```
1  // Using Barriers
2  // George F. Riley, Georgia Tech, Spring 2012
3  // This illustrates how a barrier would be implemented (both a "buggy" version
4  // and a good one).
5
6  #include <iostream>
7
8  #include "gthread.h"
9  #include "math.h"
10 #include <sys/time.h>
11
12 #include "complex.h"
13 #include "InputImage.h"
14
15 using namespace std;
16
17 // Implement a "buggy" barrier for illustration
18 class BuggyBarrier {
19 public:
20   BuggyBarrier(int P0);   // P is the total number of threads
21   void Enter(int);        // Enter the barrier, don't exit till alll there
22 private:
23   int  P;
24   int  count;             // Number of threads presently in the barrier
25   int FetchAndIncrementCount();
26   pthread_mutex_t countMutex;
27
28 };
29
30 BuggyBarrier::BuggyBarrier(int P0)
31   : P(P0), count(0)
32 {
33   // Initialize the mutex used for FetchAndIncrement
34   pthread_mutex_init(&countMutex, 0);
35 }
36
37 void BuggyBarrier::Enter(int)
38 { // This is buggy!  Why?
39   // Also, we include the "int" parameter, but it's not neede for this
40   // implementation. It is needed for the GoodBarrier, so we add a
41   // dummy paramter to make switching between the good and buggy one
42   // easier.
43   int myCount = FetchAndIncrementCount();
44   if (myCount == (P - 1))
45     { // All threads have entered, reset count and exit
46       count = 0;
47     }
48   else
49     { // Spin until all threads entered
50       while(count != 0) { } // Spin waiting for others
51     }
52 }
53
54 int BuggyBarrier::FetchAndIncrementCount()
55 { // We don't have an atomic FetchAndIncrement, but we can get the
56   // same behavior by using a mutex
```

Program barriers.cc

```
57    pthread_mutex_lock(&countMutex);
58    int myCount = count;
59    count++;
60    pthread_mutex_unlock(&countMutex);
61    return myCount;
62  }
63
64  // Implement a "good" barrier.  This is called the "sense reversing" barrier
65  class GoodBarrier {
66  public:
67    GoodBarrier(int P0);   // P is the total number of threads
68    void Enter(int myId);  // Enter the barrier, don't exit till alll there
69  private:
70    int  P;
71    int  count;             // Number of threads presently in the barrier
72    int FetchAndDecrementCount();
73    pthread_mutex_t countMutex;
74    bool* localSense;       // We will create an array of bools, one per thread
75    bool  globalSense;      // Global sense
76  };
77
78  GoodBarrier::GoodBarrier(int P0)
79    : P(P0), count(P0)
80  {
81    // Initialize the mutex used for FetchAndIncrement
82    pthread_mutex_init(&countMutex, 0);
83    // Create and initialize the localSense arrar, 1 entry per thread
84    localSense = new bool[P];
85    for (int i = 0; i < P; ++i) localSense[i] = true;
86    // Initialize global sense
87    globalSense = true;
88  }
89
90  void GoodBarrier::Enter(int myId)
91  { // This works.  Why?
92    localSense[myId] = !localSense[myId];  // Toggle private sense variable
93    if (FetchAndDecrementCount() == 1)
94      { // All threads here, reset count and toggle global sense
95        count = P;
96        globalSense = localSense[myId];
97      }
98    else
99      {
100       while (globalSense != localSense[myId]) {  } // Spin
101     }
102 }
103
104
105 int GoodBarrier::FetchAndDecrementCount()
106 { // We don't have an atomic FetchAndDecrement, but we can get the
107   // same behavior by using a mutex
108   pthread_mutex_lock(&countMutex);
109   int myCount = count;
110   count--;
111   pthread_mutex_unlock(&countMutex);
112   return myCount;
```

Program barriers.cc (continued)

```
113 }
114
115 gthread_mutex_t printingMutex;
116 //BuggyBarrier* barrier;    // The barrier for thread coordinatino
117 GoodBarrier* barrier;    // The barrier for thread coordinatino
118
119 void MyThreadThreeArgs(int myId, int count1, int count2)
120 {
121   for (int i = 0; i < count1; ++i)
122     {
123       LockMutex(printingMutex);
124       cout << "Hello from thread " << myId << " count1 " << i
125            << endl;
126       UnlockMutex(printingMutex);
127       // Use a random count2, so that each thread takes a different
128       // amount of time.  drand48() returns a random number between
129       // zero and one.
130       count2 = count2 * drand48();
131       for (int j = 0; j < count2; ++j)
132         {
133         }
134       // Here we want to wait for all other threads to get to this
135       // point, hence we need a barrier.
136       barrier->Enter(myId);
137     }
138   EndThread();  // Required by the GThreads library
139 }
140
141 int main( int argc, char** argv)
142 {
143   if (argc < 4)
144     {
145       cout << "Usage: testGthread nThreads count1 count2" << endl;
146       exit(1);
147     }
148   int nThreads = atol(argv[1]);
149   int count1   = atol(argv[2]);
150   int count2   = atol(argv[3]);
151   //barrier = new BuggyBarrier(nThreads + 1);
152   barrier = new GoodBarrier(nThreads);
153
154   for (int i = 0; i < nThreads; ++i)
155     { // Start each thread
156       cout << "Creating thread " << i << endl;
157       CreateThread(MyThreadThreeArgs, i, count1, count2);
158     }
159   // Now wait for all to complete
160   WaitAllThreads();
161   cout << "Main exiting" << endl;
162 }
163
```

Program barriers.cc (continued)