Lecture 11: More on invariants and textual reasoning - examples

K. V. S. Prasad
Dept of Computer Science
Chalmers University
Friday 30 Sep 2016

New GU student reps – please see me

ABOU ZIDAN	NASHWAN	gusabouzna@student.gu.se				
CORDERO MIRANDA	ENRIQUE	guscoren@student.gu.se				
FU	KAI	guskaifu@student.gu.se				
HENNE	BENJAMIN	gushennbe@student.gu.se				
SINGH	DEEPAK KUMAR	gussinde@student.gu.se				

Plan for today

Chap 6 almost complete

Chap 7 intro +

Deadlock now, not livelock

- Software abstractions provided by OS or RTS:
  - processes, semaphores, monitors, protected objects, channels, and messages
  - Blocking of processes, so busy-waits not needed
  - Events managed and mediated:
    - change of variable, semaphore or condition, or arrival of a message
- None of these occur in “nature”
  - They were invented – read the history!
- Deadlock = everyone blocked
  - Reasoning very similar to that for livelock

Primitives and Machines

- We see this repeatedly in Computer Science
  - Whether for primitives or whole machines
- Recognise pattern in nature or in use
  - Critical section motivating ex. for semaphores
- Specify primitive or machine
  - Set or queue? Direct handover upon signal?
- Figure out range of use and problems
  - today
- Figure out (efficient) implementation
  - Maybe later

Pet examples

- Passing a door from opposite directions
  - If both sleep until the other passes – deadlock
  - If both eager – livelock (busy waiting)
- Library
- The knife (atomic; deadlock if fork+knife picked up in either order)
- The printer (grab then file, or atomic per sheet?)
- Count up to 20
- Max, sort by chemical machine
- Max and grabbing by broadcast

Safety and liveness - reminder

- Deadlock and violation of mutex
  - occur in a state (a “bad” state)
    - Safety property
  - so to check for mutex and absence of deadlock
    - Check that no reachable state is bad
- To show absence of progress
  - Find a loop through the states with no progress
  - Liveness (progress, starvation) is harder to prove
- Deadlock = mutual starvation
  - But important special case
    - Also because easier to prove

Semaphore ops

- Signal (S)
  - If S. L = {} then S.V ++ else S.L:= S.L-{q};
    q.state := ready
    (for q in S.L)
- Wait(S)
  - If S.V > 0 then S.V - - else S.L:= S.L U {p};
    p.state := blocked
    (where p did wait)

CS by semaphores

- Slide 6.2 (p 112) onwards
  - For mutex, prove ¬(p3 ^ q3) is invariant
    - True at start
    - Suppose p3. Then show q cannot get past q2
      - Why? Because S.V=0 after one process has done wait
  - For deadlock, show (p2^q2)->S.V=1
    - True the first time
    - Subsequent visits to p2 mean p4 has executed.
  - Fairness – the semaphore op guarantees it!
    - If p waits, then q signals, p will become ready
    - Subsequent wait by q blocks q

Invariants recap

- Have to hold outside atomic actions
  - Do not need to hold during atomic actions
- Help to prove loops correct
  - Game example with straight and wavy lines
- Semaphore invariants (abbreviating S.V by k)
  - 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 for N≥2 processes

- 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 for N=2
    - But possible for N>2. P blocks, while Q and R alternate.

CS problem for n processes

- See alg 6.3 (p 113, s 6.5)
  - The same algorithm works for n procs
  - The proofs for mutex and deadlock freedom work
    - We never used special properties of binary sems
  - But starvation is now more likely
    - p and q can release each other and leave r blocked
- Exercise: If k is set to m initially, at most m processes can be in their CS's.

CS correctness (contd.)

- No starvation (if just two processes, p and q)
  - If p is starved, it is indefinitely blocked
  - So k = 0 and p is on the sem queue, and #CS=1
  - So q is in its CS, and p is the only blocked process
  - By progress assumption, q must exit CS
  - Q will signal, which immediately unblocks p
- Why “immediately”?

Why so many 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)

Mergesort using semaphores

- See p 115, alg 6.5 (s 6.8)
  - The two halves can be sorted independently
    - No need to synch
  - Merge, the third process,
    - has to wait for both halves
  - Note semaphores initialised to 0
    - Signal precedes wait
    - Done by process that did not do a wait
  - Not a CS problem, but a synchronisation one

Producer - consumer

- Yet another meaning of ”synchronous”
  - Buffer of 0 size
- Buffers can only even out transient delays
  - Average speed must be same for both
- Infinite buffer first. Means
  - Producer never waits
  - Only one semaphore needed
  - Need partial state diagram
  - Like mergesort, but signal in a loop
- See algs 6.6 and 6.7

Infinite buffer (slide 6.9, p 119)

- State space now infinite – how to draw diagram?
  - See p 118 of the book (slide 6.10, p 120)
- Invariant
  - #sem = #buffer
    - 0 initially
    - Incremented by append-signal
      - Need more detail if this is not atomic
    - Decremented by wait-take
- So consumer cannot take from empty buffer
- Only consumer waits – so no deadlock or starvation, since prod will always signal

Bounded buffer

- See alg 6.8 (p 119, s 6.12)
  - Two semaphores
    - Consumer waits if buffer empty
    - Producer waits if buffer full
  - Each process needs the other to release ”its” sem
    - Different from CS problem
  - ”Split semaphores”
  - Invariant
    - notEmpty + notFull = initially empty places

Different kinds of semaphores

- ”Strong semaphores”
  - use queue instead of set of blocked processes
    - No starvation
- Busy wait semaphores
  - No blocked processes, simply keep checking
    - See book re problems about starvation
  - Simpler.
    - Useful in multiprocessors where each proc has own CPU
      - The CPU can’t be used for anything else anyway
    - Or if there is very little contention

Dining Philosophers

- Obvious solution deadlocks (alg 6.10)
- Break by limiting 4 phils at table (6.11)
- Or by asymmetry (6.12)

Dining philosophers with semaphores

- Slide 6.14 to 6.18 (p. 124 to 128)
- Requirements
  - Can only eat with lhs and rhs fork
  - Mutex over each fork
  - Deadlock-free
  - Starvation-free
  - (efficient if no contention)

Semaphores, Monitors, Protected Objects

K. V. S. Prasad
Dept. of Computer Science
Chalmers University
30 Sep 2016

Semaphore recap

- Designed for CS problem or atomic actions
  - (even with n-proc)
  - Avoid busy waiting
- But for the producer-consumer problem
  - The correctness of each proc
    - Depends on the correctness of the other
  - Not modular
- Monitors modularise synchronisation
  - for shared memory

Monitors = synchronised objects

- A type of monitors looks like a class with sync
- An operation on a monitor
  - Looks atomic
  - All operations are mutex w.r.t. each other
    - i.e., only one operation at a time
- So alg 7.1 can only result in n=2 at the end.

Confusions with O-O programming

- Monitors are static
  - They don’t ”send messages” to each other
- Processes are the running things
  - They can enter the monitor one at a time
  - There is no queue of processes waiting to get in,
    - Only a set

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

Condition Variables = named queues

- Mutex?
  - Monitors provide it, by definition (See alg 7.1)
- But often, need explicit synchronisation
  - i.e., processes wait for different events
    - Producer waits till (someone makes) buffer notFull
    - Consumer waits till (someone makes) buffer notEmpty
  - They need to be unblocked
    - when the corresponding event occurs
- In monitors, each such event
  - Has a queue associated with it
    - In fact, for the monitor, the ”event” ***is*** just the queue
    - These queues are called ”condition variables”

Semaphore implemented by monitor

- Alg 7.2
- No explicit release of monitor lock
  - Leave when done
- waitC always blocks
  - This is not the semaphore's wait
  - When unblocked by signal
    - Must wait till signalling proc leaves monitor
- signalC has no effect on empty queue
  - Semaphore signal always has an effect

waitC (on monitor condition var) vs wait on semaphore

waitC (on monitor condition var)
Append p to cond
p.State <- blocked
Monitor release

So waitC always blocks!

Wait(S)
If S.V > 0 then S.V := S.V-1
else S.L := S.L + {p}; block p

signalC (on monitor condition var) vs signal on semaphore

signalC (on monitor condition var)

If cond not empty

q <- head of queue

ready q

Signal(S)

If S.L empty then S.V := S.V+1

else S.L := S.L –{q}; ready q *(for abitrary q)*

Correctness of semaphore by monitor

- See p 151 in book (slide 7.5, p 164)
- Identical to fig 6.1, p 112 in book (slide 6.4, p 114)
- Note that state diagrams simplify
  - Whole operations are atomic
- Check: for well-behaved program
  - There are 3 states per process, incl. Blocked
    - The variable values are determined by the proc states
  - 4 unreachable states
    - blocked-blocked (deadlock)
    - signal-signal (no mutex)
    - wait-blocked or blocked-wait (deadlock coming!)
  - For mutex starting with k=1, and two user processes

Producer-consumer

- Alg 7.3
- All interesting code gathered in monitor
- Very simple user code

Immediate resumption

- So signalling proc cannot again falsify cond
  - If signal is the last op, allow proc to leave?
    - How? See protected objects
- Many other choices possible
  - Check what your language implements

Semaphores vs monitors: examples

- Semaphores
  - Library- user returning book chooses sleeper and wakes them
  - Prod-cons – each wakes the other
  - Can't tell at a glance what the semaphore is for
    - Mutex? Synchronisation signal?
- Monitor
  - mutex access; synchronisation by condition variables
  - Library- users only contract with the library
    - takes care of returns, chooses sleeper and wakes them
  - Prod-cons – each only contracts with the buffer

Design issues with monitors

- A borrower has to wait (where?)
  - The returner and woken up borrower
    - Can be active together?
    - If not, who waits? Where?
  - “Hoare semantics”(immediate resumption)
    - the returner has to wait – where?
    - Why? So the borrower doesn’t find book gone
  - “Mesa semantics”
    - Returner signals and leaves, then wake up borrower
      - Who must again check if book is available

More monitor design issues

- When do you check if book is available?
  - Why not right away?
  - Whatever you do before that cannot change cond
  - Because that is signalled by the returner
- So you can check in a cond.var ante-room
- Drop explicit signal by returner
- Then who checks cond-vars?
  - The system
  - check all c-v's whenever anyone leaves

So: protected objects

- = monitors with cond. Vars -> entry guards
  - Call to entry blocks till guard is true
  - No signals
    - Simply check all guards whenever a user leaves

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!

Dining philosophers again

- Alg 7.5

Protected objects

- Monitors need waitC and signalC programmed
- Protected objects combine this with queueing
- See alg 7.6 for readers-writers
  - Each operation starts only when its cond is met
    - Called a ”barrier”
  - What happened to signalC?
    - When any op exits, all barriers are checked

Protected objects (contd.)

- See alg 7.6 (p 164, s 7.16)
- Tidies 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)

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!

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!