

State diagrams, interleaving, atomic actions, critical sections

K. V. S. Prasad

Dept of Computer Science

Chalmers University

1 September 2016

Functional Programming (F.P.)

- If you have never done F. P., get started early on the Erlang tutorial.
- Even if you have, take a quick look
 - to get an overview
 - to understand how hard or easy it will be for you to learn Erlang

Reminder

- Next course rep meeting Monday 5 Sep after the lecture
 - Say what you think / need / would like to your rep
 - Say what you think is going well as well as what could be better (for you)

Plan for today

- Example: Sharing a meal, or a bank a/c
- State diagrams
- Concurrency models – (a)synchrony, time, ...
- Critical sections
 - Atomic actions
- History
- Chaps from Ben-Ari (for 1 and 2 Sep 2016)
 - 2.1 to 2.5
 - 3.1 to 3.2 (maybe more if time permits)
 - 6.1 to 6.4 (maybe 6.7)
 - 7.1-7.4, 7.10

Sharing a meal

- Consider a program
 - *proctype P {grab knife; grab fork; eat}*
 - *proctype Q {grab fork; grab knife; eat}*

Here, “*proctype*” (as in Promela, for those who are following it)

- is a type declaration.
- does not produce a process
- That happens when “*run P*” or “*run Q*” is executed
 - Both instantiates and starts a copy of P or Q

Then *{run P; run P}* will result in both eating one after the other

But *{run P; run Q}* might result in P eating after Q or the other way, or in deadlock.

Shared bank account

- ***proctype W(i) {loc(i):= bal; loc(i)--; out(i) 1; bal:=loc(i)}***
 - *bal* is shared global balance
 - *loc(i)* is local register
 - *out(i)* is payout
- Then ***{run W(5); run W(6)}*** could result in both succeeding in their withdrawals, but with the account being debited just once, as in the following scenario
- ***loc(5):=bal; loc(6):=bal;
loc(5)--; loc(6)--;
out(5) 1; out(6) 1;
bal:=loc1; bal:=loc2***

Interleaving

- Each process executes a sequence of atomic commands (usually called "statements", though I don't like that term).
- Each process has its own control pointer, see Alg 2.1 of Ben-Ari
- For Alg 2.2, see what interleavings are impossible
- See slides 2.3 – 2.7 of Ben-Ari

State diagrams

- In slides 2.4 and 2.5, note that the state describes variable values before the current command is executed.
- In 2.6, note that the "statement" part is a pair, one statement for each of the processes
- Not all thinkable states are reachable from the start state

Scenarios

- A scenario is a sequence of states
 - A path through the state diagram
 - See Ben-Ari slide 2.7 for an example
 - Each row is a state
 - The statement to be executed is in bold

The counting example

- See algorithm 2.9 on slide 2.24
 - What are the min and max possible values of n ?
- How to say it in C-BACI, Ada and Java
 - 2.27 to 2.32

Atomic statements

- The 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?

The Critical Section Problem

- Attempts to solve them
 - without special hardware instructions
 - Assuming load and store are atomic
 - Designing suitable hardware instructions

Requirements and Assumptions

- Correctness requirements
 - 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)
- Assumptions
 - A process in its CS will leave eventually (progress)
 - Progress in non-CS optional

Comments

- Pre- and post-protocols
 - These don't share local or global vars with the rest of the program
- The CS models access to data shared between p and q

First try (alg 3.2, slide 3.3)

- The full state diagram shows only 16 states are reachable, out of 32
- These exclude states $(p3, q3, *)$ so mutex is OK.
- The abbreviated program reduces state space
- if p1 is stuck in NCS with $turn=1$, q starves
- Deadlock free in the sense that p can enter CS
- Error: p and q both set and test "turn"; if one dies the other is stuck

Semaphores, Monitors, Protected Objects

K. V. S. Prasad

Dept. of Computer Science

Chalmers University

2 Sep 2016

Questions?

- Reminder: course rep meeting next Monday 5 Sep
 - give your rep notes, suggestions, etc. during the break
- Anything you want to say
 - Comments, questions, stray thoughts, etc.
 - Are we too fast/slow?
- More status questions
 - How did the demo/viva go?
 - Mail us if there are problems (of any kind)

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

Plan

- Chap 6 examples
- Chap 7
 - Monitors (contd.)
 - protected objects
- Transition to message passing

Chap 3 & 4 (skipped for now)

REMINDER: do the exercises in Chaps. 1, 2, 3, 6

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

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.

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 is correct

- Invariant
 - $\#sem = \#buffer$
 - 0 initially
 - Incremented by append-signal
 - Need more detail if this is not atomic
 - Decremented by wait-take
- So cons cannot take from empty buffer
- Only cons waits – so no deadlock or starvation, since prod will always signal

Bounded buffer

- See alg 6.8 (p 119, s 6.12)
 - Two semaphores
 - Cons waits if buffer empty
 - Prod waits if buffer full
 - Each proc needs the other to release "its" sem
 - Different from CS problem
 - "Split semaphores"
 - Invariant
 - $\text{notEmpty} + \text{notFull} = \text{initially empty places}$

Different kinds of semaphores

- "Strong semaphores"
 - use queue instead of set of blocked procs
 - 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)

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

Correctness, and software processes

- 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

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

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 arbitrary 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 priority
 - 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!