Concurrent Programming

K. V. S. Prasad
Dept of Computer Science
Chalmers University
September – October 2012

Website

- http://www.cse.chalmers.se/edu/year/2012/course/TDA382_Concurrent_Programming_2012-2013_LP1/
- Should be reachable from student portal
  - Search on ”concurrent”
  - Go to their course plan
  - From there to our home page

Teaching Team

- K. V. S. Prasad
- Michal Palka
- Ann Lilliesträm
- Staffan Björnesjö

Contact

- Join the Google group
  - https://groups.google.com/forum/?fromgroups#!forum/tda381-concurrent-programming-period-1-2012
- From you to us: mail Google group
  - Or via your course rep (next slide)
- From us to you
  - Via Google group if one person or small group
  - News section of Course web page otherwise

Course representatives

- Need one each for
  - CTH
  - GU
  - Masters (students from abroad)
- Choose during first break
  - Reps then mail Google group
  - Meet at end of weeks 2, 4 and 6
    - Exact dates to be announced
    - Contact your reps for anonymous feedback

Practicalities

- An average of two lectures per week: for schedule, see
  - http://www.cse.chalmers.se/edu/year/2012/course/TDA382_Concurrent_Programming_2012-2013_LP1/info/timetable/
- Pass = >40 points, Grade 4 = >60p, Grade 5 = >80p out of 100
- Written Exam 68 points (4 hours, closed book)
- Four programming assignments – labs – 32 points
  - To be done in pairs
  - Must pass all four to pass course
  - See schedule for submission deadlines
    - (8 points on first deadline, 6 on second, 4 on third)
  - Supervision available at announced times
- Optional exercise classes

Textbook

- M. Ben-Ari, ”Principles of Concurrent and Distributed Programming”, 2nd ed

  Addison-Wesley 2006

Other resources

- Last year's slides
- Ben-Ari's slides with reference to the text
- Language resources – Java, JR, Erlang
- Gregory R. Andrews
  - *Foundations of Multithreaded, Parallel, and Distributed Programming*
    - Recommended reading
- Joe Armstrong
  - *Programming in Erlang*
    - Recommended reading

Course material

- Shared memory from 1965 – 1975 (semaphores, critical sections, monitors)
  - Ada got these right 1980 and 1995
  - And Java got these wrong in the 1990's!
- Message passing from 1978 – 1995
  - Erlang is from the 1990's
- Blackboard style (Linda) 1980's
- Good, stable stuff. What's new?
  - Machine-aided proofs since the 1980's
  - Have become easy-to-do since 2000 or so

Course still in transition!

- Good text book
  - but still no machine-aided proofs in course
- We now use Jave, JR and Erlang
  - Only as implementation languages in the labs
- For discussion
  - pseudo-code as in book
- Graded labs new
  - so bear with us if there are hiccups

To get started:

- What is computation?
  - States and transitions
  - Moore/Mealy/Turing machines
  - Discrete states, transitions depend on current state and input
- What is ”ordinary” computation?
  - Sequential. Why? Historical accident?

Example: the Frogs

- Slides 39 – 42 of Ben-Ari
- Pages 37 – 39 in book

Examples (make your own notes)

1. Natural examples we use (why don't we program like this?)
    1. Largest of multiset by handshake
    2. Largest of multiset by broadcast
    3. Sorting children by height
2. Occurring in nature (wow!)
    1. Repressilator
3. Actual programmed systems (boring)
    1. Shared bank account

Some observations

1. Concurrency is simpler!
   a. Don't need explicit ordering
   b. The real world is not sequential
   c. Trying to make it so is unnatural and hard
      - Try controlling a vehicle!
2. Concurrency is harder!
   1. many paths of computation (bank example)
   2. Cannot debug because non-deterministic

      so proofs needed
3. Time, concurrency, communication are issues

History

- 1950's onwards
  - Read-compute-print records in parallel
  - Needs timing
- 1960's onward
  - slow i/o devices in parallel with fast and expensive CPU
  - Interrupts, synchronisation, shared memory
- Late 1960's : timesharing expensive CPU between users
- Modern laptop: background computation from which the foreground process steals time

How to structure all this? Answers from the 60's

- Each I/O device can be a process
- What about the CPU?
  - Each device at least has a "virtual process" in the CPU
- Context switching
  - move next process data into CPU
  - When? On time signal or "interrupt"
  - How? CPU checks before each instruction
- What does each process need to know?
- What does the system need to know about each process?

Operating Systems (60's thru 70's)

- Divided into kernel and other services
  - which run as processes
- The kernel provides
  - Handles the actual hardware
  - Implements abstractions
    - Processes, with priorities and communication
  - Schedules the processes (using time-slicing or other interrupts)
- A 90's terminology footnote
  - When a single OS process structures itself as several processes, these are called "threads"

Terminology

- A ”process” is a sequential component that may interact or communicate with other processes.
- A (concurrent) ”program” is built out of component processes
- The components can potentially run in parallel, or may be interleaved on a single processor. Multiple processors may allow actual parallelism.

Interleaving

- Each process executes a sequence of atomic commands (usually called ”statements”, though I don’t like that term).
- Each process has its own control pointer, see 2.1 of Ben-Ari
- For 2.2, see what interleavings are impossible

State diagrams

- In slides 2.4 and 2.5, note that the state describes variable values before the current command is executed.
- In 2.6, note that the ”statement” part is a pair, one statement for each of the processes
- Not all thinkable states are reachable from the start state

Scenarios

- A scenario is a sequence of states
  - A path through the state diagram
  - See 2.7 for an example
  - Each row is a state
    - The statement to be executed is in bold

Why arbitrary interleaving?

- Multitasking (2.8 is a picture of a context switch)
  - Context switches are quite expensive
  - Take place on time slice or I/O interrupt
  - Thousands of process instructions between switches
  - But where the cut falls depends on the run
- Runs of concurrent programs
  - Depend on exact timing of external events
  - Non-deterministic! Can't debug the usual way!
  - Does different things each time!

Arbitrary interleaving (contd.)

- Multiprocessors (see 2.9)
  - If no contention between CPU's
    - True parallelism (looks like arbitrary interleaving)
  - Contention resolved arbitrarily
    - Again, arbitrary interleaving is the safest assumption

The counting example

- See algorithm 2.9 on slide 2.24
  - What are the min and max possible values of n?
- How to say it in C-BACI, Ada and Java
  - 2.27 to 2.32

But what is being interleaved?

- Unit of interleaving can be
  - Whole function calls?
  - High level statements?
  - Machine instructions?
- Larger units lead to easier proofs but make other processes wait unnecessarily
- We might want to change the units as we maintain the program
- Hence best to leave things unspecified

Why not rely on speed throughout?

- Don’t get into the train crash scenario
  - use speed and time throughout to design
  - everyday planning is often like this
    - Particularly in older, simpler machines without sensors
    - For people, we also add explicit synchronisation
- For our programs, the input can come from the keyboard or broadband
  - And the broadband gets faster every few months
- So allow arbitrary speeds

Atomic statements

- The thing that happens without interruption
  - Can be implemented as high priority
- Compare algorithms 2.3 and 2.4
    - Slides 2.12 to 2.17
  - 2.3 can guarantee n=2 at the end
  - 2.4 cannot
    - hardware folk say there is a ”race condition”
- We must say what the atomic statements are
  - In the book, assignments and boolean conditions
  - How to implement these as atomic?

What are hardware atomic actions?

- Setting a register
- Testing a register
- Is that enough?
- Think about it (or cheat, and read Chap. 3)

The standard Concurrency model

1. What world are we living in, or choose to?
   a. Synchronous or asynchronous?
   b. Real-time?
   c. Distributed?
2. We choose an abstraction that
   a. Mimics enough of the real world to be useful
   b. Has nice properties (can build useful and good programs)
   c. Can be implemented correctly, preferably easily

Obey the rules you make!

1. For almost all of this course, we assume single processor without real-time (so parallelism is only potential).
2. Real life example where it is dangerous to make time assumptions when the system is designed on explicit synchronisation – the train
3. And at least know the rules! (Therac).

Goals of the course

- covers parallel programming too – but it will not be the focus of this course
- Understanding of a range of programming language constructs for concurrent programming
- Ability to apply these in practice to synchronisation problems in concurrent programming
- Practical knowledge of the programming techniques of modern concurrent programming languages

Theoretical component

- Introduction to the problems common to many computing disciplines:
  - Operating systems
  - Distributed systems
  - Real-time systems
- Appreciation of the problems of concurrent programming
  - Classic synchronisation problems

Semantics

- What do you want the system to do?
- How do you know it does it?
- How do you even say these things?
  - Various kinds of logic
- Build the right system (Validate the spec)
- Build it right (verify that system meets spec)