Concurrent Programming

Language Level

Ch 2 [BenA 06]

Abstraction

Pseudo-language

BACI

Ada, Java, etc.

16.10.2009 Copyright Teemu Kerola 2009
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]

16.10.2009
Copyright Teemu Kerola 2009
Concurrent Program

• Sequential process
  – Successive **atomic** statements
    
P: \( p_1 \rightarrow p_2 \rightarrow p_3 \rightarrow p_4 \ldots \)
  – Control pointer (= program counter)

• Concurrent program
  – Finite set of sequential processes **working for** same goal
  – **Arbitrary interleaving** of atomic statements in different processes
Program State, Pseudo-language

- **Sequential program**

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

- Concurrent program

<table>
<thead>
<tr>
<th>Algorithm 2.1: Trivial concurrent program</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer n ← 0</td>
</tr>
<tr>
<td>p: integer k1 ← 1</td>
</tr>
<tr>
<td>p1: n ← k1</td>
</tr>
<tr>
<td>q: integer k2 ← 2</td>
</tr>
<tr>
<td>q1: n ← k2</td>
</tr>
</tbody>
</table>

- Local state for each process:
  - cp
  - Variable values
    - Local & global
- Global state for program:
  - All cp’s
  - All local variables
  - All global variables

execute p1

execute q1
Possible Program States

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

```
state: {  { p1: n ← k1
  k1 = 1 }
  { q1: n ← k2  
  k2 = 2 }
  n = 0         
  }  

– process p

– process q

– shared variable
```

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

(saavutettavissa, saavutettava tila)
State Diagram and Scenarios

- **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!

### Scenario 1 (left side)

<table>
<thead>
<tr>
<th>Process p</th>
<th>Process q</th>
<th>n</th>
<th>k1</th>
<th>k2</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1: n ← k1</td>
<td>q1: n ← k2</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>(end)</td>
<td>q1: n ← k2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>(end)</td>
<td>(end)</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

State diagram:

- **Transition**: exec. p1
- **Transition**: exec. q1
- **Transition**: exec. p1

Copyright Teemu Kerola 2009
**Algorithm 2.1: Trivial concurrent program**

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer n ← 0</td>
<td></td>
</tr>
<tr>
<td>integer k1 ← 1</td>
<td>integer k2 ← 2</td>
</tr>
<tr>
<td>p1: n ← k1</td>
<td>q1: n ← k2</td>
</tr>
</tbody>
</table>

- **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?

16.10.2009

Copyright Teemu Kerola 2009
• Two scenarios for execution
  • Both correct
  • Both have the same result

Algorithm 2.3: Atomic assignment statements

<table>
<thead>
<tr>
<th>integer n ← 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
</tr>
<tr>
<td>p₁: n ← n + 1</td>
</tr>
</tbody>
</table>

Process p | Process q | n
---|---|---
pl: n ← n + 1 | q₁: n ← n + 1 | 0
(end) | q₁: n ← n + 1 | 1
(end) | (end) | 2

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

P first, and then Q

Q first, and then P
Algorithm 2.3: Atomic assignment statements

<table>
<thead>
<tr>
<th></th>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>n ← n + 1</td>
<td>q1: n ← n + 1</td>
</tr>
</tbody>
</table>

Same statements with smaller atomic granularity:

Algorithm 2.4: Assignment statements with one global reference

<table>
<thead>
<tr>
<th></th>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>integer n ← 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>integer temp</td>
<td>integer temp</td>
</tr>
<tr>
<td>p1:</td>
<td>temp ← n</td>
<td>q1: temp ← n</td>
</tr>
<tr>
<td>p2:</td>
<td>n ← temp + 1</td>
<td>q2: n ← temp + 1</td>
</tr>
</tbody>
</table>

16.10.2009  Copyright Teemu Kerola 2009
Too Small Atomic Granularity

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

- **Safety property**
  - property must be true all the time
    - “Identity”
      - \( \text{memFree} + \text{memAllocated} = \text{memTotal} \)
    - Mouse cursor is displayed
    - System responds to new commands

- **Liveness property**
  - Property must eventually become true
    - 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**
  - \( \{ P_i \text{ will get his turn to eat} \} \equiv \text{not} \{ P_i \text{ will never get his turn to eat} \} \)
  - \( \{ \text{n value will become 2} \} \equiv \text{not} \{ \text{n value is always } \neq 2 \} \)
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

- (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?

<table>
<thead>
<tr>
<th>Algorithm 2.5: Stop the loop A</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer n ← 0</td>
</tr>
<tr>
<td>boolean flag ← false</td>
</tr>
<tr>
<td>p</td>
</tr>
<tr>
<td>p1: while flag = false</td>
</tr>
<tr>
<td>p2: n ← 1 − n</td>
</tr>
<tr>
<td>q</td>
</tr>
<tr>
<td>q1: flag ← true</td>
</tr>
<tr>
<td>q2:</td>
</tr>
</tbody>
</table>

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

16.10.2009
Copyright Teemu Kerola 2009
Machine Language Code

• What is atomic and what is not?
  – Assignment?
    \[ X = Y; \]
  – Increment?
    \[ X = X+1; \]

Algorithm 2.6: Assignment statement for a register machine

<table>
<thead>
<tr>
<th>integer n ← 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>p1: load R1,n</td>
</tr>
<tr>
<td>p2: add R1,#1</td>
</tr>
<tr>
<td>p3: store R1,n</td>
</tr>
</tbody>
</table>
Critical Reference

- Reference to variable v is critical reference, if …
  - Assigned value in P and read in Q
    - Read directly or in a statement
- **Program** satisfies limited-critical-reference (LCR)
  - Each statement has at most one critical reference
  - Easier to analyze than without this property
  - Each program is easy to transform into similar program with LCR

\[
\begin{align*}
\text{Not LCR:} & \quad n = n+1; \quad n = n+1 & \text{Bad} \\
\text{Not LCR:} & \quad n = m+1; \quad m = n+1 & \text{Bad} \\
\text{LCR:} & \quad \text{tempP} = n+1; \quad \text{tempQ} = n+1; \quad n = \text{tempP}; \quad n = \text{tempQ}; & \text{Good}
\end{align*}
\]

LCR vs. atomicity? (ouch)
Volatile and non-atomic variables

- **Volatile variable**
  - Can be modified by many processes (must be in shared memory)
  - Advice for compiler (pragma)
    - Keep something in memory, not in register
    - Pseudocode – does not generate code

- **Non-atomic variables**
  - Multiword data structures: long ints, arrays, records, …
  - Force access to be indivisible 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

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>[\text{integer local1, local2}]</td>
<td>[\text{integer local}]</td>
</tr>
<tr>
<td>p1: [n \leftarrow \text{some expression}]</td>
<td>q1: [\text{local} \leftarrow n + 6]</td>
</tr>
<tr>
<td>p2: [\text{computation not using n}]</td>
<td>q2:</td>
</tr>
<tr>
<td>p3: [\text{local1} \leftarrow (n + 5) \times 7]</td>
<td>q3:</td>
</tr>
<tr>
<td>p4: [\text{local2} \leftarrow n + 5]</td>
<td>q4:</td>
</tr>
<tr>
<td>p5: [n \leftarrow \text{local1} \times \text{local2}]</td>
<td>q5:</td>
</tr>
</tbody>
</table>
Example Program with Volatile Variables

Algorithm 2.9: Concurrent counting algorithm

<table>
<thead>
<tr>
<th></th>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>integer n ← 0</td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>integer temp</td>
<td>integer temp</td>
</tr>
<tr>
<td>p1</td>
<td>do 10 times</td>
<td>q1: do 10 times</td>
</tr>
<tr>
<td>p2</td>
<td>temp ← n</td>
<td>q2: temp ← n</td>
</tr>
<tr>
<td>p3</td>
<td>n ← temp + 1</td>
<td>q3: n ← temp + 1</td>
</tr>
</tbody>
</table>

- 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
possibly volatile

possibly volatile

**n** is volatile, because... it is assigned in one thread, and read in the other.
Concurrent Program in C

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

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

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

What if compiler optimized and kept \texttt{n} in a register? Let's 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;
Concurrent Program in Java

```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;
        }
    }
}

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

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

How many threads really in parallel? • how to control it?

Thread.yield(); // force?

> javac Adder8.java
> java Adder8

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

16.10.2009

Copyright Teemu Kerola 2009
BACI

http://www.mines.edu/fs_home/tcamp/baci/

• 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
  – Just like BACI
  – GUI for Windows

• Installation
  – 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

16.10.2009
Copyright Teemu Kerola 2009
BACI Overall Structure

C-- to PCODE Compiler

PCODE Interpreter

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


16.10.2009 Copyright Teemu Kerola 2009
jBACI

Just like BACI, but with Java

- requires Java v. 1.4 (SDK or JRE)
- Built-in compiler and interpreter
- edit state
- run state


16.10.2009 Copyright Teemu Kerola 2009
Add a breakpoint to selected (PCode) line.
Summary

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