ETH Zürich

Chair of Software Engineering

Concepts and Constructs of Concurrent Computation

Bertrand Meyer
Sebastian Nanz

Lecture 1: Overview

Practical Details

- Schedule
  - Tuesday 10-12, RZ F21: **course**
  - Wednesday 14-15, RZ F21: **exercise**
  - Wednesday 15-16, RZ F21: **seminar**
  - Wednesday 16-17, RZ F21: **seminar** or **exercise**
- Course page
  - http://se.inf.ethz.ch/courses/2012a_spring/ccc/
- Lecturers
  - Prof. Dr. Bertrand Meyer
  - Dr. Sebastian Nanz
  - Guest lecturer: Prof. Hassan Gomaa, George Mason University (Va, USA)
- Assistants
  - Benjamin Morandi
  - Scott West

firstname.lastname@inf.ethz.ch

Seminar

- The seminar has lectures of two types:
  - Lectures given by international experts (e.g. Moti Ben-Ari, Bill Roscoe, Eric Jul, André Seznec)
  - Short student presentations (20 min + questions) on a research paper on concurrency
- Paper selection:
  - You will get an email today, with a list of papers and instructions for e-mailing us your choice
  - You must respond no later than **Friday, 24 February, 16:00**
  - If you don't get the email today or miss the deadline, please email the assistants

Grading

Exam: 50%

- End of semester (not in the semester break)
- Date: **29 May 2012** at usual lecture time

Project: 35% (build a small concurrent system)

Seminar talk: 15%

This is a challenging course; the project will be demanding. Hence the 7 credit points. Do not take the course unless you are prepared to devote significant effort to it.

Purpose of the course

- To give you a practical grasp of the excitement and difficulties of building modern concurrent applications
- To expose you to newer forms of concurrency
- To introduce you to the main concurrency approaches and give you an idea of their strength and weaknesses
- To present some of the concurrency calculi
- To study in on particular approach in depth: SCOOP
- To enable you to get a concrete grasp of the issues and solutions through a course project
- To connect to recent research through a seminar

Course overview

Introduction

Concurrent and parallel programming, Multitasking and multiprocessing, **Shared-memory and distributed-memory multiprocessing**, Notion of process and thread, **Performance of concurrent systems**

Approaches to concurrent programming

Issues (data races, deadlock, starvation), **Synchronization algorithms**, **Semaphores**, **Monitors**, Java and .NET multithreading

The SCOOP model

Processors, Synchronous and asynchronous feature calls, **Separate objects and entities**, Synchronization, Examples and applications

Programming approaches to concurrency

Message-passing vs. shared-memory communication, **Language examples** (Ada, Polyphonic C#, Erlang (Actors), X10, Linda, Cilk and others), **Lock-free programming**, **Software Transactional Memory**

Reasoning about concurrent programs

Properties of concurrent programs, **Temporal logic**, **Process calculi** (CSP, CCS), Proofs of concurrent programs

ETH Zürich

Chair of Software Engineering

Concurrency:
benefits and challenges

Why concurrency?

Concurrency is not a new topic but one most programmers have been able to avoid

Previously perceived as a very specialized topic: high-performance computing, systems programming, databases

Reasons for introducing concurrency into programs:

- Efficiency
  - Time (load sharing)
  - Cost (resource sharing)
- Availability
  - Multiple access
- Convenience
  - Perform several tasks at once
- Modeling power
  - Describing systems that are inherently parallel

Modeling a concurrent world

Computer systems are used for modeling objects in the real world

- Object-oriented programming

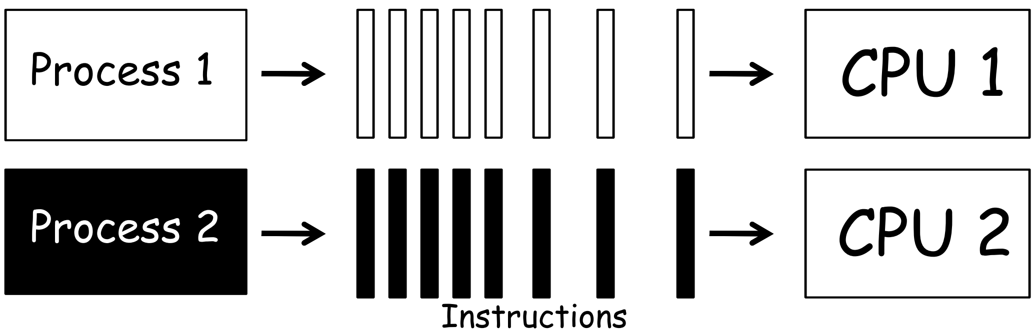
The world often includes parallel operation

Typical example:

- Limited number of seats on the same plane
- Several booking agents active at the same time

Multiprocessing, parallelism

Many of today's computations can take advantage of multiple processing units (through *multi-core* processors):



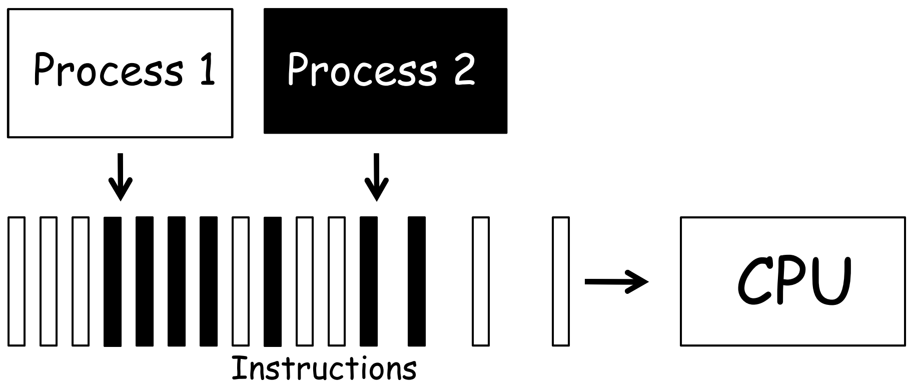
Terminology:

- **Multiprocessing** : the use of more than one processing unit in a system
- **Parallel execution**: processes running at the same time

Multitasking, concurrency

Even on systems with a single processing unit we may give the illusion of that several programs run at once

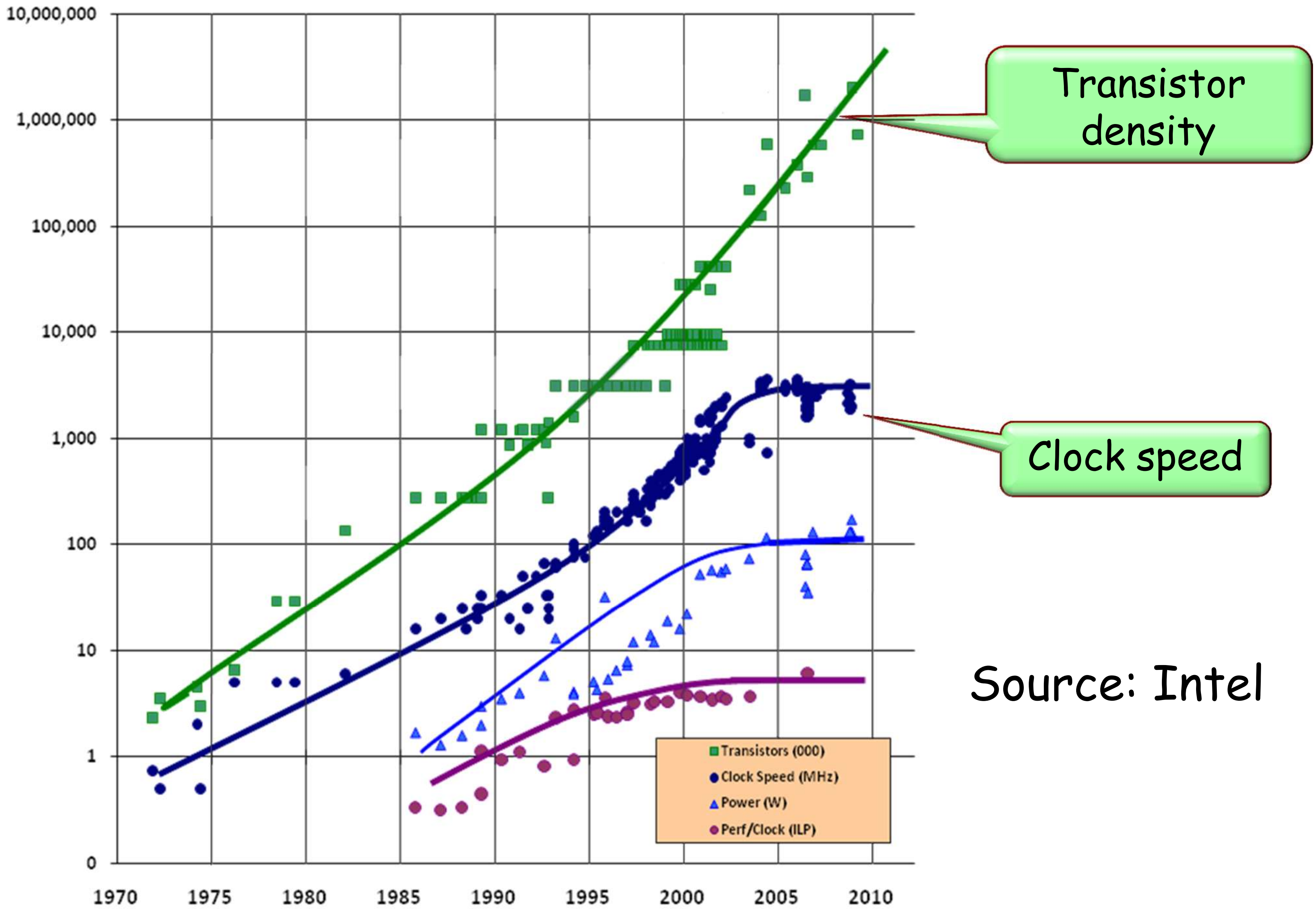
The OS switches between executing different tasks



Terminology:

- **Interleaving**: several tasks active, only one running at a time
- **Multitasking**: the OS runs interleaved executions
- **Concurrency**: multiprocessing, multitasking, or any combination

The end of Moore's Law as we knew it



Source: Intel

Why do we care?

- The "end of Moore's law as we knew it" has important implications on the software construction process
- Computing is taking an irreversible step toward parallel architectures
    - Hardware construction of ever faster sequential CPUs has hit physical limits
    - Clock speed no longer increases for every new processor generation
    - Moore's Law expresses itself as exponentially increasing number of processing cores per chip
- If we want programs to run faster on the next processor generation, the software must exploit more concurrency

Amdahl's Law*

We go from 1 processor to *n*. What gain may we expect?

Amdahl's law severely limits our hopes!

Define gain as: $$speedup = \frac{old\_execution\_time}{new\_execution\_time}$$

Not everything can be parallelized!

$$speedup = \frac{1}{1 - p + (p / n)}$$

Sequential part: $1 - p$

Parallel part: (p / n)

p: % parallelizable

n: Number of processors

*3 slides adapted from material by Maurice Herlihy

Amdahl's law: Example (1)*

Assume 10 processing units. How close are we to a 10-fold speedup?

- 60% concurrent, 40% sequential:

$$speedup = \frac{1}{1 - 0.6 \quad + \quad (0.6\ /\ 10\ )} = 2.17$$

- 80% concurrent, 20% sequential:

$$speedup = \frac{1}{1 - 0.8 \quad + \quad (0.8\ /\ 10\ )} = 3.57$$

Amdahl's law: Example (2)*

- 90% concurrent, 10% sequential:

$$speedup = \frac{1}{1 - 0.9 \quad + \quad (0.9 \,/\, 10\,)} = 5.26$$

- 99% concurrent, 1% sequential:

$$speedup = \frac{1}{1 - 0.99 + (0.99 \,/\, 10\,)} = 9.17$$

Types of parallel computation

Flynn's taxonomy: classification of computer architectures

Considers relationship of instruction streams to data streams:

	Single Instruction	Multiple Instruction
Single Data	SISD	
Multiple Data	SIMD	MIMD

- **SISD:** No parallelism (uniprocessor)

- **SIMD:** Vector processor, GPU
- **MIMD:** Multiprocessing (predominant today)

MIMD variants

SPMD (Single Program Multiple Data):

- All processors run same program, but at independent speeds; no lockstep as in SIMD

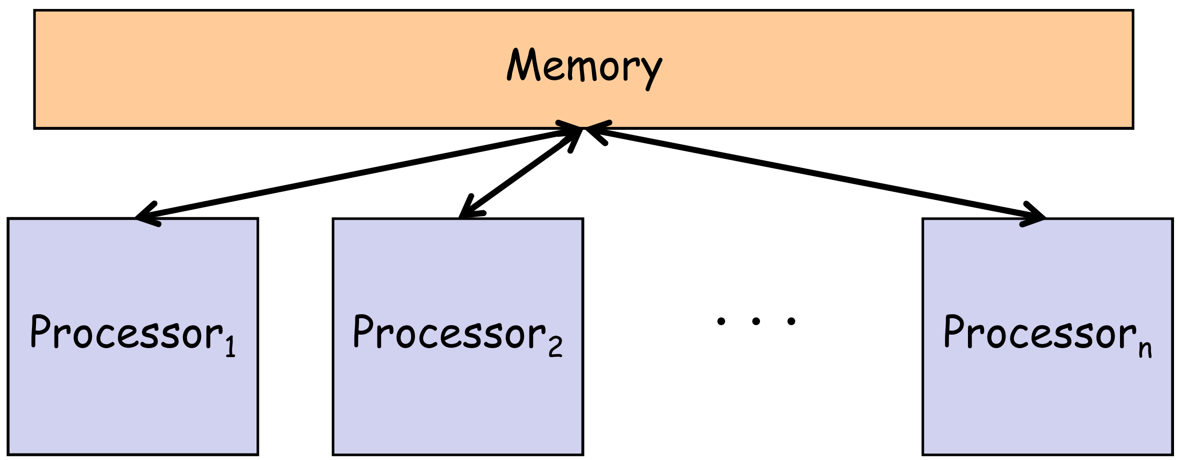
MPMD (Multiple Program Multiple Data):

- Often manager/worker strategy: manager distributes tasks, workers return result to manager

Shared memory model

All processors share a common memory

Shared-memory communication

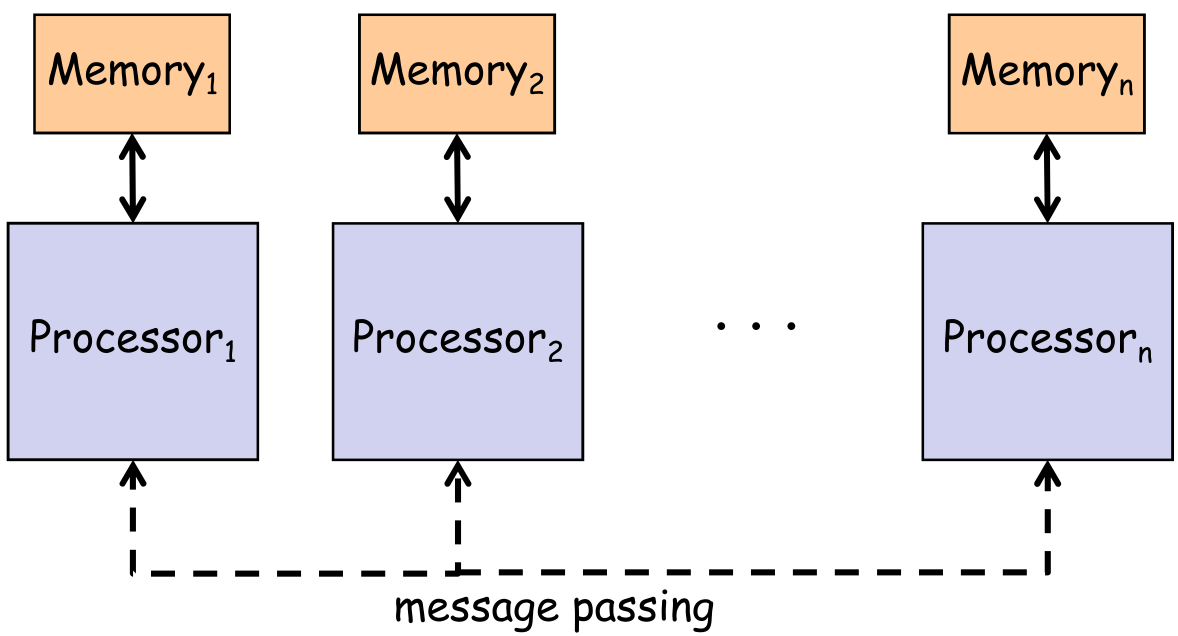


Distributed memory model

Each processor has own local memory, inaccessible to others

Message passing communication

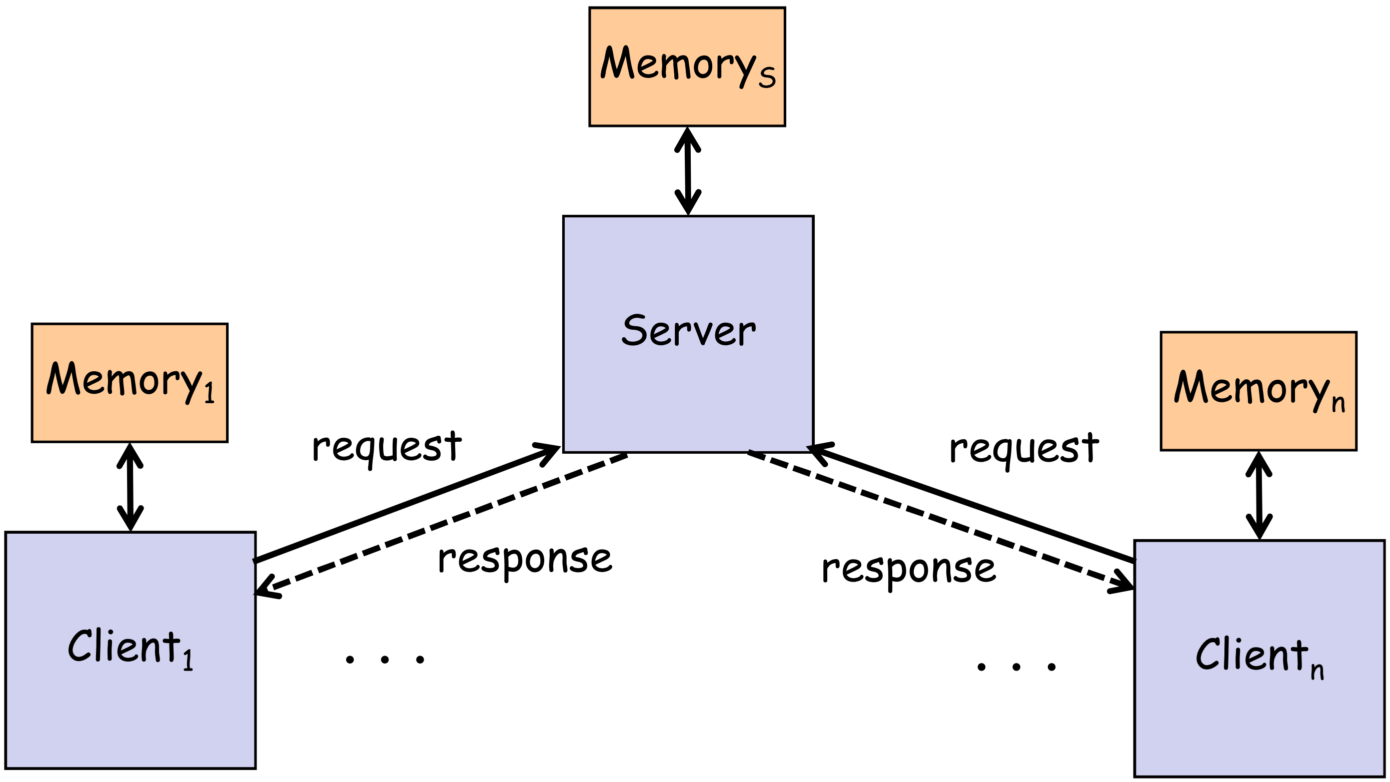
Common for SPMD architecture



Client-server model

Specific case of the distributed model

Examples: Database-centered systems, World-Wide Web



SCOOP: the trailer

SCOOP mechanism

Simple Concurrent Object-Oriented Programming

Evolved through previous two decades; CACM (1993) and chap. 32 of *Object-Oriented Software Construction*, 2nd edition, 1997

Prototype-implementation at ETH in 2007

Implementation integrated within EiffelStudio in 2011 (by Eiffel Software)

Current reference: ETH PhD Thesis by Piotr Nienaltowski, 2008; articles by Benjamin Morandi, Sebastian Nanz and Bertrand Meyer, 2010-2011

SCOOP preview: a sequential program

```
transfer (source, target:                 ACCOUNT;
          amount: INTEGER)
      -- If possible, transfer amount from source to target.
   do
      if source.balance >= amount then
          source.withdraw  (amount)
          target.deposit   (amount)
      end
   end
```

Typical calls:

```
    transfer (acc1, acc2, 100)
    transfer (acc1, acc3, 100)
```

In a concurrent setting, using SCOOP

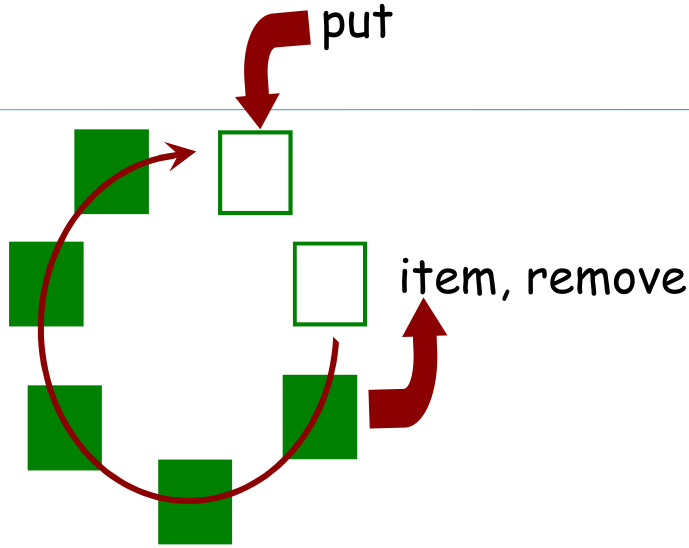
```
transfer (source, target: separate ACCOUNT;
          amount: INTEGER)
        -- If possible, transfer amount from source to target.
    do
        if source.balance >= amount then
            source.withdraw  (amount)
            target.deposit   (amount)
        end
    end
```

Typical calls:

```
    transfer (acc1, acc2, 100)
    transfer (acc1, acc3, 100)
```

A better SCOOP version

```
transfer (source, target: separate ACCOUNT;
            amount: INTEGER)
        -- Transfer amount from source to target.
    require
         source.balance >= amount
    do
        source.withdraw  (amount)
        target.deposit    (amount)
    ensure
        source.balance = old source.balance – amount
        target.balance = old target.balance + amount
    end
```

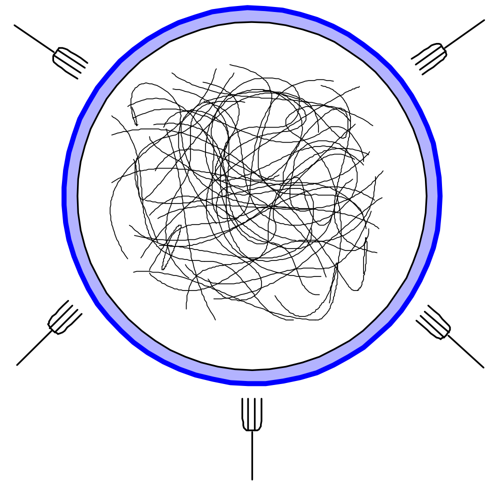



```
my_queue : BUFFER QUEUE [T ]
...
if not my_queue.is_full then

    put (my_queue, t )
end
```

```
put (b: BUFFER QUEUE [G ] ; v: G )
        -- Store v into b.
    require
        not b.is_full
    do
        ...
    ensure
        not b.is_empty
    end
```

Dining philosophers



```
class PHILOSOPHER inherit
      PROCESS
            rename
                  setup as getup
            redefine step end

feature {BUTLER}
      step
            do
                  think ;   eat (left, right)
            end

      eat (l, r: separate FORK)
                  -- Eat, having grabbed l and r.
            do ... end
end
```

The issue

Concurrency everywhere:

- Multithreading
- Multitasking
- Networking, Web services, Internet
- Multicore

Can we bring concurrent programming
to the same level
of abstraction and convenience
as sequential programming?

Previous advances in programming

	"Structured programming"	"Object technology"
Use higher-level abstractions	✓	✓
Helps avoid bugs	✓	✓
Transfers tasks to implementation	✓	✓
Lets you do stuff you couldn't before	NO	✓
Removes restrictions	NO	✓
Adds restrictions	✓	✓
Has well-understood math basis	✓	✓
Doesn't require understanding that basis	✓	✓
Permits less operational reasoning	✓	✓

Then and now

Sequential programming:

Used to be messy

Still hard but key improvements:

- Structured programming
- Data abstraction & object technology
- Design by Contract
- Genericity, multiple inheritance
- Architectural techniques

Concurrent programming:

Used to be messy

Still messy

Example: threading models in most popular approaches

Development level: sixties/seventies

Only understandable through operational reasoning

The chasm

Theoretical models, process calculi provide an elegant theoretical basis, but

- have little connection with practice (some exceptions, e.g. BPEL)
- handle concurrency aspects only

Practice of concurrent & multithreaded programming

- Little influenced by above
- Low-level, e.g. semaphores
- Poorly connected with rest of programming model