More on Semaphores, on to Monitors

K. V. S. Prasad
Dept of Computer Science
Chalmers University
9 September 2013

Questions?

- Anything you did not get
- Was I too fast/slow?
- Have you joined the google group? Found a lab partner?
- Haven't yet heard from all course reps

Natural Example

1. Clocks occur in nature
    1. Repressilator – human-made, incorporated in E-coli

Primitives and Machines

- We see this repeatedly in Computer Science
  - Whether for primitives or whole machines
- Recognise pattern in nature or in use
- Specify primitive or machine
- Figure out range of use and problems
- Figure out (efficient) implementation

Many Concurrency models

1. What world are we living in, or choose to?
   a. Synchronous or asynchronous?
   b. Real-time?
   c. Distributed?
2. We choose any abstraction that
   a. Mimics enough of the real world to be useful
   b. Has nice properties (can build useful and good programs)
   c. Can be implemented correctly, preferably easily

Concurrency Primitives in History

- 1950's onwards
  - Read-compute-print records ***in parallel***
  - Needs ***timing***
- 1960's onward
  - slow i/o devices in parallel with fast and expensive CPU
  - ***Interrupts, synchronisation, shared memory***
- ***Processes and context switching***
  - Late 1960's : timesharing expensive CPU between users
  - Modern laptop: background computation from which the foreground process steals time

Terminology

- A ”process” is a sequential component that may interact or communicate with other processes.
- A (concurrent) ”program” is built out of component processes
- The components can potentially run in parallel, or may be interleaved on a single processor. Multiple processors may allow actual parallelism.

How to structure processes?

- What are they?
    - How do you answer that?
- Start with examples from real life
    - Each I/O device can be a process
    - Then what about the CPU?
    - Each device at least has a ”virtual process” in the CPU
    - Context switching
        - move next process data into CPU
        - When? On time signal or ”interrupt”
        - How? CPU checks before each instruction
- What does a process look like to others?
    - What does each process need to know?
    - What does the system need to know about each process?

These ideas became standard in Operating Systems (60’s thru 70’s)

- Divided into kernel and other services
  - Other services run as processes
- The kernel
  - Handles the actual hardware
  - Implements abstractions
    - Processes, with priorities and communication
  - Schedules the processes
    - using time-slicing or other interrupts
- A 90’s terminology footnote
  - When a single OS process structures itself as several processes, these are called ”threads”

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

The Critical Section Problem

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

Requirements and Assumptions

- Correctness
  - 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

Rethink

- P checks wantq
  - Finds it false, enters CS,
    - but q enters before p can set wantp
- Could we prevent that?
  - When I find the book free, I take it
    - Before anyone else even sees it free
- Test-and-set(common, local) = atomic{local:=common; common:=1}
  - Now see Ben-Ari p76, slide 3.22, alg 3.11
  - See Wikipedia article, also Herlihy 1991

Exchange and other atomics

- Slides 3.22 and 3.23
- Other atomic instructions
  - Compare and swap
  - Fetch-and-add
- All use busy waits
  - OK in multiprocessors
    - Particularly if low contention

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

- Do you know what they are?
  - Help to prove loops correct
  - Game example
- 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)

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
    - notEmpty + notFull = initially empty places

Different kinds of semaphores

- ”Strong semaphores”
  - use queue insteadof 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