Introduction to Message Passing
Fri 8 Sep 2017

K. V. S. Prasad
Dept of Computer Science
Chalmers University
Lecture 5 of TDA384/DIT301, August –October 2017

Plan for today

- Shared memory: recap
- Chap 8: Message passing
  - (the book does not do many examples; we'll try to do more)
- Skipped for now
  - the rest of
    - Chap 3 (Critical Section)
    - Chap 6 (Semaphores)
    - Chap 7 (Monitors)
  - And all of
    - Chap 4 (Proofs)
    - Chap 5 (Further algorithms for CS)
- REMINDER: exercises in Chaps. 1, 2, 3, 6, 7
  - Try them in Promela. Use various assertions.

Why concurrency at all?

- Speed (parallelism)
- Modelling real life agents/actors/processes
- Historically
  - I/O devices running in parallel with CPU
  - Multiprogramming, programs sharing a CPU
  - Time sharing
    - Between people, back when they shared a CPU

Communication and Concurrency

- Shared memory is a means of communication
- Concurrent processes that don't communicate
  - Are simply leading independent lives
  - Nothing much to say about them
    - No deadlocks or mutex issues
    - No benefits either from concurrency
- Are there other means of communication?
  - Of course! Look at us!

Historical Transition

- Why did we need other models?
  - Computers started talking to each other – late 60's
    - Not just to I/O devices
- Hoare 1978
  - arrived before distributed systems
  - I see it as the first realisation that
    - Atomic actions, critical regions, semaphores, monitors…
    - Can be replaced by just I/O as primitives!
- Advent of distributed systems
  - Mostly by packages such as MPI
    - Message passing interface

Models of Communication

- Speech = broadcast
  - Synchronous communication
  - Asynchronous actions (not clocked)
  - Speaker autonomous
- Post or email = asynchronous channel (buffer)
  - Both communication and action asynchronous
  - Speaker autonomous
- Telephone = synchronous channel = 0 size buffer
  - Synchronous communication and actions
  - Only internal actions autonomous

Addressing

- Broadcast
  - Sender and/or receiver anonymous
    - Can be named (maybe) in message
- Post, email, telephone
  - Receiver named (envelope, header, number)
    - Sender need not be (but can)
- What is addressed?
  - Processes? Channels?

What do processes communicate or share?

- Data
  - Tell me what you've heard
- Resources
  - Databases – don't want inconsistent DB
  - printer – don't want interleaved printouts
- Timing signals
  - Pure timing signals: empty envelopes, beeps, etc.
- So expect (equivalents of) semaphores, etc.
- Channels can be shared between processes
  - In some languages
  - But in Erlang, e.g., only one proc can input from it

Semaphore by synchronous channels

```
Each user:
        loop
           chwait => token
           crit sec
          chsignal <= token
Semaphore:
        loop
           chwait  <= token
           chsignal => token
```

1: Information flows along the arrows => and <=, so that <= means output value, and => means input to variable.

2. Only one of contending users gets the token from chwait, and the semaphore then waits till this user returns the token.

3. The token is just a dummy (uint type, empty envelope)

Notational quarrel with Ben-Ari

- The => and <= have a clear logic about which way the information flows, but
  - Output can be written 5 => chan or chan <= 5, which makes it hard to keep track.
  - The notation => also means “implies” in logic, so clashed often in discussions
  - The notation chan!5 and chan?x are to my mind both clearer, and traditional. Output always has a ! and input a ? after the channel name.
  - The ! and ? Notation also goes well with a functional notation for processes.

CS using synchronous semaphore

P:
loop
p1: chwait => token
p2: crit sec
p3: chsignal <= token

Q:
loop
q1: chwait => token
q2: crit sec
q3: chsignal <= token

Mutex: p2 implies P has successfully done p1; P has the token. Then Semaphore permits only chsignal (return token), so Q cannot get the token.

Deadlock-free: If Semaphore is busy (the token is out), either P@p2 or P@p3 or Q@q2or Q@q3 (either P or Q has the token). So if P@p1 and Q@q1, then Semaphore has the token. It will accept chwait, from either P or Q.

Starvation: Possible, if P wins every time. A ***fair*** semaphore will ensure that when Q repeatedly asks, it will be granted at some point.

Detour: Pure signals

- A pure (synchronisation) signal is
  - A dummy variable with only one value
  - Or empty envelope in the post
  - Or missed call on the phone
    - How to communicate for free using these calls
- Along a channel or broadcast or stored in a shared variable
- Can be used as a timing signal saying agreed event has happened.

Broadcast channel is a semaphore!

```
Each user (i):
        loop
          either                                    -
                ch <= i
                crit sec
                ch <= done
          or
                ch => j
                ch => done
```

1: Did I succeed in speaking (tjing)? If so, I enter my CS. The others can't enter theirs.

2. Those who did not get to make their request wait till they hear another message.

3. Here the channel is used only for this semaphore. If it is used for other things too, the losing process should test what it hears till it hears done.

Examples from the book

- Producer-consumer
  - Doesn't matter whether synch/asynch
- Matrix-multiplication
  - Here, could be synchronous action : gangstepped
- Dining philosophers
  - With synchronous channels only.
  - Each fork behaves like a semaphore
  - Both deadlock and starvation seem possible!

Examples from the book

- Producer-consumer
  - Doesn't matter whether synch/asynch
- Matrix-multiplication
  - Here, could be synchronous action : gangstepped
- Dining philosophers
  - With synchronous channels only.
  - Each fork behaves like a semaphore
  - Both deadlock and starvation seem possible!

The matrix example

```
--        --      --        --
| 1 2 3 |     | 1 0 2 |
| 4 5 6 |  X  | 0 1 2 |
| 7 8 9 |     | 1 0 0 |
--        --
```

Have to do a series of dot products like

```
[7, 8, 9] X  | 2 |  =  7*2  + 8*2  +  9*0  = 14+16+0 = 30
             | 2 |
             | 0 |
```

Rendezvous

- Like synchronous channel, except
  - Addressing asymmetric
    - Sender knows receiver's address (entry), not v-v.
  - The communication may involve computation and return of value by the receiver
  - So made for client-server

Ada

- Uses protected objects
  - Since the 1980's
    - though the concept was around earlier
  - Thus has the cleanest shared memory model
- Also has a very good communication model
  - Rendezvous
- Ada was decided carefully through the 1970s
  - Open debates and process of definition
- Has fallen away because of popularity of C, etc.
  - Use now seen as a proprietary secret!

Robin Milner (1934-2010)

- Turing Award 1992 for CCS, ML and LCF!
- Went on to develop pi-calculus
  - Functions as processes
- Bigraphs
- CCS uses synchronous channels to make a complete calculus (programming and reasoning)