Lecture 8: Introduction to Message Passing (m.p.)

K. V. S. Prasad

TDA384/DIT391 Principles of Concurrent Programming
Chalmers Univ. and Univ. of Gothenburg

Monday 10 Feb 20

Shared memory recap

Examples:

- Critical section (atomic actions)
  - Mutex needed
  - Avoid deadlock, livelock, starvation and deadlock/busy-waiting
- Producer consumer
  - With semaphores, each depends on the other for correctness
- Dining philosophers
  - Show deadlock/livelock problems with symmetric waiting
- Readers and Writers (not yet done in class)

Other examples Dining philosophers Readers and Writers

Shared memory solutions

- Test-and-set (hardware) with busy wait
- Semaphores
  - Correctness of processes can be interdependent
  - Clear modularity must remain implicit
- Monitors (saw very briefly)
  - Need condition queues with explicit waitC and signalC operations
  - mutex ops, and modular, but suppose producer waits because buffer is full, and consumer takes an item. Should
    - ★ producer immediately resume, interrupting consumer's finishing activity? (Hoare semantics, messy implementation)
    - ★ or should consumer finish up, with producer then coming in? (Mesa semantics). If there are multiple producers, the woken up process may find buffer full again. Need to check again on waking up.
- Protected objects
  - Guarded entries, so no condition variables, and no Hoare/Mesa problem
  - Run-time must check every guard after every call. Needs fair scheduler.

Verifying shared memory programs

- By state diagram — look for counterexamples
  - bad state
    - ★ breaks safety property - nothing bad ever happens
  - bad loop with no progress
    - ★ breaks liveness property - something good eventually happens
- By invariants or other reasoning on code

Why another model of Concurrency?

Shared memory

- seems to work
- covers many applications
  - including multi-core, to be studied later
- *Dijkstra, Hoare, Lampson* won the *Turing award* for their work here

so why do we need other models?

What is shared memory used for? To *communicate*. What?

- data
- *events* (e.g., buffer not full), i.e., *timing* or *synchronisation*

How do people communicate?

- *talking*—we shall briefly look at *broadcast* later
- *messages* letters, email, or telephone (we talk, but after ringing)

By 1970, computers were talking to each other by phone—Arpanet.

The time was ripe for ...

CSP and CCS

Hoare 1978 introduced Communicating Sequential Processes (CSP)

- a formalism/mini-programming-language that
  - used just I/O to solve the concurrency problems for which we use
    - ★ Atomic actions, critical regions, semaphores, monitors . . .

How? Email

- Can carry data
- Can synchronise
  - We can agree that I wait until you send me email.

Milner (1970's, 1980, 1989): a Calculus of Communicating Systems (CCS)

- similar in manyways to CSP, but
- also connected to automata and to the λ-calculus.

Distributed systems became wide-spread after 1990 or so

- m.p. the obvious programming technique
  - packages such as MPI (Message passing interface)

Logic and semantics of CCS

Neither CSP nor CCS are part of the exam; they are included here for general background.

Transition diagrams are not restricted to concurrency. They could be non-deterministic systems resulting from any kind of program or circuit. Model checking applied to these developed almost as independent field since the late 1980's.

The pictures we draw later are inspired by CCS and CSP. Communicating transition systems.

CCS showed not only that communication is central but also worked out a theory of communication behaviour of a process (all we can observe from the outside).

More Turing awards in Concurrency: Milner, Pnueli, Lamport, Gray, Clarke et al ...

Message passing via channels

In Erlang, messages on a channel

- are all received by a unique process
  - one sends a message to a process identifier (Pid)
  - the channel has no *name* (or separate identity)
- but can be sent from any processes
  - the channel has infinite capacity, so the sender never waits
    - ★ sending is *asynchronous*

Promela channels are more general. A channel

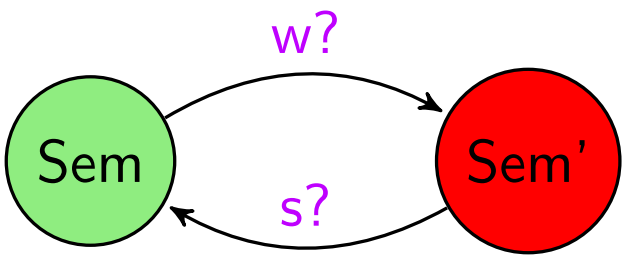
- has a name, say *a*, so
  - any process can send a message *m* on *a*,
  - any process can receive a message from *a*.
- can have 0 or any other finite capacity. As with producer/consumer,
  - a sender has to wait if the channel is full
  - a receiver has to wait if the channel is empty

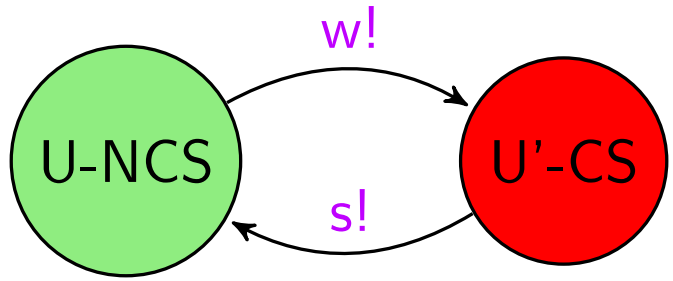
Only Erlang is exam material, not Promela.

- But Promela and our informal notation below can help understand.

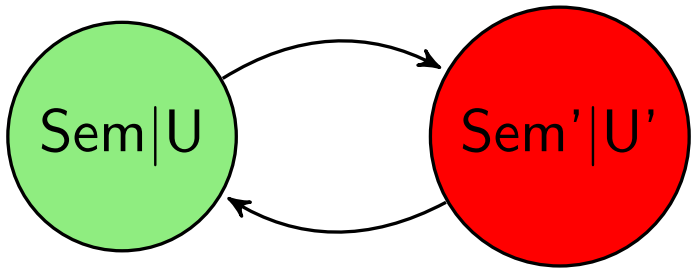
Semaphore and mutex using synchronous m.p.

The names of the channels *s*, *w* areswapped from the usual ones. Unfortunate old error, but illustrates the symmetry!





Put the two in parallel, and we get



Why no labels on the arrows now? S and U spoke to each other. We don't get to eavesdrop. But Sem can be reached by other users? We need more careful notation to say all that.

Copyright