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
    • Process switches may occur, but they do not affect result
  – 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]
Java synchronized methods

11.1.2012 Copyright Teemu Kerola 2012
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
Program State, Pseudo-language

- Sequential program

<table>
<thead>
<tr>
<th>Algorithm 2.2: Trivial sequential program</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer n ← 0</td>
</tr>
<tr>
<td>integer k1 ← 1</td>
</tr>
<tr>
<td>integer k2 ← 2</td>
</tr>
<tr>
<td>p1: n ← k1</td>
</tr>
<tr>
<td>p2: n ← k2</td>
</tr>
</tbody>
</table>

- 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</td>
</tr>
<tr>
<td>integer k1 ← 1</td>
</tr>
<tr>
<td>p1: n ← k1</td>
</tr>
<tr>
<td>q</td>
</tr>
<tr>
<td>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
Possible Program States

- List of processes in program
  - List of values for each process
    - cp
    - value of each local/global/shared variable
  - Nr of possible states can be (very) large
    - Not all states are reachable states!
    - Different executions do not go through same states (even with same input)

state: 

\[
\begin{array}{l}
\{ \{ p_1: n \leftarrow k_1 \} \quad \text{– process } p \\
\{ k_1 = 1 \} \\
\{ q_1: n \leftarrow k_2 \} \quad \text{– process } q \\
\{ k_2 = 2 \} \\
\{ n = 0 \} \quad \text{– shared variable} \\
\} \\
\end{array}
\]

unreachable

(saavutettavissa, saavutettava tila)
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!
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?

11.1.2012

Copyright Teemu Kerola 2012
**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>

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

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

<table>
<thead>
<tr>
<th>Process p</th>
<th>Process q</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1: n ← n + 1</td>
<td>q1: n ← n + 1</td>
<td>0</td>
</tr>
<tr>
<td>(end)</td>
<td>(end)</td>
<td>1</td>
</tr>
<tr>
<td>(end)</td>
<td>(end)</td>
<td>2</td>
</tr>
</tbody>
</table>
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>$n \leftarrow n + 1$</td>
<td>$q1: n \leftarrow n + 1$</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 $n \leftarrow 0$</td>
<td></td>
</tr>
<tr>
<td>p1:</td>
<td>$\text{integer temp}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\text{temp} \leftarrow n$</td>
<td>$\text{temp} \leftarrow n$</td>
</tr>
<tr>
<td>p2:</td>
<td>$n \leftarrow \text{temp} + 1$</td>
<td>$q2: n \leftarrow \text{temp} + 1$</td>
</tr>
</tbody>
</table>
Too Small Atomic Granularity

Intent: Increment n by 1

- 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 the intended semantics for 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 ("bad" never happens)
    • “Identity”
      – memFree + memAllocated = memTotal
    • Mouse cursor is always displayed
    • System responds always to new commands

• Liveness property
  – Property must eventually become true ("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 is always not eating \} \equiv \neg \{ P_i will get his turn to eat \}
  – \neg \{ n value is always \neq 2 \} \equiv \{ n value will become 2 \}

How can we prove these?
Linear Temporal Logic (LTL)

• 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?
    • p and q are both scheduled to run eventually

<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>
</tbody>
</table>

• All scenarios should be fair
  – One requirement in correct solution

11.1.2012
Machine Language Code

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

\[ X = Y; \]
\[ X = X+1; \]

<table>
<thead>
<tr>
<th>Algorithm 2.6: Assignment statement for a register machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer n ← 0</td>
</tr>
<tr>
<td>\begin{tabular}{</td>
</tr>
</tbody>
</table>
Critical Reference

- Reference to (shared) variable v is critical reference, if …
  - Assigned value in (process) 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 with LCR than without LCR
  - Each program is easy to transform into similar program with LCR

```
Not LCR:
P: n = n+1;  Q: n = n+1
Bad

Not LCR:
P: n = m+1;  Q: m = n+1
Bad

LCR:
P: tempP = n+1;  n = tempP;
Q: tempQ = n+1;  n = tempQ;
Good
```

LCR vs. atomicity? (ouch)

11.1.2012  Copyright Teemu Kerola 2012
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 (multiword) data always volatile?**
  - 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**

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer local1, local2</td>
<td>integer local</td>
</tr>
<tr>
<td>p1: n ← some expression</td>
<td>q1: local ← n + 6</td>
</tr>
<tr>
<td>p2: computation not using n</td>
<td>q2:</td>
</tr>
<tr>
<td>p3: local1 ← (n + 5) * 7</td>
<td>q3:</td>
</tr>
<tr>
<td>p4: local2 ← n + 5</td>
<td>q4:</td>
</tr>
<tr>
<td>p5: n ← local1 * local2</td>
<td>q5:</td>
</tr>
</tbody>
</table>

store n? exec. order? store n? which n?
Example Program with Volatile Variables

Algorithm 2.9: Concurrent counting algorithm

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

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

```
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";
}
```
Concurrent Program in Ada

```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 l in 1..10 loop
        Temp := N;
        N := Temp + 1;
      end loop;
    end Count_Task;

    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

```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?

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

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

Thread.yield(); // force?

> javac Adder8.java
> java Adder8
Run Multi-threaded Java

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

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 yourself?
(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**
  - 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 path to bin/bacc.exe compiler, click jbaci.jar

- **Use in class and with homework**

11.1.2012

Copyright Teemu Kerola 2012
BACI Overall Structure

C-- to PCODE Compiler

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

Executing PCODE ...

C
n = 1
i = A
n = 1
C 2
i = 1
A
C
n = 4
i = 2
C
B
n = A
n = 5
i = 24
A

C-- (Concurrent C)

bacc.exe
add.lst
(add.cm
add.pco

many tables)

bainterp.exe

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

PCODE Interpreter


11.1.2012
Copyright Teemu Kerola 2012
jBACI

Just like BACI, but with Java

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

11.1.2012
Copyright Teemu Kerola 2012

```java
/*
Add 10 to a variable in each of two processes.
The answer can be between 2 and 20.
Local variable enables bad scenario with source-level interleaving.
*/
int sum = 0;

void add10() {
    int i;
    int local;
    for (i = 1; i <= 10; i++) {
        local = sum;
        sum = local + 1;
    }
}

void main() {
    cobegin {
        add10();
        add10();
    }
    cout << "Sum = " << sum << endl;
```
Add a breakpoint to selected (PCode) line
Summary

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