

Feb 07, 24 9:19

handout05.txt

Page 1/4

```

1 CS 202, Spring 2024
2 Handout 5 (Class 6)
3
4 The previous handout demonstrated the use of mutexes and condition
5 variables. This handout demonstrates the use of monitors (which combine
6 mutexes and condition variables).
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, &mutex);
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, &mutex);
63     mutex.release();
64     return ret;
65 }
66

```

Feb 07, 24 9:19

handout05.txt

Page 2/4

```

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

```

Feb 07, 24 9:19

handout05.txt

Page 3/4

```

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(); // check self out of system
126 }
127
128 Database::startRead() {
129     acquire(&mutex);
130     while((AW + WW) > 0){
131         WR++;
132         wait(&okToRead, &mutex);
133         WR--;
134     }
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, &mutex); // active, wake up writer
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         WW--;
161     }
162     AW++;
163     release(&mutex);
164 }
165
166 Database::doneWrite() {
167     acquire(&mutex);
168     AW--;
169     if (WW > 0) {
170         signal(&okToWrite, &mutex); // give priority to writers
171     } else if (WR > 0) {
172         broadcast(&okToRead, &mutex);
173     }
174     release(&mutex);
175 }
176
177 NOTE: what is the starvation problem here?
178

```

Feb 07, 24 9:19

handout05.txt

Page 4/4

```

179
180 3. Shared locks
181
182 struct sharedlock {
183     int i;
184     Mutex mutex;
185     Cond c;
186 };
187
188 void AcquireExclusive (sharedlock *sl) {
189     acquire(&sl->mutex);
190     while (sl->i) {
191         wait (&sl->c, &sl->mutex);
192     }
193     sl->i = -1;
194     release(&sl->mutex);
195 }
196
197 void AcquireShared (sharedlock *sl) {
198     acquire(&sl->mutex);
199     while (sl->i < 0) {
200         wait (&sl->c, &sl->mutex);
201     }
202     sl->i++;
203     release(&sl->mutex);
204 }
205
206 void ReleaseShared (sharedlock *sl) {
207     acquire(&sl->mutex);
208     if (!--sl->i)
209         signal (&sl->c, &sl->mutex);
210     release(&sl->mutex);
211 }
212
213 void ReleaseExclusive (sharedlock *sl) {
214     acquire(&sl->mutex);
215     sl->i = 0;
216     broadcast (&sl->c, &sl->mutex);
217     release(&sl->mutex);
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.)

```