This illustrates how a barrier would be implemented (both a "buggy" version and a good one).

George F. Riley, Georgia Tech, Fall 2010

#include <iostream>
#include "pthread.h"

using namespace std;

// Implement a "buggy" barrier for illustration
class BuggyBarrier {
public:
  BuggyBarrier(int P0); // P is the total number of threads
  void Enter(int); // Enter the barrier, don’t exit until all there
private:
  int P;
  int count; // Number of threads presently in the barrier
  int FetchAndIncrementCount();
  pthread_mutex_t countMutex;
};

BuggyBarrier::BuggyBarrier(int P0)
: P(P0), count(0)
{
  // Initialize the mutex used for FetchAndIncrement
  pthread_mutex_init(&countMutex, 0);
}

void BuggyBarrier::Enter(int)
{ // This is buggy! Why?
  // Also, we include the "int" parameter, but it’s not needed for this
  // implementation. It is needed for the GoodBarrier, so we add a
  // dummy parameter to make switching between the good and buggy one
  // easier.
  int myCount = FetchAndIncrementCount();
  if (myCount == (P - 1))
    { // All threads have entered, reset count and exit
      count = 0;
    } // Spin until all threads entered
  else
    while(count != 0) {} // Spin waiting for others
}

int BuggyBarrier::FetchAndIncrementCount()
{ // We don’t have an atomic FetchAndIncrement, but we can get the
  // same behavior by using a mutex
  pthread_mutex_lock(&countMutex);
  int myCount = count;
  count++;
  pthread_mutex_unlock(&countMutex);
  return myCount;
}

Program barrier.cc
Implement a "good" barrier. This is called the "sense reversing" barrier.

class GoodBarrier {
public:
    GoodBarrier(int P0); // P is the total number of threads
    void Enter(int myId); // Enter the barrier, don’t exit till all there
private:
    int P;
    int count; // Number of threads presently in the barrier
    int FetchAndDecrementCount();
    pthread_mutex_t countMutex;
    bool* localSense; // We will create an array of bools, one per thread
    bool globalSense; // Global sense
};

GoodBarrier::GoodBarrier(int P0) : P(P0), count(P0) {
    // Initialize the mutex used for FetchAndIncrement
    pthread_mutex_init(&countMutex, 0);
    // Create and initialize the localSense array, 1 entry per thread
    localSense = new bool[P];
    for (int i = 0; i < P; ++i) localSense[i] = true;
    // Initialize global sense
    globalSense = true;
}

void GoodBarrier::Enter(int myId) { // This works. Why?
    localSense[myId] = !localSense[myId]; // Toggle private sense variable
    if (FetchAndDecrementCount() == 1) {
        // All threads here, reset count and toggle global sense
        count = P;
        globalSense = localSense[myId];
    }
    else {
        while (globalSense != localSense[myId]) { } // Spin
    }
}

int GoodBarrier::FetchAndDecrementCount() {
    // We don’t have an atomic FetchAndDecrement, but we can get the
    // same behavior by using a mutex
    pthread_mutex_lock(&countMutex);
    int myCount = count;
    count--;
    pthread_mutex_unlock(&countMutex);
    return myCount;
}

int main( int argc, char** argv) {
    // Create the good barrier
    barrier = new GoodBarrier(nThreads + 1);
    // Create some threads here. Each thread enter the barrier

    Program barrier.cc (continued)
// when it has completed the assigned task.
// Enter the barrier and wait for
// all threads to enter the same barrier
barrier->Enter(nThreads);
cout << "All threads finished pass 1" << endl;
}