Lecture 6: Message Passing

K. V. S. Prasad
Dept of Computer Science
Chalmers University
17 Sep 2011

Questions?

- Anything you want to say
  - Comments, questions, stray thoughts, etc.
  - Are we too fast/slow?
- Practical problems?
  - Found a lab partner? Ask tutors for help if needed

Rough lecture plan

Monday	17 Sep 1315	Message Passing	Chap 8
Thursday	20 Sep 1000	Linda	Chap 9
Monday	24 Sep 1315	Guest: JR	JR resources on web page
Monday	24 Sep 1515	Functional progr. + Chaps 8 and 9	
Monday	1 Oct 1315	Guest: Erlang	Erlang resources on web page
Monday	1 Oct 1515	Reasoning	Chaps 3 and 4
Monday	8 Oct 1315	Reasoning	Chaps 3 and 4
Monday	15 Oct 1315	Guest: SPIN	parts of book on Primula
Monday	15 Oct 1515	revision	
Thursday	18 Oct 1000	revision	

Plan for today

- Shared memory: recap
- Chap 8: Message passing

Chap 3 & 4 (skipped for now)

REMINDER: exercises in Chaps. 1, 2, 3, 6, 7

Shared memory problems

- Critical section (atomic actions)
  - Mutex needed
  - Avoid deadlock, livelock, starvation and busy waiting
- Other examples
  - Producer – consumer
  - Dining philosophers
  - Readers and Writers

Shared memory solutions

- Test-and-set (hardware) with busy wait
- Semaphores
  - Correctness of processes interdependent
  - Not modular
- Monitors: mutex ops, and modular, but
  - Need condition queues
    - With explicit waitC and signalC operations
  - Need immediate resumption or other discipline
- Protected objects
  - Barrier entries solve monitor problems
  - But can cause starvation with unfair scheduler

Monitors centralise

- Access to the data
  - Natural generalisation of objects in OO, but
    - With mutex
    - With synchronisation conditions
- Could dump everything in the kernel
  - But this centralises way too much
    - So monitors are a compromise

Protected objects

- Tidy up the mess
  - No separate condition variables
    - Or queues for them
    - Or detailed choices ”immediate release”, etc.
- The simplicity of 7.6 is worth gold!
  - Price: starvation possible
  - Can be fixed, at small price in mess (see exercises)

Correctness of shared memory programs

- By state diagram (p 112, s 6.4)
  - Mutex, because we don't have a state (p2, q2, ..)
  - No deadlock
    - Both blocked, no hope of release
  - No starvation scenario with fair scheduler
- By invariants or other reasoning on code
  - E.g., A wait will be executed
    - A blocked process will be released

Transition

- Why do we need other models?
- Advent of distributed systems
  - Mostly by packages such as MPI
    - Message passing interface
- But 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!

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
```

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

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

Semaphore by asynchronous channels

Each user (i):

```
loop
  chrequest <= i
   chwait(i) => token
   crit sec
  chsignal <= token
```

Semaphore:

```
local integer i
loop
   chrequest => i
    chwait(i) <= token

    chsignal  => token
```

Explanation: Channels request and signal are received by the semaphore .

Simplest to have only receiver, with asynchronous channels.

One user gets its request accepted; other requests stay in the buffer.

The accepted request says on which channel the token is to be sent.

If all the users share a wait cannel, the token can be stolen.

Broadcast channel is a semaphore!

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

Explanation: Did I succeed in speaking (tjing)? If not, I wait till I hear another thing.

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!

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)