Semaphores — Last Time

A semaphore S supports two atomic operations:

S→Wait(): The process that issues a wait, waits until semaphore S is available

S→Signal(): The process that issues a signal, notifies other processes that S is free. If processes are waiting, the OS wakes one up.

A Binary Semaphore guarantees mutual exclusive access to a resource (only one process enters the critical section at a time). It is usually initialized to 1.

Too Much Milk:

```
Thread A           Thread B

S->Wait();         S->Wait();
if (noMilk)        if (noMilk)
   buy milk;          buy milk;
S->Signal();       S->Signal();
```

A Counting Semaphore represents a resource with many units available. The initial count to which the semaphore is initialized is usually the number of resources. A counting semaphore lets a process continue as long as more instances are available.

Semaphores are good for implementing:

- Scheduling constraints like waitpid or thread join (initial count == 0)
- Mutual exclusion (initial count == 1)
- Multi-instance resources like bounded buffers (inital count > 1)

Readers / Writers Problem — Today

Problem statement:

- An object is shared among many threads, most only read the object but some write it
- To get good performance we want to allow **multiple readers** at a time
- To get correct operation we want **only one writer** at a time (and zero readers when there is a writer)
- How do we control access to the object to permit this protocol?

Motivation

Reader / writer asymmetry is common!

A few examples:

- Book database at the library
  - Most accesses are searches (read-only)
  - Less often there are checkouts, returns, new books, etc. (writes)
- Making stock market data available on the web
  - Thousands of clients view the data (read-only)
  - A few times per hour the data is updated (writes)
  - If writes are not atomic clients will sometimes get an inconsistent view of the data — potentially an expensive problem
- Render farm
  - Rendering machines need constant, high-bandwidth access to models
  - Models are infrequently updated
- Anonymous CVS
  - Many people view the sources (for `gcc`, for example)
  - Only a few people contribute

Motivation

Readers/writers is purely a performance optimization. Since it adds complexity how can we tell when it is needed?

Example:

You are running the CVS server for `gcc`, the GNU C Compiler.

- It is important for people to check out a consistent snapshot of the sources
- gcc contains 18,000 files, 177 MB — 22 minutes to sync up with the sources using a fast cable modem

$\Longrightarrow$ At most 65 readers per day if there is no concurrency between readers

- If there are more readers than this, reader/writer locks are a necessity

Summary: Use reader/writer locks when mutual exclusion is too restrictive, and prevents performance goals from being met.

Reader/Writer Implementation using Semaphores

```
class ReadWrite {
   public:
     void Read();
     void Write();
   private:
     int       readers; // counts readers
     Semaphore my_read_lock;   // controls access to readers
     Semaphore wrt;     // toggles entry to first
}                       // writer or reader

ReadWrite::ReadWrite {
   readers      = 0;
   mutex->count = 1;     // mutex
   wrt->count   = 1;     // mutex
}

ReadWrite::Write(){
   wrt->Wait();         // any writers or readers?
   <perform write>
   wrt->Signal();       // enable write or read
}

ReadWrite::Read(){
   my_read_lock->Wait(); // reader mutual exclusion
      readers += 1;
      if (readers == 1)  // first reader
         wrt->Wait();    // blocks writers
   my_read_lock->Signal();
   <perform read>
   my_read_lock->Wait(); // reader mutual exclusion
      readers -= 1;
      if (readers == 0)  //
         wrt->Signal();  // enable writers or readers
   my_read_lock->Signal();
}
```


Readers/Writers Scenario 1

```
R1:                   R2:                W1:
Read ()
                    Read ()
                                       Write ()
```

Readers/Writers Scenario 2

```
R1:                R2:              W1:
                                  Write ()
Read ()
                 Read ()
```


Reader/Writers Scenario 3

```
R1:                R2:              W1:
Read ()
                                  Write ()
                 Read ()
```

Readers/Writers Solution Discussion

Implementation notes:

1. The first reader blocks if there is a writer; any other readers who try to enter block on `mutex`
2. The last reader to exit signals a waiting writer
3. When a writer exits, if there is both a reader and writer waiting, which goes next depends on the scheduler
4. If a writer exits and a reader goes next, then all readers that are waiting will fall through (at least one is waiting on `wrt` and zero or more can be waiting on `mutex`)
5. Does this solution guarantee all threads will make progress?

Alternative desirable semantics:

- Let a writer enter its critical section as soon as possible

Readers/Writers Solution Favoring Writers

```
ReadWrite::Write () {
   my_write_lock->Wait(); // writer mutual exclusion
      writers += 1;       // pending writer
      if (writers == 1)   // block readers
         block_readers->Wait();
   my_write_lock->Signal();

   wrt->Wait();   // writer/reader
   <perform write>      // mutual exclusion
   wrt ->Signal();

   my_write_lock->Wait();  // writer mutual exclusion
      writers -= 1;
      if (writers == 0)    // sync with readers
         block_readers->Signal();
   my_write_lock->Signal();
}

ReadWrite::Read () {
   block_readers->Wait();   // block if there’s a writer
   my_read_lock->Wait();    // reader mutual exclusion
      readers += 1;         //
      if (readers == 1)     // synchronize with writers
         wrt->Wait();
   my_read_lock->Signal();
   block_readers->Signal();

   <perform read>
    my_read_lock->Wait();   // reader mutual exclusion
      readers -= 1;         // reader done
      if (readers == 0)     // enable writers
         wrt->Signal();
    my_read_lock->Signal();
}
```

Readers/Writers Scenario 4

```
R1:          R2:        W1:         W2:
Read ()
          Read ()
                     Write ()
                                 Write ()
```


Readers/Writers Scenario 5

```
R1:          R2:        W1:         W2:
                     Write ()
Read ()
          Read ()
                                 Write ()
```

Reader/Writers Scenario 6

```
R1:          R2:          W1:          W2:
Read ()
                        Write ()
             Read ()
                                     Write ()
```