Concurrent Programming (RIO) 18.1.2011

Lecture 2: Conc Progr Lang Level 1

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

-Concurrent Program

- Sequential process
  - Successive atomic statements

- Concurrent program
  - Finite set of sequential processes working for same goal
  - Arbitrary interleaving of atomic statements in different processes

Monitors
Ch 7 [BenA 06]
Program State, Pseudo-language

- **Sequential program**
  
  Algorithm 2.2: Trivial sequential program
  
  ```
  integer k1 ← 1
  integer k2 ← 2
  p1: n ← k1
  p2: n ← k2
  ```

- **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
  p1: n ← k1
  q1: n ← k2
  ```

- **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

  state: \[
  \{ \{ p_1: n \leftarrow k_1 \},
  \{ q_1: n \leftarrow k_2 \},
  n = 0 \} \]

- 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 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!
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 2.1: Trivial concurrent program

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<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>

---

Algorithm 2.3: Atomic assignment statements

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1: q ← n + 1</td>
<td>q1: 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>
<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>
<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>
<td>(end)</td>
<td>p1: n ← n + 1</td>
<td>(end)</td>
</tr>
<tr>
<td>(end)</td>
<td>(end)</td>
<td>2</td>
<td>(end)</td>
<td>(end)</td>
<td>2</td>
</tr>
</tbody>
</table>
### Algorithm 2.3: Atomic assignment statements

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

- **Scenario 1**  
  - OK

- **Scenario 2**  
  - Bad result

- **From now on**  
  - Assignments and Boolean evaluations are atomic!

---

**Too Small Atomic Granularity**

### Algorithm 2.4: Assignment statements with one global reference

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

<table>
<thead>
<tr>
<th>Process p</th>
<th>Process q</th>
<th>n</th>
<th>p.temp</th>
<th>q.temp</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1: temp ← n</td>
<td>q1: temp ← n</td>
<td>1</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>p2: n ← temp + 1</td>
<td>q2: n ← temp + 1</td>
<td>0</td>
<td>0</td>
<td>?</td>
</tr>
<tr>
<td>(end)</td>
<td>(end)</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process p</th>
<th>Process q</th>
<th>n</th>
<th>p.temp</th>
<th>q.temp</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1: temp ← n</td>
<td>q1: temp ← n</td>
<td>1</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>p2: n ← temp + 1</td>
<td>q2: n ← temp + 1</td>
<td>0</td>
<td>0</td>
<td>?</td>
</tr>
<tr>
<td>(end)</td>
<td>(end)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
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

13

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, will get his turn to eat }  \equiv  not { P, will never get his turn to eat }
- { n value will become 2 }  \equiv  not { n value is always \neq 2 }
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></th>
<th>Algorithm 2.5: Stop the loop A</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>integer n ← 0</td>
</tr>
<tr>
<td></td>
<td>boolean flag ← false</td>
</tr>
<tr>
<td>p</td>
<td>q</td>
</tr>
<tr>
<td>p1: while flag = false</td>
<td>q1: flag ← true</td>
</tr>
<tr>
<td>p2: n ← 1 − n</td>
<td>q2:</td>
</tr>
</tbody>
</table>

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

```plaintext
<table>
<thead>
<tr>
<th></th>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>load R1,n</td>
<td>load R1,n</td>
</tr>
<tr>
<td>p2</td>
<td>add R1,#1</td>
<td>add R1,#1</td>
</tr>
<tr>
<td>p3</td>
<td>store R1,n</td>
<td>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

**LCR vs. atomicity?**

Not LCR:
- Bad
  - \( n = n+1; \)
  - \( n = n+1; \)

Not LCR:
- Bad
  - \( n = m+1; \)
  - \( m = n+1; \)

LCR:
- Good
  - \( tempP = n+1; \)
  - \( tempP = n+1; \)
  - \( tempQ = n+1; \)
  - \( n = tempQ; \)
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 (atomic) in given order

---

### Example Program with Volatile Variables

#### Algorithm 2.8: Volatile variables

<table>
<thead>
<tr>
<th>P</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1: n ← some expression</td>
<td>q1: local ← f + 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>

---

- 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
Concurrent Programming (RIO) 18.1.2011

Lecture 2: Conc at Progr Lang Level 11

possibly volatile

Concurrent Program in Pascal

```
program count;

var n: integer := 0;

procedure p;
begin
  var temp, i: integer;
  begin
    for i := 1 to 10 do
      begin
        temp := 0;
        n := temp + 1;
      end;
  end;

procedure q:
begin
  var temp, i: integer;
  begin
    for i := 1 to 10 do
      begin
        temp := 0;
        n := temp + 1;
      end;
  end;
```

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

18.1.2011 Copyright Teemu Kerola 2011

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 C

```
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 << endl;
}
```

18.1.2011 Copyright Teemu Kerola 2011

possibly volatile, use carefully

(volatile, if critically referenced)

18.1.2011 Copyright Teemu Kerola 2011

Lecture 2: Conc at Progr Lang Level

11
Concurrent Program in Ada

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Count is
  N: Integer := 0;

pragma Volatile(N);  \ advice compiler to keep N in memory

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

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

Concurrent Program in Java

```java
import java.util.concurrent.*;

class Count extends Thread {
  public static void main(String[] args) {
    Count p = new Count();
    Count q = new Count();
    p.start();
    q.start();
    p.join();
    q.join();
  }
  
  try {
    p.join();
    q.join();
  } catch (InterruptedException e) {
  }
  System.out.println('The value of n is " + n);
}
```

> 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 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
  - 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
**BACI Overall Structure**

`bacc.exe` → `PCODE` Compiler

```c
void main(){
  cobegin {add10(); add10();}
```

**EXECUTING PCODE** ...

```c
C n = 1 i = A n = 1 C2 i = 1 A
c
C n = 4 i = 2 C
c
B n = A n = 5 i = 24 A
```


---

**jBACI**

Just like BACI, but with Java

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

jBACI IDE (integrated development environment)

Add a breakpoint to selected (PCode) line

jBACI IDE (integrated development environment)

Add a breakpoint to selected (PCode) line

Lecture 2: Conc at Progr Lang Level
Summary

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