The Well-Tempered Semaphore: Theme With Variations

Kenneth A. Reek
Rochester Institute of Technology
1 Lomb Memorial Drive
Rochester, NY USA 14623-0887
+1-585-475-6155
kar@cs.rit.edu

Abstract

This paper discusses differences in the semantics of various semaphore implementations and their ramifications in developing solutions to synchronization problems. An approach is described to insulate students from these differences and simplify the task of teaching semaphores.

Keywords

Synchronization, semaphore, monitor, operating system

1 Introduction

Process synchronization is one of the more difficult concepts taught in the Operating Systems course. Many students have trouble with it partly because this is their first experience with nonsequential programming.

But there is a more fundamental problem. There are many different definitions of semaphores presented in various operating systems textbooks. If the semaphores provided by your operating system have different semantics than those described in your textbook, students trying to write and debug real synchronization programs are likely to become even more confused.

This paper describes a solution to this problem that uses an instructor-supplied semaphore class built atop the operating system's native synchronization primitives.

2 Semaphores

2.1 First Theme: Dijkstra's Semantics

Dijkstra defined a semaphore as a shared variable that can be manipulated only by the special operations **P** and **V**. In [5], he describes these operations as shown in Figure 1.

```
P( s ):
    s = s - 1
    if s < 0 then
        wait on s
    end

V( s ):
    s = s + 1
    if s <= 0 then
        unblock one process waiting on s
    end
```

Figure 1 — Dijkstra's P and V operations

The important property is the manner in which the **V** operation works. If processes are blocked on the semaphore *s*, a **V** causes one of them to be unblocked. The unblocked process has nothing further to check; it can proceed as soon as it gets the CPU.

If a new process arrives and does a **P** before the unblocked process starts running again, the new process is blocked. Thus, an unblocked process has priority over newly arriving processes. For reasons that will soon become clear, I call these semantics *strong semaphores.*

2.2 Variation: POSIX Semantics

POSIX semaphores are a variation on this theme because they have different semantics. According to [8], the POSIX **P** (called **sem_wait**) and **V** (called **sem_post**) are implemented as shown in Figure 2. If there are sleeping processes **sem_post** awakens one of them. But the awakened process must check the count again—there is nothing to prevent a newly arriving process from doing a **sem_wait** and decrementing the count before the awakened process actually starts running. So unlike Dijkstra's semantics, the unblocked process competes with newly arriving processes.

Stark [13] investigated the semantics of different semaphore implementations to see whether they were vulnerable to starvation. When two processes are competing for a critical resource, Dijkstra's semaphores ensure that both make progress. But with POSIX semaphores, starvation is possible. For this reason, Stark labels the POSIX semantics *weak semaphores.*

```
sem_wait( s ):
    lock mutex
    while count <= 0 do
        unlock mutex
        sleep
        lock mutex
    done
    count = count - 1
    unlock mutex

sem_post( s ):
    lock mutex
    count = count + 1
    unlock mutex
    wakeup one sleeping process
```

Figure 2 — POSIX semaphores

To illustrate this difference, consider the mutual exclusion program in Figure 3. Assume that process *A* is in its critical section and *B* is blocked trying to enter. What happens when *A* leaves its critical section and does a **V**? With Dijkstra's (strong) semantics, *B* is made ready to run; if process *A* reaches its **P** operation before *B* actually gets the CPU, *A* will be blocked.

With POSIX (weak) semantics, process *B* is awakened but must check the semaphore count again. To enter its critical section, *B* must do this before *A* reaches its next **P** operation. On single-CPU systems process *A* continues to run after the **V** until its quantum expires. If the noncritical section is short, it is possible for *A* to repeatedly enter its critical section and starve process *B*.

With two processes, strong semaphores have a clear advantage over weak semaphores. The advantage is less clear with three or more processes, however. As Stark points out, it is possible for any two of them to starve the remaining ones even with strong semaphores.

One potential solution to this problem is to unblock processes according to a fixed ordering such as First Come First Served to ensure "fair" treatment. There are at least two drawbacks with this approach. First, a program that depended on any particular ordering would likely fail if ported to a system whose semaphores had different ordering semantics. Second, an ordering policy would prevent the operating system from unblocking processes according to their priorities.

```
                shared semaphore s = 1;

Process A:                  Process B:
    while true do               while true do
        noncritical section         noncritical section
        P( s )                      P( s )
        critical section            critical section
        V( s )                      V( s )
    end                         end
```

Figure 3 — Starvation problem

In summary, strong semaphores ensure that an unblocked process will be able to continue even if a new process does a **P** before the unblocked process begins to run again. Weak semaphores cannot make this guarantee.

3 Monitors

Monitors are a higher-level synchronization technique built upon semaphores. The semantics of a monitor ensure mutual exclusion and give previously blocked processes priority over newly arriving processes.

3.1 Second Theme: Brinch Hansen's Semantics

Brinch Hansen's monitor semantics [1] specify that a process doing a signal should continue executing; the process unblocked by the signal does not run until the signaller has left or blocked itself. Figure 4 shows the traditional implementation of Brinch Hansen's **wait** and **signal** operations. But they may fail if weak semaphores are used.

Consider the monitor routine in Figure 5 which is used by two processes to force alternation with one another. Each time a process calls *alternate*, it is supposed to signal the condition (unblocking the other process) and then block itself. The process remains blocked until the other process calls *alternate*.

```
wait( x ):
1   x.count = x.count + 1
2   if urgentcount > 0 then
3       urgentcount = urgentcount - 1
4       V( urgent )
5   else
6       V( gate )
7   end
8   P( x.semaphore )
9   P( urgent )

signal( x ):
10  if x.count > 0 then
11      x.count = x.count - 1
12      urgentcount = urgentcount + 1
13      V( x.semaphore )
14  end
```

Figure 4 — Brinch Hansen's wait and signal

```
alternate():
    signal( other_guy )
    wait( other_guy )
```

Figure 5 — Monitor routine to enforce alternation

Here is what may happen when weak semaphores are used. Suppose process *A* has called *alternate* and is waiting on the condition (it will blocked on *other_guy.semaphore* at line 8 in Figure 4). Process *B* now calls *alternate* and signals the condition. The **V** in line 13 increments the count on *other_guy.semaphore*. But if *B* continues executing, as is likely on single-CPU systems, it will then do the wait operation where the **V** in line 4 will increment the count on the *urgent* semaphore.

The counts of both *other_guy.semaphore* and *urgent* have been incremented. Assuming that *A* has not yet begun to run, *B* will breeze right through the **P** operations in lines 8 and 9 without blocking. The result is that *B* continues executing and *A* remains blocked—the reverse should have happened, so the monitor has failed.

With strong semaphores the problem disappears: when process *A* is unblocked from *other_guy.semaphore*, it has precedence over any newly arriving processes (namely, *B* at line 8 in the wait operation). Thus, *B* is blocked and *A* runs as expected.

3.2 Variation: Hoare's Semantics

It is interesting to note that Hoare's variation of monitor semantics [7] are not vulnerable to this failure. When a process does a signal in Hoare's monitor, it is delayed. The process that received the signal runs right away—only when the signalled process has left or blocked itself will the signaller be allowed to continue. This ordering ensures that the signalling process cannot do a wait on the same condition before the signalled process runs.

In summary, monitors using Brinch Hansen's semantics may fail when implemented with weak semaphores. Monitors of both types implemented with strong semaphores will work correctly, and monitors using Hoare's semantics will work correctly with both types of semaphores.

4 Teaching Synchronization

4.1 Main theme: Variations in Textbooks

It is interesting to compare the many variations in the presentation of semaphores found in operating systems textbooks over the years. One irritating difference is the lack of a uniform naming convention for the semaphore operations themselves.

Some authors [4,9,10,11,15] have used Dijkstra's names despite the fact that the mnemonic values of **P** (*proberen*) and **V** (*verhogen*) are lost on non-Dutch speakers.[1] Other authors, for whatever reason, have used a variety of other names: **wait** and **send** [2], **wait** and **signal** [3,12], **up** and **down** [14], **lower** and **raise** [6]. There are probably other variations.

These variations don't affect students directly. But instructors thinking of adopting a new textbook must consider the daunting task of revising their notes, handouts, web pages and other materials to conform to the conventions used in the new text.

More serious is the fact that some texts present weak semaphores semantics [2,9] while some present strong semantics [3,4,6,7,10,12,14,15]. Few texts discuss the differences between the strong and weak semantics. [9] discusses adding a yield to the weak **V** to make it more likely that the unblocked process will get a chance to run. [11] shows both strong and weak types and describes how the busy loop in the weak form is less CPU-efficient than the strong form, but does not discuss the semantic differences between the forms.

Unfortunately, while most texts present strong semantics, many modern operating systems (including POSIX-compliant ones) implement weak semaphores. This poses two problems for instructors wishing to assign their students real synchronization programs. First, the OS semaphore semantics may be different than those presented in the textbook. Second, the interface to the OS semaphores will certainly differ from what the students have seen in the text and in class.

The problem with the interface is easily overcome by writing a wrapper to shield the student from the operating system's interface with its flags and options. If the instructor uses this wrapper to illustrate synchronization problems in class, students should be comfortable using the same interface to write their own programs.

The problem with the semantics is perhaps less obvious but far more important. Students have enough difficulty implementing synchronization programs without the added confusion of the system's semaphores working differently than those they learned in class.

4.2 Variation: Experience Using a Wrapper

I have implemented such a wrapper and have used it in my Operating Systems class for several years with great success. The wrapper uses either POSIX or Solaris condition variables to build counting semaphores with strong semantics and an object-oriented (OO) interface in C++.

In class, I describe both the weak and strong semantics, and then use the wrapper in all subsequent discussions and examples. When they write their synchronization programs, my students use the same wrapper so everything works exactly as they have seen it in class.

Our students are taught OO techniques from their first year, so the choice of an OO interface was obvious. However, this interface provides some additional advantages. First, the class enforces the otherwise informal restrictions that semaphores can be manipulated only by the **P** and **V** operations, and that their internal counts are not accessible. Second, the semaphore constructor *requires* an initializer, reinforcing the idea that semaphores must be given an initial value. Figure 6 shows fragments from a C++ program that uses this semaphore class to provide mutual exclusion.

[1] This problem is easily solved merely by using other words—**P**: permission or pause; **V**: vacate or vamoose.

```
#include "Semaphore.h"

Semaphore mutex( 1 ); // Initialize count to 1

while( true ){
    mutex.P();
    // critical section goes here
    mutex.V();
}
```

Figure 6 — Mutual exclusion using the wrapper

The existence of the semaphore class made it easy to extend the wrapper with classes that implement both Hoare's and Brinch Hansen's monitors. To write a monitor, the student writes a class that inherits from one of these. The inheritance provides the monitor functionality for free, and the student only needs to add the code for the desired synchronization. Figure 7 shows the complete monitor that enforces alternation among two processes.

4.3 Variation: Changing the Ordering

Solutions to synchronization problems should not depend on the ordering (if any) provided by the underlying semaphores. Detecting these dependencies, however, is non-trivial. The final variation implemented in my wrapper is an option to shuffle the order in which processes are unblocked from semaphores.

To test their programs for order dependencies, students simply compile the wrapper with a flag that enables the shuffling code. If their solution no longer works, then there is an ordering dependency in it. While this is not foolproof (the absence of a failure does not guarantee that there are no dependencies), it is certainly a useful tool. I find it especially helpful when testing assignments that students have submitted to me.

```
class TwoProcessAlternate:
    public Gladiator { // Brinch Hansen's semantics
public:
    void alternate();

private:
    Condition other_guy;
};

void TwoProcessAlternate::alternate(){
    enter();  // enter monitor's mutual exclusion
    signal( other_guy ); // let the other guy go
    wait( other_guy );   // now wait for his signal
    leave();  // leave monitor's mutual exclusion
}
```

Figure 7 — Monitor to enforce alternation

5 Coda: Availability of the Software

All of the code described in this paper is freely available from http://www.cs.rit.edu/~kar/papers . In addition, this link also leads to documents that describe the classes, handouts showing solutions for several synchronization problems, and some programming assignments that I have given students in the past.

References

[1] Brinch Hansen, P. The Programming Language Concurrent Pascal. *IEEE Transactions on Software Engineering* Vol. SE-1, No. 6, pp. 199-207, 1975.

[2] Coffman, E.G. Jr, and Denning, P.J. *Operating Systems Theory*. Prentice Hall, Englewood Cliffs, N.J., 1973

[3] Comer, D. *Operating System Design, the XINU Approach* Bell Telephone Laboratories, Inc. 1984

[4] Deitel, H. *An Introduction to Operating Systems, Revised First Edition* Addison Wesley, Reading Mass., 1984

[5] Dijkstra, E.W. The Structure of the THE-Multiprogramming System *Communications of the ACM* (May 1968) 341-346

[6] Galli, D. *Distributed Operating Systems Concepts & Practice* Prentice Hall, Upper Saddle River, N.J., 2000

[7] Hoare, C. A. R. Monitors, An Operating System Structuring Concept. *Communications of the ACM* Vol. 17, No 10, pp 549-557; 1974

[8] Lewis, Bill, and Berg, Daniel *Multithreaded Programming with PThreads* Sun Microsystems Press, 2550 Garcia Avenue, Mountain View, Ca, 1998

[9] Nutt, G. *Operating Systems, A Modern Perspective, 2nd ed.* Addison Wesley Longman, Reading, Mass., 2000

[10] Shay, W.A. *Introduction to Operating Systems* HarperCollins, New York, 1993

[11] Silberschatz, A., Galvin, P., and Gagne, G. *Applied Operating System Concepts* John Wiley & Sons, 605 Third Avenue, NY, 2000

[12] Stallings, W. *Operating Systems, Internals and Design Principles, 3rd ed.* Prentice Hall, Upper Saddle River, N.J., 1998

[13] Stark, E.W. Semaphore Primitives and Starvation-Free Mutual Exclusion *Journal of the ACM* (Oct 1982) 1049-1072

[14] Tanenbaum, A. *Modern Operating Systems, 2nd ed.* Prentice Hall, Upper Saddle River, N.J., 2001

[15] Tsichritzis, D.C., and Bernstein, P.A. *Operating Systems* Academic Press, 111 Fifth Avenue, N.Y., 1974