The Semantics of Concurrent Programming, 3

K. V. S. Prasad
Dept of Computer Science
Chalmers University
February – March 2013

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

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)

Towards Dekker: the problem, again

- Specification
  - Both p and q cannot be in their CS at once (mutex)
  - If p and q both wish to enter their CS, one must succeed eventually (no deadlock)
  - If p tries to enter its CS, it will succeed eventually (no starvation)
- GIVEN THAT
  - A process in its CS will leave eventually (progress)
  - Progress in non-CS optional

Different kinds of requirement

- Safety:
  - Nothing bad ever happens on any path
  - Example: mutex
  - In no state are p and q in CS at the same time
  - If state diagram is being generated incrementally, we see more clearly that this says ”in every path, mutex”
- Liveness
  - A good thing happens eventually on every path
  - Example: no starvation
  - If p tries to enter its CS, it will succeed eventually
  - Often bound up with fairness
  - We can see a path that starves, but see it is unfair

Mutex for Alg 4.1

- Invariant Inv1: (p3 or p4 or p5) -> wantp
  - Base: p1, so antecedent is false, so Inv1 holds.
  - Step: Process q changes neither wantp nor Inv1.

    Neither p1 nor p3 nor p4 change Inv1.

    p2 makes both p3 and wantp true.

    p5 makes antecedent false, so keeps Inv1.

So by induction, Inv1 is always true.

Mutex for Alg 4.1 (contd.)

- Invariant Inv2: wantp -> (p3 or p4 or p5)
  - Base: wantp is initialised to false , so Inv2 holds.
  - Step: Process q changes neither wantp nor Inv1.

    Neither p1 nor p3 nor p4 change Inv1.

    p2 makes both p3 and wantp true.

    p5 makes antecedent false, so keeps Inv1.

So by induction, Inv2 is always true.

Inv2 is the converse of Inv1.

Combining the two, we have

Inv3: wantp <-> (p3 or p4 or p5) and

wantq <-> (q3 or q4 or q5)

Mutex for Alg 4.1 (concluded)

- Invariant Inv4: not (p4 and q4)
  - Base: p4 and q4 is false at the start.
  - Step: Only p3 or q3 can change Inv4.

p3 is ”await (not wantq)”. But at q4, wantq is true by Inv3, so p3 cannot execute at q4.

Similarly for q3.

So we have mutex for Alg 4.1

Proof of Dekker's Algorithm (outline)

- Invariant Inv2: (turn = 1) or (turn = 2)
- Invariant Inv3: wantp <-> p3..5 or p8..10
- Invariant Inv4: wantq <-> q3..5 or q8..10
- Mutex follows as for Algorithm 4.1
- Will show neither p nor q starves
  - Effectively shows absence of livelock

Liveness via Progress

- Invariants can prove safety properties
  - Something good is always true
  - Something bad   is always false
- But invariants cannot state liveness
  - Something good happens eventually
- Progress A to B
  - if we are in state A, we will progress to state B.
  - Weak fairness assumed
- to rule out trivial starvation because process never scheduled.
  - A scenario is weakly fair if
  - B is continually enabled at state Ain scenario ->
  -     B will eventually appear in the scenario

Liveness in Dekker's algorithm

- We used the Utrecht slides

Monitors

- To implement semaphores
- To do readers/writers

waitC(cond)

Append p to cond

p.State <- blocked

Monitor release

signalC(cond)

If cond not empty

    q <- head of queue

    ready q

Correctness of semaphore

- See p 151
- Exactly the same as fig 6.1 (s 6.4)
- Note that state diagrams simplify
  - Whole operations are atomic

Readers and writers

- Alg 7.4
- Not hard to follow, but lots of detail
  - Readers check for no writers
  - But also for no blocked writers
    - Gives blocked writers prioroty
  - Cascaded release of blocked readers
    - But only until next writer shows up
  - No starvation for either reader or writer
- Shows up in long proof (sec 7.7, p 157)
  - Read at home!

Readers-writers invariants

- Readers exclude writers but not other readers
- Writers write alone
- Invariants R >= 0 and W >= 0
- Theorem
  - (W <= 1) and (W=1 -> R=0) and (R>0 -> W=0)
  - Monitor operations are atomic! Check each one.
  - Don’t forget to check the Signal operations

Readers progress

- i.e., Won’t block forever on OKtoRead
- Invariants (lemma)
  - Not empty(OKtoRead) -> (W ne 0) or not empty(OKtoWrite)
  - not empty(OKtoWrite) -> (R ne 0) or (W ne 0)
- not empty(OKtoRead)-> <> signalC(OKtoRead)
  - Case W ne 0
  - Writer progresses, executes the signalC(OKtoRead)
  - Case R ne 0
  - Last reader releases a writer, reduces to previous case

Exchange

- ex(a,b) = atomic{local t; t:=a; a:=b; b:=t}
- See slide 3.23, alg 3.12
- Prove correct
- We did this as an example exercise from the book