Implementing counting semaphores using binary semaphores

A. Udaya Shankar

`http://www.cs.umd.edu/users/shankar/412-Notes/10x-CountingFromBinarySemaphores.html`
August 11, 2011

1 Introduction

Two implementations of counting semaphores using binary semaphores are given below. Implementation 1 is incorrect. Thanks to Timothy Alicie for pointing this out. Implementation 2 (by Barz) is proved to be correct.

Assignment is ←. Equality is "=". `CSem` stands for counting semaphores. `BSem` stands for binary semaphores.

References

- Hans W. Barz. 1983. Implementing semaphores by binary semaphores. SIGPLAN Not. 18, 2 (February 1983), 39-45.
  DOI=10.1145/948101.948103 http://doi.acm.org/10.1145/948101.948103).
- David Hemmendinger. 1989. Comments on "A correct and unrestrictive implementation of general semaphores". SIGOPS Oper. Syst. Rev. 23, 1 (January 1989), 7-8.
  DOI=10.1145/65762.65763 http://doi.acm.org/10.1145/65762.65763.

2 Implementation 1 (incorrect)

```
CSem(K) cs {    // counting sem, init K
   int val ← K;  // value of csem
   BSem wait(0);  // to block on csem
   BSem mutex(1); // protects val

   Pc(cs) {
      P(mutex);
      val ← val - 1;
      if val < 0 {
         V(mutex);
    1:   P(wait);
      } else
         V(mutex);
   }

   Vc(cs) {
      P(mutex);
      val ← val + 1;
      if val ≤ 0
         V(wait);
      V(mutex);
   }
}
```

Evolution showing error

```
Initial:
  cs = 0; val = 0; wait = 0; mutex = 1.
Thread t1 attempts Pc(cs):
  t1 at 1; val = -1; wait = 0; mutex = 1.
Thread t2 attempts Pc(cs):
  t1, t2 at 1; val = -2; wait = 0; mutex = 1.
Thread t3 executes Vc(cs) twice:
  t1, t2 at 1; val = 0; wait = 1; mutex = 1.
Thread t2 gets past 1:
  t1 at 1; val = 0; wait = 0; mutex = 1.
Thread t1 is blocked, but it should not be.
```

3 Implementation 2 (correct)

```
CSem(K) cs {           // counting semaphore initialized to K
   int val ← K;           // the value of csem
   BSem gate(min(1,val));   // 1 if val > 0; 0 if val = 0
   BSem mutex(1);           // protects val

   Pc(cs) {
      P(gate)
  a1: P(mutex);
      val ← val - 1;
      if val > 0
         V(gate);
      V(mutex);
   }

   Vc(cs) {
      P(mutex);
      val ← val + 1;
      if val = 1
         V(gate);
      V(mutex);
   }
}
```

3.1 Criteria for correct implementation

Note that `val` is decremented at the end of `Pc(cs)` and incremented at the end of `Vc(cs)`. Thus `val` always equals the correct value of counting semaphore `cs`. Thus the program implements the counting semaphore if (and only if) the following hold:

A_1 : `val` $\geq$ `0` always holds.

A_2 : If a thread *t* is at `Pc(cs)` and `val` > `0` holds,
then eventually either thread *t* gets past `Pc(cs)` or `val` = `0` holds.

A_1 ensures that a thread gets past `Pc(cs)` only if `val` is higher than zero just before getting past (otherwise, A_1 would not hold just after the thread got past).

A_2 ensures that if threads are waiting on `Pc(cs)` and `val` is positive, one thread will get past. (This is so-called "weak fairness". One can also prove "strong fairness" assuming the same of the binary semaphores.)

3.2 Effective atomicity

When analyzing the above program each of functions `Pc(cs)` and `Vc(cs)` can be treated as atomically executed.

Proof: While a thread is inside `Vc(cs)`, it is not affected by its environment nor does it affect the environment. The former is obvious. The latter is almost obvious.

While a thread *t* is executing a code chunk, the environment learns nothing about the state of its execution. Another thread blocked on `gate` may get past `P(gate)` before thread *t* exits the code chunk (i.e., before it executes `V(mutex)`). But the environment cannot distinguish this from the situation where thread *t* executes `V(mutex)` first (but the news was slow to get to the environment).

The argument for `Pc(cs)` is the same. (Blocking occurs only at the start, before thread *t* gets inside `Pc(cs)`.)

End of proof

3.3 Proof of A_1 and A_2

Exactly one of the following always holds:

B_1 : `(no thread at a1) and (val` $\geq$ `0) and (gate` $=$ `1 iff val > 0)`
B_2 : `(exactly one thread at a1) and (val > 0) and (gate` $=$ `0)`

Proof: It is easy to check that each of functions `Pc(cs)` and `Vc(cs)` preserves `(`B_1 `or` B_2`)`: i.e., establishes `(`B_1 `or` B_2`)` if `(`B_1 `or` B_2`)` held before the step.

`(`B_1 `or` B_2`)` implies A_1. (B_1 implies `val` $\geq$ `0`, and B_1 implies `val > 0`.)

B_1 implies A_2. (If `val` is non-zero then `gate` equals 1, and so any thread at `Pc(cs)` is not blocked.

End of proof