Reasoning about Monitors and Protected Objects

K. V. S. Prasad
Dept of Computer Science
Chalmers University
Lecture 9, TDA384/DIT391: Friday 22 Sep 2017

Correctness - safety

- A safety property must always hold
  - In every state of every computation
- = "nothing bad ever happens"
  - Typically, partial correctness (Program is correct if it terminates)
    - E.g., "loop until head, toss"
      - sure to produce a head if it terminates
      - But not sure it will terminate
      - Will do so with increasing probability the longer we go on
    - How about "loop until sorted, shuffle deck"?
      - Sure to produce sorted deck if it terminates
      - Needs much longer expected run to terminate
    - Can guarantee neither progress nor termination

Correctness - Liveness

- A liveness property must eventually hold
  - Every computation has a state where it holds
- = a good thing happens eventually
  - Termination
  - Progress = get from one step to the next
  - Non-starvation of individual process
- Sort by shuffle is safe but cannot guarantee liveness - either progress or termination

(Weak) Fairness assumption

- If at any state in the scenario, a statement is continually enabled, that statement will eventually appear in the scenario.
- So an unfair version of coin tossing cannot guarantee we will eventually see a head.
- We usually assume fairness

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.

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 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 (p did wait)

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

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
- Exactly the same as fig 6.1 (s 6.4)
- Note that state diagrams simplify
  - Whole operations are atomic
- Check: for well-behaved program
  - 4 unreachable states
    - blocked-blocked (deadlock)
    - signal-signal (no mutex)
    - wait-blocked (deadlock coming!)
  - For mutex starting with k=1, and two user processes
    - The variable values are determined by the proc states

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!

monitor+user are correct for readers-writers

- Lemma 7.1 Show that R>=0 and W>=0
  - Proof: Initialised to 0. Increased in StartRead and StartWrite and decreased in the End ops. So these invariants only follow because of user.
  - Note: to prove that StartRead can only increase, we need two cases – was Signal(OktoRead) invoked? So R will increase by #OktoRead.
  - EndRead might start a writer. But this is a brief digression. Corresponding EndWrite only after EndRead terminates.

monitor+user correct for readers-writers - 2

- Theorem 7.2 (R>0 -> W=0) & (W<=1) &(W=1 -> R=0)
  - Base case: at start, the premises are false.
  - Steps: 4 ops without startC operations (and 4 with,later).
    - StartRead could falsify either implication. But the if says W=0
    - EndRead could falsify R>0 (1^{st} implication true) or make R=0 (2^{nd} implication true).
    - StartWrite could falsify W<=1 or W=0, but only operates when W=0 and R=0. So all true.
    - EndWrite can only make W=0. So cannot falsify anything.

monitor+user correct for readers-writers - 3

- Theorem 7.2 (R>0 -> W=0) & (W<=1) & (W=1 -> R=0)
  - Steps: 4 ops with startC operations.
    - SignalC(OktoRead) in StartRead: only when W=0. So the awoken reader finds the same. So no implication falsified.
    - SignalC(OktoRead) in EndWrite: only when W=0, by middle invariant. So the awoken reader finds the same. So no implication falsified.
      - NOTE EASIER TO PROVE ALL THREE INVARIANTS TOGETHER.
    - SignalC(OktoWrite) in EndRead: only when R=0. So the awoken writer no implication falsified. W=0 when EndRead began, by first inv.
    - SignalC(OktoWrite) in EndWrite: only when W=R=0 and readers wait. So the awoken writer no implication falsified.

RW monitor: Lemmas to prove no starvation

- W waiting -> R>0 or W>0
  - Base: true (no W waiting)
  - Step: Preserved by StartWrite. Can R=0 and W=0 while a W waits? Examine EndRead and EndWrite to see that the inv holds after each monitor op.
- R waiting -> W waiting or W>0
  - Base: true (no R waiting)
  - Step 1: An R can start waiting only if consequent is true.
  - Step 2: Can consequent become false while an R waits?
    - EndWrite makes W=0 but the signals will ensure either W=1 again, or a cascade of waiting readers.

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
- DO EXAMPLES AND PROOFS FOR PROTECTED OBJECTS

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!