallelization of a Loop

```
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 ip and jp and new loop bounds.

Solution of the Transformation and Parallelization Example



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

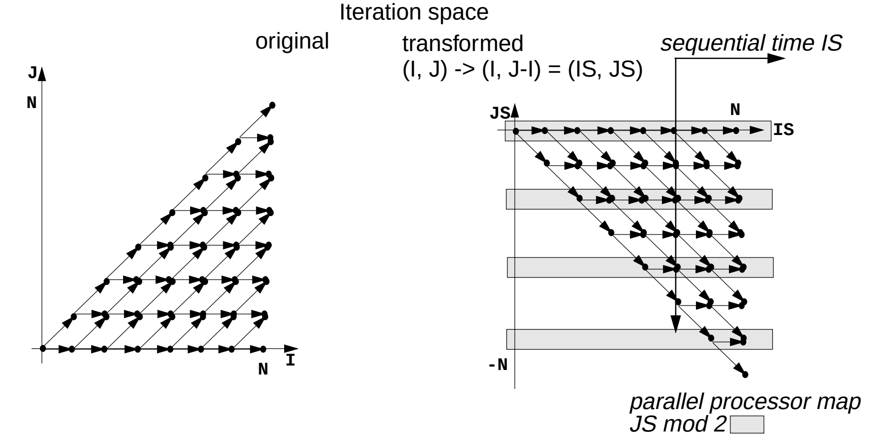
6.: Inverse $\begin{pmatrix}0 & 1\\ 1 & -1\end{pmatrix}$

7. Bounds: orig.: B $\begin{pmatrix}-1 & 0\\ 1 & 0\\ 0 & -1\\ 0 & 1\end{pmatrix}$ c $\begin{pmatrix}0\\ N\\ 0\\ M\end{pmatrix}$ new: $B * T^{-1}$ $\begin{pmatrix}0 & -1\\ 0 & 1\\ -1 & 1\\ 1 & -1\end{pmatrix}$

1	$-jp \le 0$	**1, 3 => $0 \le ip$**
2	$jp \le N$	**2, 4 => $ip \le M+N$**
3	$-ip+jp \le 0$	**1, 4 => max (0, ip-M) $\le$ jp**
4	$ip - jp \le M$	**2, 3 => jp $\le$ 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;
```


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
```


Data Mapping

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

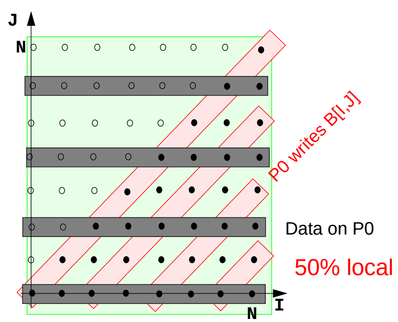
Index space of an array:
n-dimensional space of integral index points (polytope)

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

Data distribution for parallel loops

index space of B

original

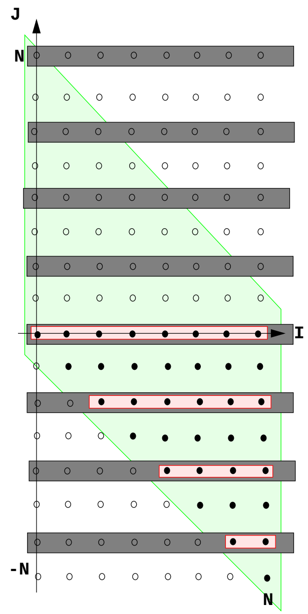


Data on P0

50% local

transformed
skewing f=-1
(i,j) -> (i,j-i)

100%local




```
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
```

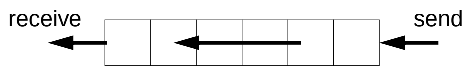
```
DECLARE B[-1..N,-N..N]
...
   B[IS,JS] :=
      B[IS-1,JS-1]+B[IS-1,JS-1]
```

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; **block**s 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.

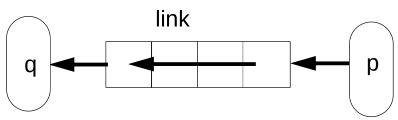
Channels implemented in Java

```
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 (); }
}
```

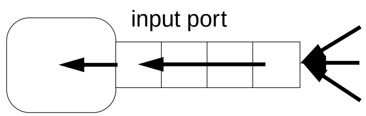
All waiting processes wait for the same condition => notify() is sufficient.
After a notify-call a new receive-call may have stolen the only message => wait loop is needed

Processes and channels

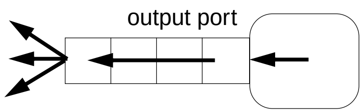
link:
one sender is connected to **one receiver**;
e. g. processes form chains of transformation steps (pipeline)



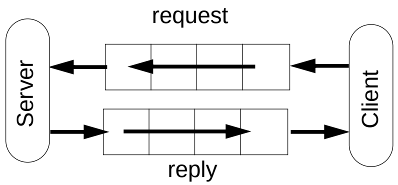
input port of a process:
many senders - one receiver;
channel belongs to the receiving process;
e. g. a server process receives tasks from several client processes



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



Termination conditions

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

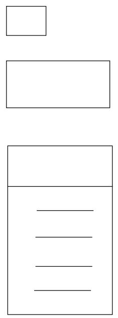
1. **All channels are empty.**
2. **No** processes are **blocked on a receive** operation.
3. **All** processes are **terminated**.

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**.

Message structures

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



empty — **synchronization** only

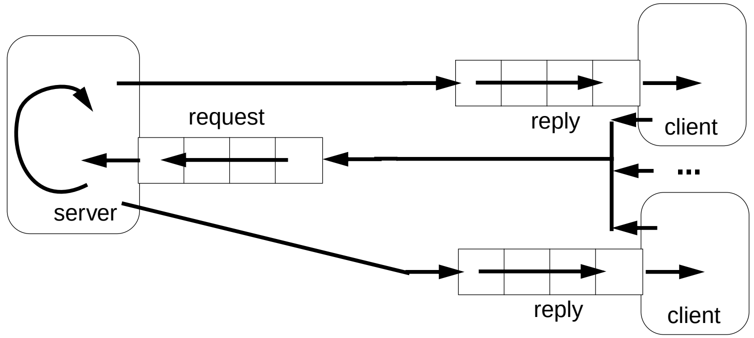
kind — different **kinds** of messages, without data
e. g. signal different kinds of events

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 ()

Client / server: basic channel structure

One **server process responds to requests of several client processes**



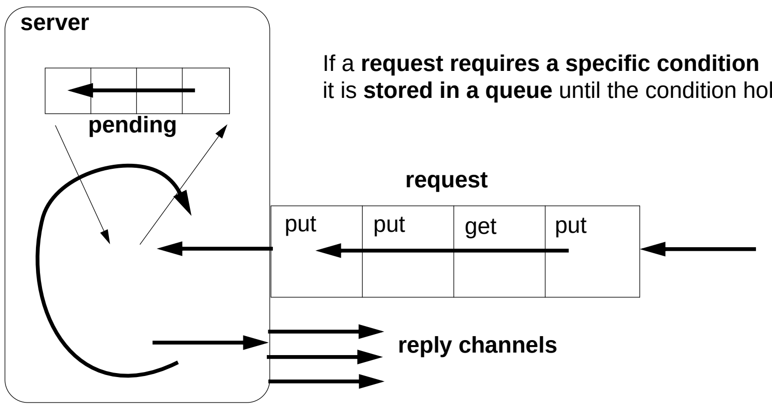
request channel:
input port of the server

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

Server processes: different kinds of operations

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

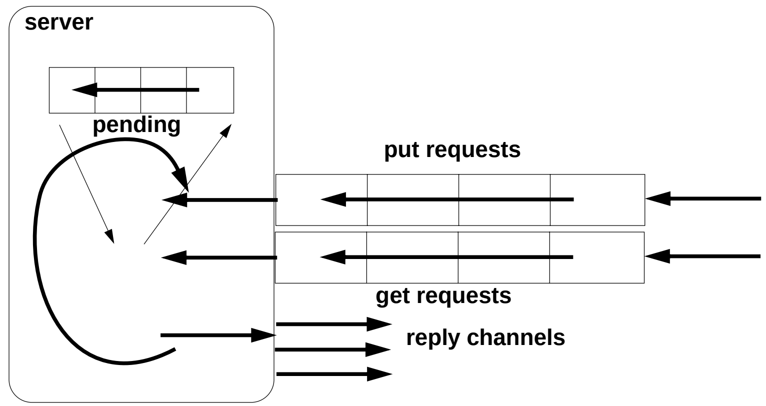


If a **request requires a specific condition** it is **stored in a queue** until the condition holds.

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.

Different kinds of operations on different channels

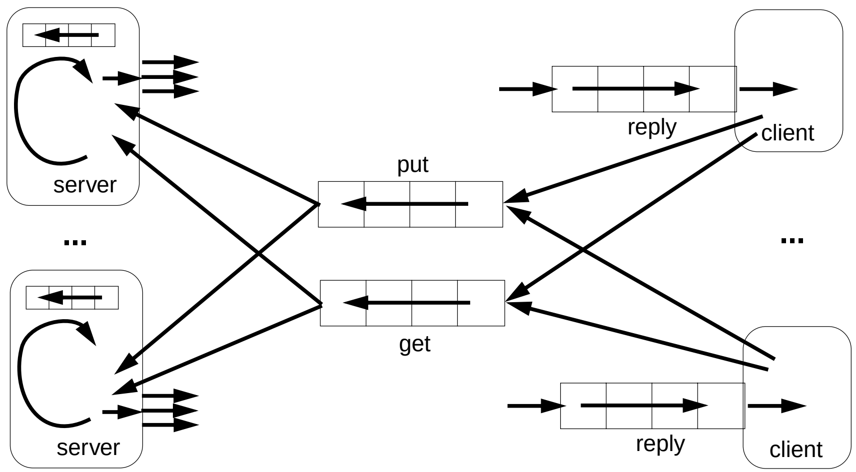


Server must not block on an empty input port while another port may be non-empty:

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

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!

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;
}  }
```

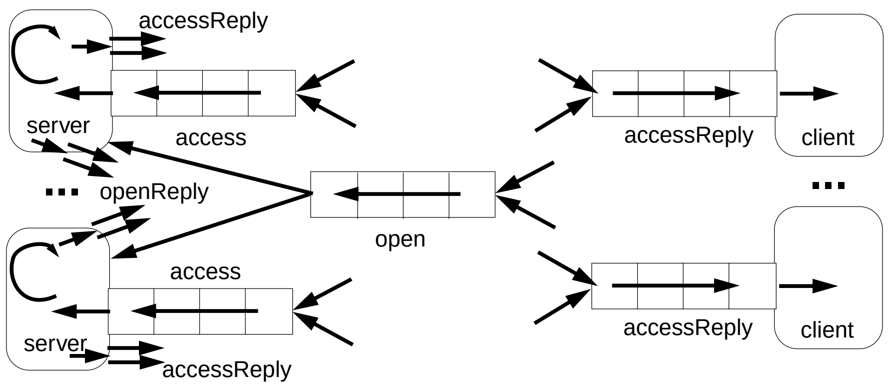
Checking several channels:

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


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.



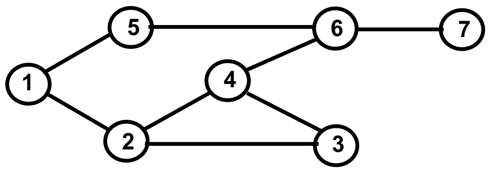
Active monitor (server) vs. passive monitor

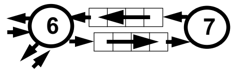
active monitor		passive monitor
	1. program structure	
active process		passive program module
	2. client communication	
request - reply via channels		calls of entry procedures
	3. server operations	
kinds of messages and/or different channels		entry procedures
	4. mutual exclusion	
requests are handled sequentially		guaranteed for entry procedure calls
	5. delayed service	
queue of pending requests replies are delayed		client processes are blocked condition variables, wait - signal
	6. multiple servers	
may cooperate on the same request channels		multiple monitors are not related

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

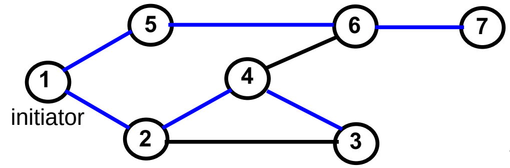
Broadcast method

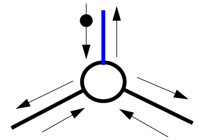
Method (for all nodes but the initiator node):
1. The node waits for a message on its incoming links.
2. After having **received the first message** it sends a **copy to all of its n neighbours** - including to the sender of the first message
3. The node then receives **n-1 redundant messages** from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



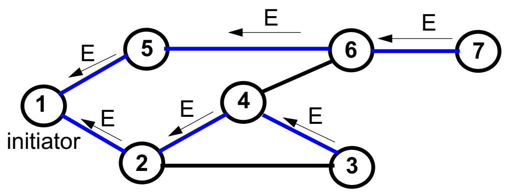


total number of messages: 2*|edges|

Probe and echo in a net

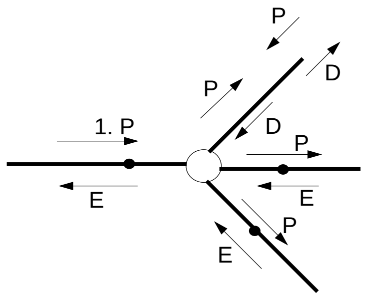
Task: An initiator requests combined **information from all nodes** in the graph (**probe**).
The information is **combined** on its way through the net (**echo**);
e. g. sum of certain values local to each node, topology of the graph, some global state.

Method (roughly):
- distribute the probes like a broadcast,
- let the first reception determine a spanning tree,
- return the echoes on the spanning tree edges.



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**.

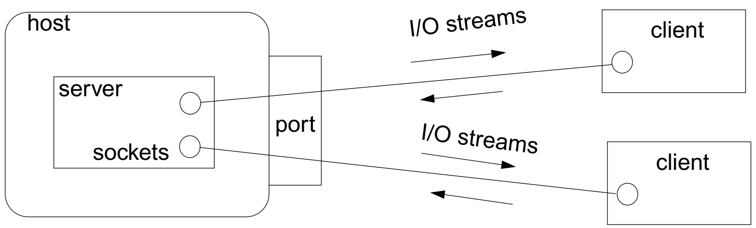


2 messages are sent on each **spanning tree edge**.

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

Connections via ports and sockets

Port:
- an **abstract connection point** of a computer; numerically encoded
- a **sever 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.

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();
```


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**.

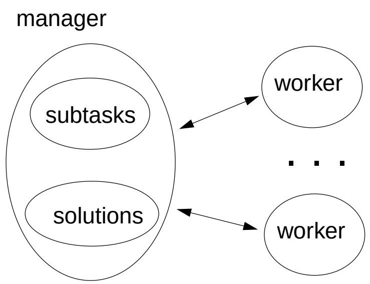
Speedup if the processes are executed
in parallel on different processors.

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.



Branch and Bound

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

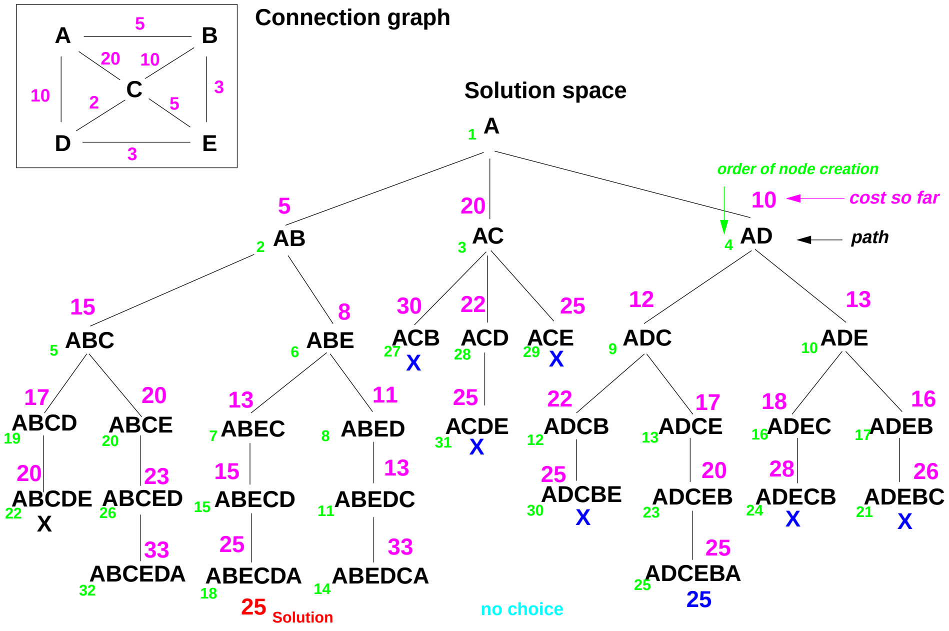
tree structured solution space is searched for a best solution

General scheme of operations:
- **partial solution S is extended** to S_1, S_2, ... (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

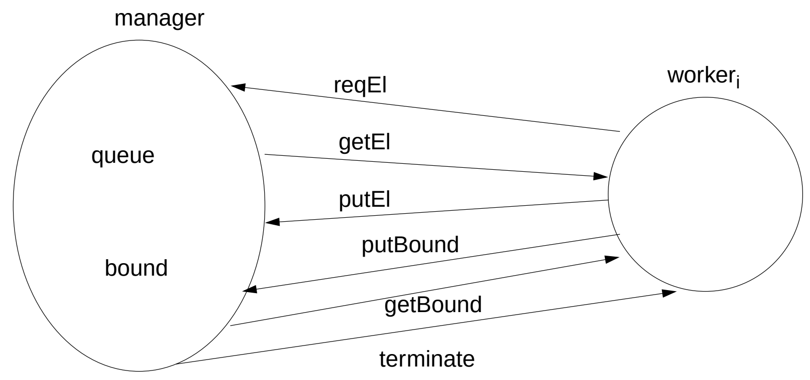
B&B example: Travelling sales person



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

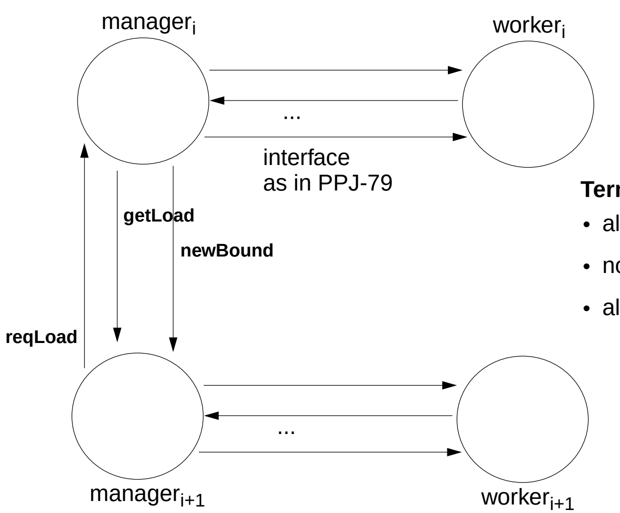


Protocol: reqEl (getEl [getBound] (putEl | putBound)* reqEl)* terminate
for a single Worker

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



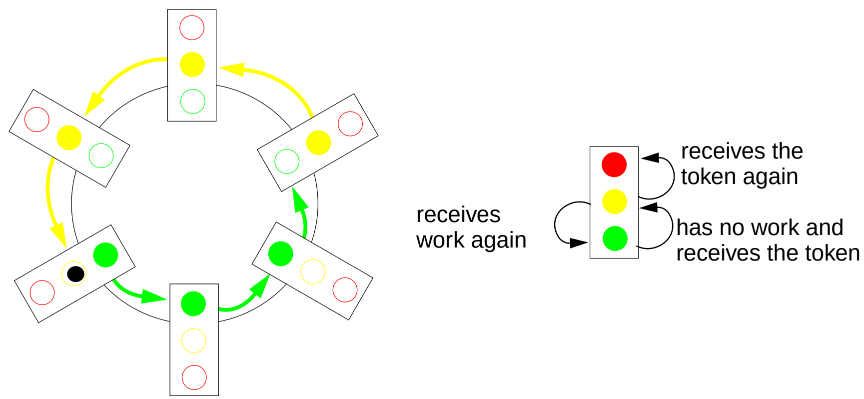
Termination condition:

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

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

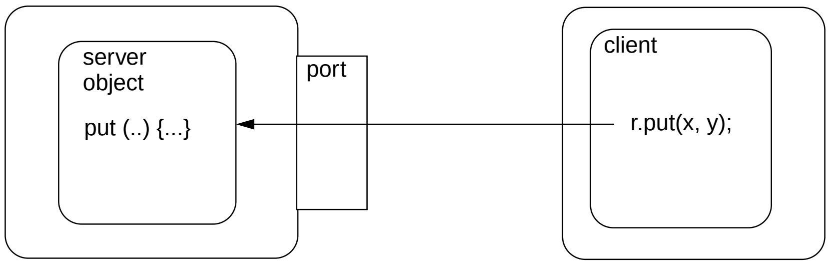


Method calls for objects on remote machines (RMI)

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.

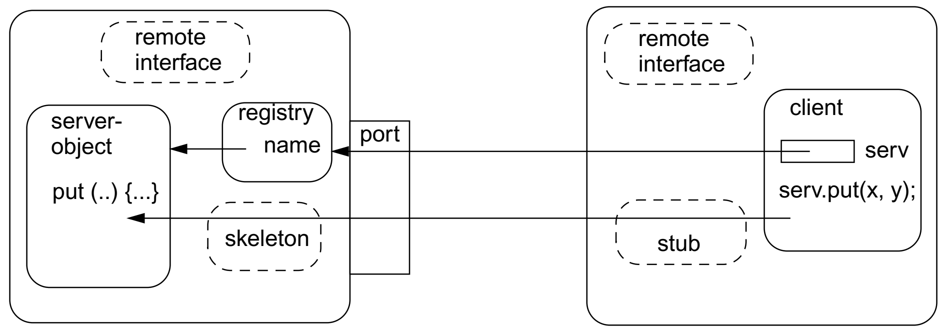
Comparable techniques: CORBA with IDL, Microsoft DCOM with COM



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)

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

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
```


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.

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; }
}
```


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]

Notations for synchronous message passing

Notation in CSP und Occam:

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

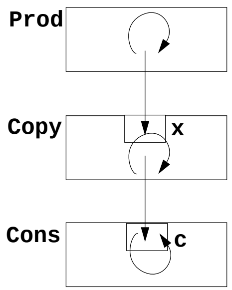
`q:   ...  p ? v   ...` **receive** a value from process **p** and assign it to variable **v**

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
```



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
```

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
```


Prefix sums computed with synchronous messages

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

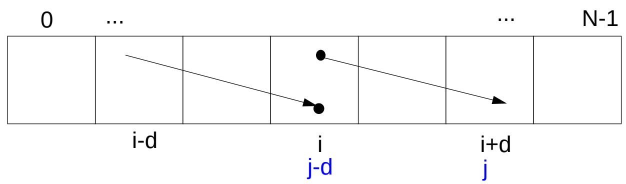
Necessary synchronization only (cf. synchronous barriers, PPJ-48)

```
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
      d := 2*d                                     double the distance
   od                                              {SUM and d >= N-1}
end
```

Why can deadlocks not occur?

No deadlocks in synchronous prefix sums

sychronization pattern



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

  if i+d < N and i>=0 process i executes `Worker(i+d)!sum`
  let j = i+d, i.e. j-d = i >= 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 >= N, then process i only receives:
  Prove by induction.

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

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
```

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.

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
```


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)
```


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;
```

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);
                end Get;
                dequeue(buf);
        end select;
    end loop;
end Buffer;
```

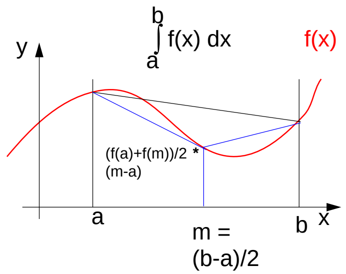
```
task type Producer;

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;
```


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

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 eps from the **area of the big trapezoid**.

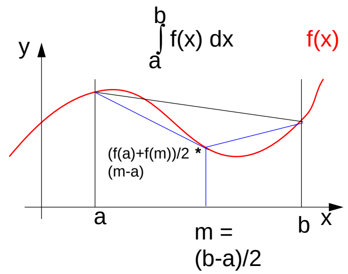
See [G. Andrews: Foundations of Multithreaded, Parallel, and Distributed Programming, Addison Wesley, 2000, pp. 17-19]

```
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

      lar = quad(f,l,m,larea,eps) and

      rar = quad(f,m,r,rarea,eps)

   in (lar+rar)
   end
   else area
end
```

initial call:

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


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 eps 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)
```

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
```

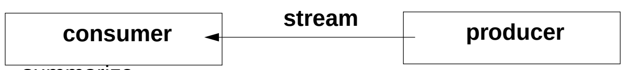
Eager evaluation: all elements of a list are to be computed, before any can be accessed.
Lazy evaluation only those elements of a list are computed which are going to be accessed.

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**.



consumer	producer
summarize	sequence of numbers
use random numbers	random numbers
decide upon convergence	iterate approximations
decide upon solution	enumerate solutions space

Functions on streams can be understood as communicating concurrent processes.

Examples for stream functions (1)

produce a stream of numbers: `int -> int seq`

```
fun from k = Cons (k, fn()=> from (k+1));
```

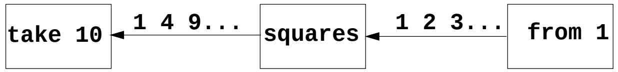
consume the first n elements into a list: `'a seq * int -> 'a list`

```
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()));
```

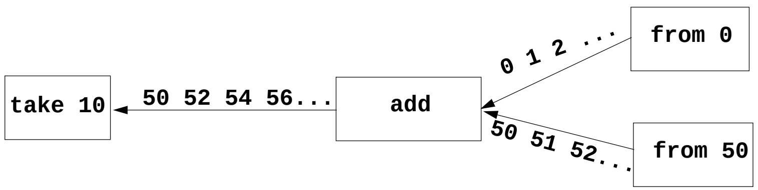
```
take        (squares        (from 1), 10);
```



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;
```



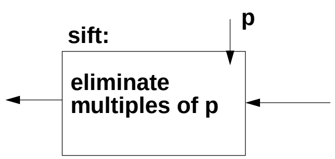
Filter-Schema:

```
('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());
```

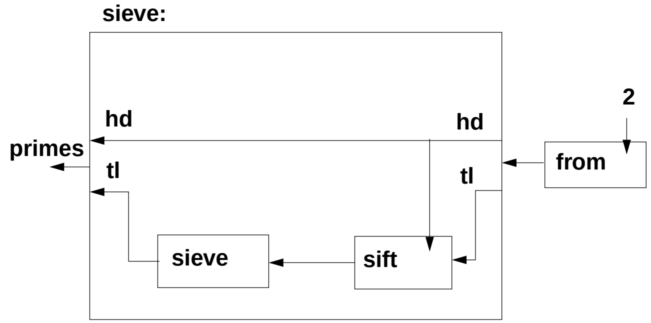
Recursive stream composition

```
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!

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
```

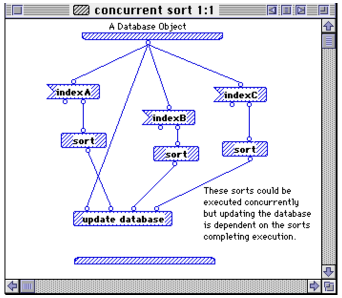
[G. Andrews: Foundations of Multithreaded, Parallel, and Distributed Programming, Addison Wesley, 2000, pp. 326-328]

Dataflow languages

Textual languages:

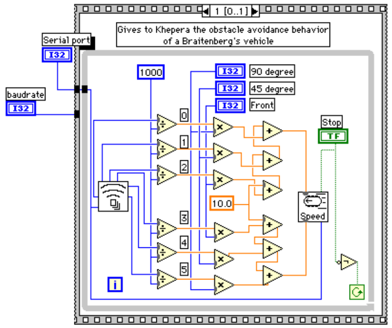
Lucid: stream computations by equations, no side effects; 1976, Wadge, Ashcroft

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.

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

Basic communication constructs

process creation:

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

asynchonous 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

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.
```


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.
```


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.
```


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

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

[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]

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) => ...
   }
}
```


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.

[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]

Example: orders and cancellations

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


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?

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.

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.

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.

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.