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
    - SW
      - Manually coded inside
      - Provided automatically to the user by programming environment

---

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

3 processes (P, R, Q) interleaved execution

P: $p_1 \rightarrow p_2$
Q: $q_1 \rightarrow q_2$

$p_3, \ldots$ $t_{cp_P}$

$P: p_1 \rightarrow q_1 \rightarrow q_2 \rightarrow p_2, q_1 \rightarrow p_1 \rightarrow q_2 \rightarrow p_2$}

16.10.2009 Copyright Teemu Kerola 2009
Program State, Pseudo-language

- Sequential program
  
  Algorithm 2.2: Trivial sequential program

  integer n ← 0

  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

  integer k1 ← 1
  integer k2 ← 2

  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

  Possible Program States

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

  state:

  \[
  \begin{align*}
  \{ & \{ \text{p1: } n \leftarrow k_1 \} \quad \text{process p} \\
  & \{ q_1: n \leftarrow k_2 \} \quad \text{process q} \\
  & n = 0 \quad \text{shared variable} \\
  \} \\
  \{ & \{ \text{p1: } n \leftarrow k_1 \} \quad \text{process p} \\
  & \{ q_1: n \leftarrow k_2 \} \quad \text{process q} \\
  & n = 0 \quad \text{shared variable} \\
  \}
  \end{align*}
  \]

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

  (saavutettavissa, saavutettava tila)

state:

- Transition 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?
Lecture 2 summary: Concurrency at Programming Language Level

---

**Algorithm 2.3: Atomic assignment statements**

```
integer n ← 0

p1: n ← n + 1
q1: n ← n + 1
```

---

**Same statements with smaller atomic granularity:**

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

```
integer n ← 0

p

integer temp
p1: temp ← n
p2: n ← temp + 1

q

integer temp
q1: temp ← n
q2: n ← temp + 1
```

---

**Too Small Atomic Granularity**

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

```
integer n ← 0

p

integer temp
p1: temp ← n
p2: n ← temp + 1

Process p

q

integer temp
q1: temp ← n
q2: n ← temp + 1

Process q

n | p.temp | q.temp
---|---|---
p1: temp ← n | q1: temp ← n | n | p.temp | q.temp
p2: n ← temp + 1 | q1: temp ← n | 0 | 0 | ?
@end | q1: temp ← n | 0 | 0 | ?
@end | q2: n ← temp + 1 | 1 | 0 | 0
```

- **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”
      - memFree + memAllocated = 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, 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?

  Algorithm 2.5: Stop the loop A
  
  \[
  \begin{align*}
  &\text{integer } n \leftarrow 0 \\
  &\text{boolean } flag \leftarrow false
  \end{align*}
  \]

  \[
  \begin{array}{l|l}
  p & q \\
  \hline
  a_1: \text{while } flag = false & a_1: \text{flag } \leftarrow true \\
  p_2: \quad n \leftarrow 1 \rightarrow n & q_2:
  \end{array}
  \]

- 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

<table>
<thead>
<tr>
<th>( p )</th>
<th>( q )</th>
</tr>
</thead>
<tbody>
<tr>
<td>load ( R1,n )</td>
<td>load ( R1,n )</td>
</tr>
<tr>
<td>add ( R1,#1 )</td>
<td>add ( R1,#1 )</td>
</tr>
<tr>
<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
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>integer local1, local2</td>
<td>integer local</td>
</tr>
<tr>
<td>p1: ( n \leftarrow \text{some expression} )</td>
<td>q1: local ( \leftarrow n + 6 )</td>
</tr>
<tr>
<td>p2: \text{computation not using} ( n )</td>
<td>q2:</td>
</tr>
<tr>
<td>p3: local1 ( \leftarrow (n + 5) \times 7 )</td>
<td>q3:</td>
</tr>
<tr>
<td>p4: local2 ( \leftarrow n + 5 )</td>
<td>q4:</td>
</tr>
<tr>
<td>p5: ( n \leftarrow \text{local1*local2} )</td>
<td>q5:</td>
</tr>
</tbody>
</table>

What if local1 & local2 were volatile?

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 temp</td>
<td>integer temp</td>
</tr>
<tr>
<td>p1: do 10 times</td>
<td>q1: do 10 times</td>
</tr>
<tr>
<td>p2: temp ( \leftarrow n )</td>
<td>q2: temp ( \leftarrow n )</td>
</tr>
<tr>
<td>p3: ( n \leftarrow \text{temp + 1} )</td>
<td>q3: ( n \leftarrow \text{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
Lecture 2 summary: Concurrency at Programming Language Level

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;

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.
```

What if compiler optimized and kept 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 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;
}
```
Lecture 2 summary: Concurrency at Programming Language Level

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;
   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
public 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();

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

    Execute on 8-processor vera.cs.helsinki.fi?
```
Lecture 2 summary: Concurrency at Programming Language Level

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

BACI Overall Structure

C-- to PCODE Compiler

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

Executing PCODE ...

C n =1 i =A n =1 C z i =
1 A
C n =4 i =2 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

Summary

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