Principles of Concurrent Programming TDA384/DIT391

K. V. S. Prasad
Dept of Computer Science
Chalmers University
August –October 2017

Teaching Team

- K. V. S. Prasad (main lecturer, course-in-charge)
- Carlo Furia (guest lectures on multi-core programming)
- Course Assistants
  - Raul Pardo Jimenez (also, guest lectures on Promela and Java)
  - John Camilleri (also, guest lectures on Erlang)
  - Alexander Sjösten
  - Asefa Abel

Website

- http://www.cse.chalmers.se/edu/course/TDA384_LP1/
- Should be reachable from student portals
  - Search on ”concurrent”
  - Go to their course plan
  - From there to our home page
- Or just search for TDA384 or DIT391, possibly with ”concurrent”added

Communication channels

- Best of all, face-to-face: in lectures or supervised lab sessions
- At other times:
  - From you to us:
    - Use Ping-Pong discussion forum (go to TDA384/DIT391)
    - 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)
  - Will announce names and email addresses when admin tells us
- Usually we get a few from CTH and a few from GU
- Preliminary 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/TDA384_LP1/lectures/
- 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 70 points (4 hours, two open books)
- Three programming labs – 30 points
  - To be done in pairs
  - See http://www.cse.chalmers.se/edu/year/2017/course/TDA384_LP1/labs/ for submission deadlines and marks
  - Supervision available at announced times

Textbooks

- **M. Ben-Ari, ”Principles of Concurrent and Distributed Programming”, 2nd ed., Addison-Wesley 2006**
  - Central to your study. Chaps 1 through 9 of book.
- **M. Herlihy & N. Shavit: *The Art of Multiprocessor Programming*, Morgan Kaufmann** (available online through Chalmers library)
  - Parallelizing computations: Herlihy & Shavit 16.1, 16.4
  - Parallel linked lists: Herlihy & Shavit 9
  - Lock free programming: Herlihy & Shavit 10.1, 10.2, 10.5, 10.6, 18.1, 18.2

Other resources

- Old slides (both mine and Carlo Furio’s)
- Ben-Ari’s slides with reference to the text
- Language resources – Java, Erlang, Promela
  - E.g., Joe Armstrong, *Programming in Erlang*
- Recommended reading
  - http://www.cse.chalmers.se/edu/course/TDA384_LP1/reading/
  - http://www.cse.chalmers.se/edu/year/2016/course/TDA383_LP1/lit_inf.html

Programming Languages

- For labs
  - Java (labs 1 and 3), Erlang (lab 2)
- Erlang untyped functional language with asynchronous channels
  - Tutorials on Erlang week 3
    - GET STARTED NOW WITH ERLANG TUTORIALS
- For lectures
  - Ben-Ari's pseudo code
  - Java/Erlang, or pseudo-code based on them
  - Spin/Promela as teaching aid (ignore if you wish)
- All but Erlang supported by Ben-Ari's textbook
- Exam will not use Promela/SPIN

Formal Entry Requirements

- **GU**
  - 7.5 hec in imperative/object-oriented programming such as DIT012, DIT948 or equivalent,
  - an additional course in programming or data structures.
  - Moreover, the student must also have knowledge in propositional logic, which is acquired by successfully completing courses such as DIT980, DIT725, the part on introductory algebra from MMGD200, or equivalent.
- **CTH**
  - Solid background in programming, including object oriented languages (for example Java), and basic knowledge of (propositional) logic. Some knowledge of functional programming (for example Haskell) is a plus.

Informal Prerequisite warning

For some students the formal prerequisites suffice, but many need more:

1. Comfort with sizable Java code (for the labs). See lab 3.
2. Some students find Functional programming ("FP") hard to pick up. Are you one of these? Find out by doing the Erlang tutorial. If you find that hard, consider taking this course after doing some FP.
3. The course is about Concurrent Programming in general, not just as it appears in Java and Erlang. Hence the pseudo-code. You must be comfortable adapting to new notation.
4. Concurrent programs cannot be debugged in the usual way, so we need to reason about them. Glance quickly through Ben-Ari and make sure you are happy with the content.

Multi-agent systems around us

- Biological (sub)systems
  - the circulatory system, say (at a suitable level of abstraction)
- Ecological systems
- Parts of an economy
- Various views of an industrial plant
- …

- We often wish to simulate such systems to understand how they work, and to diagnose systems gone wrong. So:
  - How should we represent an agent?

Processes (or agents, or threads)

- Abstraction from an active, autonomous entity
  - People (cooperating, coordinating or competing on a job)
  - Machines (ditto)
    - E.g., I/O devices and CPU
- Each person or machine works at their own speed
  - Some slow, some fast, some even changing speeds as they go
  - Some might be taking breaks (ready=runnable, but not actually running),
  - or waiting=blocked=sleeping (for input, say, or for a message)
- The first abstraction is from actual speed
  - So not real time
  - Explicit synchronisation, not clock based
  - Follow your own rules – example of train crash

What should processes see of each other?

- People working together can
  - Speak (or phone or email), or leave messages on a bulleting board
  - They can also look over and see whether the others are waiting, or taking a break, or actually working, and how fast
- But this is too rich for a programming abstraction
  - We limit processes to only communicate in limited ways, typically either
    - Shared memory (like bulletin boards, or memory cells)
    - Messages (like email or phone or radio broadcast)
      - We begin with shared memory, for a few weeks
  - Also, from the outside
    - We say we don't know if a process is blocked or running, and if the latter, how fast
    - So we don't care about actual real time, but only sequences of actions

But just shared memory is problematic

- Consider CDR -> R -> CPU -> P -> LPT, where
  - CDR = card reader, LPT = line printer, and R and P are shared memory cells
  - The arrows -> show the way the data flows
  - Believe it or not, this is a reasonable representation of a 1950's URE system
    - URE = Unit Record Equipment
- Suppose CDR is supposed to input records 1, 2, 3, …
  - which the CPU passes on to LPT in the form 1*1, 2*2, 3*3, … (squares)
  - But if the CPU is too fast or the CDR is too slow or stuck, then the CPU might pick up 1, 2, 2, 3, ... And so on. (Repeating records).
  - If the CPU is too slow, it might pick up 1, 3, … (Skipping records)

So need (synchronisation) signals or events

- So that the CDR can tell the CPU "read the next record now", and so that the CPU can tell the CDR "write the next record now".
- We can imagine adding a flag with values "full" and "empty".
  - CDR waits till flag=empty, then writes the new record into R, and sets the flag:=full.
  - CPU waits till flag=full, then reads R, and sets the flag:=empty.
- What does wait mean?
  - Check the flag every cycle till the flag has the value you want.
  - This is called a busy wait. This is not wasting cycles, since the machine has nothing else to do.
- How do you show the solution works?

Psuedo-code for our flag attempt

flag := empty

- CDR code
- loop
  d1: loop until flag=empty;
  R := new record;
  d2: flag := full

- CPU code (partial)
- loop
  p1: loop until flag=full;
  temp:= R;
  p2: flag := empty

d1, d2, p1, p2 are command labels

Simply “loop” means “loop forever”

Can we avoid busy waits?

- Yes, we can sleep until the flag=full
  - (or whatever condition we are waiting for)
  - Provided we are woken up when the condition holds
- Who will wake us up?
- The operating system (“OS”) or the language run-time support (“RTS”)
- Thus we have Ben-Ari’s slide 6.1
  - This has been standard software view for fifty years
- What does the CPU do when a process is sleeping?
  - Busy-waiting for the condition defeats the software sleep
  - It can run another process
  - There can be any number of CPU’s and any number of processes! Scheduling!

An example of interference in shared memory

- Suppose spouses share a bank account and ATMs dispense one unit of cash using the following procedure
- ***proctype W(loc)***
  {int temp;
  temp := bal;
  temp--;
  out(1);
  bal:=temp}
  - ***bal*** is shared global balance between spouses
  - ***temp*** is local register in ATM
  - ***out*** is payout – users always withdraw 1 unit of money

Shared bank account – simultaneous withdrawals from different locations

- Then ***{run W(A); run W(B)}***
  could result in locations A and B both succeeding in withdrawals, but with the account being debited just once, as in the following scenario
- ***W(A): temp := bal; W(B): temp := bal;***
  W(A): temp-- ; W(B): temp-- ;
  W(A): out(1); W(B): out(1);
  W(A): bal:=temp; W(B): bal:=temp
- Commands on the same line are executed in parallel (or, in either order), but before commands on lines below.

Mutual exclusion ("mutex")

- Each ATM needs mutually exclusive access to *bal.*
- If we try a flag solution (busy/free), then each ATM could find the flag free, and go ahead.

Psuedo-code for flag attempt for the bank

flag := free

- ATM code
- loop
  a1: loop until flag=free;
  a2: flag := busy;
  a3: temp := bal; temp--; bal:=temp;
  a4: flag := free

- ATM code
- loop
  b1: loop until flag=free;
  b2: flag := busy;
  b3: temp := bal; temp--; bal:=temp;
  b4: flag := free

Consider the scenarion a1, b1, a2, b2, … We get the same interference problem.

The solution is that a1 and a2 must happen in one step.
Atomic action to prevent unwanted interleaving

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 Alg 2.1 of Ben-Ari
- For Alg 2.2, see what interleavings are impossible
- See slides 2.3 – 2.7 of Ben-Ari

Scenarios

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

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

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?

FP begins with evaluation

- The first computations we all learn: 5+3=8
  - Neither 5 nor 3 "became" 8. Nothing changed!
  - The = is often better replaced by ->.
    - The = is true enough. It says that ->, evaluation, does not change the value.
  - Then why do a -> at all?
    - We go towards whatever is regarded as simpler.
    - The simplest is the canonical value, often a name, such as 8.

Factorial

```
fac 0 = 1
fac n = n * fac (n-1)  -- use if parm <> 0
```

In the context of this program of two definitions,

an expression is evaluated as follows: for a non-canonical term, find a matching pattern, and replace lhs by rhs

```
fac 3 = 3 * fac 2
      = 3 * (2 * fac 1)
      = 3 * (2 * (1 * fac 0))
      = 3 * (2 * (1 * 1))
      = 3 * (2 * 1)
      = 3 * 2
      = 6
```

More on FP

- So the hunt for matching patterns is the new control flow
- The replace is the new basic command, as assignment is for imperative programming (IP)
- We can use if-then-else or case expressions to branch
- We don't need loops, because the recursion does that job
- FP and IP can each do what the other does
- Erlang is IP as far as the I/O goes (state changes), and the FP part of it is incidental to this course – but needed when you program in Erlang!
- The O-O part of Java is incidental to CP (concurrent programming), if not inimical to it – but you need to at least follow the syntax when you program in Java!