

Sep 21, 2025 12:13

handout04.txt

Page 1/3

```

1 CS 202, Fall 2025
2 Handout 4
3
4 Handout 3 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. Producer/consumer revisited [also known as bounded buffer]
10
11 2a. Producer/consumer [bounded buffer] with mutexes
12
13     Mutex mutex;
14
15     void producer (void *ignored) {
16         for (;;) {
17             /* next line produces an item and puts it in nextProduced */
18             nextProduced = means_of_production();
19
20             mutex_lock(&mutex);
21             while (count == BUFFER_SIZE) {
22                 mutex_unlock(&mutex);
23                 yield(); /* or schedule() */
24                 mutex_lock(&mutex);
25             }
26
27             buffer[in] = nextProduced;
28             in = (in + 1) % BUFFER_SIZE;
29             count++;
30             mutex_unlock(&mutex);
31         }
32     }
33
34     void consumer (void *ignored) {
35         for (;;) {
36
37             mutex_lock(&mutex);
38             while (count == 0) {
39                 mutex_unlock(&mutex);
40                 yield(); /* or schedule() */
41                 mutex_lock(&mutex);
42             }
43
44             nextConsumed = buffer[out];
45             out = (out + 1) % BUFFER_SIZE;
46             count--;
47             mutex_unlock(&mutex);
48
49             /* next line abstractly consumes the item */
50             consume_item(nextConsumed);
51         }
52     }
53

```

Sep 21, 2025 12:13

handout04.txt

Page 2/3

```

54
55 2b. Producer/consumer [bounded buffer] with mutexes and condition variables
56
57     Mutex mutex;
58     Cond nonempty;
59     Cond nonfull;
60
61     void producer (void *ignored) {
62         for (;;) {
63             /* next line produces an item and puts it in nextProduced */
64             nextProduced = means_of_production();
65
66             mutex_lock(&mutex);
67             while (count == BUFFER_SIZE)
68                 cond_wait(&nonfull, &mutex);
69
70             buffer[in] = nextProduced;
71             in = (in + 1) % BUFFER_SIZE;
72             count++;
73             cond_signal(&nonempty, &mutex);
74             mutex_unlock(&mutex);
75         }
76     }
77
78     void consumer (void *ignored) {
79         for (;;) {
80
81             mutex_lock(&mutex);
82             while (count == 0)
83                 cond_wait(&nonempty, &mutex);
84
85             nextConsumed = buffer[out];
86             out = (out + 1) % BUFFER_SIZE;
87             count--;
88             cond_signal(&nonfull, &mutex);
89             mutex_unlock(&mutex);
90
91             /* next line abstractly consumes the item */
92             consume_item(nextConsumed);
93         }
94     }
95
96
97     Question: why does cond_wait need to both mutex_unlock the mutex and
98     sleep? Why not:
99
100         while (count == BUFFER_SIZE) {
101             mutex_unlock(&mutex);
102             cond_wait(&nonfull);
103             mutex_lock(&mutex);
104         }
105

```

```

106 2c. Producer/consumer [bounded buffer] with semaphores
107
108 Semaphore mutex(1);          /* mutex initialized to 1 */
109 Semaphore empty(BUFFER_SIZE); /* start with BUFFER_SIZE empty slots */
110 Semaphore full(0);           /* 0 full slots */
111
112 void producer (void *ignored) {
113     for (;;) {
114         /* next line produces an item and puts it in nextProduced */
115         nextProduced = means_of_production();
116
117         /*
118          * next line diminishes the count of empty slots and
119          * waits if there are no empty slots
120          */
121         sem_down(&empty);
122         sem_down(&mutex); /* get exclusive access */
123
124         buffer[in] = nextProduced;
125         in = (in + 1) % BUFFER_SIZE;
126
127         sem_up(&mutex);
128         sem_up(&full); /* we just increased the # of full slots */
129     }
130 }
131
132 void consumer (void *ignored) {
133     for (;;) {
134
135         /*
136          * next line diminishes the count of full slots and
137          * waits if there are no full slots
138          */
139         sem_down(&full);
140         sem_down(&mutex);
141
142         nextConsumed = buffer[out];
143         out = (out + 1) % BUFFER_SIZE;
144
145         sem_up(&mutex);
146         sem_up(&empty); /* one further empty slot */
147
148         /* next line abstractly consumes the item */
149         consume_item(nextConsumed);
150     }
151 }
152
153 Semaphores *can* (not always) lead to elegant solutions (notice
154 that the code above is fewer lines than 2b) but they are much
155 harder to use.
156
157 The fundamental issue is that semaphores make implicit (counts,
158 conditions, etc.) what is probably best left explicit. Moreover,
159 they *also* implement mutual exclusion.
160
161 For this reason, you should not use semaphores. This example is
162 here mainly for completeness and so you know what a semaphore
163 is. But do not code with them. Solutions that use semaphores in
164 this course will receive no credit.

```