```
// $Id: prodconsbuf.cpp,v 1.3 2014-05-30 22:59:13-07 - - $

// Producer/consumer problem using mutex and condition_variable.

#include <array>
#include <condition_variable>
#include <iomanip>
#include <cassert>
#include <iostream>
#include <mutex>
#include <sstream>
#include <string>
#include <thread>
#include <vector>
using namespace std;

#include <sys/time.h>

//
// Timer.
//
class elapsed_time {
   private:
      struct timeval start;
   public:
      elapsed_time() { gettimeofday (&start, nullptr); }
      string elapsed() {
         struct timeval now;
         gettimeofday (&now, nullptr);
         double secs = (double) (now.tv_sec - start.tv_sec)
                     + (double) (now.tv_usec - start.tv_usec) * 1e-6;
         ostringstream result;
         result << setw(7) << setprecision(3) << fixed << secs;
         return result.str();
      }
} timer;

```

```

//
// class bounded_buffer
// NOT synchronized.
// Just your ordinary 12B Data Structures queue.
//

template <typename T, size_t size>
class bounded_buffer {
   public:
      using value_type = T;
   private:
      static constexpr ssize_t EMPTY = -1;
      ssize_t head = EMPTY;
      ssize_t tail = EMPTY;
      array<T,size> items;
   public:
      // Six synthesizeable are all OK.
      bool empty() const { return head == EMPTY; }
      bool full() { return (tail + 1) % size == head; }
      const value_type& front() const;
      void pop();
      void push (const value_type& val);
};

template <typename T, size_t size>
const T& bounded_buffer<T,size>::front() const {
   if (empty()) throw runtime_error ("bounded_buffer::front (empty)");
   return items[head];
}

template <typename T, size_t size>
void bounded_buffer<T,size>::pop() {
   if (empty()) throw runtime_error ("bounded_buffer::pop (empty)");
   if (head == tail) head = tail = EMPTY;
                else head = (head + 1) % size;
}

template <typename T, size_t size>
void bounded_buffer<T,size>::push (const value_type& val) {
   if (full()) throw runtime_error ("bounded_buffer::push (full)");
   if (empty()) head = tail = 0;
           else tail = (tail + 1) % size;
   items[tail] = val;
}

```

```

//
// class synchronized_buffer
// prevents concurrent access and uses the bounded_buffer.
//

template <typename T, size_t size>
class synchronized_buffer {
   public:
      using value_type = T;
   private:
      bounded_buffer<T,size> buffer;
      mutex lock;
      condition_variable condvar;
      bool ready {false};
   public:
      void put (const value_type& val);
      value_type get();
};

template <typename T, size_t size>
void synchronized_buffer<T,size>::put (const value_type& val) {
   unique_lock<mutex> ulock (lock);
   while (buffer.full()) condvar.wait (ulock);
   buffer.push (val);
   condvar.notify_all();
}

template <typename T, size_t size>
T synchronized_buffer<T,size>::get() {
   unique_lock<mutex> ulock (lock);
   while (buffer.empty()) condvar.wait (ulock);
   value_type result = buffer.front();
   buffer.pop();
   condvar.notify_all();
   return result;
}

```

```

//
// Data counter.
// Counts messages so that consumers know when to stop.
// Consumers stop when all producers have quit and data is done.
//

class counter {
   private:
      ssize_t data_count {0};
      ssize_t producer_count {0};
      bool producer_started {false};
      mutex lock;
   public:
      enum ADJUST {INCR = +1, NONE = 0, DECR = -1};
      void adjust (ADJUST data, ADJUST producer = NONE);
      bool end_of_data();
      friend string to_string (const counter&);
};

void counter::adjust (ADJUST data, ADJUST producer) {
   lock.lock();
   assert ((data == NONE and producer != NONE)
         or (data != NONE and producer == NONE));
   if (producer == INCR) producer_started = true;
   data_count += (ssize_t) data;
   producer_count += (ssize_t) producer;
   assert (data_count >= 0);
   assert (producer_count >= 0);
   lock.unlock();
}

bool counter::end_of_data() {
   // Should be const, but then couldn't lock it.
   lock.lock();
   bool end = producer_started and producer_count == 0
              and data_count == 0;
   lock.unlock();
   return end;
}

string to_string (const counter& ctr) {
   return "[" + to_string (ctr.data_count) + ","
          + to_string (ctr.producer_count) + "]"
          + (ctr.producer_started ? "+" : "-");
}

```

```

//
// Buffer and data declarations and printer.
//

using buf_data = pair<string,size_t>;
using synch_buffer = synchronized_buffer<buf_data,5>;
string to_string (const buf_data& data) {
   return "[\"" + data.first + "\"," + to_string (data.second) + "]";
}

struct printer {
   mutex lock;
   void print (const string& name, size_t id, const buf_data& data,
               const counter* count) {
      lock.lock();
      cout << timer.elapsed() << " " << name << " " << id
           << " " << to_string (data) << " ... " << to_string (*count)
           << endl << flush;
      lock.unlock();
   }
   void print (const string& name, const string &status) {
      lock.lock();
      cout << timer.elapsed() << " " << name << " " << status << endl;
      lock.unlock();
   }
} print;

```

```

//
// Producer and consumer threads.
//

void producer (size_t id, counter* count, synch_buffer* buffer,
               const vector<string>* words) {
   count->adjust (counter::NONE, counter::INCR);
   print.print ("producer " + to_string (id), "STARTING");
   for (const auto& word: *words) {
      this_thread::sleep_for (chrono::milliseconds (id * 200));
      buf_data data {word, id};
      buffer->put (data);
      count->adjust (counter::INCR);
      print.print ("producer", id, data, count);
   }
   count->adjust (counter::NONE, counter::DECR);
   print.print ("producer " + to_string (id), "FINISHED");
}

void consumer (size_t id, counter* count, synch_buffer* buffer) {
   print.print ("consumer " + to_string (id), "STARTING");
   do {
      this_thread::sleep_for (chrono::milliseconds (id * 400));
      if (count->end_of_data()) break;
      auto data = buffer->get();
      count->adjust (counter::DECR);
      print.print ("consumer", id, data, count);
   }while (not count->end_of_data());
   print.print ("consumer " + to_string (id), "FINISHED");
}

//
// Main.
//

int main() {
   cout << boolalpha;
   counter count;
   print.print ("main " + to_string (count), "STARTING");
   synch_buffer buffer;
   vector<thread> vec;
   vector<string> words {"Hello", "World", "foo", "bar", "baz", "qux"};
   for (size_t i = 1; i <= 3; ++i) {
      vec.push_back (thread (producer, i, &count, &buffer, &words));
      vec.push_back (thread (consumer, i, &count, &buffer));
   }
   for (auto& t: vec) t.join();
   print.print ("main " + to_string (count), "FINISHED");
   return 0;
}

//TEST// prodconsbuf >prodconsbuf.out 2>&1
//TEST// mkpspdf prodconsbuf.ps prodconsbuf.cpp* prodconsbuf.out

```

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ mkc: starting prodconsbuf.cpp
prodconsbuf.cpp:
     $Id: prodconsbuf.cpp,v 1.3 2014-05-30 22:59:13-07 - - $
g++ -g -O0 -Wall -Wextra -std=gnu++11 prodconsbuf.cpp -o prodconsbuf -lg
lut -lGLU -lGL -lX11 -lm -lrt
rm -f prodconsbuf.o
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ mkc: finished prodconsbuf.cpp
```

```
0.000 main [0,0]- STARTING
0.001 producer 1 STARTING
0.001 producer 2 STARTING
0.001 consumer 1 STARTING
0.001 consumer 2 STARTING
0.001 producer 3 STARTING
0.001 consumer 3 STARTING
0.201 producer 1 ["Hello",1] ... [1,3]+
0.401 producer 2 ["Hello",2] ... [2,3]+
0.401 producer 1 ["World",1] ... [3,3]+
0.401 consumer 1 ["Hello",1] ... [2,3]+
0.601 producer 1 ["foo",1] ... [3,3]+
0.601 producer 3 ["Hello",3] ... [4,3]+
0.801 producer 2 ["World",2] ... [5,3]+
0.801 consumer 2 ["Hello",2] ... [4,3]+
0.801 consumer 1 ["World",1] ... [3,3]+
0.801 producer 1 ["bar",1] ... [4,3]+
1.001 producer 1 ["baz",1] ... [5,3]+
1.201 consumer 1 ["foo",1] ... [4,3]+
1.201 producer 2 ["foo",2] ... [5,3]+
1.201 consumer 3 ["Hello",3] ... [5,3]+
1.202 producer 3 ["World",3] ... [5,3]+
1.601 consumer 2 ["World",2] ... [4,3]+
1.601 producer 1 ["qux",1] ... [5,3]+
1.601 producer 1 FINISHED
1.601 consumer 1 ["bar",1] ... [4,2]+
1.602 producer 2 ["bar",2] ... [5,2]+
2.002 consumer 1 ["baz",1] ... [4,2]+
2.002 producer 3 ["foo",3] ... [5,2]+
2.401 consumer 2 ["foo",2] ... [4,2]+
2.401 producer 2 ["baz",2] ... [5,2]+
2.402 consumer 1 ["World",3] ... [4,2]+
2.402 consumer 3 ["qux",1] ... [3,2]+
2.602 producer 3 ["bar",3] ... [4,2]+
2.802 producer 2 ["qux",2] ... [5,2]+
2.802 producer 2 FINISHED
2.802 consumer 1 ["bar",2] ... [4,1]+
3.202 consumer 2 ["foo",3] ... [3,1]+
3.202 consumer 1 ["baz",2] ... [2,1]+
3.202 producer 3 ["baz",3] ... [3,1]+
3.602 consumer 3 ["bar",3] ... [2,1]+
3.602 consumer 1 ["qux",2] ... [1,1]+
3.802 producer 3 ["qux",3] ... [2,1]+
3.802 producer 3 FINISHED
4.002 consumer 2 ["baz",3] ... [1,0]+
4.002 consumer 1 ["qux",3] ... [0,0]+
4.002 consumer 1 FINISHED
4.802 consumer 2 FINISHED
4.802 consumer 3 FINISHED
4.802 main [0,0]+ FINISHED
```