```
1   CS 202, Fall 2021
2   Handout 4 (Class 5)
3
4   The handout from the last class gave examples of race conditions. The following
5   panels demonstrate the use of concurrency primitives (mutexes, etc.). We are
6   using concurrency primitives to eliminate race conditions (see items 1
7   and 2a) and improve scheduling (see item 2b).
8
9   1. Protecting the linked list......
10
11          Mutex list_mutex;
12
13          insert(int data) {
14              List_elem* l = new List_elem;
15              l->data = data;
16
17              acquire(&list_mutex);
18
19              l->next = head;
20              head = l;
21
22              release(&list_mutex);
23          }
24
```

```
25  2.  Producer/consumer revisited [also known as bounded buffer]
26
27      2a. Producer/consumer [bounded buffer] with mutexes
28
29      Mutex mutex;
30
31      void producer (void *ignored) {
32          for (;;) {
33              /* next line produces an item and puts it in nextProduced */
34              nextProduced = means_of_production();
35
36              acquire(&mutex);
37              while (count == BUFFER_SIZE) {
38                  release(&mutex);
39                  yield(); /* or schedule() */
40                  acquire(&mutex);
41              }
42
43              buffer [in] = nextProduced;
44              in = (in + 1) % BUFFER_SIZE;
45              count++;
46              release(&mutex);
47          }
48      }
49
50      void consumer (void *ignored) {
51          for (;;) {
52
53              acquire(&mutex);
54              while (count == 0) {
55                  release(&mutex);
56                  yield(); /* or schedule() */
57                  acquire(&mutex);
58              }
59
60              nextConsumed = buffer[out];
61              out = (out + 1) % BUFFER_SIZE;
62              count--;
63              release(&mutex);
64
65              /* next line abstractly consumes the item */
66              consume_item(nextConsumed);
67          }
68      }
69
```

```
 70
 71      2b. Producer/consumer [bounded buffer] with mutexes and condition variables
 72
 73          Mutex mutex;
 74          Cond nonempty;
 75          Cond nonfull;
 76
 77          void producer (void *ignored) {
 78              for (;;) {
 79                  /* next line produces an item and puts it in nextProduced */
 80                  nextProduced = means_of_production();
 81
 82                  acquire(&mutex);
 83                  while (count == BUFFER_SIZE)
 84                      cond_wait(&nonfull, &mutex);
 85
 86                  buffer [in] = nextProduced;
 87                  in = (in + 1) % BUFFER_SIZE;
 88                  count++;
 89                  cond_signal(&nonempty, &mutex);
 90                  release(&mutex);
 91              }
 92          }
 93
 94          void consumer (void *ignored) {
 95              for (;;) {
 96
 97                  acquire(&mutex);
 98                  while (count == 0)
 99                      cond_wait(&nonempty, &mutex);
100
101                  nextConsumed = buffer[out];
102                  out = (out + 1) % BUFFER_SIZE;
103                  count--;
104                  cond_signal(&nonfull, &mutex);
105                  release(&mutex);
106
107                  /* next line abstractly consumes the item */
108                  consume_item(nextConsumed);
109              }
110          }
111
112
113          Question: why does cond_wait need to both release the mutex and
114          sleep? Why not:
115
116              while (count == BUFFER_SIZE)  {
117                  release(&mutex);
118                  cond_wait(&nonfull);
119                  acquire(&mutex);
120              }
121
```

```
122      2c.  Producer/consumer [bounded buffer] with semaphores
123
124          Semaphore mutex(1);             /* mutex initialized to 1 */
125          Semaphore empty(BUFFER_SIZE);  /* start with BUFFER_SIZE empty slots */
126          Semaphore full(0);             /* 0 full slots */
127
128          void producer (void *ignored) {
129              for (;;) {
130                  /* next line produces an item and puts it in nextProduced */
131                  nextProduced = means_of_production();
132
133                  /*
134                   * next line diminishes the count of empty slots and
135                   * waits if there are no empty slots
136                   */
137                  sem_down(&empty);
138                  sem_down(&mutex);  /* get exclusive access */
139
140                  buffer [in] = nextProduced;
141                  in = (in + 1) % BUFFER_SIZE;
142
143                  sem_up(&mutex);
144                  sem_up(&full);    /* we just increased the # of full slots */
145              }
146          }
147
148          void consumer (void *ignored) {
149              for (;;) {
150
151                  /*
152                   * next line diminishes the count of full slots and
153                   * waits if there are no full slots
154                   */
155                  sem_down(&full);
156                  sem_down(&mutex);
157
158                  nextConsumed = buffer[out];
159                  out = (out + 1) % BUFFER_SIZE;
160
161                  sem_up(&mutex);
162                  sem_up(&empty);   /* one further empty slot */
163
164                  /* next line abstractly consumes the item */
165                  consume_item(nextConsumed);
166              }
167          }
168
169      Semaphores *can* (not always) lead to elegant solutions (notice
170      that the code above is fewer lines than 2b) but they are much
171      harder to use.
172
173      The fundamental issue is that semaphores make implicit (counts,
174      conditions, etc.) what is probably best left explicit. Moreover,
175      they *also* implement mutual exclusion.
176
177      For this reason, you should not use semaphores. This example is
178      here mainly for completeness and so you know what a semaphore
179      is. But do not code with them. Solutions that use semaphores in
180      this course will receive no credit.
```