Lesson 2

Concurrency at Programming Language Level

Ch 2 [BenA 06]

Abstraction
Pseudo-language
BACI
Ada, Java, etc.

Levels of Abstraction

- Granularity of operations
  - Invoke a library module
  - Statement in high level programming language
  - Instruction in machine language
- Atomic statement
  - Anything that we can guarantee to be atomic
    - Executed completely “at once”
    - Always the same correct atomic result
    - Result does not depend on anybody else
  - Can be at any granularity
  - Can *trust* on that atomicity

Atomic Statement

- Atomicity guaranteed somehow
  - Machine instruction: HW
    - Memory bus transaction
  - Programming language statement, set of statements, or set of machine instructions
    - SW
      - Manually coded
      - Disable interrupts
      - OS synchronization primitives
  - Library module
    - SW
      - Manually coded inside
      - Provided automatically to the user by programming environment

Load R1, Y

Read mem(0x35FA8300)

?

```
-- start atomic
Load R1, Y
Sub R1, =1
Jpos R1, Here
-- end atomic
```

Monitors
Ch 7 [BenA 06]

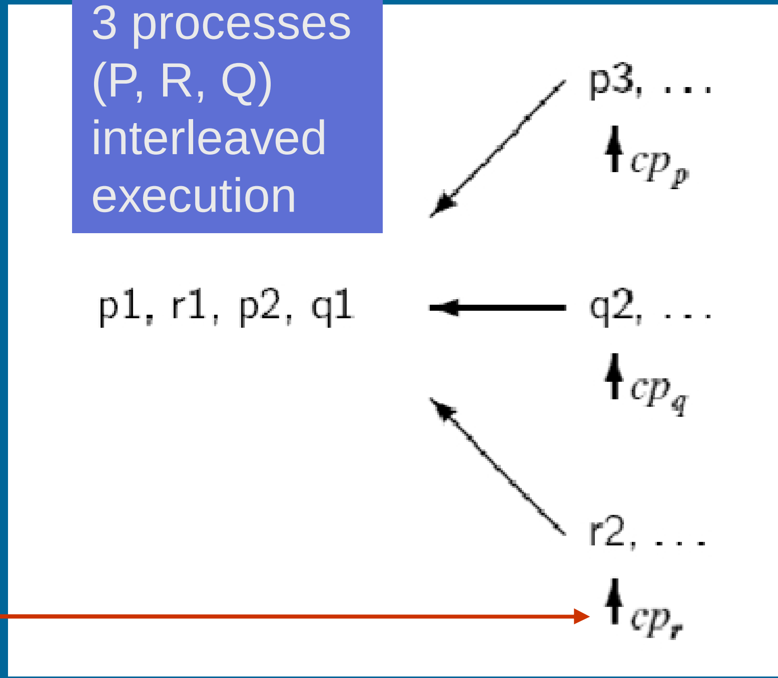
Concurrent Program

- Sequential process
  - Successive atomic statements

P: p1 → p2 → p3 → p4 …

  - Control pointer (= program counter)
- Concurrent program
  - Finite set of sequential processes working for same goal
  - Arbitrary interleaving of atomic statements in different processes

3 processes (P, R, Q) interleaved execution



P: p1 → p2

Q: q1 → q2

p1→q1→p2→q2,
p1→q1→q2→p2,
p1→p2→q1→q2,
q1→p1→q2→p2,
q1→p1→p2→q2,
q1→q2→p1→p2.

?

~~p1 → q2 → p2 → q1~~

Program State, Pseudo-language

pseudo-kieli

- Sequential program

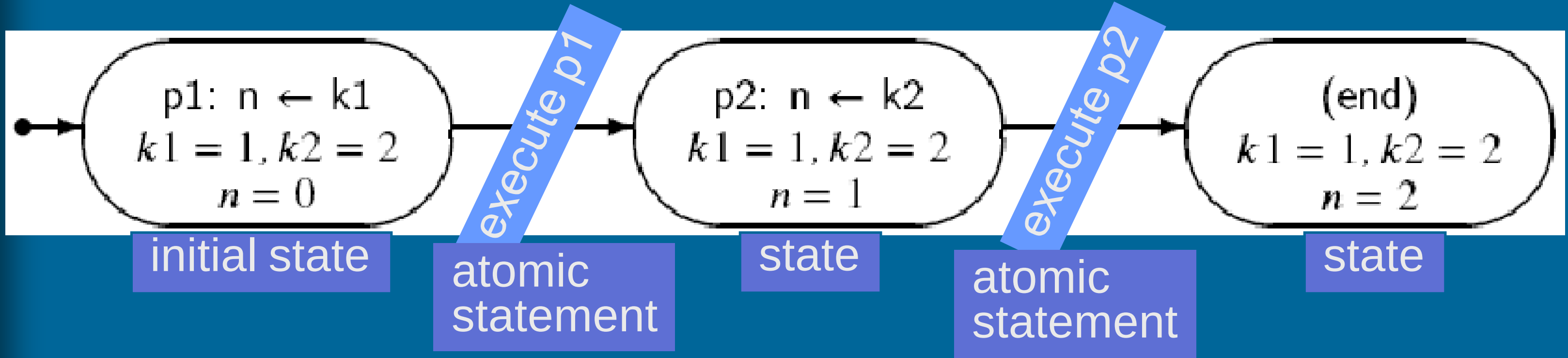
Algorithm 2.2: Trivial sequential program

```
integer n ← 0
      integer k1 ← 1
      integer k2 ← 2
p1:   n ← k1
p2:   n ← k2
```

cp

state

- State
  - next statement to execute (cp, i.e., PC)
  - variable values

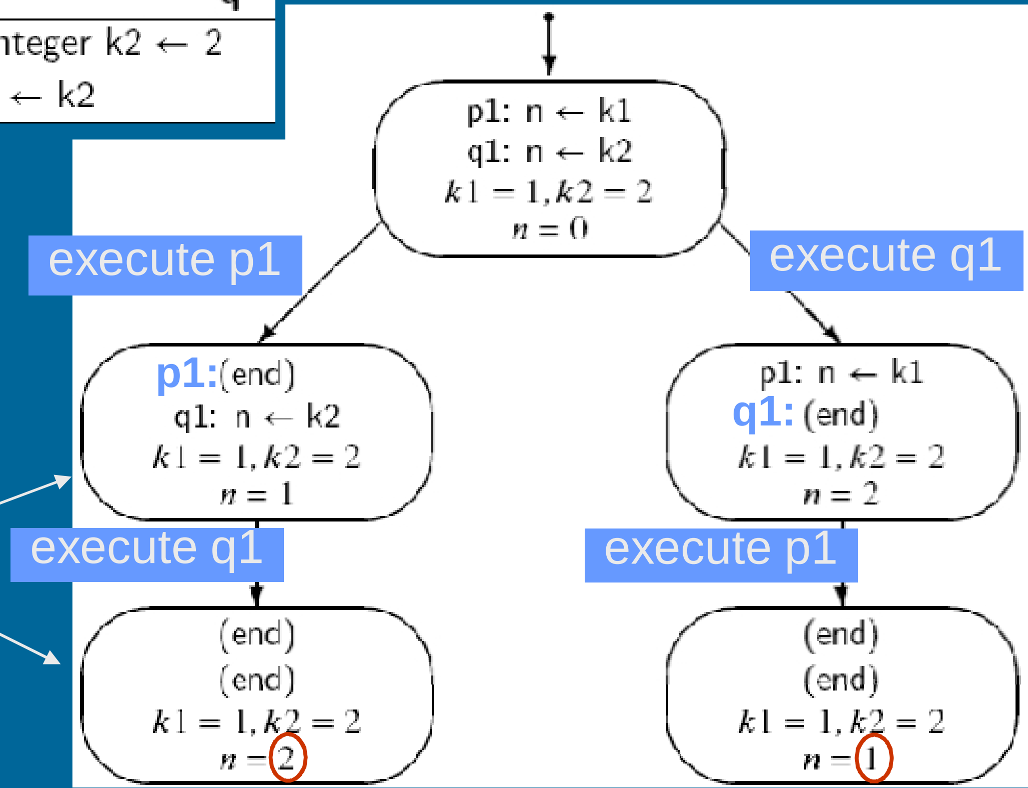


(Global) Program State

- Concurrent program

Algorithm 2.1: Trivial concurrent program	
integer n ← 0	
p	**q**
integer k1 ← 1	integer k2 ← 2
p1: n ← k1	q1: n ← k2

- <u>Local state</u> for each *process*:
  - cp
  - Variable values
    - Local <u>& global</u>
- <u>Global state</u> for *program*
  - *All* cp's
  - *All* local variables
  - *All* global variables



Possible Program States

- List of processes in program
  - List of values for each process
    - cp
    - value of each local/global/shared variable

p1: n ← k1
q1: n ← k2
$k1 = 1, k2 = 2$
$n = 0$

state: {
  { p1: n ← k1 – **process p**
  k1 = 1 }
  { q1: n ← k2 – **process q**
  k2 = 2 }
  n = 0 – **shared variable**
}

- Nr of possible states can be (very) large
  - Not all states are reachable states!

(saavutettavissa, saavutettava tila)

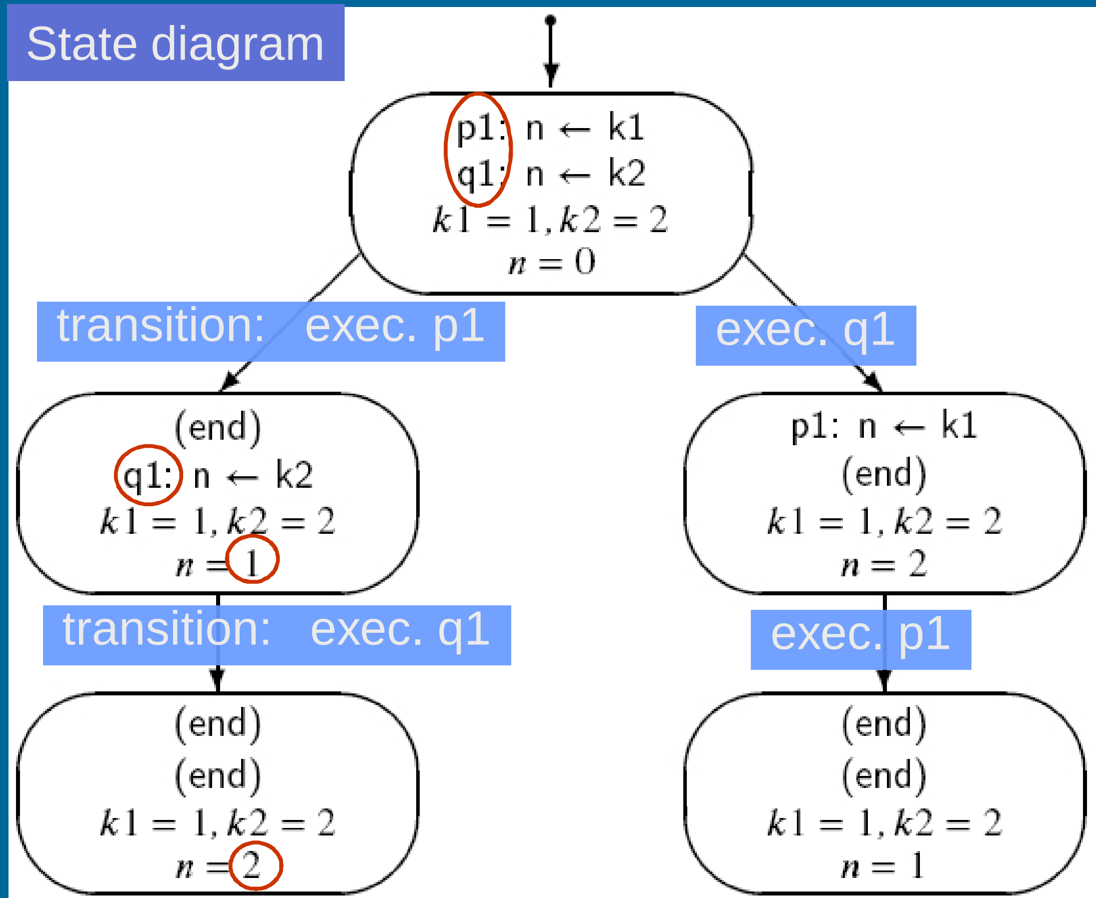
  - Different executions do not go through same states (even with same input)

unreachable state: {
  { p1: n ← k1
  k1 = 2 }
  { q1: n ← k2
  k2 = 1 }
  n = 3
}

State Diagram and Scenarios

Process p	Process q	n	k1	k2
p1: n←k1	q1: n←k2	0	1	2
(end)	**q1: n←k2**	1	1	2
(end)	(end)	2	1	2

Scenario 1 (left side)



- Transitions from one possible state to another
  - Executed statement must be one of those in the 1st state
- State diagram for concurrent program
  - Contains all reachable states and transitions
  - All possible executions are included, they are all correct!

Discuss * 2

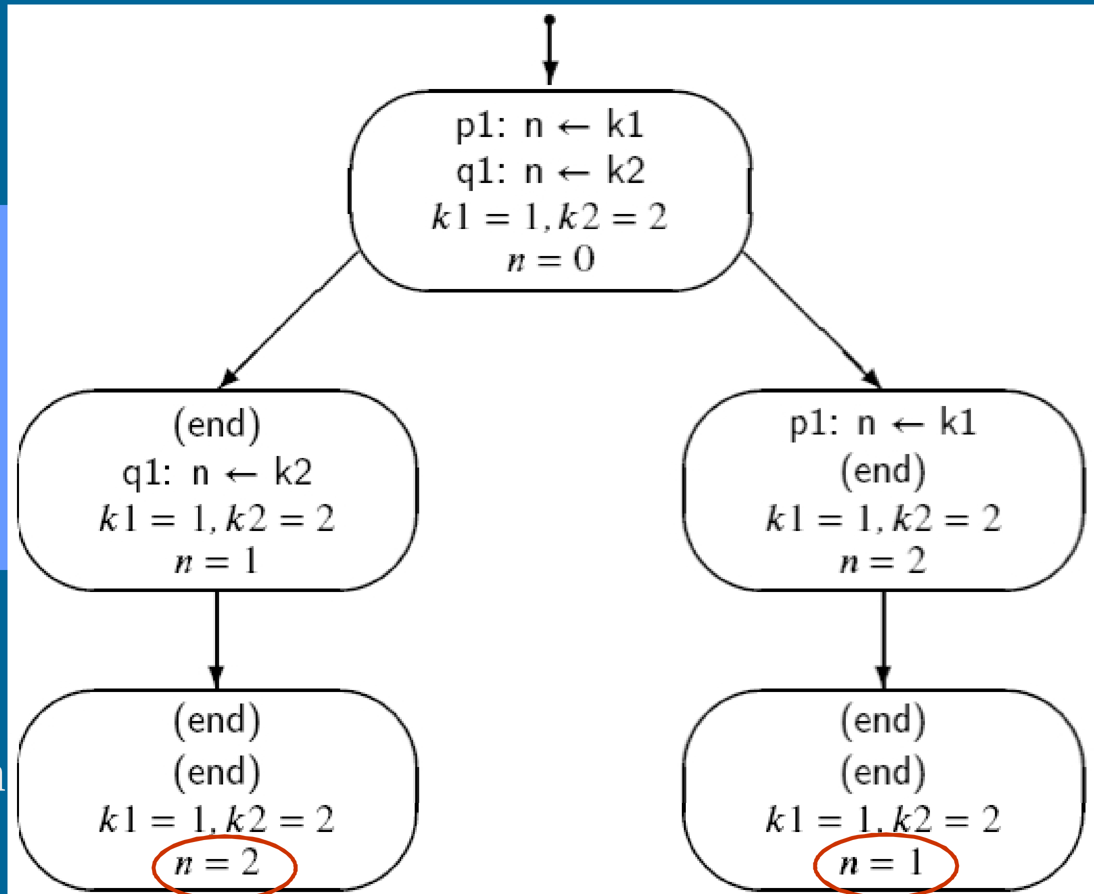
Algorithm 2.1: Trivial concurrent program	
integer n ← 0	
p	q
integer k1 ← 1	integer k2 ← 2
p1: n ← k1	q1: n ← k2

Atomic Statements

- Two scenarios
  - Both correct
  - Different result!

NO need to have the same result! Statements do the same, but overall result may be different. (see p. 19 [BenA 06])

- Atomic?
  - Assignment?
  - Boolean evaluation
  - Increment?



Algorithm 2.3: Atomic assignment statements	
integer n ← 0	
p	q
p1: n ← n + 1	q1: n ← n + 1

- Two scenarios for execution
  - Both correct
  - Both have the same result

P first, and then Q

Process p	Process q	n
p1: n←n+1	q1: n←n+1	0
(end)	**q1: n←n+1**	1
(end)	(end)	2

Q first, and then P

Process p	Process q	n
p1: n←n+1	**q1: n←n+1**	0
p1: n←n+1	(end)	1
(end)	(end)	2

Algorithm 2.3: Atomic assignment statements

integer n ← 0	
p	**q**
p1: $n \leftarrow n + 1$	q1: $n \leftarrow n + 1$

Same statements with smaller atomic granularity:

Algorithm 2.4: Assignment statements with one global reference

integer n ← 0	
p	**q**
integer temp	integer temp
p1: $temp \leftarrow n$	q1: $temp \leftarrow n$
p2: $n \leftarrow temp + 1$	q2: $n \leftarrow temp + 1$

Too Small Atomic Granularity

Algorithm 2.4: Assignment statements with one global reference	
integer n ← 0	
p	q
integer temp	integer temp
p1: temp ← n	q1: temp ← n
p2: n ← temp + 1	q2: n ← temp + 1

- Scenario 1
  - OK
- Scenario 2
  - Bad result
- From now on
  - Assignments and Boolean evaluations are atomic!

Process p	Process q	n	p.temp	q.temp
p1: temp←n	q1: temp←n	0	?	?
p2: n←temp+1	q1: temp←n	0	0	?
(end)	**q1: temp←n**	1	0	?
(end)	**q2: n←temp+1**	1	0	1
(end)	(end)	2	0	1

Process p	Process q	n	p.temp	q.temp
p1: temp←n	q1: temp←n	0	?	?
p2: n←temp+1	**q1: temp←n**	0	0	?
p2: n←temp+1	q2: n←temp+1	0	0	0
(end)	**q2: n←temp+1**	1	0	0
(end)	(end)	1	0	0

Correctness

- What is the correct answer?
- Usually clear for sequential programs
- Can be fuzzy for concurrent programs
  - Many correct answers?
  - What is intended semantics of the program?
  - Run programs 100 times, each time get different answer?
    - Each answer is correct, if program is correct!
    - Does not make debugging easier!
    - Usually can not test all possible scenarios (too many!)
  - How to define correctness for concurrent programs?
    - Safety properties = properties that are always true
    - Liveness properties = properties that eventually become true

"turvallisuus"

"elävyys"

Safety and Liveness

- Safety property

  safety-ominaisuus, turvallisuus

  – property must be true <u>all the time</u> ("bad" never happens)
    - "Identity"

      identiteetti, invariantti

      – memFree + memAllocated = memTotal
    - Mouse cursor is always displayed
    - System responds always to new commands
- Liveness property

  elävyys, liveness-ominaisuus

  – Property must <u>eventually become true</u> ("good" eventually happens)
    - Variable n value = 2
    - System prompt for next command is shown
    - Control will resume to calling program
    - Philosopher will get his turn to eat
    - Eventually the mouse cursor is not displayed
    - Program will terminate
- Duality of safety and liveness properties
  – { P_i will get his turn to eat } ≡ not { P_i will never get his turn to eat }
  – { n value will become 2 } ≡ not { n value is always ≠ 2 }

Linear Temporal Logic (LTL)

(lineaarinen) temporaalilogiikka

- Define safety and liveness properties for certain state in some (arbitrary) scenario
  - Example of Modal Temporal Logic (MDL), logic on concepts like possibility, impossibility, and necessity
- Alternative: Branching Temporal Logic (BTL)
  - Properties true in some or all states starting from the given state
    - More complex, because all future states must be covered
  - Common Temporal Logic (CTL)
    - Can be checked automatically
      - Every time computation reaches given state
    - SMV model checker
    - NuSMV model checker

Fairness

reiluus

- (Weakly) fair scenario
  - Wanted condition eventually occurs
    - Nobody is locked out forever?
    - Will a philosopher ever get his turn to eat?
    - Will an algorithm eventually stop?
    - p and q are both scheduled to run eventually

Algorithm 2.5: Stop the loop A	
integer n ← 0 boolean flag ← false	
p	q
p1: while flag = false	q1: flag ← true
p2: n ← 1 − n	q2:

- All scenarios should be fair
  - One requirement in correct solution

Machine Language Code

- What is atomic and what is not?
  - Assignment?
  - Increment?

X = Y;

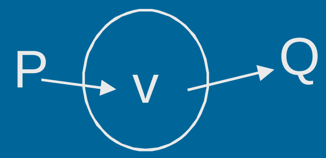
X = X+1;

Algorithm 2.6: Assignment statement for a register machine	
integer n ← 0	
p	**q**
p1: load R1,n	q1: load R1,n
p2: add R1,#1	q2: add R1,#1
p3: store R1,n	q3: store R1,n

Critical Reference

kriittinen viite

- Reference to variable v is critical reference, if …
  - **Assigned value in P and read in Q**
    - Read directly or in a statement



- <u>Program</u> satisfies limited-critical-reference (LCR)
  - Each <u>statement</u> has <u>at most one</u> critical reference
  - Easier to analyze than without this property
  - Each program is easy to transform into similar program with LCR

rajoitettu kriittinen viite

	P	Q	
Not LCR:	<u>n</u> = <u>n</u>+1;	<u>n</u> = <u>n</u>+1	Bad
Not LCR:	<u>n</u> = <u>m</u>+1;	<u>m</u> = <u>n</u>+1	Bad
LCR:	tempP = <u>n</u>+1; <u>n</u> = tempP;	tempQ = <u>n</u>+1; <u>n</u> = tempQ;	Good

LCR vs. atomicity?
(ouch)

Volatile and non-atomic variables

riskialtis

- Volatile variable
  - Can be modified by many processes (<u>must</u> be in shared memory)
  - Advice for compiler (pragma)
    - Keep something in memory, <u>not</u> in register
    - Pseudocode – does not generate code
- Non-atomic variables
  - Multiword data structures: long ints, arrays, records, …
  - Force access to be indivisible (atomic) in given order

What if compiler/hw decides to keep value of n in a register/cache? When is it stored back to memory? What if local1 & local2 were volatile?

Algorithm 2.8: Volatile variables	
integer n ← 0	
p	**q**
integer local1, local2	integer local
p1: n ← *some expression*	q1: local ← n + 6
p2: *computation not using n*	q2:
p3: local1 ← (n + 5) ∗ 7	q3:
p4: local2 ← n + 5	q4:
p5: n ← local1 * local2	q5:

store n?

exec. order?

store n?

which n?

Example Program with Volatile Variables

Algorithm 2.9: Concurrent counting algorithm	
integer n ← 0	
p	q
integer temp	integer temp
p1: do 10 times	q1: do 10 times
p2: temp ← n	q2: temp ← n
p3: n ← temp + 1	q3: n ← temp + 1

- Can implement it in any concurrent programming language
  - (Extended) Pascal and (Extended) C
  - BACI (Ben-Ari Concurrency Interpreter)
    - Code automatically compiled (from Extended Pascal or C)
  - Ada
  - Java

Discuss

(Ben-Ari Concurrent Pascal)

possibly volatile

Concurrent Program in Pascal

```
program count;
var n: integer := 0;

procedure p;
var temp, i : integer;
begin
  for i := 1 to 10 do
    begin
    temp := n;
    n := temp + 1
    end
end;
```

n is volatile, because… it is assigned in one thread, and read in the other

```
procedure q;
var temp, i : integer;
begin
  for i := 1 to 10 do
    begin
    temp := n;
    n := temp + 1
    end
end;

begin { main program }
  cobegin p; q coend;
  writeln(' The value of n is ', n)
end.
```

Concurrent Program in C

(Ben-Ari Concurrent C, C--)

possibly volatile, use carefully

(volatile, if critically referenced)

```
int n = 0;

void p() {
   int temp, i;
   for (i = 0; i < 10; i++) {
      temp = n;
      n = temp + 1;
   }
}

void q() {
   int temp, i;
   for (i = 0; i < 10; i++) {
      temp = n;
      n = temp + 1;
   }
}

void main() {
   cobegin { p(); q(); }
   cout << "The value of n is " << n << "\n";
}
```

What if compiler optimized and kept n in a register?
Lets hope not!
(in ExtPascal or C--
global (volatile) variables are seemingly kept in memory by default)

Concurrent Program in Ada

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Count is
    N: Integer := 0;
    pragma Volatile(N);

    task type Count_Task;
    task body Count_Task is
        Temp: Integer;
    begin
        for I in 1..10 loop
            Temp := N;
            N := Temp + 1;
        end loop;
    end Count_Task;

begin
    declare
        P, Q: Count_Task;
    begin
        null;
    end;
    Put_Line("The value of N is " & Integer'Image(N));
end Count;
```

advice compiler to keep N in memory

Concurrent Program in Java

```
class Count extends Thread {
    static volatile int n = 0;

    public void run() {
        int temp;
        for (int i = 0; i < 10; i++) {
            temp = n;
            n = temp + 1;
        }
    }
```

Thread.yield(); // force?

```
    public static void main(String[] args) {
        Count p = new Count();
        Count q = new Count();
        p.start ();
        q.start ();
```

How many threads really in parallel?

- how to control it?

```
        try {
            p.join ();
            q.join ();
        }
        catch (InterruptedException e) { }
        System.out.println ("The value of n is " + n);
    }
}
```

> javac Adder8.java
> java Adder8

Execute on 8-processor vera.cs.helsinki.fi?

http://www.cs.helsinki.fi/u/kerola/rio/Java/examples/Adder8.java

http://www.cs.helsinki.fi/u/kerola/rio/Java/examples/Adder8b.java

Run Multi-threaded Java

Execute on 8-processor vera.cs.helsinki.fi?

http://www.cs.helsinki.fi/u/kerola/rio/Java/examples/Adder8.java

```
kerola@vera:~/public_html/rio/Java/examples$   javac Adder8.java
kerola@vera:~/public_html/rio/Java/examples$   java Adder8

        finally n = 80000 = 37358

kerola@vera:~/public_html/rio/Java/examples$   java Adder8

        finally n = 80000 = 34464
```

- Why different result?
- What is correct result?

Run them your self?
(Copy source code in your own directory)

BACI

http://inside.mines.edu/~tcamp/baci/baci.html

- Ben-Ari Concurrency Interpreter
  - Write concurrent programs with
    - *C--* or *Ben-Ari Concurrent Pascal* (.cm and .pm suffixes)
    - Compile and run in BACI
  - GUI for Unix/Linux
- jBACI http://stwww.weizmann.ac.il/g-cs/benari/jbaci/
  - Just like BACI
  - GUI for Windows

See BACI instructions
http://www.cs.helsinki.fi/u/kerola/rio/ohjeet/ohjeet.html

- Installation http://stwww.weizmann.ac.il/g-cs/benari/jbaci/jbaci1-4-5.zip
  - load version 1.4.5 jBACI executable files and example programs, unzip, edit config.cfg to have correct paths to bin/bacc.exe and bin/bapas.exe translators, click run.bat
- Use in class, homeworks and in project

BACI Overall Structure

add.cm

C- - (Concurrent C)

```
…
void main() {
  cobegin { add10();
            add10(); }
….
```

bacc.exe

C-- to PCODE Compiler

add.lst

```
…
17   24  void main() {
18   25      cobegin {add10(); add10(); }
….
```

add.pco

(many tables)

```
LOAD_ADDR, push sum
LOAD_VALUE, push local
PUSH_LIT 1
DO_ADD, pop(1), s[t]=(s[oldt-1]+s[oldt])
STORE, s[s[t-1]]=s[t], pop(2)
```

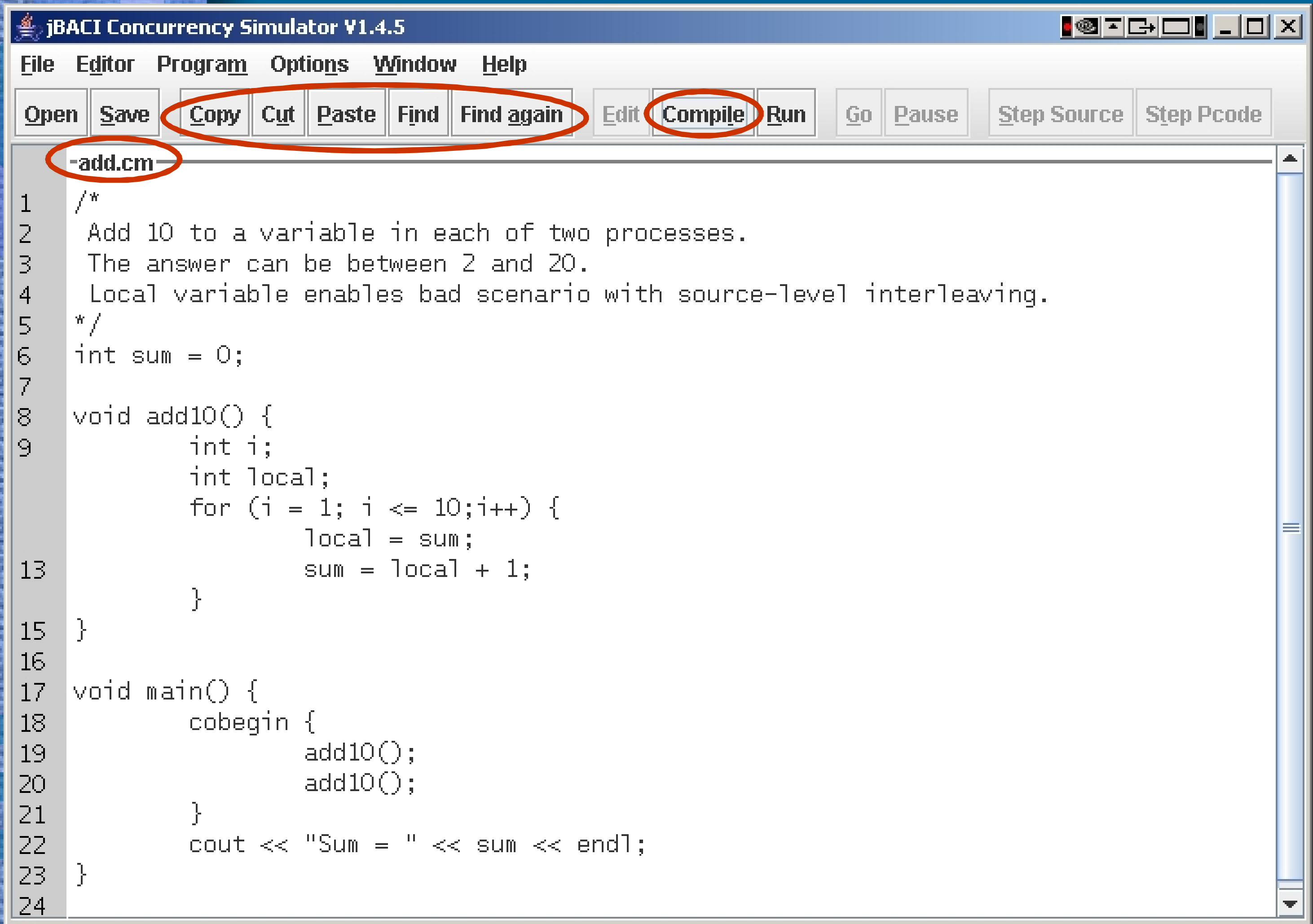
PCODE Interpreter

bainterp.exe

```
Executing PCODE ...
C  n =1  i =A  n =1 C2  i =
1 A
C  n =4  i =2 C
B  n =A  n =5  i =24 A
```

http://www.cs.helsinki.fi/u/kerola/rio/BACI/baci-c.pdf

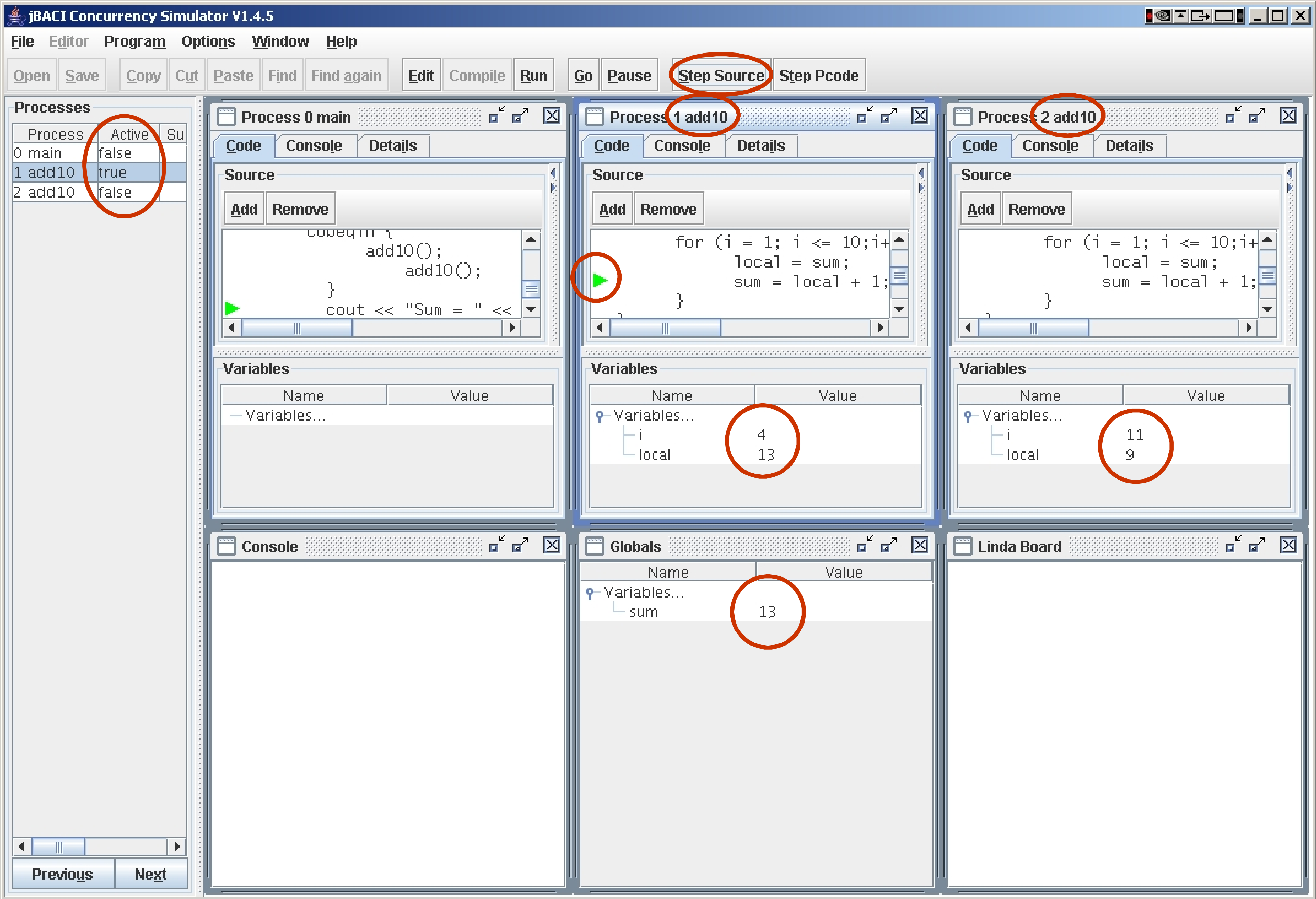
jBACI



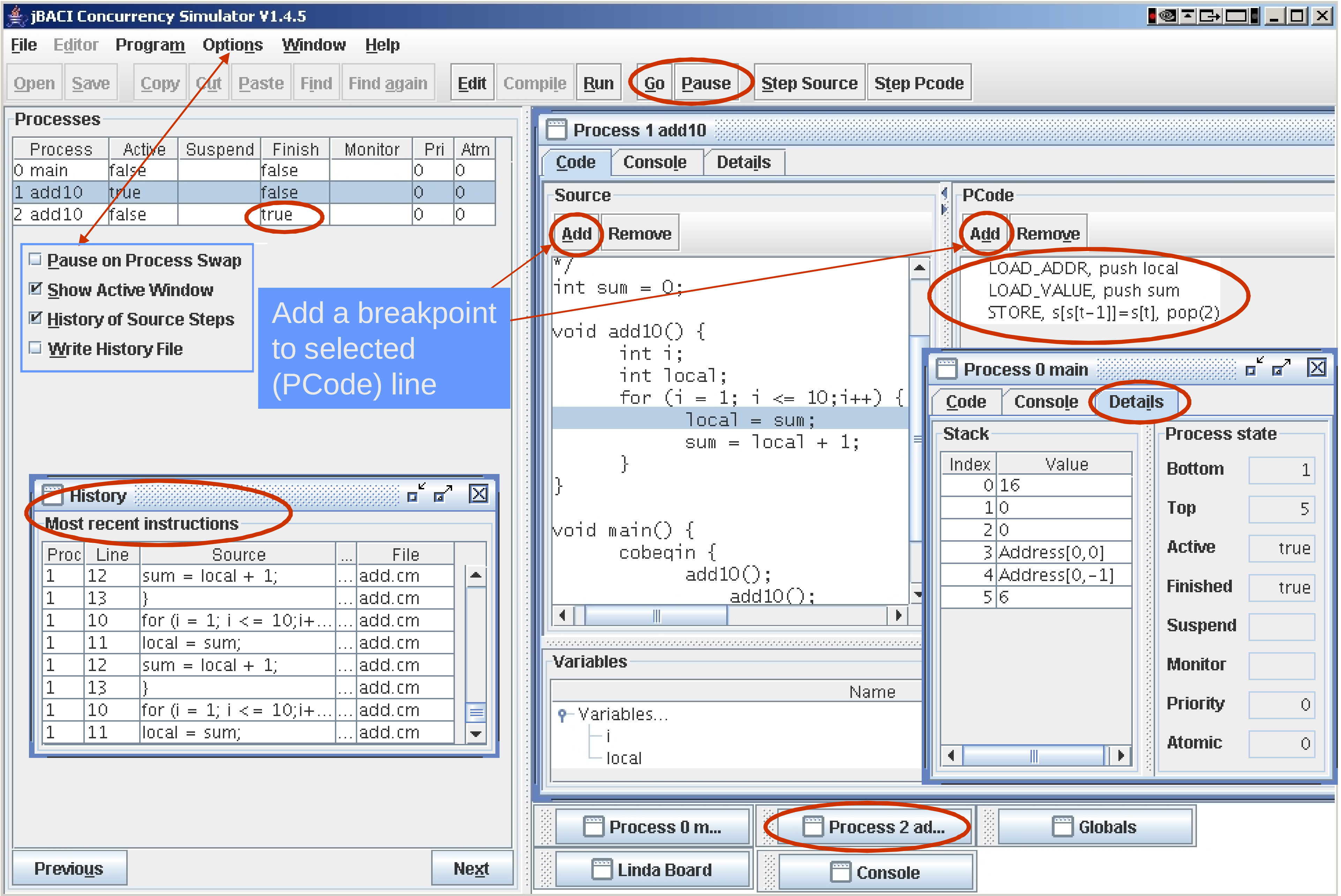
Just like BACI, but with Java
- requires Java v. 1.4 (SDK or JRE)
- Built-in compiler and interpreter
- edit state
- run state

http://www.cs.helsinki.fi/u/kerola/rio/BACI/jbaci.pdf

jBACI IDE (integrated development environment)



jBACI IDE (integrated development environment)



Summary

- Abstraction, atomicity
- Concurrent program, program state
- Pseudo-language algorithms
- High level language algorithms
- BACI