**15-213 Spring 2005** **March 21, 2005**
**Section F**

# Concurrency Example – Readers/Writers Locks

Consider the following data structure and functions:

```
typedef struct blackbox {

  // Some stuff that we don't know about

} Blackbox;

char *read(Blackbox *b);

void write(BlackBox *b, const char *buf);
```

We don't know how `Blackbox` has been implemented, but we have been told that it was written with no knowledge of threads in mind. Our manager wants us to write wrappers arounds the functions to allow for use of `Blackbox` in a multi-threaded environment, following these rules:

(1) If a `write` is occurring, there cannot be any `read`s occurring simultaneously

(2) There is at most one `write` occurring at any time

(3) If any thread is capable of proceeding without violating (1) or (2), then at least one such thread should be allowed to do so (i.e., there cannot be deadlock)

The declarations for these thread-safe wrappers is the same:

```
char *ts_read(Blackbox *b);

void ts_write(BlackBox *b, const char *buf);
```

**Question 1:**

What tools do we have to work with in solving this problem?

**Question 2:**

What is the simplest possible way to satisfy these criteria? What are the drawbacks of this implementation?

**Question 3:**

What if we add a fourth criteria: (4) If there are no `write`s occurring, then multiple `read`s should be able to run at the same time

## Implementation 1 - Mutex

```
/* Add a mutex and a count to Blackbox: */

typedef struct blackbox {
  // Some stuff that we don't know about

  volatile int reader_count;
  pthread_mutex_t lock;
} Blackbox;

char *ts_read(Blackbox *b)
{
  char *buf;

  pthread_mutex_lock(&b->lock);
  b->reader_count++;
  pthread_mutex_unlock(&b->lock);

  buf = read(b);

  pthread_mutex_lock(&b->lock);
  b->reader_count--;
  pthread_mutex_unlock(&b->lock);
}

void ts_write(BlackBox *b, const char *buf) {
  int written = FALSE;
  while (!written)
  {
    pthread_mutex_lock(&b->lock);
    if (b->reader_count == 0)
    {
      write(b, buf);
      written = TRUE;
    }
    pthread_mutex_unlock(&b->lock);
  }
}
```

### Question 4

One problem that is common is high-concurrency applications is *starvation* – that is, one thread or group of threads being perpetually denied a chance to run, even though there is no rule preventing them from running.

Can you see any reasons why Implementation 1 might lead to starvation? If so, how? How might we fix it?

## Implementation 2 - Condition Variable

```
/* This implementation ensures that if there are any writes
 * pending, they will occur before any reads are permitted,
 * minimizing the risk of starvation in the common case
 * (many readers, few writers).
 */

/* Add a mutex and a count to Blackbox: */

typedef struct blackbox {
  // Some stuff that we don't know about

  volatile int num_reads_in_prog;
  volatile int num_writes; // waiting or in progress
  pthread_cond_t reader_cv;
  pthread_cond_t writer_cv;
  pthread_mutex_t lock;
} Blackbox;

/* Pre-condition: b->lock must be locked before this function
 * is called
 */
void signal_next(BlackBox *b)
{
  if (b->num_writes > 0)
  {
    // If any writes are waiting, wake one up
    pthread_cond_signal(&b->writer_cv);
  }
  else
  {
    // If there are no writes pending, wake up all the
    // readers (there may not be any but that's fine)
    pthread_cond_broadcast(&b->reader_cv);
  }
}
```

```
char *ts_read(Blackbox *b)
{
  char *buf;

  pthread_mutex_lock(&b->lock);
  while (b->num_writes > 0)
  {
    // cond_wait unlocks the mutex, waits to be signaled,
    // then re-acquires the mutex
    pthread_cond_wait(&b->reader_cv, &b->lock);
  }
  // By here b->num_writes must be 0
  b->num_reads_in_prog++;
  pthread_mutex_unlock(&b->lock);

  buf = read(b);

  pthread_mutex_lock(&b->lock);
  b->num_reads_in_prog--;
  signal_next(b);
  pthread_mutex_unlock(&b->lock);
}

void ts_write(BlackBox *b, const char *buf) {
  pthread_mutex_lock(&b->lock);
  b->num_writes++;

  // Don't need a while here because we will always be able to
  // run if we are signaled
  if (b->num_writes > 1 || b->num_reads_in_prog > 0)
  {
    // cond_wait unlocks the mutex, waits to be signaled,
    // then re-acquires the mutex
    pthread_cond_wait(&b->writer_cv, &b->lock);
  }
  pthread_mutex_unlock(&b->lock);

  write(b, buf);

  pthread_mutex_lock(&b->lock);
  b->num_writes--;
  signal_next(b);
  pthread_mutex_unlock(&b->lock);
}
```