```
1  //Zachery Davis
2  #include <pthread.h>
3  #include <iostream>
4  #include <unistd.h>
5  #include <cstdlib>
6
7  using namespace std;
8
9  #define MAX 10
10 #define N 4
11
12 // Data structure to represent a simplified Order
13 // that has an order number and an item number.
14 struct Order
15 {
16     int order_num;
17     int item_num;
18 };
19
20 Order new_orders [N];       // array of elements of type Order to be used as a shared buffer
21 int num_new_orders = 0;     // count of number of new (i.e., unprocessed) orders
22 int order_num = 0;          // global variable used to generate unique order numbers
23
24 // TODO: Define and initialize necessary mutex and condition variables here
25 pthread_mutex_t data_mutex = PTHREAD_MUTEX_INITIALIZER;
26 pthread_mutex_t console_mutex = PTHREAD_MUTEX_INITIALIZER;
27
28 pthread_cond_t cond_order_added = PTHREAD_COND_INITIALIZER;
29 pthread_cond_t cond_space_available = PTHREAD_COND_INITIALIZER;
30
31 void* takeOrders(void* arg)
32 {
33     int item;
34     int index = 0;
35
36     for(int i = 0; i < MAX; ++i) {
37         // Beginning of critical region 1
38         pthread_mutex_lock(&console_mutex);
39         // Get user input
40         cout << "Enter a menu item number between 1 and 50: ";
41         cin >> item;
42
43         // Print new order's details
44         cout << "Got new order! Order number is " << order_num << " and item number: " << item
           << std::endl;
45
46         // End of critical region 1
47         pthread_mutex_unlock(&console_mutex);
48
49         // Beginning of critical region 2
50         pthread_mutex_lock(&data_mutex);
51         while(num_new_orders == N) { //full buffer
52             pthread_cond_wait(&cond_space_available, &data_mutex); //wait for something to be
               removed
53         }
54         // Put new order into new orders buffer and update number of new orders
55         new_orders[index].order_num = order_num;
56         new_orders[index++].item_num = item;
57         ++num_new_orders;
58
59         // End of critical region 2
60         pthread_cond_signal(&cond_order_added);
61         pthread_mutex_unlock(&data_mutex);
62
63         // Update order number so that next order gets a different number
```

```
64          ++order_num;
65
66          // If the end of the new orders buffer is reached, wrap back around
67          if(index == N)
68              index = 0;
69      }
70
71      pthread_exit(NULL);
72  }
73
74  void* processOrders(void* arg)
75  {
76      int item;
77      int index = 0;
78      int o_num;
79
80      for(int i = 0; i < MAX; ++i) {
81          // Beginning of critical region 3
82          pthread_mutex_lock(&data_mutex);
83          while(num_new_orders == 0) { //empty buffer
84              pthread_cond_wait(&cond_order_added, &data_mutex); //wait for something to be added
85          }
86          // Retrieve new order details from buffer and update number of new orders
87          o_num = new_orders[index].order_num;
88          item = new_orders[index++].item_num;
89          --num_new_orders;
90
91          // End of critical region 3
92          pthread_cond_signal(&cond_space_available);
93          pthread_mutex_unlock(&data_mutex);
94
95          // Beginning of critical region 4
96          pthread_mutex_lock(&console_mutex);
97          // Print retrieved order's details
98          cout << "Processing order number " << o_num << " with item number: " << item <<
            std::endl;
99
100         // End of critical region 4
101         pthread_mutex_unlock(&console_mutex);
102
103         // Suspend self for 1 second
104         sleep(1);
105
106         // If the end of the new orders buffer is reached, wrap back around
107         if(index == N)
108             index = 0;
109     }
110
111     pthread_exit(NULL);
112 }
113
114 int main()
115 {
116     // Create threads to take and process orders
117     pthread_t id1, id2;
118     pthread_create(&id1, NULL, processOrders, NULL);
119     pthread_create(&id2, NULL, takeOrders, NULL);
120
121     // TODO: Add code to wait for both threads to finish
122     pthread_join(id1, NULL);
123     pthread_join(id2, NULL);
124
125     // Print goodbye message
126     cout << "Phew! Done with orders for today!" << endl;
127
```

```
128         pthread_exit(NULL);
129     }
130
```