Lecture 3: Semaphores (chap. 6)

K. V. S. Prasad
Dept of Computer Science
Chalmer University
6 Sep 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

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 2.1 of Ben-Ari
- For 2.2, see what interleavings are impossible

Why arbitrary interleaving?

- Multitasking (2.8 is a picture of a context switch)
  - Context switches are quite expensive
  - Take place on time slice or I/O interrupt
  - Thousands of process instructions between switches
  - But where the cut falls depends on the run
- Runs of concurrent programs
  - Depend on exact timing of external events
  - Non-deterministic! Can't debug the usual way!
  - Does different things each time!

Arbitrary interleaving (contd.)

- Multiprocessors (see 2.9)
  - If no contention between CPU's
    - True parallelism (looks like arbitrary interleaving)
  - Contention resolved arbitrarily
    - Again, arbitrary interleaving is the safest assumption

But what is being interleaved?

- Unit of interleaving can be
  - Whole function calls?
  - High level statements?
  - Machine instructions?
- Larger units lead to easier proofs but make other processes wait unnecessarily
- We might want to change the units as we maintain the program
- Hence best to leave things unspecified

Why not rely on speed throughout?

- Don't get into the train crash scenario
  - use speed and time throughout to design
  - everyday planning is often like this
    - Particularly in older, simpler machines without sensors
    - For people, we also add explicit synchronisation
- For our programs, the input can come from the keyboard or broadband
  - And the broadband gets faster every few months
- So allow arbitrary speeds

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?

What are hardware atomic actions?

- Setting a register
- Testing a register
- Is that enough?
- Think about it (or cheat, and read Chap. 3.10)

Obey the rules you make!

1. For almost all of this course, we assume single processor without real-time (so parallelism is only potential).
2. Real life example where it is dangerous to make time assumptions when the system is designed on explicit synchronisation – the train
3. And at least know the rules! (Therac).

Semaphores to solve Critical Sections

- We saw that the CS problem can be solved by
  - Test-and-set, Compare-and-swap, …
    - Two things at once: minimal atomic actions
  - But these are low level machine instructions
  - Semaphores: same trick at language level
- So we expect semaphores to solve CS
  - Why is the CS problem so important?
  - It is how we restrict interleaving
- What else can they do? What problems in use?
- How do we implement them?

Processes revisited

- We didn't really say what "waiting" was
  - Define it as "blocked for resource"
    - If run will only busy-wait
  - If not blocked, it is "ready"
    - Whether actually running depends on scheduler
  - Running -> blocked transition done by process
  - Blocked -> ready transition due to external event
- Now see B-A slide 6.1
- Define "await" as a non-blocking check of boolean condition

Semaphore definition

- Is a pair < value, set of blocked processes>
- Initialised to <k, empty>
  - k depends on application
    - For a binary semaphore, k=1 or 0, and k=1 at first
- Two operations. When proc p calls sem S
  - Wait (S) =
    - if k>0 then k:=k-1 else block p and add it to set
  - signal (S)
    - If empty set then k:=k+1 else take a q from set and unblock it
- Signal undefined on a binary sem when k=1

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

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 possible
    - 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

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