3. Semaphores and Locks

Semaphores are used to provide mutual exclusion and condition synchronization. Locks provide mutual exclusion and have special properties that make them useful in object-oriented programs.

3.1 Counting Semaphores

A counting semaphore is a synchronization object that is initialized with an integer value and then accessed through two operations, which are named *P* and *V* (or *down* and *up*, *decrement* and *increment*, *wait* and *signal*).

```
class countingSemaphore {
   public countingSemaphore(int initialPermits) {permits = initialPermits;}
   public void P() {…};
   public void V() {…};
   private int permits;
}
```

When defining the behavior of methods *P*() and *V*(), it is helpful to interpret a counting semaphore as having a pool of permits.

- A thread calls method *P*() to request a permit. If the pool is empty, the thread waits until a permit becomes available.
- A thread calls method *V*() to return a permit to the pool.

A counting semaphore *s* is declared and initialized using

```
countingSemaphore s(1);
```

The initial value, in this case 1, represents the initial number of permits in the pool.

A sketch of a possible implementation of methods P() and V().

```
public void P( ) {
if (permits > 0)    // permits holds the current number of permits in the pool:
     --permits;   // take a permit from the pool
   else             // the pool is empty so wait for a permit
      wait until permits becomes positive and then decrement permits by one.
}


public void V( ) {
   ++permits;     // return a permit to the pool
}
```

There may be many threads waiting in *P()* for a permit. The waiting thread that gets a permit as the result of a *V()* operation is not necessarily the thread that has been waiting the longest.

The invariant for semaphore *s*:

(the initial number of permits) + (the number of completed *s.V()* operations)

≥ (the number of *completed s.P()* operations).

The number of completed *P()* operations may be less than the number of started *P()* operations.

There are no methods for accessing the value of a semaphore. We rely on a semaphore's invariant to define its behavior.

3.2 Using Semaphores

There are some common idioms or mini-patterns in the way semaphores are used to solve problems. Being able to recognize and apply these patterns is a first step towards understanding and writing semaphore-based programs.

3.2.1 Resource Allocation

Problem: Three threads are contending for two resources. If neither resource is available, a thread must wait until one of the resources is released.

Solution:

```
              countingSemaphore s(2); // two resources are available initially
   Thread 1                 Thread 2                 Thread 3
   s.P();                   s.P();                   s.P();
   /* use the resource */   /* use the resource */   /* use the resource */
   s.V();                   s.V();                   s.V();
```

Listing 3.1 Resource allocation using semaphores.

The invariant for semaphore s and the placement of the $P()$ and $V()$ operations guarantees that there can be no more than two consecutive completed $s.P()$ operations without an intervening $s.V()$ operation.

The pool of permits represented by semaphore s maps directly to the managed pool of resources. Methods $P()$ and $V()$ do all of the necessary bookkeeping internally - counting resources, checking the number of available resources, and blocking threads when no resources are available.

3.2.2 More Semaphore Patterns

Listing 3.2 shows an alternate solution to the resource allocation problem.

```
// variables shared by the threads

int count = 2;                       // number of available resources
int waiting = 0;                     // number of waiting threads
// provides mutual exclusion for count and waiting
countingSemaphore mutex = new countingSemaphore(1);
// used as a queue of blocked threads
countingSemaphore resourceAvailable = new countingSemaphore(0);
Thread i { // each thread executes the following code:
mutex.P();                           // enter the critical section
if (count>0) {                       // is a resource available?
   count--;                          // one less resource available
   mutex.V();                        // exit the critical section
}
else {
   waiting++;                        // one more waiting thread
   mutex.V();                        // exit the critical section
   resourceAvailable.P();            // wait for a resource
}
/* use the resource */
mutex.P();                           // enter the critical section
if (waiting>0) {                     // are there waiting threads?
   --waiting;                        // one less waiting thread
   resourceAvailable.V();            // notify a waiting thread
}
else count++;                        // return a resource to the pool
mutex.V();
}
```
Listing 3.2 An alternate solution to the resource allocation problem.

- Variable *count* tracks the number of available resources.

- Shared variable *waiting* tracks the number of waiting threads.

- Notice that *count* is not incremented when a waiting thread is notified, just as *count* is

  not decremented after a waiting thread receives a resource.

This solution contains several patterns.

Mutex: A semaphore, typically named *mutex* (for "<u>mut</u>ual <u>ex</u>clusion") is initialized to one. A critical section begins with a call to *mutex.P()* and ends with a call to *mutex.V()*:

```
mutex.P()
/* critical section */
mutex.V()
```

The semaphore invariant ensures that the completion of *P()* and *V()* operations alternates, which allows one thread at a time to be inside the critical section.

Enter-and-Test: Often a thread will enter a critical section and then test a condition that involves shared variables.

```
mutex.P();
if (count>0) {        // test for available resources; count is a shared variable
  …;                  // count>0 means that resource is available
}
else {
…;
resourceAvailable.P(); // wait for a resource
}
```

One of the alternatives of the if-statement will contain a *P()* operation so that the thread can block itself until it is notified that the condition is satisfied.

Exit-before-Wait: A thread executing inside a critical section will exit the critical section before blocking itself on a *P()* operation.

```
mutex.V();             // exit critical section
resourceAvailable.P(); // wait for a resource
```

If a thread did not call *mutex.V()* to leave the critical section before it called *resourceAvailable.P()*, then no other threads would be able to enter the critical section and no calls to *resourceAvailable.V()* would ever occur.

Condition Queue: A semaphore can be used as a queue of threads that are waiting for a condition to become true. If the initial value of a semaphore *s* is 0, and the number of started *s.P()* operations is never less than the number of completed *s.V()* operations, then the semaphore invariant ensures that every *s.P()* operation is guaranteed to block the calling thread.

```
if (waiting>0)        // if one or more threads are waiting
    resourceAvailable.V(); // then notify a blocked thread
```

A call to *resourceAvailable.P()* will always block the calling thread and a call to *resourceAvailable.V()* will always unblock a waiting thread.

3.3 Binary Semaphores and Locks

A binary semaphore must be initialized with the value 1 or the value 0 and the completion of *P()* and *V()* operations must alternate. (Note that *P()* and *V()* operations can be started in any order, but their completions must alternate.)

▪ If semaphore's initial value is 1, the first completed operation must be *P()*.

▪ If semaphore's initial value is 0, the first completed operation must be V().

Thus, the *P()* and *V()* operations of a binary semaphore may block the calling threads.

```
binarySemaphore mutex = new binarySemaphore(1);
Thread 1          Thread 2
mutex.P();        mutex.P();
/* critical section *//* critical section */
mutex.V();        mutex.V();
```

A "mutex lock" (or "lock"), can also be used to solve the critical section problem. (Locks can provide mutual exclusion but not condition synchronization.)

```
mutexLock mutex;
```

Thread 1	Thread 2
mutex.lock();	mutex.lock();
/* critical section */	/* critical section */
mutex.unlock();	mutex.unlock();

Unlike a semaphore, a lock has an owner, and ownership plays an important role in the behavior of a lock:

- A thread requests ownership of lock L by calling *L.lock()*.

- A thread that calls *L.lock()* becomes the owner if no other thread owns the lock; otherwise the thread is blocked.

- A thread releases its ownership of L by calling *L.unlock()*. If the thread does not own L, the call to *L.unlock()* generates an error.

- A thread that already owns lock L and calls *L.lock()* again is not blocked. In fact, it is common for a thread to request and receive ownership of a lock that it already owns. But the owning thread must call *L.unlock()* the same number of times that it called *L.lock()* before another thread can become L's owner.

- A lock that allows its owning thread to lock it again is called a *recursive* lock.

Locks are commonly used in the methods of classes.

```
class lockableObject {
  public void F() {
    mutex.lock();
    ...;
    mutex.unlock();
  }
  public void G() {
    mutex.lock();
    ...; F(); ...;                    // method G() calls method F()
    mutex.unlock();
  }
  private mutexLock mutex;
}
```

Methods *F()* and *G()* are critical sections.

When a thread calls method *G()*, the *mutex* is locked. When method *G()* calls method *F()*, *mutex.lock()* is executed in *F()*, but the calling thread is not blocked since it already owns *mutex*.

Differences between locks and binary semaphores:

- For a binary semaphore, if two calls are made to *P*() without any intervening call to *V*(), the second call will block. But a thread that owns a lock and requests ownership again is not blocked. (Beware of the fact that locks are not always recursive, so check the documentation before using a lock.)

- The owner for successive calls to *lock()* and *unlock()* must be the same thread. But successive calls to *P*() and *V*() can be made by different threads.

3.4 Implementing Semaphores

Semaphores can be implemented at the user level, the operating system level, or with hardware support.

3.4.1 Implementing P() and V()

Implementation 1 in Listing 3.3 uses busy waiting to delay threads. It's possible that an awakened thread will always find that the value of *permits* is zero and will thus never be allowed to complete its *P()* operation.

Semaphores with this type of implementation are called *weak semaphores*.

Implementation 2 blocks waiting threads in a (operating system) queue until they are notified. When a thread blocked in *P()* is awakened by a *V()* operation, the thread is allowed to complete its *P()* operation. This is a s*trong semaphore*.

```
// Implementation 1 uses semi-busy-waiting.
P():    while (permits == 0) {
            Sleep(..);      // voluntarily relinquish CPU
        }
        permits = permits - 1;
V():    permits = permits + 1;
```

```
// Implementation 2 uses a queue of blocked threads; permits may be negative.
P():    permits = permits - 1;
        if (permits < 0) wait on a queue of blocked threads until notified;
V():    permits = permits + 1;
        if (permits <= 0) notify one waiting thread;
```

Listing 3.3 Implementations of *P()* and *V()* for counting semaphores.

Implementation 3 in Listing 3.4 is a (strong) binary semaphore.

According to this implementation, the invariant for a binary semaphore s is

((the initial value of s (which is 0 or 1)) +
    (the number of completed $s.V()$ operations)
       - (the number of completed $s.P()$ operations)) = 0 or 1.

```
// Implementation 3 uses two queues of blocked threads. V() may block caller.
P():     if (permits == 0)
              wait in a queue of blocked threads until notified;
         permits = 0;
         if (queue of threads blocked in V() is not empty)
            notify one blocked thread in V();

V():     if (permits == 1)
              wait in a queue of blocked threads until notified;
         permits = 1;
         if (queue of threads blocked in P() is not empty)
            notify one blocked thread in P();
```
Listing 3.4 Implementation of *P( )* and *V( )* for binary semaphores.

Since methods *P()* and *V()* access shared variables (e.g., permits), they must be critical sections. Here's an implementation using the techniques in Chapter 2:

```
P(): entry-section;                          V(): entry-section;
     while (permits == 0) {                       permits = permits + 1;
        exit-section;                              exit-section;
        ;   // null-statement
        entry-section;
     };
     permits = permits - 1;
     exit-section;
```

Semaphores can also be implemented at the operating system level.
Listing 3.5 shows implementations of *P()* and *V()* as operations in the kernel of an operating system. Critical sections are created by disabling and enabling interrupts.

```
P(s):disable interrupts;
      permits = permits - 1;
      if (permits < 0) {
          add the calling thread to the queue for s and change its state to blocked;
          schedule another thread;
      }
      enable interrupts;


V(s):disable interrupts;
      permits = permits + 1;
      if (permits <= 0)  {
          select a thread from the queue for s; change the thread's state to ready;
      }
      enable interrupts;
```

Listing 3.5 Implementation of *P()* and *V()* in the kernel of an operating system.

For a shared-memory multiprocessor machine, disabling interrupts may not work, so atomic hardware instructions like those described in Section 2.3 can be used.

3.4.2 The VP() Operation

Instead of writing

   s.V();
   t.P();

we combine the separate *V()* and *P()* operations into a single atomic *VP*() operation:

   t.VP(s);

An execution of $t.VP(s)$ is equivalent to $s.V()$; $t.P()$, except that during the execution of $t.VP(s)$, no intervening *P*(), *V*(), or *VP*() operations are allowed to be started on s and t.

Operation $t.VP(s)$ can only be used in cases where the *V()* operation on s cannot block the calling thread. This means that s is a counting semaphore, or s is or a binary semaphore that is guaranteed to be in a state in which a *V()* operation will not block.

A context switch between successive *V*() and *P*() operations is often a source of subtle programming errors. A *V*() operation followed by a *P*() operation appears in the *Exit-before-Wait* pattern:

```
mutex.V();              // exit critical section
someCondition.P();      // wait until some condition is true
```

- A thread T that executes *mutex.V*() releases mutual exclusion.

- Other threads may enter their critical sections and perform their *mutex.V()* and *someCondition.P()* operations before thread T has a chance to execute *someCondition.P*().

- This can create a problem, which is often in the form of a deadlock. The *VP*() operation removes these types of errors.

3.5 Semaphore-Based Solutions to Concurrent Programming Problems

These problems demonstrate how semaphores can be used to solve a commonly occurring synchronization problem.

3.5.1 Event Ordering

Assume that code segment *C1* in *Thread1* has to be executed after code segment *C2* in *Thread2*. Let *s* be a counting or binary semaphore initialized to 0.

```
Thread1    Thread2
s.P();     C2;
C1;        s.V();
```

The *s.P()* operation will block *Thread1* until *Thread2* does its *s.V()* operation. This guarantees that code segment *C1* is executed after segment *C2*.

3.5.2 Bounded Buffer

A bounded buffer has *n* slots. Each slot is used to store one item. A producer is not permitted to deposit an item when all the slots are full. A consumer is not permitted to withdraw an item when all the slots are empty.

The solution to the bounded buffer problem in Listing 3.6 uses semaphores *fullSlots* and *emptySlots* as resource counters, counting the full and empty slots, respectively, in the buffer.

The condition (number of full slots + number of empty slots == n) is true before and after each *deposit* and *withdraw* operation.

```
int buffer[ ] = new int[n];
countingSemaphore emptySlots(n);            // the number of empty slots
countingSemaphore fullSlots(0);             // the number of full slots

Producer {
   int in = 0;
   int item;
   ...
   /* produce item */
   emptySlots.P();      // wait if there are no empty slots
   buffer[in] = item;   // in is the index for a deposit
   in = (in + 1) % n;   // 0 ≤ in < n; and in = (out+#items in buffer)%n
   fullSlots.V();       // signal that a slot was filled
   ...
}

Consumer {
   int out = 0;
   int item;
   fullSlots.P();           // wait if there are no full slots
   item = buffer[out];   // out is the index for a withdraw
   out = (out + 1) % n; // 0 ≤ out < n
   emptySlots.V();       // signal that a slot was emptied
   /* consume item */
}
```

Listing 3.6 Bounded buffer using semaphores.

Since *in* and *out* cannot have the same value when the buffer is accessed, no critical section is required for accessing the slots in the *buffer*.

3.5.3 Dining Philosophers

There are *n* philosophers who spend their time eating and thinking [Dijkstra 1971]. They sit at a table with *n* seats. A bowl of rice sits in the center of the table. There is one chopstick between each pair of philosophers.

- When a philosopher is hungry, she picks up the two chopsticks that are next to her, one at a time.
- When she gets her chopsticks, she holds them until she is finished eating.
- Then she puts down her chopsticks one at a time and goes back to thinking.

Solutions to the dining philosophers problem are required to be free from deadlock and starvation. An additional property, called "maximal parallelism", may also be required. This property is satisfied by a solution that allows a philosopher to eat as long as her neighbors are not eating.

3.5.3.1 Solution 1. In the solution in Listing 3.7, each chopstick is represented as a binary semaphore, which serves as a simple resource counter that is initialized to 1. Picking up a chopstick is implemented as a *P()* operation, and releasing a chopstick is implemented as a *V()* operation. A global deadlock occurs when each philosopher holds her left chopstick and waits forever for her right chopstick.

```
philosopher(int i  /* 0..n-1 */) {
   while (true) {
      /*  think */
      chopsticks[i].P();            // pickup left chopstick
      chopsticks[(i+1) % n].P();   // pickup right chopstick
      /*  eat */
      chopsticks[i].V();            // put down left chopstick
      chopsticks[(i+1) % n].V();   // put down right chopstick
   }                               }
```
Listing 3.7 Dining philosophers using semaphores – Solution 1.

3.5.3.2 Solution 2. This solution is the same as Solution 1 except that only (n-1) philosophers are allowed to sit at a table that has *n* seats. A semaphore *seats* with initial value (n-1) is used as a resource counter to count the number of available seats. Each philosopher executes *seats.P()* before she picks up her left chopstick and *seats.V()* after she puts down her right chopstick.

- maximal parallelism is not satisfied (two neighboring philosophers hold a single chopstick but are unwilling to let each other eat).
- deadlock-free.
- if semaphores with First-Come-First-Serve (FCFS) *P()* and *V()* operations are used this solution is also starvation free.

3.5.3.3 Solution 3. This solution is the same as Solution 1 except that one philosopher is designated as the "odd" philosopher. The odd philosopher picks up her right chopstick first (instead of her left chopstick).

- maximal parallelism is not satisfied
- deadlock-free.
- if semaphores with First-Come-First-Serve (FCFS) *P()* and *V()* operations are used this solution is also starvation free.

3.5.3.4. Solution 4. In the solution in Listing 3.8, a philosopher picks up two chopsticks only if both of them are available. Each philosopher has three possible states: thinking, hungry and eating.

- A hungry philosopher can eat if her two neighbors are not eating.
- After eating, a philosopher unblocks a hungry neighbor who is able to eat.
- This solution is deadlock-free and it satisfies maximal parallelism, but it is not starvation-free. A hungry philosopher will starve if whenever one of her neighbors puts her chopstick down the neighbor on her other side is eating.

```java
final int thinking = 0; final int hungry = 1; final int eating = 2;
int state[] = new int[n];
for (int j = 0; j < n; j++) state[j] = thinking;
// mutex provides mutual exclusion for accessing state[].
binarySemaphore mutex = new binarySemaphore(1);
// philosopher i blocks herself on self[i] when she is hungry but unable to eat
binarySemaphore self[] = new binarySemaphore[n];
for (int j = 0; j < n; j++) self[j] = new binarySemaphore(0);

philosopher(int i /* 0..n-1 */) {
   while (true) {
      /* think */
      mutex.P()
      state[i] = hungry;
      test(i);           // performs self[i].V() if philosopher i can eat
      mutex.V();
      self[i].P();       // self[i].P() will not block if self[i].V() was
      /* eat */          //    performed during call to test(i)
      mutex.P();
      state[i] = thinking;
      test((i - 1) % n); // unblock left neighbor if she is hungry and can eat
      test((i + 1) % n); // unblock right neighbor she is hungry and can eat
      mutex.V();
   }
}
void test(int k /* 0..n-1 */) {
// philosopher i calls test(i) to check whether she can eat.
// philosopher i calls test((i - 1) % n) when she is finished eating to unblock a
// hungry left neighbor.
// philosopher i calls test((i + 1) % n) when she is finished eating to unblock a
// hungry right neighbor.
   if ((state[k] == hungry) && (state[(k+n-1) % n)] != eating) &&
       (state[(k+1) % n] != eating)) {
     state[k] = eating;
     self[k].V();   // unblock philosopher i's neighbor, or guarantee that
   }                //  philosopher i will not block on self[i].P().
}
```

Listing 3.8 Dining philosophers using semaphores – Solution 4.

Array *self[]* is an array of semaphores used to block philosophers who are unable to eat.

Philosopher i blocks itself on semaphore *self[i]* when she is hungry but unable to eat.

Philosopher *i*'s first call to *test()* is to check whether her neighbors are eating:

- if neither neighbor is eating, Philosopher *i* executes *self[i].V()* in function *test()*. Since only Philosopher i executes a *P()* operation on *self[i]*, and since Philosopher *i* is obviously not blocked on semaphore *self[i]* at the time she executes *self[i].V()*, no thread is unblocked by this *V()* operation. Furthermore, since *self[i]* is initialized to 0, this *V()* operation does not block and Philosopher *i* is allowed to continue. The purpose of this *V()* operation is not clear until we notice that Philosopher *i* immediately thereafter executes *self[i].P()*, which, thanks to the just performed *V()* operation, is guaranteed not to block.
- if one or both of Philosopher *i*'s neighbors are eating when *test(i)* is called, then *self[i].V()* is not executed, causing Philosopher *i* to be blocked when she executes *self[i].P()*.

The two other calls to *test()* that Philosopher *i* makes are to unblock hungry neighbors that are able to eat when Philosopher *i* finishes eating.

3.5.4 Readers and Writers

Data is shared by multiple threads [Courtois et al. 1971]. When a thread reads (writes) the shared data, it is considered to be a reader (writer). Readers may access the shared data concurrently, but a writer always has exclusive access.

Table 3.1 shows six different strategies for controlling how readers and writers access the shared data. These strategies fall into one of three categories based on whether readers or writers get priority when they both wish to access the data:

1. R=W: readers and writers have equal priority; served together in FCFS order.
2. R>W: readers generally have a higher priority than writers.
3. R<W: readers generally have a lower priority than writers.

Access Strategy	Description
R=W.1	One reader or one writer with equal priority
R=W.2	Many readers or one writer with equal priority
R>W.1	Many readers or one writer with readers having a higher priority
R>W.2	Same as R>W.1 except that when a reader arrives, if no other reader is reading or waiting, it waits until all writers that arrived earlier have finished
R<W.1	Many readers or one writer with writers having a higher priority
R<W.2	Same as R<W.1 except that when a writer arrives, if no other writer is writing or waiting, it waits until all readers that arrived earlier have finished

Table 3.1 Strategies for the readers and writers problem.

- In strategies R>W.1 and R>W.2, writers will starve if before a group of readers finishes reading there is always another reader requesting to read.
- In strategies R<W.1 and R<W.2, readers will starve if before a writer finishes writing there is always another writer requesting to write.
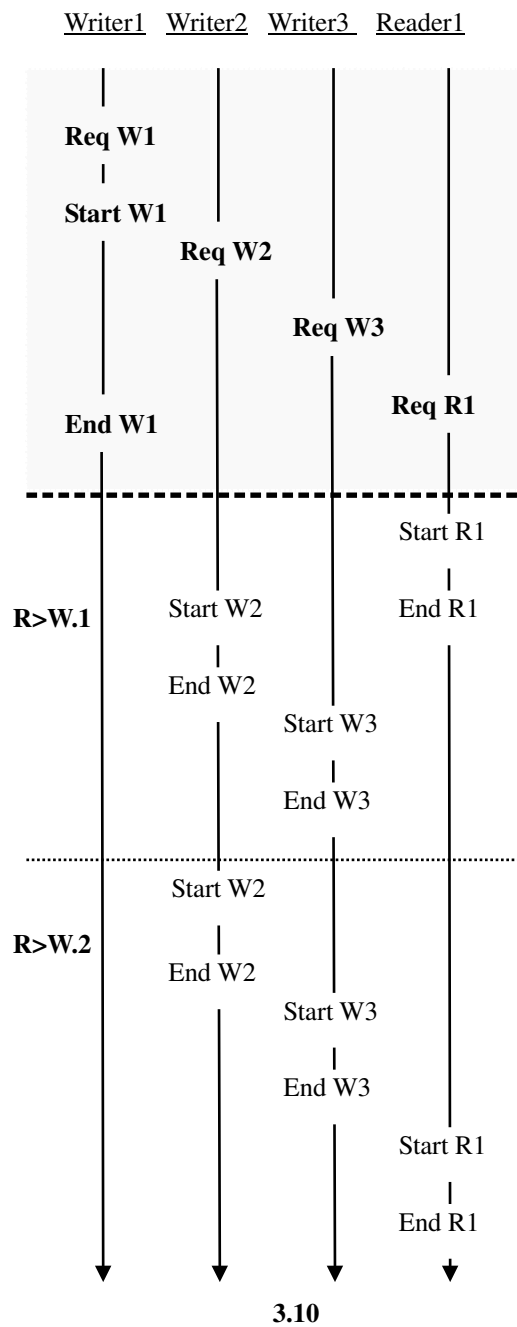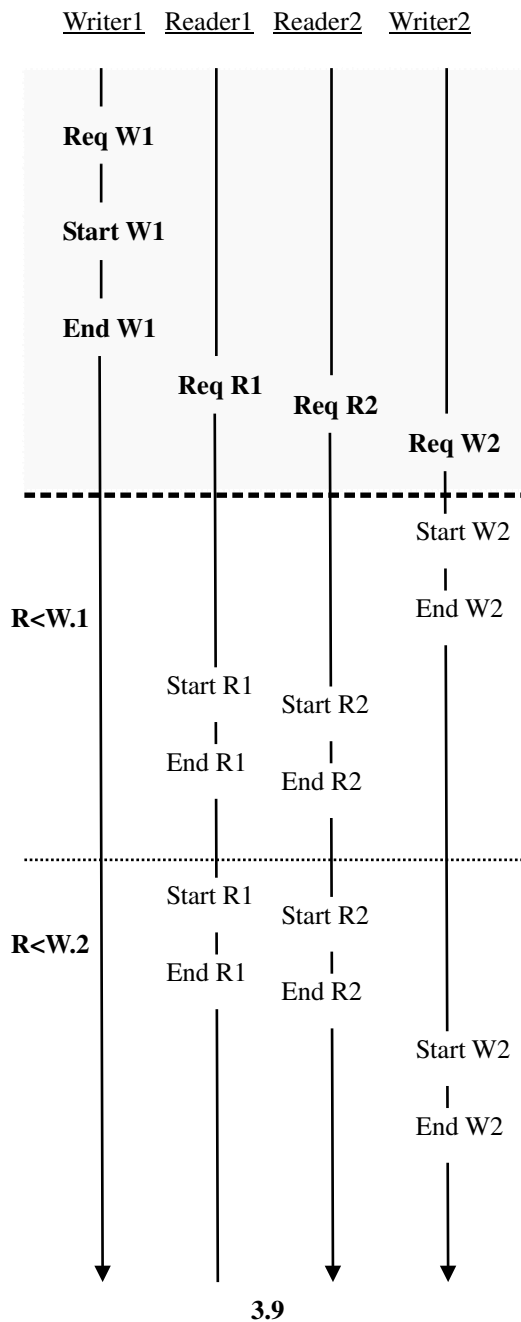- In strategies R=W.1 and R=W.2, no readers or writers will starve.

Fig. 3.9 compares strategies R<W.1 and R<W.2. In the shaded part at the top, Reader 1, Reader 2, and Writer 2 issue a request after Writer 1 finishes writing. Below the dashed line, the scenario is completed using the two different strategies.
- For strategy R<W.1, Writer 2 writes before the readers read.
- For strategy R<W.2, both readers are allowed to read before Writer 2 writes. Strategy R<W.2 requires Writer 2 to wait until the readers that arrived earlier have finished since Writer 2 requested to write when no other writer was writing or waiting.

We are assuming that multiple request events can occur before a decision is made as to whether to allow a reader or writer to start. This assumption is important in order to distinguish between the various strategies.

Fig. 3.10 compares strategies R>W.1 and R>W.2. In the shaded part at the top, Reader 1, Writer 2, and Writer 3 issue a request while Writer 1 is writing.
- For strategy R>W.1, Reader 1 reads before the writers writes.
- For strategy R>W.2, both writers are allowed to write before Reader 1 reads. This is because Reader 1 requested to read when no other reader was reading or waiting, so Reader 1 must wait until the writers who arrived earlier have finished.

Figure 3.9

Writer1	Reader1	Reader2	Writer2

Req W1

Start W1

End W1

Req R1

Req R2

Req W2

Start W2

End W2

R<W.1

Start R1

Start R2

End R1

End R2

Start R1

Start R2

R<W.2

End R1

End R2

Start W2

End W2

3.9

Figure 3.10

Writer1	Writer2	Writer3	Reader1

Req W1

Start W1

Req W2

Req W3

End W1

Req R1

Start R1

End R1

R>W.1

Start W2

End W2

Start W3

End W3

R>W.2

Start W2

End W2

Start W3

End W3

Start R1

End R1

3.10

3.5.4.1 R>W.1. This strategy gives readers a higher priority than writers and may cause waiting writers to starve.

In Listing 3.11, semaphore *mutex* provides mutual exclusion for the *Read()* and *Write()* operations, while semaphores *readers_que* and *writers_que* implement the Condition Queue pattern from section 3.2.2.

In operation *Write()*, a writer can write if no readers are reading and no writers are writing. Otherwise, the writer releases mutual exclusion and blocks itself by executing writers_que.P(). The *VP()* operation ensures that delayed writers enter the *writer_que* in the same order that they entered operation *Write()*.

In operation *Read()*, readers can read if no writers are writing; otherwise, readers block themselves by executing readers_que.P(). At the end of the read operation, a reader checks to see if any other readers are still reading. If not, it signals a writer blocked on *writers_que*. A continuous stream of readers will cause waiting writers to starve.

At the end of a write operation, waiting readers have priority. If readers are waiting in the *readers_que*, one of them is signaled. This first reader checks to see if any more readers are waiting. If so, it signals the second reader, which signals the third, and so on. This cascaded wakeup continues until no more readers are waiting. If no readers are waiting when a writer finishes, a waiting writer is signaled.

```
int activeReaders = 0;              // number of active readers
int activeWriters = 0;              // number of active writers
int waitingWriters = 0;             // number of waiting writers
int waitingReaders = 0;             // number of waiting readers
binarySemaphore mutex = new binarySemaphore(1); // exclusion
binarySemaphore readers_que = new binarySemaphore(0);// waiting readers
binarySemaphore writers_que = new binarySemaphore(0);// waiting writers
sharedData x ... ;                  // x is shared data


   Read() {                          Write() {
     mutex.P();                        mutex.P();
     if (activeWriters > 0) {          if (activeReaders > 0 ||
                                         activeWriters > 0 ) {

        waitingReaders++;                 waitingWriters++;
        readers_que.VP(mutex);            writers_que.VP(mutex);
     }                                 }
     activeReaders++;                  activeWriters++;
     if (waitingReaders > 0) {         mutex.V();
        waitingReaders--;
        readers_que.V();               /* write x */
     }
     else                             mutex.P();
        mutex.V();                     activeWriters--;
                                       if (waitingReaders > 0) {
     /* read x */                         waitingReaders--;
                                          readers_que.V();
     mutex.P();                        }
     activeReaders--;                  else if (waitingWriters > 0) {
     if (activeReaders == 0 &&            waitingWriters--;
        waitingWriters > 0) {            writers_que.V();
           waitingWriters--;           }
           writers_que.V();            else
     }                                    mutex.V();
     else                            }
        mutex.V();
   }
```

Listing 3.11 Strategy R>W.1.

This implementation illustrates another important semaphore pattern called "passing-the-baton".

Notice that delayed readers and writers exit their critical sections before they block themselves on *readers_que.P()* or *writers_que.P()*. This is part of the Exit-and-Wait pattern mentioned in Section 3.2. However, after these delayed threads are signaled, they never execute *mutex.P()* to reenter the critical section.

That is, we expect to see readers executing

```
readers_que.VP(mutex);    // exit critical section (mutex.V()) and wait
                          //   (readers_que.P())
mutex.P();                // reenter the critical section before continuing
```

but the *mutex.P()* operation is missing.

To understand why this operation is not needed, it helps to think of mutual exclusion as a baton that is passed from thread to thread.

- When a waiting thread is signaled, it receives the baton from the signaling thread. Possession of the baton gives the thread permission to execute in its critical section.
- When that thread signals another waiting thread, the baton is passed again.
- When there are no more waiting threads to signal, a *mutex.V()* operation is performed to release mutual exclusion. This will allow some thread to complete a future *mutex.P()* operation and enter its critical section for the first time.

This technique is implemented using an if-statement, such as the one used in the cascaded wakeup of readers:

```
if (waitingReaders > 0) {  // if another reader is waiting
   waitingReaders--;
   readers_que.V();   // pass the baton to a reader (i.e., do not release mutual
}                     // exclusion)
else
   mutex.V();         // else release mutual exclusion
```

- When a waiting reader is awakened, it receives mutual exclusion for accessing shared variables *activeReaders* and *waitingReaders*.
- If another reader is waiting, the baton is passed to that reader.
- If no more readers are waiting, mutual exclusion is released by executing *mutex.V()*.

3.5.4.2 R>W.2. This strategy allows concurrent reading and generally gives readers a higher priority than writers. Writers have priority in the following situation: when a reader requests to read, if it is a "lead reader" (i.e., no other reader is reading or waiting), it waits until all writers that arrived earlier have finished writing. This strategy may cause waiting writers to starve.

In Listing 3.12, when a reader R executes *Read()*:

- if one or more other readers are reading, then R starts reading immediately
- if one or more other readers are waiting for writers to finish, then R is blocked on *mutex*
- if no reader is reading or waiting then R is a lead reader, so R executes *writers_r_que.P()*.
  - If a writer is writing, then R will be blocked on *writers_r_que* behind any waiting writers that arrived before R. (Writers that arrived before R executed *writers_r_que.P()* before R did.)
  - Otherwise, R can start reading, and writers will be blocked when they execute *writers_r_que.P()*.

When a *Write()* operation ends, semaphore *writers_r_que* is signaled. Waiting writers that arrived before a lead reader will be ahead of the reader in the queue for *writers_r_que*; thus, the lead reader will have to wait for these writers to finish writing.

```
int activeReaders = 0;          // number of active readers
mutexLock mutex = new mutexLock();   // mutual exclusion for activeReaders
// condition queue for waiting writers and the first waiting reader
binarySemaphore writers_r_que = new binarySemaphore(1);
sharedData x ... ;               // x is shared data

Read() {
   mutex.lock();// block readers if lead reader waiting in writers_r_que
   ++activeReaders;
   if (activeReaders == 1)
      writers_r_que.P();  // block lead reader if a writer is writing
   mutex.unlock();

   /* read x */

   mutex.lock();
   --activeReaders;
   if (activeReaders == 0)
      writers_r_que.V(); // allow waiting writers, if any, to write
   mutex.unlock();
}

Write() {
   writers_r_que.P(); // block until no readers are reading or waiting and
   /* write x */    //   no writers are writing
   writers_r_que.V();// signal lead reader or a writer at the front of the queue
}
```

Listing 3.12 Strategy R>W.2.

This solution is interesting because it violates the *Exit-and-Wait* pattern described in Section 3.2.

As a rule, a thread will exit a critical section before blocking itself on a *P*() operation. However, lead readers violate this rule when they execute *writers_r_que.P*() without first executing *mutex.V()* to exit the critical section.

This is a key part of the solution.

- Only a lead reader (i.e. when *activeReaders* == 1) can enter the queue for *writers_r_que* since any other readers are blocked by *mutex.P*() at the beginning of the read operation.

- When the lead reader is released by *writers_r_que.V*(), the lead reader allows the other waiting readers to enter the critical section by signaling *mutex.V*().

- These other readers will not execute *writers_r_que.P*() since (*activeReaders* == 1) is only true for the lead reader. (Note that the name of the semaphore "*writers_r_que*" indicates that multiple writers but only one reader may be blocked on the semaphore.)

3.5.4.3 R<W.2. This strategy allows concurrent reading and generally gives writers a higher priority than readers. Readers have priority in the following situation: when a writer requests to write, if it is a lead writer (i.e., no other writer is writing or waiting), it waits until all readers that arrived earlier have finished reading.

This strategy and strategy R>W.2 are symmetrical. The solution in Listing 3.13 for R<W.2 differs from the solution in Listing 3.12 for R>W.2 as follows:

- semaphore *readers_w_que* is used to allow a lead writer to block readers. Only one writer (a lead writer) can be blocked in this queue.
- semaphore *writers_r_que* is renamed *writers_que,* since readers are never blocked in this queue.
- variable *waitingOrWritingWriters* is used to count the number of waiting or writing writers. At the end of a write operation, readers are permitted to read only if there are no waiting or writing writers.

When a writer W executes *Write()*:

- if one or more other writers are waiting for readers to finish, then W is blocked on *mutex_w*.
- if no writer is writing or waiting then W is a lead writer, so W executes *readers_w_que.P()*.
    - o (If a writer is writing, then *waitingOrWritingWriters* will be greater than one after it is incremented by W, so W will not execute *readers_w_que.P()*.)
    - o If a reader is reading, then W will be blocked on *readers_w_que* behind any waiting readers; otherwise, W can start writing, and readers will be blocked when they execute *readers_w_que.P()*.
- W can be followed by a stream of writers. When the last of these writers finishes, it executes *readers_w_que.V()* to signal waiting readers. If a finishing writer always finds that another writer is waiting, readers will starve.

The use of *readers_w_que* ensures that readers that arrive before a lead writer have a higher priority. This is because readers and lead writers both call *readers_w_que.P()*, and they are served in FCFS order.

- When a reader blocked in *readers_w_que* is given permission to read, it executes *readers_w_que.V()* to signal the next waiting reader, who executes *readers_w_que.V()* to signal the next waiting reader, and so on.
- This cascaded wakeup continues until there are no more waiting readers or until the *readers_w_que.V()* operation wakes up a lead writer.
- The lead writer executes *writers_que.P()* to block itself until the readers have finished.
- The last reader executes *writers_que.V()* to signal the lead writer that readers have finished.

```
int activeReaders = 0;  // number of active readers
int waitingOrWritingWriters = 0;// number of writers waiting or writing
mutexLock mutex_r = new mutexLock();   // exclusion for activeReaders
// exclusion for waitingOrWritingWriters
mutexLock mutex_w = new mutexLock();
binarySemaphore writers_que = new binarySemaphore(1);// waiting writers
// condition queue for waiting readers and the first waiting writer
binarySemaphore readers_w_que = new binarySemaphore(1);
sharedData x ... ;                 // x is shared data
Read() {
   readers_w_que.P(); // serve waiting readers and lead writer FCFS
   mutex_r.lock();
   ++activeReaders;
   if (activeReaders == 1)
   writers_que.P();      // block writers reads are occurring
   mutex_r.unlock();
   readers_w_que.V();// signal the next waiting reader or a lead writer

   /* read x */

   mutex_r.lock();
   --activeReaders;
   if (activeReaders == 0)
   writers_que.V();     // allow writing
   mutex_r.unlock();
}
```

```
Write () {
    // if a lead writer is waiting in readers_w_que, this blocks other writers
    mutex_w.lock();
    ++waitingOrWritingWriters;
    if (waitingOrWritingWriters == 1)  // true if this is a lead writer
    readers_w_que.P();  // block lead writer if there are waiting readers
    mutex_w.unlock();
    writers_que.P();       // block if a writer is writing or a reader is reading

    /* write x */

    writers_que.V();  // signal writing is over; wakes up waiting writer (if any)
    mutex_w.lock();
    -- waitingOrWritingWriters;
    if (waitingOrWritingWriters == 0)     // no writers are waiting or writing
    readers_w_que.V();          // so allow reading
    mutex_w.unlock();
}
```
Listing 3.13 Strategy R<W.2.

3.5.5 Simulating Counting Semaphores

Suppose that a system provides binary semaphores but not counting semaphores. Listing 3.14 shows how to use binary semaphores to implement the *P*() and *V*() operations in a *countingSemaphore* class.

Here is a possible execution scenario for four threads T1, T2, T3, and T4 that are using a *countingSemaphore s* initialized to 0:

1. T1 executes *s.P()*: T1 decrements *permits* to -1 and has a context switch immediately after completing the *mutex.V*() operation but before it executes *delayQ.P*().
2. T2 executes *s.P()*: T2 decrements *permits* to -2 and has a context switch immediately after completing the *mutex.V*() operation but before it executes *delayQ.P*().
3. T3 executes s.V(): T3 increments *permits* to -1 and executes *delayQ.V*().
4. T4 executes s.V(): T4 increments *permits* to 0 and executes *delayQ.V*(). Since T3 just previously completed an *s.V()* operation in which the value of *delayQ* was incremented to 1, T4 is blocked at *delayQ.V*().
5. T1 resumes execution, completes *delayQ.P*(), and then completes its *s.P()* operation.
6. T4 resumes execution and completes its *delayQ.V()* operation.
7. T2 resumes execution, completes *delayQ.P*(), and then completes its *s.P()* operation.

In step (4), T4 executes *s.V()* and is blocked by the *delayQ.V()* operation.

3.6 Semaphores and Locks in Java: see Chapter3NotesJava

3.7 Semaphores and Locks in Win32: see Chapter3NotesWin32

3.8 Semaphores and Locks in Pthreads: see Chapter3NotesPthreads

3.9 Another Note on Shared Memory Consistency

Recall from Section 2.5.6 the issues surrounding shared memory consistency.

Compiler and hardware optimizations may reorder read and write operations on shared variables making it difficult to reason about the behavior of multithreaded programs.

Critical sections created using Java's built-in synchronization operations or the operations in the Win32 or Pthreads library provide mutual exclusion and also protect against unwanted re-orderings.

For example, the shared variable values that a thread can see when it unlocks a mutex can also be seen by any thread that later locks the *same* mutex.

Thus, an execution in which shared variables are correctly protected by locks or semaphores is guaranteed to be sequentially consistent
   $\Rightarrow$ Rule: always access shared variables inside critical sections.

We will see that this rule also simplifies testing and debugging.