Further Comments on Implementation of General Semaphores

C. Samuel Hsieh

Department of Computer Science
Vanderbilt University
Nashville, TN 37235, U.S.A.
(615) 343-4404

1. Semantics of Two Previous Implementations

Two different implementations of general semaphores in terms of binary semaphores have been presented in recent issues of *Operating Systems Review* by Hemmendinger[1] and by Kearns [2]. Both implementations use an integer to simulate a general semaphore. The integer and other data shared by the P and V operations are protected in critical sections guarded by a binary semaphore *mutex*. The two implementations differ in the invariants that their executions satisfy. We precisely distinguish the semantics of the two implementations below.

Using Habermann's notation [3], we define the following quantities for formally describing the state of synchronization in a general semaphore:

- C(s): initial value of a general semaphore s.
- nw(s): how many times P(s) was executed.
- ns(s): how many times V(s) was executed.
- np(s): how many times P(s) was passed, i.e., how many times a process was enabled to continue with the instruction following P(s).

Upon every exit from a critical section protected by *mutex*, the state of synchronization in Hemmendinger's construction satisfies the following invariant relation:

(1) $$\mathrm{np}(s) = \min(\,\mathrm{nw}(s), \mathrm{C}(s) + \mathrm{ns}(s)\,)$$

which is exactly the invariant relation that Habermann used to define the effect of executing the primitives *wait* and and *signal* [3]. On the other hand, the invariant satisfied by Kearns' implementation upon every exit from a critical section is

(2) $$\mathrm{np}(s) \le \min(\,\mathrm{nw}(s), \mathrm{C}(s) + \mathrm{ns}(s)\,)$$

which is a somewhat weaker condition than (1). This difference arises from the circumstance described below. When there are processes blocked on s (i.e., np(s) < nw(s)) and a V(s) is executed, Kearns' implementation releases *mutex* before a blocked process is awakened and completes its P operation, thus allowing ns(s) to be incremented (possibly several times) without incrementing np(s); on the other hand, Hemmendinger's implementation does not release *mutex* until a logically blocked process is awakened and completes its *P* operation and, thus, an increment of ns(s) is always followed by an increment in np(s) before other changes can be made to the quantities used in the invariant.

2. An Efficient and Concise Algorithm

Since semaphores were informally described by Dijsktra[4], several versions of *semaphores* have appeared in the literature. Whether (1) corresponds to the semantics of a general semaphore *better* than (2), or vice versa, can be a question subject to much debate. If (2) is an acceptable invariant, the following is an efficient and concise algorithm for implementing a general semaphore in terms of binary semaphores.

```
type
  semaphore = record
              mutex = 1, delay = 0;  (*binary semaphores*)
              n = 0  (*integer to simulate general semaphore*)
              end record;
```

```
Procedure V(var s: semaphore)
begin
PB(s.mutex);
s.n := s.n + 1;
if s.n = 1 then VB(s.delay);
VB(s.mutex)
end;
```

```
Procedure P(s:semaphore);
begin
PB(s.delay);
PB(s.mutex);
s.n := s.n - 1;
if s.n > 0 then VB(s.delay);
VB(s.mutex)
end;
```

Correctness of the algorithm can be established by noting that s.delay is set to 1 (open) when s.n > 0, and is set to 0 (closed) when s.n = 0. Thus, a P operation will block on s.delay until s.n > 0. Like Kearns' algorithm [2], this algorithm satisfies the invariant (2), but uses fewer shared variables, and requires fewer context switches. Executing a P operation can cause up to three context switches in Kearns' algorithm, whereas the above algorithm requires at most two context switches to complete a P operation.

References

1. D. Hemmendinger, "A Correct Implementation of General Semaphores," ACM OSR 22(3) (July 1988) pp. 42-43.
2. P. Kearns, "A Correct and Unrestrictive Implementation of General Semaphores," ACM OSR 22(4) pp. 46-48.
3. A.N. Habermann, "Synchronization of Communicating Processes," Comm. ACM, vol. 15 no.3 (March 1972), pp. 171-176.
4. E.W. Dijkstra, "Cooperating Sequential Processes," in Programming Languages (F. Genuys, ed.), Academic Press, 1968, pp.43-112.