Design Patterns for Semaphores

Kenneth A. Reek

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

ABSTRACT

This paper describes two design patterns that are powerful tools to help teach how semaphores are used to solve synchronization problems. The patterns are general enough to be used with many different types of problems, yet are simple enough for students to understand and apply effectively.

Categories and Subject Descriptors

D.4.1 [**Process Management**]: Synchronization

General Terms

Algorithms, Design

Keywords

Design pattern, semaphore, synchronization, operating system

1. INTRODUCTION

Synchronization is one of the more difficult concepts taught in the first Operating Systems course. Synchronization among cooperating processes is necessary because accesses to shared resources must be controlled to avoid conflicts. The bulk of students' previous programming experience has been sequential, though, so many of them have a hard time with the idea that multi-process programs can do many things at once and therefore do not fully appreciate the risks of uncontrolled access to resources. To make matters worse, the task of debugging parallel programs by tracing all possible combinations of execution orderings is time consuming, tedious, and error-prone.

Students (such as ours) who have seen Java `synchronized` classes prior to the Operating Systems course have some exposure to parallel execution, yet I find that most of them are still confused by synchronization. Because Java hides the implementation of the synchronization, students who have learned this high-level method but not the low-level synchronization techniques will be unprepared to construct synchronization solutions using other languages.

SIGCSE'04, March 3–7, 2004, Norfolk, Virginia, USA.

Few OS textbooks provide any help for students struggling with all but the simplest synchronization programs. Most ([2,3,6,7,8] are typical, [1] is somewhat better) merely describe semaphore semantics, and explain how to use semaphores to achieve mutual exclusion. They then show solutions to a couple classic problems such as Dining Philosophers or Readers/Writers, but do not present any general strategies for using semaphores.

Design patterns [5] simplify the task of implementing software systems as well as improving their reliability. This power can be applied to semaphores—[4] describes patterns for basic tasks (rendezvous, mutual exclusion), advanced tasks (multiplex, barrier, reusable barrier), and goes on to show how to use them to solve many classic (and some interesting new) synchronization problems.

This paper describes two semaphore design patterns, called **I'll Do It For You** and **Pass the Baton,** that take a different approach than those mentioned above. Rather than implementing a specific tool, these patterns describe general ways to implement solutions. My students have found them useful in designing their own solutions to synchronization problems.

2. SAMPLE PROBLEM

To illustrate the patterns, consider a sharable resource with the following characteristics:

(1) As long as there are fewer than three processes using the resource, new processes can start using it right away.

(2) Once there are three process using the resource, all three must leave before any new processes can begin using it.

The first hurdle for students is to realize that counters are needed to keep track of how many processes are waiting and active, and that these counters are themselves shared resources that must be protected with mutual exclusion. Then a student might create a solution like the one shown in Figure 1.

The solution appears to do everything right: all accesses to the shared variables are protected by mutual exclusion, processes do not block themselves while in the mutual exclusion, new processes are prevented from using the resource if there are (or were) three active users, and the last process to depart unblocks up to three waiting processes. The program is nevertheless incorrect.

There are two problems. First, because unblocked processes must *reenter* the mutual exclusion (line 10) there is a chance that newly arriving processes (at line 5) will beat them into the critical section. Second, there is a time delay between when the waiting processes are unblocked and when they resume execution and update the counters. The waiting processes must be

```
1  Semaphore mutex( 1 ), block( 0 );       // Shared variables: semaphores,
2  int     active( 0 ), waiting( 0 );      //   counters, and
3  bool    must_wait( false );             //   state information
4
5  mutex.P();                              // Enter the mutual exclusion
6  if( must_wait ){                        // If there are (or were) 3, then
7          waiting += 1;                   //   we must wait, but we must leave
8          mutex.V();                      //   the mutual exclusion first!
9          block.P();                      // Wait for all current users to depart
10         mutex.P();                      // Reenter the mutual exclusion
11         waiting -= 1;                   //   and update the waiting count
12 }
13 active += 1;                            // Update active count, and remember
14 must_wait = active == 3;                //   if the count reached 3
15 mutex.V();                              // Leave the mutual exclusion
16
17 // Critical section goes here
18
19 mutex.P();                              // Enter mutual exclusion
20 active -= 1;                            //   and update the active count
21 if( active == 0 ){                      // Last one to leave?
22         int n( min( waiting, 3 ) );     // If so, unblock up to 3
23         while( n > 0 ){                 //  waiting processes
24                 block.V();
25                 n -= 1;
26         }
27         must_wait = false;              // All active processes have left
28 }
29 mutex.V();                              // Leave the mutual exclusion
```

Figure 1 — Incorrect solution

accounted for as soon as they are unblocked (because they might resume execution at any time), but it may be some time before the processes actually do resume and update the counters to reflect this.

To illustrate, consider the case where three processes are blocked at line 9. The last active process will unblock them (lines 23-26) as it departs. But there is no way to predict when these processes will resume executing and update the counters to reflect the fact that they have become active.

If a new process reaches line 6 before the unblocked ones resume, the new one *should* be blocked. But the status variables have not yet been updated so the new process will gain access to the resource. When the unblocked ones eventually resume execution, they will also begin accessing the resource. The solution has failed because it has allowed four processes to access the resource together.

One possible solution is to change the `if` in line 6 to a `while`, forcing unblocked processes to recheck whether they can begin using the resource. But this solution is more prone to starvation because it encourages new arrivals to "cut in line" ahead of those that were already waiting.

3. I'LL DO IT FOR YOU

A better approach is to eliminate the time delay. If the departing process updates the waiting and active counters as it unblocks waiting processes, the counters will accurately reflect the new state of the system before any new processes can get into the mutual exclusion.

This is the **I'll Do It For You** pattern—the departing process updates the system state on behalf of the processes it unblocks. Because the updating is already done, the unblocked processes need not reenter the critical section at all.

Implementing this pattern is easy. Identify all of the work that would have been done by an unblocked process and make the unblocking process do it instead. Figure 2 shows a solution using this pattern.

This solution does not completely prevent newly arriving processes from cutting in line, but does make it less likely. Suppose three processes arrived when the resource was busy, but one of them lost its quantum just before blocking itself at line 5 in Figure 2 (which is unlikely, but certainly possible). When the last active process departs, it will do three `V` operations and set `must_wait` to `true`. If a new process arrives before the older ones resume, the new one will decide to block itself. However, it will breeze past the `P` in line 5 without blocking, and when the process that lost its quantum earlier runs it will block itself instead. This is not an error—the problem doesn't dictate *which* processes access the resource, only *how many* are allowed to access it. Indeed, because the unblocking order of semaphores is implementation dependent, the only portable way to ensure that processes proceed in a particular order is to block each on its own semaphore.

```
1  mutex.P();                              // Enter the mutual exclusion
2  if( must_wait ){                        // If there are (or were) 3, then
3          waiting += 1;                   //   we must wait, but we must leave
4          mutex.V();                      //   the mutual exclusion first!
5          block.P();                      // Wait for all current users to depart
6  } else {
7          active += 1;                    // Update active count, and remember
8          must_wait = active == 3;        //   if the count reached 3
9          mutex.V();                      // Leave the mutual exclusion
10 }
11
12 // Critical section goes here
13
14 mutex.P();                              // Enter mutual exclusion
15 active -= 1;                           //   and update the active count
16 if( active == 0 ){                     // Last one to leave?
17         int n( min( waiting, 3 ) );     // If so, see how many procs to unblock
18         waiting -= n;                   // Deduct this number from waiting count
19         active = n;                     //   and set active to this number
20         while( n > 0 ){                 // Now unblock the processes
21                 block.V();              //   one by one
22                 n -= 1;
23         }
24         must_wait = active == 3;        // Remember if the count is 3.
25 }
26 mutex.V();                              // Leave the mutual exclusion
```

Figure 2 — I'll Do It For You

```
1  mutex.P();                              // Enter the mutual exclusion
2  if( must_wait ){                        // If there are (or were) 3, then
3          waiting += 1;                   //   we must wait, but we must leave
4          mutex.V();                      //   the mutual exclusion first!
5          block.P();                      // Wait for all current users to depart
6          waiting -= 1;                   // We've got the mutual exclusion: update count
7  }
8  active += 1;                            // Update active count, and remember
9  must_wait = active == 3;                //   if the count reached 3
10 if( waiting > 0 && !must_wait ){         // If there are others waiting and we don't
11         block.V();                      //   yet have 3 active, unblock one,
12 } else {                                // otherwise
13         mutex.V();                      //   open the mutual exclusion
14 }
15
16 // Critical section
17
18 mutex.P();                              // Enter the mutual exclusion
19 active -= 1;                           //   and update the active count
20 if( active == 0 ){                     // Last one to leave?
21         must_wait = false;              // Set up to let new processes enter
22 }
23 if( waiting > 0 && !must_wait ){        // If there are others waiting and we don't
24         block.V();                      //   have 3 active, unblock a waiting process
25 } else {                                // otherwise
26         mutex.V();                      //   open the mutual exclusion
27 }
```

Figure 3 — Pass the Baton

One drawback to this pattern is that the cohesion is not as good as before. It is not immediately obvious why a *departing* process should decrement the waiting count (line 18, Figure 2) or increase the active count (line 19).

Another potential drawback to this pattern is that newly unblocked processes cannot access the shared variables after they resume because they are not in the mutual exclusion. One classic textbook [9] describes a problem where each process is identified by a unique integer and is allowed to access a resource only if the sum of the integers of the processes already using it is a multiple of three. After resuming, an unblocked process needs mutual exclusion to safely add its private value to the shared sum, but with this pattern the process doesn't have it.

4. PASS THE BATON

This pattern synchronizes processes like runners in a relay race. As each runner finishes her laps, she passes the baton to the next runner. "Having the baton" is like having permission to be on the track.

In the synchronization world, being in the mutual exclusion is analogous to having the baton—only one person can have it. After you unblock a waiting process, you "pass the baton" to it by simply leaving (or blocking yourself) without opening the mutual exclusion. The unblocked process doesn't reenter the mutual exclusion—it takes over your ownership of it. The process can therefore safely update the system state on its own. When it is finished, it reopens the mutual exclusion. Newly arriving processes can no longer cut in line because they cannot enter the mutual exclusion until the *unblocked* process has finished. This pattern is illustrated in Figure 3.

Because the unblocked process takes care of its own updating, the cohesion of this solution is better. However, once you have unblocked a process, you must immediately stop accessing the variables protected by the mutual exclusion. The safest approach is to immediately leave (after line 24, the process leaves without opening the mutex) or block yourself.

A drawback of this pattern is that you can only pass the baton to one process at a time. In this example, only one waiting process can be unblocked even if several are waiting—to unblock more would violate the mutual exclusion of the status variables. This problem is solved by having the newly unblocked process check whether more processes should be unblocked (line 10, Figure 3). If so, it passes the baton to one of them (line 11); if not, it opens up the mutual exclusion for new arrivals (line 13).

5. SUMMARY

Here are concise descriptions of these patterns. Note that their simplicity makes them easy to teach and to apply.

I'll Do It For You

- The unblocked processes do not reenter the mutual exclusion.
- The unblocking process updates the system state on behalf of the processes that it unblocks.

Pass the Baton

- When a waiting process is unblocked, the unblocking process leaves (or blocks itself) without opening the mutual exclusion.
- When an unblocked process resumes, it is already in the mutual exclusion and can update the system state on its own behalf.
- Before opening the mutual exclusion, the unblocked process may have to check whether another process should be unblocked instead.

Figure 4 summarizes the different characteristics of both design patterns, and helps students select the one that is most appropriate for a particular problem.

6. CONCLUSIONS

I have been describing these techniques informally in my Operating Systems class for several years, and more formally for the last two years. Despite being simple, the patterns have proven to be very useful in helping my students to understand semaphore programming.

Like other design patterns, these rules describe how to construct solutions. By using them, students can use low-level techniques without having to do as much low-level analysis, which reduces the likelihood of errors.

7. AVAILABILITY

A class handout describing these patterns in more detail is available from http://www.cs.rit.edu/~kar/papers . This link also leads to documents that describe a C++ Semaphore class, handouts showing how these patterns can be used in solutions for several synchronization problems, and some programming assignments that I have given students in the past.

I'll Do It For You	*Pass the Baton*
Always opens the mutual exclusion before exiting	Unblock one of these waiting processes, OR unblock one of those waiting processes, OR open the mutual exclusion
Can unblock any number of processes	Can only unblock one process at a time
Unblocking process remains in the mutual exclusion	Unblocking process must cease accessing shared variables as soon as another process is unblocked
Unblocked processes cannot access shared variables	Unblocked processes can access shared variables
Cohesion suffers	Cohesion is better

Figure 4 — Comparison of Design Patterns

8. REFERENCES

[1] Bacon, J. and Harris, T. *Operating Systems, Concurrent and Distributed Software Design* Addison Wesley, Reading, Mass. 2003

[2] Bic, L. and Shaw, A. *Operating Systems Principles* Prentice Hall, Upper Saddle River, NJ, 2003

[3] Chow, R, and Johnson, T. *Distributed Operating Systems & Algorithms* Addison Wesley, Reading Mass., 1997

[4] Downey, Allen B. The Little Book of Semaphores *http://allendowney.com/semaphores* 2003

[5] Gamma, E., et. al. *Design Patterns* Addison-Wesley, Reading, Mass. 1985

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

[7] Silberschatz, A. et. al. *Operating Systems Concepts, 6e* John Wiley & Sons, New York, NY, 2003

[8] Stallings, W. *Operating Systems, Internals and Design Principles* Prentice-Hall, Upper Saddle River, NJ, 1998

[9] Tsichritzis, D., and Bernstein, P. *Operating Systems* Academic Press, New York, NY, 1974