Analysis of a Solution to a Synchronization Problem

In discussion section, I presented the following solution to the following problem.

Problem. A *binary semaphore* is most commonly defined as a semaphore whose integer value can range only between 0 and 1. Implement the usual type of semaphore using binary semaphores.

I gave the following solution:

```
 1   type semaphore = record of
 2      value : integer = 0;   (* value of the usual semaphore *)
 3      bsem : binsemaphore = 0; (* semaphore for block *)
 4      mutex : binsemaphore = 1; (* semaphore for mutual exclusion *)
 5   end;
 6
 7   procedure udown(s: semaphore)
 8   begin
 9      down(s.mutex);
10      if s.value = 0 then begin
11         up(s.mutex);
12         down(s.bsem);
13         down(s.mutex);
14      end;
15      s.value := s.value - 1;
16      up(S.mutex);
17   end;
18
19   procedure uup(s : semaphore)
21   begin
22      down(s.mutex);
23      if s.value = 0 then
24         up(s.bsem);
25      s.value := s.value + 1;
26      up(s.mutex);
27   end;
```

The basic idea of this solution is to synchronize the *uup* and the *down* of the usual semaphores using *bsem.* The field *value* keeps track of the value of the usual semaphore. Because two processes may be calling these functins simultaneously, we need to ensure mutual exclusion; the semaphore field *mutex* does this. Note the *down*(*s.bsem*) is done outside this area of mutual exclusion, to prevent deadlock.

I also encouraged students to try to find problems with all solutions to synchronization problems, including (indeed, especially) with the ones I gave. A student did, and found a problem. The above solution does not work.

Here is a demonstration. Suppose we have 3 processes, *p*, *q*, and *r* sharing a semaphore *s*, initialized to 0.

1. Process *p* calls *udown*(*s*) first and enters the region of mutual exclusion. It releases mutual exclusion at line 11 and blocks at line 12.
2. Process *q* calls *uup*(*s*). At line 24, it increments *s.value*, and process *p* unblocks. At this point, *s.value* is 1.
3. Process *q* does not advance further at this time.
4. Before process *q* can exit the *uup* call, a third process *r* calls *udown*(*s*). It blocks at line 9.
5. Process *q* exits the *uup* call. At this point, *s.value* is 1.
6. Process *r* unblocks, and at line 10, as *s.value* is 1, the process immediately goes to line 15.
7. Process *r* unblocks, and at line 10, as *s.value* is 1, the process immediately goes to line 15. Now, *s.value* becomes 0, and process *r* leaves *udown.*
8. Process *p* now continues. It leaves the function *udown*, resetting *s.value* to -1.

Let us review what happened. The initial value of *s* was 0. One process called *uup* and two called *udown.* If the semaphore were correctly implemented, one process would still be blocked on *udown*. But as the above shows, ***no*** processes are blocked. So the soltion is flawed.

So, how do we do this right? The problem is that the process blocked on *udown* unblocked and ***then*** tried to re-enter the zone of mutual excluson. That cannot happen until the unblocking process leaves *uup*. There is a gap between the leaving of *uup* and the taking of mutual exclusion by the now-unblocked process in *udown*.

So, what we can do is simply not release mutual exclusion at the end of *uup*. Basically, we look at *s.value*. If that is non-zero, no process is blocked on the semaphore, so we release mutual exclusion. If it is zero, a process is blocked on the semaphore, so we release the blocked process. That is the basis for the following solution:

```
type semaphore = record of
   value : integer = 0;   (* value of the usual semaphore *)
   bsem : binsemaphore = 0; (* semaphore for block *)
   mutex : binsemaphore = 1; (* semaphore for mutual exclusion *)
end;

procedure udown(s: semaphore)
begin
   down(s.mutex);
   s.value := s.value - 1;
   if s.value < 0 then begin
      up(s.mutex);
      down(s.bsem);
   end;
   up(S.mutex);
end;

procedure uup(s : semaphore)
begin
   down(s.mutex);
   s.value := s.value + 1;
   if s.value < 0 then
      up(s.bsem);
   else
      up(s.mutex);
end;
```

Note the manipulations of *s.value* moves before the conditional. This prevents two processes from manipulating that field within the zone of mutual exclusion. Also, right after releasing the blocked semaphore, the process in *uup* exits that function.