

Jan 31, 13 15:08

I05-handout.txt

Page 1/7

```

1 Handout for CS 439
2 Class 5
3 31 January 2013
4
5 1. Producer/consumer [bounded buffer]
6
7     a. Producer/consumer with mutexes (last time)
8
9     b. Producer/consumer [bounded buffer] with mutexes and condition variables
10
11         Mutex mutex;
12         Cond nonempty;
13         Cond nonfull;
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                 acquire(&mutex);
21                 while (count == BUFFER_SIZE)
22                     cond_wait(&nonfull, &mutex);
23
24                 buffer[in] = nextProduced;
25                 in = (in + 1) % BUFFER_SIZE;
26                 count++;
27                 cond_signal(&nonempty, &mutex);
28                 release(&mutex);
29             }
30         }
31
32         void consumer (void *ignored) {
33             for (;;) {
34
35                 acquire(&mutex);
36                 while (count == 0)
37                     cond_wait(&nonempty, &mutex);
38
39                 nextConsumed = buffer[out];
40                 out = (out + 1) % BUFFER_SIZE;
41                 count--;
42                 cond_signal(&nonfull, &mutex);
43                 release(&mutex);
44
45                 /* next line abstractly consumes the item */
46                 consume_item(nextConsumed);
47             }
48         }
49
50
51         Question: why does cond_wait need to both release the mutex and
52         sleep? Why not:
53
54         while (count == BUFFER_SIZE) {
55             release(&mutex);
56             cond_wait(&nonfull);
57             acquire(&mutex);
58         }
59

```

Jan 31, 13 15:08

I05-handout.txt

Page 2/7

```

60     1c. Producer/consumer [bounded buffer] with semaphores
61
62     Semaphore mutex(1);          /* mutex initialized to 1 */
63     Semaphore empty(BUFFER_SIZE); /* start with BUFFER_SIZE empty slots */
64     Semaphore full(0);           /* 0 full slots */
65
66     void producer (void *ignored) {
67         for (;;) {
68             /* next line produces an item and puts it in nextProduced */
69             nextProduced = means_of_production();
70
71             /*
72              * next line diminishes the count of empty slots and
73              * waits if there are no empty slots
74              */
75             sem_down(&empty);
76             sem_down(&mutex); /* get exclusive access */
77
78             buffer[in] = nextProduced;
79             in = (in + 1) % BUFFER_SIZE;
80
81             sem_up(&mutex);
82             sem_up(&full); /* we just increased the # of full slots */
83         }
84     }
85
86     void consumer (void *ignored) {
87         for (;;) {
88
89             /*
90              * next line diminishes the count of full slots and
91              * waits if there are no full slots
92              */
93             sem_down(&full);
94             sem_down(&mutex);
95
96             nextConsumed = buffer[out];
97             out = (out + 1) % BUFFER_SIZE;
98
99             sem_up(&mutex);
100            sem_up(&empty); /* one further empty slot */
101
102            /* next line abstractly consumes the item */
103            consume_item(nextConsumed);
104        }
105    }
106
107    Semaphores *can* (not always) lead to elegant solutions (notice
108    that the code above is fewer lines than 1c) but they are much
109    harder to use.
110
111    The fundamental issue is that semaphores make implicit (counts,
112    conditions, etc.) what is probably best left explicit. Moreover,
113    they *also* implement mutual exclusion.
114
115    For this reason, you should not use semaphores. This example is
116    here mainly for completeness and so you know what a semaphore
117    is. But do not code with them. Solutions that use semaphores in
118    this course will receive no credit.
119

```

Jan 31, 13 15:08

I05-handout.txt

Page 3/7

```

120 2. Example of a monitor: MyBuffer
121
122 // This is pseudocode that is inspired by C++.
123 // Don't take it literally.
124
125 class MyBuffer {
126     public:
127         MyBuffer();
128         ~MyBuffer();
129         void Enqueue(Item);
130         Item = Dequeue();
131     private:
132         int count;
133         int in;
134         int out;
135         Item buffer[BUFFER_SIZE];
136         Mutex* mutex;
137         Cond* nonempty;
138         Cond* nonfull;
139     }
140
141 void
142 MyBuffer::MyBuffer()
143 {
144     in = out = count = 0;
145     mutex = new Mutex;
146     nonempty = new Cond;
147     nonfull = new Cond;
148 }
149
150 void
151 MyBuffer::Enqueue(Item item)
152 {
153     mutex.acquire();
154     while (count == BUFFER_SIZE)
155         cond_wait(&nonfull, &mutex);
156
157     buffer[in] = item;
158     in = (in + 1) % BUFFER_SIZE;
159     ++count;
160     cond_signal(&nonempty, &mutex);
161     mutex.release();
162 }
163
164 Item
165 MyBuffer::Dequeue()
166 {
167     mutex.acquire();
168     while (count == 0)
169         cond_wait(&nonempty, &mutex);
170
171     Item ret = buffer[out];
172     out = (out + 1) % BUFFER_SIZE;
173     --count;
174     cond_signal(&nonfull, &mutex);
175     mutex.release();
176     return ret;
177 }
178

```

Jan 31, 13 15:08

I05-handout.txt

Page 4/7

```

179 int main(int, char**)
180 {
181     MyBuffer buf;
182     int dummy;
183     tid1 = thread_create(producer, &buf);
184     tid2 = thread_create(consumer, &buf);
185
186     // never reach this point
187     thread_join(tid1);
188     thread_join(tid2);
189     return -1;
190 }
191
192 void producer(void* buf)
193 {
194     MyBuffer* sharedbuf = reinterpret_cast<MyBuffer*>(buf);
195     for (;;) {
196         /* next line produces an item and puts it in nextProduced */
197         Item nextProduced = means_of_production();
198         sharedbuf->Enqueue(nextProduced);
199     }
200 }
201
202 void consumer(void* buf)
203 {
204     MyBuffer* sharedbuf = reinterpret_cast<MyBuffer*>(buf);
205     for (;;) {
206         Item nextConsumed = sharedbuf->Dequeue();
207
208         /* next line abstractly consumes the item */
209         consume_item(nextConsumed);
210     }
211 }
212
213 Key point: *Threads* (the producer and consumer) are separate from
214 *shared object* (MyBuffer). The synchronization happens in the
215 shared object.
216
217 3. Readers/writers
218
219 state variables:
220     AR = 0; // # active readers
221     AW = 0; // # active writers
222     WR = 0; // # waiting readers
223     WW = 0; // # waiting writers
224
225     Condition okToRead = NIL;
226     Condition okToWrite = NIL;
227     Mutex mutex = FREE;
228
229 Database::read() {
230     startRead(); // first, check self into the system
231     Access Data
232     doneRead(); // check self out of system
233 }
234
235 Database::startRead() {
236     acquire(&mutex);
237     while((AW + WW) > 0){
238         WR++;
239         wait(&okToRead, &mutex);
240         WR--;
241     }
242     AR++;
243     release(&mutex);
244 }
245
246 Database::doneRead() {
247     acquire(&mutex);
248     AR--;
249     if (AR == 0 && WW > 0) { // if no other readers still
250         signal(&okToWrite, &mutex); // active, wake up writer
251     }

```

Jan 31, 13 15:08

I05-handout.txt

Page 5/7

```

252     release(&mutex);
253 }
254
255 Database::write(){ // symmetrical
256     startWrite(); // check in
257     Access Data
258     doneWrite(); // check out
259 }
260
261 Database::startWrite() {
262     acquire(&mutex);
263     while ((AW + AR) > 0) { // check if safe to write.
264         // if any readers or writers, wait
265         WW++;
266         wait(&okToWrite, &mutex);
267         WW--;
268     }
269     AW++;
270     release(&mutex);
271 }
272
273 Database::doneWrite() {
274     acquire(&mutex);
275     AW--;
276     if (WW > 0) {
277         signal(&okToWrite, &mutex); // give priority to writers
278     } else if (WR > 0) {
279         broadcast(&okToRead, &mutex);
280     }
281     release(&mutex);
282 }
283
284 NOTE: what is the starvation problem here?
285

```

Jan 31, 13 15:08

I05-handout.txt

Page 6/7

```

286 4. Shared locks
287
288 struct sharedlock {
289     int i;
290     Mutex mutex;
291     Cond c;
292 };
293
294 void AcquireExclusive (sharedlock *sl) {
295     acquire(&sl->mutex);
296     while (sl->i) {
297         wait (&sl->c, &sl->mutex);
298     }
299     sl->i = -1;
300     release(&sl->mutex);
301 }
302
303 void AcquireShared (sharedlock *sl) {
304     acquire(&sl->mutex);
305     while (sl->i < 0) {
306         wait (&sl->c, &sl->mutex);
307     }
308     sl->i++;
309     release(&sl->mutex);
310 }
311
312 void ReleaseShared (sharedlock *sl) {
313     acquire(&sl->mutex);
314     if (!--sl->i)
315         signal (&sl->c, &sl->mutex);
316     release(&sl->mutex);
317 }
318
319 void ReleaseExclusive (sharedlock *sl) {
320     acquire(&sl->mutex);
321     sl->i = 0;
322     broadcast (&sl->c, &sl->mutex);
323     release(&sl->mutex);
324 }
325
326 QUESTIONS:
327 A. There is a starvation problem here. What is it? (Readers can keep
328     writers out if there is a steady stream of readers.)
329 B. How could you use these shared locks to write a cleaner version
330     of the code in item 5., above? (Though note that the starvation
331     properties would be different.)
332

```

Jan 31, 13 15:08

I05-handout.txt

Page 7/7

```
332
333 5. Simple deadlock example
334
335     T1:
336         acquire(mutexA);
337         acquire(mutexB);
338
339         // do some stuff
340
341         release(mutexB);
342         release(mutexA);
343
344     T2:
345         acquire(mutexB);
346         acquire(mutexA);
347
348         // do some stuff
349
350         release(mutexA);
351         release(mutexB);
```