Semaphores (chap. 6)

K. V. S. Prasad
Dept of Computer Science
Chalmer University
26-28 Jan 2014

Questions?

- Anything you did not get
- Was I too fast/slow?
- Have you joined the google group? Found a lab partner?
- Last chance to talk to a course rep

Plan

- Review critical section problem
  - First two solutions
  - State diagram proofs of correctness
- Atomic actions
  - Sketch of hardware solutions to CS
- Semaphore solution to CS
  - State diagram
  - Invariant proofs
- Invariants

Avoiding bad stories

- In concurrent programming you often
  - Want to cut out unwanted interleavings
- proctype P {knife, fork,eat} in {run P; run P}
  - Fine with atomic knife
  - But if knife = {loop until knife free; grab knife}
    - Then both P's can emerge from loop at same time
  - Making the loop and grab atomic rules out the unwanted story
    - {exit loop; exit loop; grab knife; grab knife}

Atomic actions

- A 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?

Critical Sections

- The CS problem = avoid the story below
    - Preprotocol; Preprotocol; CS; CS
- The CS problem can be solved by
    - Test-and-set, Compare-and-swap, …
        - Two things at once: minimal atomic actions
    - Or just swap (”exchange” in Alg 3.12, slide 3.23)
        - Invariant proof.
- What invariants are
    - Help to prove loops correct
    - Max example

What are hardware atomic actions?

- Setting a register
- Testing a register
  - Is that enough?
  - Think about it (or cheat, and read Chap. 3)
- But these are machine instructions
  - Semaphores are the software equivalent

Semaphores to solve Critical Sections

- We saw that the CS problem can be solved by
  - Test-and-set, Compare-and-swap, …
    - Two things at once: minimal atomic actions
  - But these are low level machine instructions
  - Semaphores: same trick at language level
- So we expect semaphores to solve CS
  - What else can they do?
  - What problems in use?
  - How do we implement them?

Processes revisited

- We didn’t really say what ”waiting” was
  - Define it as ”blocked for resource”
    - If run will only busy-wait
  - If not blocked, it is ”ready”
    - Whether actually running depends on scheduler
  - Running -> blocked transition done by process
  - Blocked -> ready transition due to external event
- Now see B-A slide 6.1
- Define ”await” as a non-blocking check of boolean condition

Semaphore definition

- Is a pair < value, set of blocked processes>
- Initialised to <k, empty>
  - k depends on application
    - For a binary semaphore, k=1 or 0, and k=1 at first
- Two operations. When proc p calls sem S
  - Wait (S) =
    - if k>0 then k:=k-1 else block p and add it to set
  - signal (S)
    - If empty set then k:=k+1 else take a q from set and unblock it
- Signal undefined on a binary sem when k=1

Critical Section with semaphore

- See alg 6.1 and 6.2 (slides 6.2 through 6.4)
- Semaphore is like alg 3.6
  - The second attempt at CS without special ops
  - There, the problem was
    - P checks wantq
      - Finds it false, enters CS,
      - but q enters before p can set wantp
- We can prevent that by compare-and-swap
- Semaphores are high level versions of this

Correct?

- Look at state diagram (p 112, s 6.4)
  - Mutex, because we don't have a state (p2, q2, ..)
  - No deadlock
    - Of a set of waiting (or blocked) procs, one gets in
    - Simpler definition of deadlock now
      - Both blocked, no hope of release
  - No starvation, with fair scheduler
    - A wait will be executed
    - A blocked process will be released

More on state diagrams

- Mutex: Check that states (CS, CS, …) do not occur
  - Such states are conceivable.
  - They just should not be *reachable*
    - from the start state
    - in a *correctly programmed* CS routine.
- Deadlock/livelock in a state diagram
  - (self-)loops from the pre-protocol state
  - Either no escape arc, or escape arcs not enabled.
- Check that conceivable but unreachable states are accounted for.

Invariants

- Semaphore invariants
  - k >= 0
  - k = k.init + #signals - #waits
  - Proof by induction
    - Initially true
    - The only changes are by signals and waits

CS correctness via sem invariant

- Let #CS be the number of procs in their CS's.
  - Then #CS + k = 1
    - True at start
    - Wait decrements k and increments #CS; only one wait possible before a signal intervenes
    - Signal
      - Either decrements #CS and increments k
      - Or leaves both unchanged
  - Since k>=0, #CS <= 1. So mutex.
  - If a proc is waiting, k=0. Then #CS=1, so no deadlock.
  - No starvation – see book, page 113

Why two proofs?

- The state diagram proof
  – Looks at each state
  – Will not extend to large systems
    - Except with machine aid (model checker)
- The invariant proof
  – In effect deals with sets of states
    - E.g., all states with one proc is CS satisfy #CS=1
  – Better for human proofs of larger systems
  – Foretaste of the logical proofs we will see (Ch. 4)