

Lecture 5: Monitors and protected objects

K. V. S. Prasad
Dept of Computer Science
Chalmers University
15 Sep 2014

Questions?

- Reminder: course rep meeting next Tuesday 23 Sep
 - All: give your rep some notes, suggestions, etc. during the break
- Anything you want to say
 - Comments, questions, stray thoughts, etc.
 - Are we too fast/slow?
- More reminders
 - Joined the google group. You must, to mail us and get replies
 - Please don't mail us at our personal addresses
 - Found a lab partner? Ask tutors for help if needed

Plan for today

- Chap 6: recap
- 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

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

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

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

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

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

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!