Concurrent Programming

K. V. S. Prasad
Dept of Computer Science
Chalmers University
August –October 2016

Teaching Team

- K. V. S. Prasad
- Raul Pardo Jimenez
- Ann Lillieström
- Nicholas Smallbone
- John Camilleri (guest for two lectures on Erlang)

Website

- http://www.cse.chalmers.se/edu/course/TDA383
- Should be reachable from student portal
  - Search on ”concurrent”
  - Go to their course plan
  - From there to our home page

Contact

- Use Ping-Pong (go to TDA383/DIT390)
- From you to us: mail discussion forum
  - Or via your course rep (next slide)
- From us to you
  - Via Ping-Pong if one person or small group
  - News section of Course web page otherwise

Course representatives

- Randomly chosen by admin (email addresses on website)
  - Shamsi Abdullayev - MPALG
  - Breunis Blaauwendraad - Utbyte
  - Herman Hörnstein - TKDAT
  - Olle Månsson - TKTFY
  - Tove Svensson - TKDAT
- Usually we get two more from GU
- Plan to meet after Monday lecture, weeks 2, 4, 6

Practicalities

- An average of two lectures per week: for schedule, see
  - http://www.cse.chalmers.se/edu/course/TDA383/time_inf.html
- Rough guidelines (marks out of 100):
  - Pass = >40 points, Grade 4 = >60p, Grade 5 = >80p
  - To pass, must pass all labs and exam separately
- Written Exam 68 points (4 hours, closed book)
- Four programming labs – 32 points
  - To be done in pairs
  - See schedule for submission deadlines and marks
  - Supervision available at announced times

Textbook

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

  Addison-Wesley 2006

Central to your study. Exam based entirely on Chaps 1 through 9 of book.

See Ping-Pong for your classmates’ solutions to problems from the book, and for past exams.

Other resources

- Old slides (both mine and Alejandro Russo's)
- Ben-Ari's slides with reference to the text
- Language resources – Java, Erlang, Promela
- Gregory R. Andrews
  - *Foundations of Multithreaded, Parallel, and Distributed Programming*
    - Older text Recommended reading
- Joe Armstrong
  - *Programming in Erlang*
    - Recommended reading

Programming Languages

- For labs
  - Java (labs 1 and 2), Erlang (labs 3 and 4)
  - Erlang untyped functional language with asynchronous channels
  - Tutorials on Erlang week 3
    - GET STARTED NOW WITH ERLANG TUTORIALS
- For lectures and exam
  - Ben-Ari’s pseudo code
    - Can use Java+Erlang in exam, BUT WITH CARE
  - Spin/Promela as teaching aid (ignore if you wish)
- All but Erlang supported by Ben-Ari’s textbook

Course always in transition!

- We now use Java and Erlang
  - Only as implementation languages in the labs
- Orally graded labs
  - Worked well last term
- Good text book
  - Sadly, still no Promela/Spin officially in course
  - This year, using them informally
- For discussion
  - pseudo-code as in book

Plan for today

- Ideas from other sciences, music and cinema
- Correctness, semantics, dangers, debugging …
- State diagrams
- Example: Unit Record Equipment
- Radical concurrency

Parallelism in nature

- Everywhere!
  - The world is a parallel place
    - Physics, chemistry, biology, economics, medicine, history, football, tennis, ….
      - 10 million agents to simulate spread of infection
      - Simulate patient at various levels
        - » Cannot predict what will happen, but can show what might
  - And in art
    - Music, cinema
- Programming may be the only field where only one thing happens at a time
  - Was never really true (interrupts, etc.)
    - But education still 30 years out of date

Music

- Parallel
  - Time holds everything together (“real time” in CS)
    - What is held together?
      - Threads (themes, motifs)
        - » Can be logical or physical (which instrument, which hand)
        - » https://www.youtube.com/watch?v=A6s49OKp6aE
        - » https://www.youtube.com/watch?v=Qqe0GdUpJHs
    - Things that happen in time are called “events” in CS
    - The themes and motifs are called “processes”
  - Synchronisation is everywhere
  - Harmony and counterpoint are music’s version of “coordination”

Cinema

- Concurrent (potentially parallel)
  - There is only one screen
    - So stories go on (or pause) off screen
  - There are cuts
    - within a scene (punctuation in a story)
    - and intercuts between scenes (“meanwhile”, …)
  - The priest’s voice provides a time-stamp.
    - Without it, the other scenes could be “meanwhile”, but not necessarily at the same instant
  - With the trains, synchronisation is visual or audible (phone)

Death by concurrency

- The presence of death in those film clips was not incidental – it was intended
- Concurrent systems are often embedded (in cars, planes, medical equipment, train signals)
  - Get them wrong and you too can kill
    - Not just in your video games, but for real
- Train crash in NE India (see website)
- Therac radiation therapy machine (see website)

Debugging doesn’t work

- Concurrent systems are non-deterministic
  - Don’t know who speaks first
  - Don’t know who arrives first at a meeting
- So cannot re-run
  - So cannot set break points, backup and find bugs
- Then what do we do?
  - Use model checkers or proof checkers
  - They check spec versus implementation

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)

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

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 (2.35 onwards)
- Pages 37 – 39 in book

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”

Example: Unit Record Equipment

- 1900's - 1950's – 1970's
  - Look up Wikipedia, etc.
- Typical application: payroll
  - One card per employee input (200 cpm)
  - Process info (100 records per min, avg)
  - Print salary info or cheque (300 lpm)
  - loop

```
        read card;
        process info;
        print
```

But this is sequential. CDR waits while processing+printing

How to speed up?

Ex: URE 1

- We said the CDR waits. Do cards wait?
  - Active – passive distinction
    - Where does action come from?
      - Agents in nature. Why we see agents when there aren't any.
      - Animals vs plants+things
    - Are "objects" in CS active? No O-O in this course.
  - CDR, LPR and CPU act. How does the info move?
  - "Communication and Concurrency", Robin Milner.
    - Earlier version also from CTH library, but online.

Ex: URE 2

- CDR puts contents in shared memory
  - How does CPU know contents have arrived?
    - By interrupt, or by timing
      - Interrupt = check between instructions
  - What does CDR do meanwhile?
    - whole card is read and transferred as one?
    - If column by column, re-visit questions.

Ex: URE 3

- CDR / CPU / LPR

```
CDR                 CPU              LPR
 loop                loop             loop
   c := card           p := f(c)        paper := p
```

- This is how we show parallel processes
  - But we need coordination/synchronisation/timing
  - CDR needs another c to read the next card into?
    - Is this an internal matter for CDR, and c is all we look at?
- CDR – c – CPU – p – LPR
  - So CPU can miss a card or re-read the same one.

Ex: URE 4

- ->
  CDR c CPU
  <-
- We try to work with signals (taps on shoulder)
  - Assume that reading a card and processing it take much longer than assigning to and from c, and sending and receiving signals

Ex: URE 5

- CDR / CPU / LPR

```
CDR              CPU              LPR
 loop              loop              loop
   CR?               CF?               LF?
                     LR?
   c := card         p := f(c)         paper := p
   CF!               LF!               LR!
                     CR!
```

- Assuming signals are quick, and access to c and p are unguarded (why the post office sends you a small note to say a big parcel has arrived).
- All waiting to start with – deadlock. Kick start.

Why not use time/speed throughout?

- Remember train crash (mix speed/messages)
  - 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

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

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

Concurrent? Parallel?

- Examples:
  - Max
    - Using handshake, broadcast
  - Sort
    - Using broadcast
  - Eight queens
- Crossing a door, sharing a printer

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
      a. 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

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

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)