

Feb 22, 11 2:05

l11-handout.txt

Page 1/7

```

1 Handout for CS 372H
2 Class 11
3 22 February 2011
4
5 1. Example of a monitor: MyBuffer
6
7 // This is pseudocode that is inspired by C++.
8 // Don't take it literally.
9
10 class MyBuffer {
11     public:
12         MyBuffer();
13         ~MyBuffer();
14         void Enqueue(Item);
15         Item = Dequeue();
16     private:
17         int count;
18         int in;
19         int out;
20         Item buffer[BUFFER_SIZE];
21         Mutex* mutex;
22         Cond* nonempty;
23         Cond* nonfull;
24     }
25
26     void
27     MyBuffer::MyBuffer()
28     {
29         in = out = count = 0;
30         mutex = new Mutex;
31         nonempty = new Cond;
32         nonfull = new Cond;
33     }
34
35     void
36     MyBuffer::Enqueue(Item item)
37     {
38         mutex.acquire();
39         while (count == BUFFER_SIZE)
40             cond_wait(&nonfull, &mutex);
41
42         buffer[in] = item;
43         in = (in + 1) % BUFFER_SIZE;
44         ++count;
45         cond_signal(&nonempty, &mutex);
46         mutex.release();
47     }
48
49     Item
50     MyBuffer::Dequeue()
51     {
52         mutex.acquire();
53         while (count == 0)
54             cond_wait(&nonempty, &mutex);
55
56         Item ret = buffer[out];
57         out = (out + 1) % BUFFER_SIZE;
58         --count;
59         cond_signal(&nonfull, &mutex);
60         mutex.release();
61         return ret;
62     }
63

```

Feb 22, 11 2:05

l11-handout.txt

Page 2/7

```

64 int main(int, char**)
65 {
66     MyBuffer buf;
67     int dummy;
68     tid1 = thread_create(producer, &buf);
69     tid2 = thread_create(consumer, &buf);
70     thread_join(tid1);
71
72     // never reach this point
73     return -1;
74 }
75
76 void producer(void* buf)
77 {
78     MyBuffer* sharedbuf = reinterpret_cast<MyBuffer*>(buf);
79     for (;;) {
80         /* next line produces an item and puts it in nextProduced */
81         Item nextProduced = means_of_production();
82         sharedbuf->Enqueue(nextProduced);
83     }
84 }
85
86 void consumer(void* buf)
87 {
88     MyBuffer* sharedbuf = reinterpret_cast<MyBuffer*>(buf);
89     for (;;) {
90         Item nextConsumed = sharedbuf->Dequeue();
91
92         /* next line abstractly consumes the item */
93         consume_item(nextConsumed);
94     }
95 }
96
97 Key point: *Threads* (the producer and consumer) are separate from
98 *shared object* (MyBuffer). The synchronization happens in the
99 shared object.
100

```

Feb 22, 11 2:05

l11-handout.txt

Page 3/7

```

101 2. Readers/writers
102
103 state variables:
104     AR = 0; // # active readers
105     AW = 0; // # active writers
106     WR = 0; // # waiting readers
107     WW = 0; // # waiting writers
108
109     Condition okToRead = NIL;
110     Condition okToWrite = NIL;
111     Mutex mutex = FREE;
112
113 Database::read() {
114     startRead(); // first, check self into the system
115     Access Data
116     doneRead(); // check self out of system
117 }
118
119 Database::startRead() {
120     acquire(&mutex);
121     while((AW + WW) > 0){
122         WR++;
123         wait(&okToRead, &mutex);
124         WR--;
125     }
126     AR++;
127     release(&mutex);
128 }
129
130 Database::doneRead() {
131     acquire(&mutex);
132     AR--;
133     if (AR == 0 && WW > 0) { // if no other readers still
134         signal(&okToWrite, &mutex); // active, wake up writer
135     }
136     release(&mutex);
137 }
138
139 Database::write(){ // symmetrical
140     startWrite(); // check in
141     Access Data
142     doneWrite(); // check out
143 }
144
145 Database::startWrite() {
146     acquire(&mutex);
147     while ((AW + AR) > 0) { // check if safe to write.
148         // if any readers or writers, wait
149         WW++;
150         wait(&okToWrite, &mutex);
151         WW--;
152     }
153     AW++;
154     release(&mutex);
155 }
156
157 Database::doneWrite() {
158     acquire(&mutex);
159     AW--;
160     if (WW > 0) {
161         signal(&okToWrite, &mutex); // give priority to writers
162     } else if (WR > 0) {
163         broadcast(&okToRead, &mutex);
164     }
165     release(&mutex);
166 }
167
168 NOTE: what is the starvation problem here?
169

```

Feb 22, 11 2:05

l11-handout.txt

Page 4/7

```

170 3. Shared locks
171
172 struct sharedlock {
173     int i;
174     Mutex mutex;
175     Cond c;
176 };
177
178 void AcquireExclusive (sharedlock *sl) {
179     acquire(&sl->mutex);
180     while (sl->i) {
181         wait (&sl->c, &sl->mutex);
182     }
183     sl->i = -1;
184     release(&sl->mutex);
185 }
186
187 void AcquireShared (sharedlock *sl) {
188     acquire(&sl->mutex);
189     while (sl->i < 0) {
190         wait (&sl->c, &sl->mutex);
191     }
192     sl->i++;
193     release(&sl->mutex);
194 }
195
196 void ReleaseShared (sharedlock *sl) {
197     acquire(&sl->mutex);
198     if (!--sl->i)
199         signal (&sl->c, &sl->mutex);
200     release(&sl->mutex);
201 }
202
203 void ReleaseExclusive (sharedlock *sl) {
204     acquire(&sl->mutex);
205     sl->i = 0;
206     broadcast (&sl->c, &sl->mutex);
207     release(&sl->mutex);
208 }
209
210 QUESTIONS:
211 A. There is a starvation problem here. What is it? (Readers can keep
212    writers out if there is a steady stream of readers.)
213 B. How could you use these shared locks to write a cleaner version
214    of the code in item 2., above? (Though note that the starvation
215    properties would be different.)
216

```

Feb 22, 11 2:05

l11-handout.txt

Page 5/7

217 4. Simple deadlock example

```

218
219     T1:
220         acquire(mutexA);
221         acquire(mutexB);
222
223         // do some stuff
224
225         release(mutexB);
226         release(mutexA);
227
228     T2:
229         acquire(mutexB);
230         acquire(mutexA);
231
232         // do some stuff
233
234         release(mutexA);
235         release(mutexB);
236

```

Feb 22, 11 2:05

l11-handout.txt

Page 6/7

237 5. More subtle deadlock example

```

238
239     Let M be a monitor (shared object with methods protected by mutex)
240     Let N be another monitor
241
242     class M {
243     private:
244         Mutex mutex_m;
245
246         // instance of monitor N
247         N another_monitor;
248
249         // Assumption: no other objects in the system hold a pointer
250         // to our "another_monitor"
251
252     public:
253         M();
254         ~M();
255         void methodA();
256         void methodB();
257     };
258
259     class N {
260     private:
261         Mutex mutex_n;
262         Cond cond_n;
263         int navailable;
264
265     public:
266         N();
267         ~N();
268         void* alloc(int nwanted);
269         void free(void*);
270     }
271
272     int
273     N::alloc(int nwanted) {
274         acquire(&mutex_n);
275         while (navailable < nwanted) {
276             wait(&cond_n, &mutex_n);
277         }
278
279         // peel off the memory
280
281         navailable -= nwanted;
282         release(&mutex_n);
283     }
284
285     void
286     N::free(void* returning_mem) {
287         acquire(&mutex_n);
288
289         // put the memory back
290
291         navailable += returning_mem;
292
293         broadcast(&cond_n, &mutex_n);
294
295         release(&mutex_n);
296     }
297
298
299     void
300     M::methodA() {
301
302         acquire(&mutex_m);
303
304         void* new_mem = another_monitor.alloc(int nbytes);
305
306         // do a bunch of stuff using this nice
307         // chunk of memory n allocated for us
308
309         release(&mutex_m);

```

Feb 22, 11 2:05

l11-handout.txt

Page 7/7

```
310     }
311
312     void
313     M::methodB() {
314
315         acquire(&mutex_m);
316
317         // do a bunch of stuff
318
319         another_monitor.free(some_pointer);
320
321         release(&mutex_m);
322     }
323
324     QUESTION: What's the problem?
325
```