

```

1  CS 5600, week 6.a
2
3  The previous handout demonstrated the use of mutexes and condition
4  variables. This handout demonstrates the use of monitors (which combine
5  mutexes and condition variables).
6
7
8  1. The bounded buffer as a monitor
9
10 // This is pseudocode that is inspired by C++.
11 // Don't take it literally.
12
13 class MyBuffer {
14     public:
15         MyBuffer();
16         ~MyBuffer();
17         void Enqueue(Item);
18         Item = Dequeue();
19     private:
20         int count;
21         int in;
22         int out;
23         Item buffer[BUFFER_SIZE];
24         Mutex* mutex;
25         Cond* nonempty;
26         Cond* nonfull;
27     }
28
29     void
30     MyBuffer::MyBuffer()
31     {
32         in = out = count = 0;
33         mutex = new Mutex;
34         nonempty = new Cond;
35         nonfull = new Cond;
36     }
37
38     void
39     MyBuffer::Enqueue(Item item)
40     {
41         mutex.acquire();
42         while (count == BUFFER_SIZE)
43             cond_wait(&nonfull, &mutex);
44
45         buffer[in] = item;
46         in = (in + 1) % BUFFER_SIZE;
47         ++count;
48         cond_signal(&nonempty);
49         mutex.release();
50     }
51
52     Item
53     MyBuffer::Dequeue()
54     {
55         mutex.acquire();
56         while (count == 0)
57             cond_wait(&nonempty, &mutex);
58
59         Item ret = buffer[out];
60         out = (out + 1) % BUFFER_SIZE;
61         --count;
62         cond_signal(&nonfull);
63         mutex.release();
64         return ret;
65     }
66

```

```

67
68 int main(int, char**)
69 {
70     MyBuffer buf;
71     int dummy;
72     tid1 = thread_create(producer, &buf);
73     tid2 = thread_create(consumer, &buf);
74
75     // never reach this point
76     thread_join(tid1);
77     thread_join(tid2);
78     return -1;
79 }
80
81 void producer(void* buf)
82 {
83     MyBuffer* sharedbuf = reinterpret_cast<MyBuffer*>(buf);
84     for (;;) {
85         /* next line produces an item and puts it in nextProduced */
86         Item nextProduced = means_of_production();
87         sharedbuf->Enqueue(nextProduced);
88     }
89 }
90
91 void consumer(void* buf)
92 {
93     MyBuffer* sharedbuf = reinterpret_cast<MyBuffer*>(buf);
94     for (;;) {
95         Item nextConsumed = sharedbuf->Dequeue();
96
97         /* next line abstractly consumes the item */
98         consume_item(nextConsumed);
99     }
100 }
101
102 Key point: *Threads* (the producer and consumer) are separate from
103 *shared object* (MyBuffer). The synchronization happens in the
104 shared object.
105

```

106 2. This monitor is a model of a database with multiple readers and  
 107 writers. The high-level goal here is (a) to give a writer exclusive  
 108 access (a single active writer means there should be no other writers  
 109 and no readers) while (b) allowing multiple readers. Like the previous  
 110 example, this one is expressed in pseudocode.

```

111 // assume that these variables are initialized in a constructor
112 state variables:
113 AR = 0; // # active readers
114 AW = 0; // # active writers
115 WR = 0; // # waiting readers
116 WW = 0; // # waiting writers
117
118 Condition okToRead = NIL;
119 Condition okToWrite = NIL;
120 Mutex mutex = FREE;
121
122 Database::read() {
123   startRead(); // first, check self into the system
124   Access Data
125   doneRead();
126 }
127
128 Database::startRead() {
129   acquire(&mutex);
130   while((AW + WW) > 0){
131     WR++;
132     wait(&okToRead, &mutex);
133   }
134   WR--;
135   AR++;
136   release(&mutex);
137 }
138
139 Database::doneRead() {
140   acquire(&mutex);
141   AR--;
142   if (AR == 0 && WW > 0) { // if no other readers still
143     signal(&okToWrite);
144   }
145   release(&mutex);
146 }
147
148 Database::write(){ // symmetrical
149   startWrite(); // check in
150   Access Data
151   doneWrite(); // check out
152 }
153
154 Database::startWrite() {
155   acquire(&mutex);
156   while ((AW + AR) > 0) { // check if safe to write.
157     // if any readers or writers, wait
158     WW++;
159     wait(&okToWrite, &mutex);
160   }
161   WW--;
162   AW++;
163   release(&mutex);
164 }
165
166 Database::doneWrite() {
167   acquire(&mutex);
168   AW--;
169   if (WW > 0) {
170     signal(&okToWrite); // give priority to writers
171   } else if (WR > 0) {
172     broadcast(&okToRead);
173   }
174   release(&mutex);
175 }
176 }
177

```

178 NOTE: what is the starvation problem here?

179

### 180 3. Shared locks

```

181 struct sharedlock {
182   int i;
183   Mutex mutex;
184   Cond c;
185 };
186
187 void AcquireExclusive (sharedlock *sl) {
188   acquire(&sl->mutex);
189   while (sl->i) {
190     wait (&sl->c, &sl->mutex);
191   }
192   sl->i = -1;
193   release(&sl->mutex);
194 }
195
196 void AcquireShared (sharedlock *sl) {
197   acquire(&sl->mutex);
198   while (sl->i < 0) {
199     wait (&sl->c, &sl->mutex);
200   }
201   sl->i++;
202   release(&sl->mutex);
203 }
204
205 void ReleaseShared (sharedlock *sl) {
206   acquire(&sl->mutex);
207   if (--sl->i)
208     signal (&sl->c);
209   release(&sl->mutex);
210 }
211
212 void ReleaseExclusive (sharedlock *sl) {
213   acquire(&sl->mutex);
214   sl->i = 0;
215   broadcast (&sl->c);
216   release(&sl->mutex);
217 }
218
219
220 QUESTIONS:
221 A. There is a starvation problem here. What is it? (Readers can keep
222   writers out if there is a steady stream of readers.)
223 B. How could you use these shared locks to write a cleaner version
224   of the code in the prior item? (Though note that the starvation
225   properties would be different.)

```