Introduction to programming with semaphores
Fri 1 Sep 2017

K. V. S. Prasad
Dept of Computer Science
Chalmers University
Lecture 2 of TDA384/DIT301, August –October 2017

Assignments?

- How's it going with the Erlang tutorial?
  - Anyone seriously worried about FP?
- Did you look at the Java in lab 3? Comfortable with it?
- Did you glance through Ben-Ari or at least his slides?

Recap

- Motivations
  - Simulate multi-agent systems; 2 agents or 10 million
  - For greater speed through parallelism
- Main abstraction – a process
  - Could be a CPU, a person, an ant, a bot, …
  - Cooperates, coordinates, competes, communicates … with other processes
    - Using shared CPUs, memory cells, communication channels, …

Time

- The main trick is to manage time
  - In a purely parallel setup like music, every process follows the time axis.
  - In cinema, time expands, shrinks, is skipped, and goes back and forth.
    - Intercutting = processes share the screen (CPU) -- may or may not run off screen.

Concurrent systems

- Real time systems are a bit like music – actual time matters
- Parallel programs are like music too
  - all play all the time (counting rests as playing)
- Concurrent systems are like multi-screen cinema
  - But no skipping, no going back
  - Must show sending of a signal or message or event before receipt
  - Time reduced to sequencing – simpler, allows flexible scheduling
    - But don't sneak actual time in the back door
      - Remember the train crash "never mind token, you have time to escape"

Execution model and scenarios

- Of all the processes in the program
  - pick any runnable one, and execute one step
  - Or, pick any subset of runnable processes and do one step of each
- Since the picking is arbitrary, we get non-determinism
- So usual debugging (breakpoints, stepping, starting over, etc.) fails
- Need to check every possible run
  - Done by model checkers such as SPIN
  - Or by examining the program text

Sleep instead of busy wait

- Don't check every cycle for condition (event, message)
  - Instead, say "I'll sleep. Wake me up on condition."
  - Run time support or Operating system will wake you up.
  - It will also schedule a runnable (non-sleeping) process onto an available CPU.
    - If no CPU available, a process is runnable but not running
- Hence Ben-Ari chap 6 model of process states

People trying to borrow the lone library book - failing program

book := free

a1: loop until book=free;	b1: loop until book=free;
a2: book:= borrowed	b2: book:= borrowed

students a and b might execute a1 and b1 in parallel,
and then a2 and b2 in parallel, resulting in double booking.
And yet both politely checked availability first.

a1 and b1 use a test instruction (=, yielding true/false), while
a2 and b2 are set instructions. It is the separation between test and set that allows interference.

Hardware solution – an atomic test-and-set instruction

- tset(*mine,library*) = atomic{*mine:=library; library*:=0}
- *Mine* and *library* say how many books (0 or 1) we each have
- To begin with, *library*=1, and both students have respective *mine*=0

library := 1

a: loop tset(mine,library) until mine=1;

b : loop tset(mine,library) until mine=1;

Students a and b can execute their loops a and b in parallel, and only one will emerge with the book.
But these loops are busy waits.

Semaphore = software test-and-set, plus wake-up services

- Suppose you had a wait(book) instruction, which
  - if book=free, gives you the book
  - if book=borrowed, puts you to sleep and wakes you when it is free
- This achieves the same as the loops in the previous slide
  - You emerge with a book
  - or go to sleep (instead of being stuck in a busy loop)

Returning the book?

- In the hardware solution, simply set library:=1
  - If you only use mine via the tset, you don't need to set mine:=0
- In the software solution, we need to be a little more sophisticated

A semaphore S is a compound data type with two fields, S.V of type non-negative integer, and S.L of type set of processes. We are using the familiar dotted notation of records and structures from programming languages. A semaphore S must be initialized with a value $k \geq 0$ for S.V and with the empty set Ø for the S.L:

```
semaphore S ← (k, Ø)
```

There are two *atomic* operations defined on a semaphore S (where p denotes the process that is executing the statement):[1]

wait(S)

```
if  S.V > 0
    S.V ← S.V − 1
else
    S.L ← S.L ∪ p
    p.state ← blocked
```

110 6 Semaphores

signal(S)

```
if S.L = ∅
   S.V ← S.V + 1
else
   let q be an arbitrary element of S.L
   S.L ← S.L − {q}
   q.state ← ready
```

If S.L is empty, increment the value of the integer component; otherwise—S.L is nonempty—unblock q, an arbitrary element of the set of processes blocked *on* S.L. The state of process p does not change.

A semaphore whose integer component can take arbitrary non-negative values is

Alg 6.1 of Ben-Ari

- Each student does

```
loop
    other stuff      //non-critical section (NCS)
    sleep till you borrow book // pre-protocol
    read book        //critical section (CS)
    return book      //post-protocol
```

CS assumptions and requirements

- Assumptions
  - NCS may loop - Students may drop out of course here
  - But CS must terminate – must return borrowed book
- Requirements
  - book is issued to at most one student at a time (mutex)
  - Attempt to borrow will succeed (liveness, no starvation)
  - Multiple students trying to borrow -> one will succeed
    - i.e., no deadlock (= all sleeping, so no one wakes-up another)

Proof by state diagram, figure 6.1

- Extends easily to multiple students
  - need fair semaphores
    - ensure no one is ignored all the time
- Extends easily to multiple copies of book
  - need general semaphore

Producer - consumer

- 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
  - 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

Old Psuedo-code for single place buffer

flag := empty

- producer
- loop

```
d1: loop until flag=empty;
      R := new record;
d2: flag := full
```

- consumer
- loop

```
p1: loop until flag=full;
      temp:= R;
p2: flag := empty
```

Why does this single flag code work?

- Just one semaphore?
- The two conditions are mutually exclusive and complementary
  - Flag can only be empty or full, and not both at once
- Each process flips the flag to the value the other is waiting for
- The flag is like an ad-hoc semaphore with both wait and –wait
  - -wait = wait for 0 instead of wait for 1
- Such ad-hoc solutions might exist in many cases.
- The two semaphore solution for the bounded buffer uses standard primitives, an advantage in itself
- Ad hoc solutions require more care