Further comments on "A correct and unrestrictive implementation of general semaphores"

John A. Trono, William E. Taylor
Computer Science Department
St. Michael's College
1 Winooski Park
Colchester, VT 05439
{jtrono, wtaylor}@smcvt.edu

Abstract

Over a decade ago, a race condition was discovered in a specific implementation of the counting semaphore operations P and V. Several corrections to that implementation were published. These were subsequently critiqued and eventually the discussion ended. This article will expose a newly discovered race condition in one of the corrections.

Introduction

An exercise to implement the traditional counting semaphore operations P and V (wait and signal) appeared in at least one popular operating systems textbook in the 1980s (exercise 9.11) [11]. That exercise specified the use of only binary semaphores, and the concomitant operations PB and VB (binary wait and binary signal), in the solution. A solution for that exercise was listed in the accompanying instructor's manual for that textbook. Leszek Kotulski pointed out that this implementation, which also appeared in [10,12], would not work in all cases, and described the race condition that caused it to fail [8]. Several subsequent issues of the journal where Kotulski's paper appeared proposed other solutions to this problem [4-7], and the discussion continued for nine months after Leszek's article was published. In this paper we will first review the solution that contained the original race condition and what must occur for it to manifest itself, then briefly summarize the discussion that followed, and finally describe the new race condition discovered in one of the modified implementations.

The Original Race Condition

Kotulski illustrated how one published implementation of the P and V operations, that only used binary semaphore operations, was flawed [8]. This flaw can occur because it is possible for multiple, binary signal operations to be executed on an open binary semaphore in this implementation. The final effect is that those binary signals act like no-ops (assigning a value of true to a boolean flag that is already true), or they may produce non-deterministic results, depending upon the behavior of the VB operation on an open binary semaphore. A brief example to show how this race condition occurs is as follows.

Suppose there are four processes, Q_1 - Q_4, that will execute P operations on an initially zero-valued counting semaphore called temp. As each process is selected as the next to run, it continues its program until P(temp) is invoked. As the program counter is set to the statement at line A1 in figure 1, each process is preempted. After Q_1 - Q_4 have all done this, the internal values for temp are: count equaling -4, delay is false and mutex is true. Q_5 next completes a V(temp) operation, setting delay to true and count to -3. If Q_6 - Q_8 all complete a V(temp) operation, count will be zero, but only the first process of Q_1 - Q_4 to regain control of the CPU will proceed. The other three processes will remain blocked indefinitely on delay without any more V(temp) operations completing. This is because each VB(delay) operation executed by Q_6 - Q_8 will not have the desired effect of allowing one process that has invoked P(temp) to continue its execution.

```
type                                        // Pascal style syntax - creating a new type of variable
        semaphore = record                  // to be used as a counting semaphore.
                mutex : binarysemaphore; // internal variable used to ensure mutual exclusion
                delay : binarysemaphore; // internal variable to allow process to sleep/block
                count : integer;            // internal counter to maintain number of blocked processes
        end;

procedure P(s : semaphore);
begin
        PB(s.mutex);
        s.count := s.count – 1;
        if s.count < 0 then
        begin
             VB(s.mutex);
             PB(s.delay); // line A1
        end
        else
             VB(s.mutex);
end;

procedure V(s : semaphore);
begin
        PB(s.mutex);
        s.count := s.count + 1;
        if s.count <= 0 then
             VB(s.delay);
        VB(s.mutex);
end;
```

Figure 1 - "Traditional" implementation of counting semaphores using only binary semaphores.

The First Corrected Implementation

As outlined in the previous section, the updating of the semaphore's internal value for count inside the P (or V) operation may cause a process to block on (or to eventually become "unblocked" from) the internal binary semaphore delay. Hemmendinger [4] points out "The correct algorithm is thus a slight modification of the previous one; it moves the 'else' from P to V." (The second edition of Shaw's text [12] incorporates this change [2].)

This modification changes which process is responsible for freeing up the critical section that protects the internal variable count during those operations. Both of the P and V operations formerly released the internal binary semaphore variable mutex before completing, but Hemmendinger's modification has the V discontinue to do so if any processes are currently waiting on that counting semaphore. This implies that any subsequent processes executing the V

operation on that same counting semaphore will block on the first statement inside the V operation until a process becomes unblocked in the P operation and signals mutex before it leaves that module. (Stallings includes this as a problem in his textbook. Exercise 5.13 [13] provides the traditional counting semaphore implementation, using only binary semaphores. It asks the student to find the flaw in the code, and then to remove it. A hint is given on how to remove it - move one line of code.)

Kearns suggests that even though this simple correction does prevent the race condition from occurring, it can impose a severe scheduling restriction on the processes that invoke these P and V operations [7]. For example, consider K consumer processes that will routinely execute P(buffer), where buffer's internal value of count is initially zero, and L producer processes that will routinely execute V(buffer). (The counting semaphore variable buffer could be representing the number of items created but not taken yet in a producer-consumer type application.) Let us also assume that the K processes have executed P(buffer) and are all currently blocked. One of the L producer processes could then execute V(buffer) and then could continue executing, perhaps even producing another item before a consumer actually acquires the CPU. However, any other producer processes that attempts to invoke the V(buffer) operation will be blocked until one of the K blocked consumer processes awakes and completes its P operation, thereby signaling mutex and allowing any V(buffer) operation to proceed. This is because Hemmendinger's corrected implementation forces a lock step behavior between the processes invoking the P and V operations, once processes start to wait on the internal variable delay.

In the situation just described, if L V operations are invoked, and $L < K$, then all producers and L consumers will eventually complete their respective semaphore operations and continue to execute their individual actions. However, for efficiency reasons, Kearns points out that not allowing more than one of the producers to complete their V operations and continue is very restrictive [7]. If the producer and consumer processes were running on something more powerful than a single CPU, it would seem to be wasteful to prevent each producer from continuing until a matching blocked consumer completely finished executing their original P operation. Because of the definition of the P and V operations, it is understood that if K processes are blocked on a counting semaphore (that was initially zero) as a result of performing a P operation, then they should all become unblocked if and when K V operations have completed. The processes that are performing the V operations shouldn't become blocked during said operations unless the internal value of count is currently being accessed by another process.

The Second Corrected Implementation

Therefore, Kearns added a few lines of code to the "traditional" implementation given in figure 1, creating another improved implementation of these P and V operations. (This appears in figure 3.) Hemmendinger points out that Kearns' modifications could be further improved by an additional line of code in the V operation, and that would also obviate other objections put forth in [5]. This modification would occur at line A4 so that it could be executed conditionally; only if wakecount was equal to 1. Both Hemmendinger [5] and Hsieh [6] include a different solution to this problem that was first described in Barz [1], and is given in figure 2. (Barz also cites [9]

as another source for the "traditional implementation", and Barz outlines a different argument than Kotulski's in his paper concerning the limitations of that code.)

Barz's implementation appears to follow a simpler design because once a process has passed either PB(mutex) statement that process, when executing a P or V operation will: increment or decrement the internal count variable, conditionally execute a VB(delay) operation, and then finally release the critical section that is ensuring that only one process can be updating the value in count. The value of count is not allowed to become smaller than zero because such a process would become blocked on the PB(delay) statement that is executed immediately inside of the P operation. Only processes that can execute the P operation completely, once the count variable is free, are allowed to continue past the PB(delay) statement.

```
type
        semaphore = record                          // assumes initvalue >= 0 for this implementation and is assigned
                mutex = 1 : binarysemaphore;        // when the semaphore variable is created.
                delay = min(1, initvalue) : binarysemaphore; // choose smaller of 1 and the specified initial value.
                count = initvalue : integer;        // start this counting semaphore out at the initial value.
        end;

procedure P(s : semaphore);
begin
        PB(s.delay);
        PB(s.mutex);
        s.count := s.count – 1;
        if s.count > 0 then
            VB(s.delay);
        VB(s.mutex);
end;

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

Figure 2 - Implementation proposed by Barz[1].

Other implementations described in this article have first attempted to maintain a representative value in count of the number of processes waiting. A value of -3 stored in count would imply that three processes have begun to execute the P operation on that counting semaphore and are currently blocked from continuing their execution. The number of processes that would be allowed to complete the P operation is also maintained in the internal state of the counting semaphore. For instance, the internal variable count being equal to +2 implies that the first two processes invoking the P operation would not become blocked inside it, and would therefore continue executing their other statements after completing the P module.

The solution put forth by Kearns (in figure 3) has appeared elsewhere without any previous disclaimer to its correctness [3], and since a race condition in that implementation has been recently uncovered, we will now describe how that situation can arise.

New Race Condition

The flaw in Kearns' [7] solution arises because he allows the potential for too many VB operations to be executed. The problem lies in the fact that the V operation signals delay *and* the P operation may also signal delay, i.e. when wakecount > 0. For example, let us say that there are seven processes, R_1 - R_7, that are suspended on delay at line A2 in the P operation as given in figure 3. Now, suppose four processes, R_8 - R_{11}, begin executing V operations. These four processes execute in such a way that R_8 wakes up one of R_1 - R_7 (let's say R_1), which then resumes execution and is unfortunately preempted on PB(mutex) at line A3 after completing PB(delay). (This could happen if the producers are higher priority processes than the consumers, and individually become ready to run while the PB(delay) invocation is completing.) R_9 - R_{11} affect three more of the consumer processes in the same way (let's say R_2 - R_4). At this point, delay is closed, wakecount is four, and R_1 - R_4 are in the ready state with program counters pointing at line A3. When R_1 - R_4 resume execution of their respective P operations, they will each decrement wakecount, then check to see if wakecount is greater than zero. Since this condition will be true for the first three processes of R_1 - R_4 to continue, delay will be signaled three more times, which would eventually allow the three remaining processes, R_5 - R_7, that are blocked on delay to continue execution even though only four V operations were executed. As with Kotulski's argument, even though this scenario is unlikely, it still provides an example of a race condition that could adversely affect the behavior of any processes relying on the correctness of the code in figure 3.

```
type
        semaphore = record
                mutex = 1 : binarysemaphore;  // initial values were specified for these internal variables.
                delay = 0 : binarysemaphore;
                count = 0 : integer;
                wakecount = 0 : integer;      // variable used to remember to wake up this many blocked processes.
        end;

procedure P(s : semaphore);
begin
        PB(s.mutex);
        s.count := s.count – 1;
        if s.count < 0 then
        begin
            VB(s.mutex);
            PB(s.delay);  // line A2
            PB(s.mutex); // line A3
            s.wakecount := s.wakecount – 1;
            if s.wakecount > 0 then
                VB(s.delay);
        end;
        VB(s.mutex);
end;

procedure V(s : semaphore);
begin
        PB(s.mutex);
        s.count := s.count + 1;
        if s.count <= 0 then
        begin
            s.wakecount := s.wakecount + 1;
            VB(s.delay); // line A4
        end;
        VB(s.mutex);
end;
```

Figure 3 - Implementation by Kearns [7].

This problem in Kearns' [7] solution can be fixed by using the modification suggested in Hemmendinger [5]; changing line A4 to *if s.wakecount = 1 then VB(s.delay).* This change eliminates all but the necessary signals on the binary semaphore, thus alleviating the problem.

Summary

This article outlines how a new race condition could occur in one possible implementation of the counting semaphore operations P and V, given that these operations are built using binary semaphores and the wait and signal operations that are defined on them (PB and VB). The race condition can be eliminated in this implementation by the addition of one line of code. Another simpler, more straightforward solution presented by Barz [1] is also included, and some of its merits are briefly described.

Bibliography

[1] Barz, H. W. Implementing semaphores by binary semaphores. SIGPLAN Notices, volume 18, number 2, (February, 1983), pp 39-45.
[2] Bic, L. and Shaw, A. C. *The Logical Design of Operating Systems.* Prentice-Hall, second edition (1988).
[3] Hartley, S. *Concurrent Programming: The Java Programming Language.* Oxford University Press, (1998), page 114.
[4] Hemmendinger, D. A correct implementation of general semaphores. *Operating Systems Review*, volume 22, number 3, (July, 1988), pp. 42-44.
[5] Hemmendinger, D. Comments on "A correct implementation of general semaphores". *Operating Systems Review*, volume 23, number 1, (January, 1989), pp. 7-8.
[6] Hsieh, C. S. Further comments on implementation of general semaphores. *Operating Systems Review*, volume 23, number 1, (January,1989), pp. 9-10.
[7] Kearns, P. A correct and unrestrictive implementation of general semaphores. *Operating Systems Review*, volume 22, number 4, (October, 1988), pp. 46-48.
[8] Kotulski, L. Comments on implementation of P and V primitives with help of binary semaphores. *Operating System Review*, volume 22, number 2, (April, 1988), pp.53-59.
[9] Lipton, R. J., Snyder, L. and Zalcstein, Y. Evaluation criteria for process synchronization. Sagamore Conference on Parallel Processing, (1975), pp. 245-250.
[10] Perrot, R. H. Concurrent Programming: Microcomputers, in *Microcomputer System Design (Lecture Notes in Computer Science, number 126)*, Springer Verlag, (1982), page 254.
[11] Peterson, J. and Silberschatz, A. *Operating Systems Concepts*, Addison-Wesley, second edition, (1985).
[12] Shaw, A. C. *The Logical Design of Operating Systems.* Prentice Hall, first edition, (1974).
[13] Stallings, W. *Operating Systems: Internals and Design Principles.* Prentice Hall, third edition, (1998).