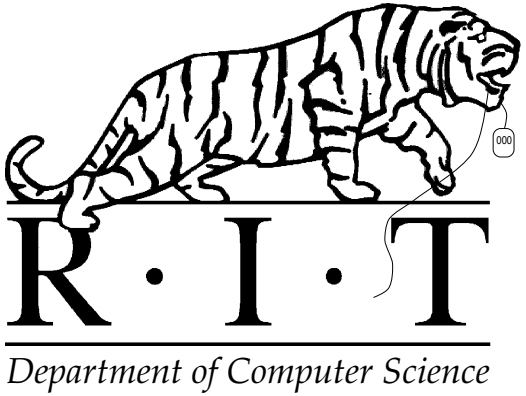


Design Patterns for Semaphores

Kenneth A. Reek, Professor
Department of Computer Science
102 Lomb Memorial Drive
Rochester Institute of Technology
Rochester, NY 14623

kar@cs.rit.edu

Introduction

Synchronization is difficult!

- Synchronization using low-level techniques is more difficult
- But high-level techniques may not always be available
- Teaching high-level techniques does not convey the underlying concepts as well

Most textbooks describe semaphores

- Illustrate simple mutual exclusion
- Show solutions for a couple of classic problems, possibly including some “tricks”
- Do not describe general approaches

Readers/Writers

Rules:

- Any number of readers can access the file simultaneously if there are no writers
- Each writer must have exclusive access to the file

Rules to avoid indefinite postponement:

- When one or more writers is waiting to access the file, newly arriving readers are blocked
- When an active writer departs, waiting readers are unblocked in preference to unblocking another writer

System State

The state of this system includes four counts:

- Number of readers active and blocked
- Number of writers active and blocked

Might be many processes testing and changing state

- The state itself is a critical resource
- Accesses to it must be protected with mutual exclusion
- But you must not block yourself inside of the mutual exclusion!

Incorrect Solution

```
// READERS
mutex.P();
if( w_active > 0 || w_waiting > 0 ){
        r_waiting += 1;
        mutex.V();
        reader.P();
        mutex.P();
        r_waiting –= 1;
}
r_active += 1;
mutex.V();
// ... read the file ...
mutex.P();
r_active –= 1;
if( r_active == 0 && w_waiting > 0 )
        writer.V();
mutex.V();
```

```
// WRITERS
mutex.P();
if( r_active > 0 || w_active > 0 ){
        w_waiting += 1;
        mutex.V();
        writer.P();
        mutex.P();
        w_waiting –= 1;
}
w_active += 1;
mutex.V();
// ... write the file ...
mutex.P();
w_active –= 1;
if( r_waiting > 0 )
        for( int i = 0; i < r_waiting; i += 1 )
                reader.V();
else if( w_waiting > 0 )
        writer.V();
mutex.V();
```

Incorrect Solution: What's Wrong

The problem is that unblocked processes must *reenter* the mutual exclusion

- Newly arriving processes might beat them to it
- When unblocked processes finally run, the system state may have changed—it may no longer be legal for them to run

Incorrect Solution: Example

A writer is accessing the file and several readers are blocked

r_active: 0 r_waiting: 3 w_active: 1 w_waiting: 0

```
mutex.P();
if( w_active > 0 || w_waiting > 0 ){
        r_waiting += 1;
        mutex.V();
        reader.P(); ← 3 blocked
        mutex.P();
        r_waiting –= 1;
}
r_active += 1;
mutex.V();
// ... read the file ...
```

```
// ... write the file ...
mutex.P();
w_active –= 1;
if( r_waiting > 0 )
        for( int i = 0; i < r_waiting; i += 1 )
                reader.V();
else if( w_waiting > 0 )
        writer.V();
mutex.V();
```

1) The writer leaves, unblocking all the blocked readers

r_active: 0 r_waiting: 3 w_active: 0 w_waiting: 0

Incorrect Solution: Example (continued)

2) Before the readers get a quantum, a new writer arrives and "cuts in line"

r_active: 0 r_waiting: 3 w_active: 0 w_waiting: 0

```
mutex.P();
if( w_active > 0 || w_waiting > 0 ){
      r_waiting += 1;
      mutex.V();
      reader.P();   ← 3 unblocked
      mutex.P();
      r_waiting –= 1;
}
r_active += 1;
mutex.V();
// ... read the file ...
```

```
// WRITERS
mutex.P();
if( r_active > 0 || w_active > 0 ){
      w_waiting += 1;
      mutex.V();
      writer.P();
      mutex.P();
      w_waiting –= 1;
}
w_active += 1;
mutex.V();
// ... write the file ...
```

r_active: 0 r_waiting: 3 w_active: 1 w_waiting: 0

Incorrect Solution: Example (continued)

3) The unblocked readers, one by one, resume and begin reading

r_active: 0 r_waiting: 3 w_active: 1 w_waiting: 0

```
// READERS
mutex.P();
if( w_active > 0 || w_waiting > 0 ){
        r_waiting += 1;
        mutex.V();
        reader.P();
        mutex.P();
        r_waiting –= 1;
}
r_active += 1;
mutex.V();
// ... read the file ...
```

← 3 unblocked processes resume

r_active: 3 r_waiting: 0 w_active: 1 w_waiting: 0

Incorrect Solution: Explanation

The problem is the delay between *unblocking* the processes and *updating the system state* to reflect the unblocking

- The unblocked processes might resume at any time
- You must consider them "running" *as soon as they are unblocked*
- Cannot depend on the unblocked processes themselves to update the system state

A Possible Fix

Change the “if” to a “while”

```
// READERS
mutex.P();
while( w_active > 0 || w_waiting > 0 ){
        r_waiting += 1;
        mutex.V();
        reader.P();
        mutex.P();
        r_waiting –= 1;
}
r_active += 1;
mutex.V();
// ... read the file ...
```

- Unfair to waiting processes
- Exacerbates indefinite postponement

A Better Fix

Update the system state when the unblocking is done

- The unblocking process does the updating on behalf of the unblocked process
- This is the "I'll Do It For You" pattern

Implementing "I'll Do It For You"

1) Identify the statements executed by unblocked processes
2) Change the code so that unblocked processes do not execute these statements

```
// READERS
mutex.P();
if( w_active > 0 || w_waiting > 0 ){
        r_waiting += 1;
        mutex.V();
        reader.P();
        mutex.P();
        r_waiting –= 1;
}
r_active += 1;
mutex.V();
// ... read the file ...
```

```
// READERS
mutex.P();
if( w_active > 0 || w_waiting > 0 ){
        r_waiting += 1;
        mutex.V();
        reader.P();
} else {
        r_active += 1;
        mutex.V();
}
```

Implementing "I'll Do It For You" (continued)

3) Put the same functionality into the unblocking code

```
// ... write the file ...
mutex.P();
w_active –= 1;
if( r_waiting > 0 )
        for( int i = 0; i < r_waiting; i += 1 )
                reader.V();
else if( w_waiting > 0 )
…
```

```
// ... write the file ...
mutex.P();
w_active –= 1;
if( r_waiting > 0 ){
        r_active = r_waiting;
        while( r_waiting > 0 ){
                r_waiting –= 1;
                reader.V();
        }
} else if( w_waiting > 0 )
…
```

The unblocked processes do not reenter the mutual exclusion at all

The "I'll Do It For You" Solution

```
// READERS
mutex.P();
if( w_active > 0 || w_waiting > 0 ){
        r_waiting += 1;
        mutex.V();
        reader.P();
} else {
        r_active += 1;
        mutex.V();
}
// ... read the file ...
mutex.P();
r_active –= 1;
if( r_active == 0 && w_waiting > 0 ){
        w_waiting –= 1;
        w_active += 1;
        writer.V();
}
mutex.V();
```

```
// WRITERS
mutex.P();
if( r_active > 0 || w_active > 0 ){
        w_waiting += 1;
        mutex.V();
        writer.P();
} else {
        w_active += 1;
        mutex.V();
}
// ... write the file ...
mutex.P();
w_active –= 1;
if( r_waiting > 0 ){
        r_active = r_waiting;
        while( r_waiting > 0 ){
                r_waiting –= 1;
                reader.V();
        }
} else if( w_waiting > 0 ){
        w_waiting –= 1;
        w_active += 1;
        writer.V();
}
mutex.V();
```

I'll Do It For You: Characteristics

The unblocked processes do not reenter the mutual exclusion

- Unblocked process cannot access any state variables
- You can unblock any number of processes
- Cohesion suffers

I'll Do It For You: Characteristics (continued)

- Still vulnerable to "cutting in line," but much less likely

1) One process is reading the file, and a writer arrives

r_active: 1 r_waiting: 0 w_active: 0 w_waiting: 0

```
// WRITERS
mutex.P();
if( r_active > 0 || w_active > 0 ){
        w_waiting += 1;
        mutex.V();
        writer.P();
} else {
        w_active += 1;
        mutex.V();
}
// ... write the file ...
```

← process loses its quantum here (very unlikely, but possible)

r_active: 1 r_waiting: 0 w_active: 0 w_waiting: 1

I'll Do It For You: Characteristics (continued)

2) The reader leaves and tries to unblock the writer

r_active: 1 r_waiting: 0 w_active: 0 w_waiting: 1

```
// WRITERS
mutex.P();
if( r_active > 0 || w_active > 0 ){
        w_waiting += 1;
        mutex.V();
        writer.P();
} else {
        w_active += 1;
        mutex.V();
}
// ... write the file ...
```

← process paused here

```
// ... read the file ...
mutex.P();
r_active –= 1;
if( r_active == 0 && w_waiting > 0 ){
        w_waiting –= 1;
        w_active += 1;
        writer.V();
}
mutex.V();
```

r_active: 0 r_waiting: 0 w_active: 1 w_waiting: 0

I'll Do It For You: Characteristics (continued)

3) A new writer arrives before the original one resumes

r_active: 0 r_waiting: 0 w_active: 1 w_waiting: 0

```
// WRITERS
mutex.P();
if( r_active > 0 || w_active > 0 ){
        w_waiting += 1;
        mutex.V();
        writer.P();
} else {
        w_active += 1;
        mutex.V();
}
// ... write the file ...
```

← process paused here

r_active: 0 r_waiting: 0 w_active: 1 w_waiting: 1

Only way to guarantee ordering is to block each process on a separate semaphore

Alternate Approach

The original problem is caused by having to *reenter* the mutual exclusion

- The unblocking process leaves without opening the mutual exclusion
- The unblocked process takes over the mutual exclusion
- This is the Pass the Baton pattern

Like runners in a relay race passing the baton to one another

- Occupancy of the mutual exclusion is our “baton”

Implementing "Pass the Baton"

1) Find where unblocked processes reenter the mutual exclusion
2) Eliminate these P operations

```
// WRITERS
mutex.P();
if( r_active > 0 || w_active > 0 ){
        w_waiting += 1;
        mutex.V();
        writer.P();
        mutex.P();
        w_waiting –= 1;
}
w_active += 1;
mutex.V();
// ... write the file ...
```

```
// WRITERS
mutex.P();
if( r_active > 0 || w_active > 0 ){
        w_waiting += 1;
        mutex.V();
        writer.P();
        w_waiting –= 1;
}
w_active += 1;
mutex.V();
// ... write the file ...
```

Implementing "Pass the Baton" (continued)

3) When unblocking a process, do not leave the mutual exclusion

```
mutex.P();
r_active –= 1;
if( r_active == 0 && w_waiting > 0 )
        writer.V();
mutex.V();
```

```
mutex.P();
r_active –= 1;
if( r_active == 0 && w_waiting > 0 )
        writer.V();
else
        mutex.V();
```

Implementing "Pass the Baton" (continued)

4) You can only unblock one process at a time

```
// ... write the file ...
mutex.P();
w_active –= 1;
if( r_waiting > 0 )
        for( int i = 0; i < r_waiting; i += 1 )
                reader.V();
else if( w_waiting > 0 )
        writer.V();
mutex.V();
```

```
// ... write the file ...
mutex.P();
w_active –= 1;
if( r_waiting > 0 )
        reader.V();
else if( w_waiting > 0 )
        writer.V();
else
        mutex.V();
```

Implementing "Pass the Baton" (continued)

5) When the first unblocked process finishes, have it pass the baton to the next one

```
// READERS
mutex.P();
if( w_active > 0 || w_waiting > 0 ){
        r_waiting += 1;
        mutex.V();
        reader.P();
        mutex.P();
        r_waiting –= 1;
}
r_active += 1;
mutex.V();
// ... read the file ...
```

```
// READERS
mutex.P();
if( w_active > 0 || w_waiting > 0 ){
        r_waiting += 1;
        mutex.V();
        reader.P();
        r_waiting –= 1;
}
r_active += 1;
if( r_waiting > 0 )
        reader.V();
else
        mutex.V();
// ... read the file ...
```

The "Pass the Baton" Solution

```
// READERS
mutex.P();
if( w_active > 0 || w_waiting > 0 ){
        r_waiting += 1;
        mutex.V();
        reader.P();
        r_waiting –= 1;
}
r_active += 1;
if( r_waiting > 0 )
        reader.V();
else
        mutex.V();
// ... read the file ...
mutex.P();
r_active –= 1;
if( r_active == 0 && w_waiting > 0 )
        writer.V();
else
        mutex.V();
```

```
// WRITERS
mutex.P();
if( r_active > 0 || w_active > 0 ){
        w_waiting += 1;
        mutex.V();
        writer.P();
        w_waiting –= 1;
}
w_active += 1;
mutex.V();
// ... write the file ...
mutex.P();
w_active –= 1;
if( r_waiting > 0 )
        reader.V();
else if( w_waiting > 0 )
        writer.V();
else
        mutex.V();
```

Pass the Baton: Characteristics

The unblocked processes take over ownership of the mutual exclusion from the unblocking processes

- The unblocked process can access the system state
- The unblocking process must no longer access system state
- Can only unblock one process at a time
- Cohesion is better
- Looks like the P's and V's are not balanced, but they are

Pass the Baton: Characteristics (continued)

Completely invulnerable to “cutting in line”

1) One process is reading the file, and a writer arrives

r_active: 1 r_waiting: 0 w_active: 0 w_waiting: 0

```
// WRITERS
mutex.P();
if( r_active > 0 || w_active > 0 ){
        w_waiting += 1;
        mutex.V();
        writer.P();
        w_waiting –= 1;
}
w_active += 1;
mutex.V();
// ... write the file ...
```

← process loses its quantum here (very unlikely, but possible)

r_active: 1 r_waiting: 0 w_active: 0 w_waiting: 1

Pass the Baton: Characteristics (continued)

2) The reader leaves and tries to unblock the writer

r_active: 1 r_waiting: 0 w_active: 0 w_waiting: 1

```
// WRITERS
mutex.P();
if( r_active > 0 || w_active > 0 ){
        w_waiting += 1;
        mutex.V();
        writer.P();
        w_waiting –= 1;
}
w_active += 1;
mutex.V();
// ... write the file ...
```

← process paused here

```
// ... read the file ...
mutex.P();
r_active –= 1;
if( r_active == 0 && w_waiting > 0 )
        writer.V();
else
        mutex.V();
```

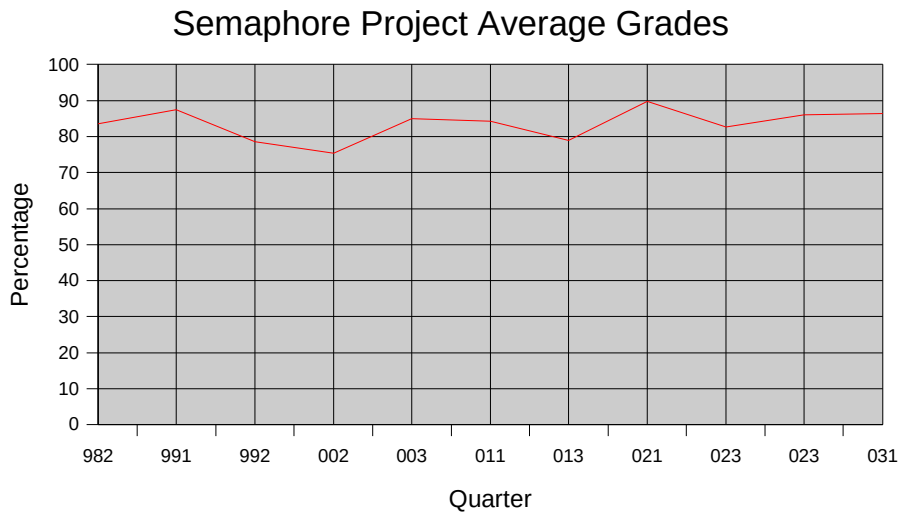
r_active: 0 r_waiting: 0 w_active: 0 w_waiting: 1

3) Newly arriving processes cannot cut in line because they cannot enter the mutual exclusion

Experience

I have used these patterns in my Operating Systems class for several years, and began to describe them more formally for almost two years.

- Students seem to be able to use the patterns easily
- No significant change in grades



Availability

`http://www.cs.rit.edu/~kar/papers` has links to:

- Class handouts describing the patterns and showing their use in several classic synchronization problems
- Descriptions of several problems I have assigned in the past
- More

Thank you!