Readers and Writers Example - Granularity Issues.
Multiple concurrent readers, but exclusive access for writers.

Original Textbook code with ERRORS - What are they?

```
 1  int readers;                              // Neg value=> active writer
 2  pthread_mutex_t lock;
 3  pthread_cond_t rBusy, wBusy;              // Use separate conditional vars
 4                                            // for readers and writers
 5  AcquireExclusive()
 6  {
 7    pthread_mutex_lock(&lock);
 8    while(readers !=0)
 9    {
10      pthread_cond_wait(&wBusy, &lock);
11    }
12    readers=-1;
13    pthread_mutex_unlock(&lock);
14  }
15
16  AcquireShared()
17  {
18    pthread_mutex_lock(&lock);
19    readWaiters++;
20    while(readers<0)
21    {
22      pthread_cond_wait(&rBusy, &lock);
23    }
24    readWaiters--;
25    pthread_mutex_unlock(&lock);
26  }
27
28  ReleaseExclusive()
29  {
30    pthread_mutex_lock(&lock);
31    readers=0;
32    pthread_cond_broadcast(&rBusy);         // Only wake up readers
33    pthread_mutex_unlock(&lock);
34  }
35
36  ReleaseShared(
37  {
38    int doSignal;
39
40    pthread_mutex_lock(&lock);
41    readers--;
42    doSignal=(readers==0)
43    pthread_mutex_unlock(&lock);
44    if(doSignal)                            // Signal executes outside
45    {                                       // of critical section
46      pthread_cond_signal(&wBusy);          // Wake up a writer
47    }
48  }
```

Corrected Textbook code????

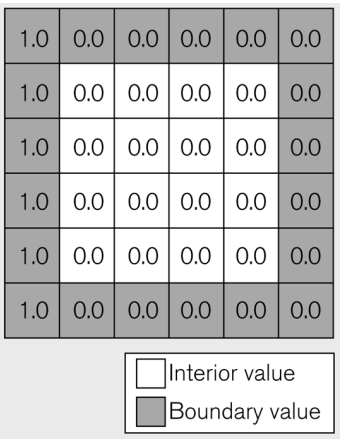
```
1  int readers;                    // Negative value => active writer
2  pthread_mutex_t lock;
3  pthread_cond_t busy;            // Use one condition variable to
4                                  // indicate whether data if busy
5  AcquireExclusive()
6  {
7    pthread_mutex_lock(&lock);    // This code suffers from spurious
8    while(readers!=0)             // wake-ups!!!
9    {
10     pthread_cond_wait(&busy, &lock);
11   }
12   readers=-1;
13   pthread_mutex_unlock(&lock);
14 }
15
16 AcquireShared()
17 {
18   pthread_mutex_lock(&lock);
19   while(readers<0)
20   {
21     pthread_cond_wait(&busy, &lock);
22   }
23   readers++;
24   pthread_mutex_unlock(&lock);
25 }
26
27 ReleaseExclusive()
28 {
29   pthread_mutex_lock(&lock);
30   readers=0;
31   pthread_cond_broadcast(&busy);
32   pthread_mutex_unlock(&lock);
33 }
34
35 ReleaseShared(
36 {
37   pthread_mutex_lock(&lock);
38   readers--;
39   if(readers==0)
40   {
41     pthread_cond_signal(&busy);
42   }
43   pthread_mutex_unlock(&lock);
44 }
```

Thread Scheduling - how kernel threads are scheduled by the OS. Two Pthread approaches:

- *bounded threads* - each pthread is mapped to its own kernel thread with the OS scheduler assigning them to processors (default which we typically want; attribute: PTHREAD_SCOPE_SYSTEM)
- *unbounded threads* - multiple pthreads (user-level threads) are mapped to a single kernel thread which the OS schedulers. When a user-level thread is blocked by I/O, the whole kernel thread would be put into a wait state, so no threads in that group are running. Only advantages is they have less creating and destroying overhead. (attribute: PTHREAD_SCOPE_PROCESS

Case Study: Successive Over-Relaxation (SOR) - often used in 3D form to solve differential equations such as Navier-Stokes equations for fluid flow.

1. 2D SOR - on each iteration replace all interior values by the average of their four nearest neighbors



a) Would it be best to perform static allocation or dynamic allocation (e.g., work queue) of the SOR calculations to threads?

b) How should we decompose the work among threads?

c) How do we synchronize the threads so all threads finish an iteration before any start the next iteration?

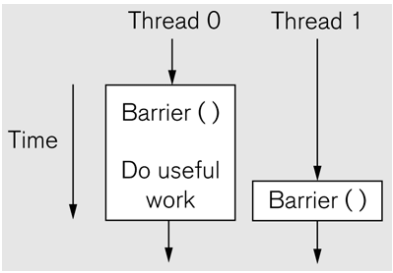
d) How can we build a barrier using mutex(es) and condition variable(s)?

```c
/*  Programmer:  Textbook Figure 16.6 Example
    File:        fig6-16.c
    Compiled by: gcc fig6-16.c -lpthread -lm
    Description:  Textbooks 2D over-relaxation program.
*/

#include <math.h>
#include <stdio.h>
#include <limits.h>
#include <time.h>
#include <pthread.h>
#include <stdlib.h>
#define MAXTHREADS 16/* max. # threads */

void * thread_main(void *);
void InitializeData();
void barrier();

pthread_mutex_t update_lock;
pthread_mutex_t barrier_lock;/* mutex for the barrier */
pthread_cond_t all_here;/* condition variable for barrier */
int count=0;/* counter for barrier */

int n, t, rowsPerThread;
double myDelta, threshold;
double **val, **new;
double delta=0.0;

/* Command line args: matrix size, number of threads, threshold
*/

int main(int argc, char * argv[])
{
   /* thread ids and attributes */
    pthread_t tid[MAXTHREADS];
    pthread_attr_t attr;
    long i, j;
    long startTime, endTime,seqTime,parTime;

    /* set global thread attributes */
    pthread_attr_init(&attr);
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

    /* initial mutex and condition variable */
    pthread_mutex_init(&update_lock, NULL);
    pthread_mutex_init(&barrier_lock, NULL);
    pthread_cond_init(&all_here, NULL);

    /* read command line arguments */
    if (argc != 4) {
        printf("usage: %s <matrix size> <# threads> <threshold>\n",
        argv[0]);
        exit(1);
    } // end if

    sscanf(argv[1], "%d", &n);
    sscanf(argv[2], "%d", &t);
    sscanf(argv[3], "%f", &threshold);

    rowsPerThread = n/t;
    InitializeData();

    for (i=0; i<t; i++) {
        pthread_create(&tid[i], &attr, thread_main, (void *) i);
    } // end for

    for (i=0; i < t; i++) {
       pthread_join(tid[i], NULL);
    } // end for

    printf("maximum difference:  %e\n", delta);
} // end main
```

```c
void* thread_main(void * arg) {

    long id=(long) arg;
    double average;
    double **myVal, **myNew;
    double **temp;
    int i, j;
    int start;

    /* determine first row that this thread owns */
    start = id*rowsPerThread+1;
    myVal = val;
    myNew = new;

    do {
        myDelta = 0.0;
        if (id == 0) {
            delta=0.0;/* reset shared value of delta */
        } // end if
        barrier();

        /* update each point */
        for (i = start; i < start+rowsPerThread; i++) {
            for (j = 1; j < n+1; j++) {
                average = (myVal[i-1][j] + myVal[i][j+1] +
                           myVal[i+1][j] + myVal[i][j-1])/4;
                if (myDelta < fabs(average - myVal[i][j]))
                    myDelta=fabs(average-myVal[i][j]);
                myNew[i][j] = average;
            } // end for j
        } // end for i

        temp = myNew;   /* prepare for next iteration */
        myNew = myVal;
        myVal = temp;

        pthread_mutex_lock(&update_lock);
        if (myDelta > delta) {
            delta = myDelta;    /* update delta */
        } // end if
        pthread_mutex_unlock(&update_lock);

        barrier();
    } while (delta >threshold);  // end do
} // end thread_main

void InitializeData() {
    int i, j;

    new = (double **) malloc((n+2)*sizeof(double *));
    val = (double **) malloc((n+2)*sizeof(double *));

    for (i = 0; i < n+2; i++) {
        new[i] = (double *) malloc((n+2)*sizeof(double));
        val[i] = (double *) malloc((n+2)*sizeof(double));
    } // end for i

    /* initialize to 0.0 except to 1.0 along the left boundary */
    for (i = 0; i < n+2; i++) {
        val[i][0] = 1.0;
        new[i][0] = 1.0;
    } // end for i
    for (i = 0; i < n+2; i++) {
        for (j = 1; j < n+2; j++) {
            val[i][j] = 0.0;
            new[i][j] = 0.0;
        } // end for j
    } // end for i
} // end InitializeData

void barrier() {
    pthread_mutex_lock(&barrier_lock);
    count++;
    if (count == t) {
        count = 0;
        pthread_cond_broadcast(&all_here);
    } else {
        while(pthread_cond_wait(&all_here, &barrier_lock) != 0);
    } // end if
    pthread_mutex_unlock(&barrier_lock);
} // end barrier
```

Case Study: Overlapping Synchronization with Computation

In general overlap long-latency operatons or communication with independent computation.




```
// Initiate synchronization
barrier.arrived();

// Do useful work

// Complete synchronization
barrier.wait();
```

Split-phase barrier allows a thread to do useful work while waiting for other threads to arrive at the barrier.

Erroneous Split-phase barrier that keeps a count of the number of arrivals.

```
 1  class Barrier
 2  {
 3    int nThreads;                    // Number of threads
 4    int count;                       // Number of threads participating
 5    pthread_mutex_t lock;
 6    pthread_cond_t all_here;
 7    public:
 8      Barrier(int t);
 9      ~Barrier(void);
10      void arrived(void);            // Initiate a barrier
11      int done(void);                // Check for completion
12      void wait(void);               // Wait for completion
13  }
14
15  int Barrier::done(void)
16  {
17    int rval;
18    pthread_mutex_lock(&lock);
19
20    rval=!count;                     // Done if the count is zero
21
22    pthread_mutex_unlock(&unlock);
23    return rval;
24  }
25
26  void Barrier::arrived(void)
27  {
28    pthread_mutex_lock(&lock);
29    count++                          // Another thread has arrived
30
31    // If last thread to arrive, then wake up any waiters
32    if(count==nThreads)
33    {
34      count=0;
35      pthread_cond_broadcast(&all_here);
36    }
37
38    pthread_mutex_unlock(&lock);
39  }
40
41  void Barrier::wait(void)
42  {
43    pthread_mutex_lock(&lock);
44
45    // If not done, then wait
46    if(count !=0)
47    {
48      pthread_cond_wait(&all_here, &lock);
49    }
50
51    pthread_mutex_lock(&lock);
52  }
```

Consider the following 2 thread situation:

	Thread 0	Thread 1	Count
Time	barrier.arrive()		0
		barrier.arrive()	1
		barrier.wait()	0
		barrier.arrive()	
		barrier.wait();	1
	barrier.wait();		

Both deadlock.

How might we fix the deadlock?

Problem in previous code "occurs because Thread 0 was looking at the state of the counter for the wrong invocation of the barrier" -- okay I buy that. "A solution then is to keep track of the current phase of the barrier. In particular, the arrive() method returns a phase number, which is then passed to the done() and wait() methods." Claimed correct code:

Can we find any problems with this code?

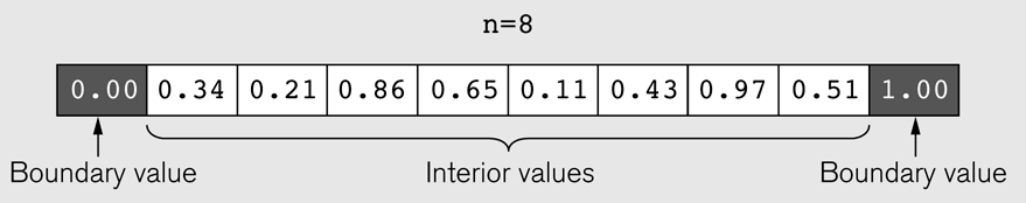
```
 1  class Barrier
 2  {
 3    int nThreads;                  // Number of threads
 4    int count;                     // Number of threads participating
 5    int phase;                     // Phase # of this barrier
 6    pthread_mutex_t lock;
 7    pthread_cond_t all_here;
 8    public:
 9      Barrier(int t);
10      ~Barrier(void);
11      void arrived(void);          // Initiate a barrier
12      int done(int p);             // Check for completion of phase p
13      void wait(int p);            // Wait for completion of phase p
14  }
15
16  int Barrier::done(int p)
17  {
18    int rval;
```

What is the purpose of the done() method?

```
19    pthread_mutex_lock(&lock);
20
21    rval=(phase!=p);               // Done if the phase # has changed
22
23    pthread_mutex_unlock(&lock);
24    return rval;
25  }
26
27  int Barrier::arrived(void)
28  {
29    int p;
30    pthread_mutex_lock(&lock);
31
32    p=phase;                       // Get phase number
33    count++                        // Another thread has arrived
34
35    // If last thread, then wake up any waiters, go to next phase
36    if(count==nThreads)
37    {
38      count=0;
39      pthread_cond_broadcast(&all_here);
40      phase=1 - phase;
41    }
42
43    pthread_mutex_unlock(&lock);
44    return p;
45  }
46
47  void Barrier::wait(int p)
48  {
49    pthread_mutex_lock(&lock);
50
51    // If not done, then wait
52    while(p==phase)
53    {
54      pthread_cond_wait(&all_here, &lock);
55    }
56
57    pthread_mutex_unlock(&lock);
58  }
```

Is this Barrier code correct?

1D Successive over-relaxation solution using “correc” split-phase barrier.




```
 1  double *val, *new;                // Hold n values
 2  int n;                            // Number of interior values
 3  int t;                            // Number of threads
 4  int iterations                    // Number of iterations to perform
 5
 6  thread_main(int index)
 7  {
 8    int n_per_thread=n/t;
 9    int start=index*n_per_thread;
10    int phase;
11
12    for(int i=0; i<iterations; i++)
13    {
14      // Update local boundary values
15      int j=start;
16      new[j]=(val[j-1]+val[j+1])/2.0;
17      j=start+n_per_thread -1;
18      new[j]=(val[j-1]+val[j+1])/2.0;
19
20      // Start barrier
21      phase=barrier.arrived();
22
23      // Update local interior values
24      for(j=start+1; j<start+n_per_thread-1; j++)
25      {
26        new[j]=(val[j-1]+val[j+1])/2.0;    // Compute average
27      }
28      swap(new, val);
29
30      // Complete barrier
31      barrier.wait(phase);
32    }
33  }
```

What is the swap(new, val) doing?

Would we expect any performance improvement with a 1D SOR with split-phase barriers?

Java - every object in Java has an associated mutex and condition variable with methods:

- wait - waits on the object's condition variable
- notifyAll - wakes up all threads waiting
- notify - wakes up only one thread

All must be called by a thread holding the object's mutex

Notes:

- A thread holding the mutex and issuing a wait appears to deadlock since it would prevent another threads from acquiring the mutex to perform a notify or notifyAll. However, the wait operation releases the mutex before putting the thread into a wait state
- When a waiting thread is awoken, it reacquires the mutex before the wait operation returns. (Java has recursive mutexes which have a counter for the number of times a thread has locked the mutex while holding the lock. When a thread invokes the unlock, the counter is decremented, and only when it reaches 0 is the mutex really unlocked.)
- One limitation of Java is that each object has only a single condition variable. In the BoundedBuffer example, all threads wait on the single condition variable. If we could have multiple condition variables with a monitor, threads waiting to insert into a full buffer could wait on one condition variable, while threads waiting to retrieve from an empty buffer could wait on another condition variable. This way the insert code could notify a thread waiting to retrieve from an empty buffer and vice versa.

The Java Thread Class

`public synchronized void start()`

- Starts this `Thread` and returns immediately after invoking the `run()` method.
- Throws `IllegalThreadStateException` if the thread was already started.

`public void run()`

- The body of this `Thread`, which is invoked after the thread is started.

```
public final synchronized void join(long millis)
  throws InterruptedException
```

- Waits for this `Thread` to die. A timeout in milliseconds can be specified, with a timeout of 0 milliseconds indicating that the thread will wait forever.

`public static void yield()`

- Causes the currently executing `Thread` object to yield the processor so that some other runnable Thread can be scheduled.

`public final int getPriority()`

- Returns the thread's priority.

`public final void setPriority(int newPriority)`

- Sets the thread's priority.

Notes:

For a complete list of public methods for the `Thread` class, see `java.lang.Thread`.

Count 3s Example in Java

```
 1  import java.util.*;
 2  import java.util.concurrent.*;
 3
 4  public class CountThrees implements Runnable
 5  {
 6    private static final int ARRAY_LENGTH=1000000;
 7    private static final int MAX_THREADS=10;
 8    private static final int MAX_RANGE=100;
 9    private static final Random random=new Random();
10    private static int count=0;
11    private static Object lock=new Object();
12    private static int[] array;
13    private static Thread[] t;
14
15    public static void main(String[] args)
16    {
17      array=new int[ARRAY_LENGTH];
18
19      //initialize the elements in the array
20      for(int i=0; i<array.length; i++)
21      {
22        array[i]=random.nextInt(MAX_RANGE);
23      }
24
25      //create the threads
26      CountThrees[] counters=new CountThrees[MAX_THREADS];
27      int lengthPerThread=ARRAY_LENGTH/MAX_THREADS;
28
29      for(int i=0; i<counters.length; i++)
30      {
31        counters[i]=new CountThrees(i*lengthPerThread,
32                                    lengthPerThread);
33      }
34
35      //run the threads
36      for(int i=0; i<counters.length; i++)
37      {
38        t[i]=new Thread(counters[i]);
39        t[i].start();
40      }
41      for(int i=0; i<counters.length; i++)
42      {
43        try
44        {
45          t[i].join();
46        }
47        catch(InterruptedException e)
48        { /*do nothing*/ }
49      }
50
51      //print the number of threes
52      System.out.println("Number of threes: " + count);
53    }
54
55    private int startIndex;
56    private int elements;
57    private int myCount=0;
58
59    public CountThrees(int start, int elem)
60    {
61      startIndex=start;
62      elements=elem;
63    }
64
65    //Overload of run method in the Thread class
66    public void run()
67    {
68      //count the number of threes
69      for(int i=0; i<elements; i++)
70      {
71        if(array[startIndex+i]==3)
72        {
73          myCount++;
74        }
75      }
76      synchronized(lock)
77      {
78        count+=myCount;
79      }
80    }
81  }
```