

```

1 Handout for CS 439
2 Class 4
3 29 January 2013
4
5 0. Recall race condition examples from last time:
6
7 --the threads calling f()/g()
8
9 --the threads simultaneously enqueueing to the head of a linked list
10
11 1. Producer/consumer example:
12
13 /*
14 "buffer" stores BUFFER_SIZE items
15 "count" is number of used slots. a variable that lives in memory
16 "out" is next empty buffer slot to fill (if any)
17 "in" is oldest filled slot to consume (if any)
18 */
19
20 void producer (void *ignored) {
21     for (;;) {
22         /* next line produces an item and puts it in nextProduced */
23         nextProduced = means_of_production();
24         while (count == BUFFER_SIZE)
25             ; // do nothing
26         buffer [in] = nextProduced;
27         in = (in + 1) % BUFFER_SIZE;
28         count++;
29     }
30 }
31
32 void consumer (void *ignored) {
33     for (;;) {
34         while (count == 0)
35             ; // do nothing
36         nextConsumed = buffer[out];
37         out = (out + 1) % BUFFER_SIZE;
38         count--;
39         /* next line abstractly consumes the item */
40         consume_item(nextConsumed);
41     }
42 }
43
44 /*
45 what count++ probably compiles to:
46 reg1 <-- count      # load
47 reg1 <-- reg1 + 1    # increment register
48 count <-- reg1      # store
49
50 what count-- could compile to:
51 reg2 <-- count      # load
52 reg2 <-- reg2 - 1   # decrement register
53 count <-- reg2      # store
54 */
55
56 What happens if we get the following interleaving?
57
58 reg1 <-- count
59 reg1 <-- reg1 + 1
60 reg2 <-- count
61 reg2 <-- reg2 - 1
62 count <-- reg1
63 count <-- reg2
64

```

```

65
66 2. Some other examples. What is the point of these?
67
68 [From S.V. Adve and K. Gharachorloo, IEEE Computer, December 1996,
69 66-76. http://rsim.cs.uiuc.edu/~sadve/Publications/computer96.pdf]
70
71 a. Can both "critical sections" run?
72
73     int flag1 = 0, flag2 = 0;
74
75     int main () {
76         tid id = thread_create (p1, NULL);
77         p2 (); thread_join (id);
78     }
79
80     void p1 (void *ignored) {
81         flag1 = 1;
82         if (!flag2) {
83             critical_section_1 ();
84         }
85     }
86
87     void p2 (void *ignored) {
88         flag2 = 1;
89         if (!flag1) {
90             critical_section_2 ();
91         }
92     }
93
94 b. Can use() be called with value 0, if p2 and p1 run concurrently?
95
96     int data = 0, ready = 0;
97
98     void p1 () {
99         data = 2000;
100        ready = 1;
101    }
102    int p2 () {
103        while (!ready) {}
104        use(data);
105    }
106
107 c. Can use() be called with value 0?
108
109     int a = 0, b = 0;
110
111     void p1 (void *ignored) { a = 1; }
112
113     void p2 (void *ignored) {
114         if (a == 1)
115             b = 1;
116     }
117
118     void p3 (void *ignored) {
119         if (b == 1)
120             use (a);
121     }
122

```

Jan 29, 13 15:06

I04-handout.txt

Page 3/8

```

123 3. Protecting the linked list.....
124
125     Mutex list_mutex;
126
127     insert(int data) {
128         List_elem* l = new List_elem;
129         l->data = data;
130
131         acquire(&list_mutex);
132
133         l->next = head;    // A
134         head = l;        // B
135
136         release(&list_mutex);
137     }
138

```

Jan 29, 13 15:06

I04-handout.txt

Page 4/8

```

139 4. Producer/consumer revisited [also known as bounded buffer]
140
141 4a. Producer/consumer [bounded buffer] with mutexes
142
143     Mutex mutex;
144
145     void producer (void *ignored) {
146         for (;;) {
147             /* next line produces an item and puts it in nextProduced */
148             nextProduced = means_of_production();
149
150             acquire(&mutex);
151             while (count == BUFFER_SIZE) {
152                 release(&mutex);
153                 yield(); /* or schedule() */
154                 acquire(&mutex);
155             }
156
157             buffer [in] = nextProduced;
158             in = (in + 1) % BUFFER_SIZE;
159             count++;
160             release(&mutex);
161         }
162     }
163
164     void consumer (void *ignored) {
165         for (;;) {
166
167             acquire(&mutex);
168             while (count == 0) {
169                 release(&mutex);
170                 yield(); /* or schedule() */
171                 acquire(&mutex);
172             }
173
174             nextConsumed = buffer[out];
175             out = (out + 1) % BUFFER_SIZE;
176             count--;
177             release(&mutex);
178
179             /* next line abstractly consumes the item */
180             consume_item(nextConsumed);
181         }
182     }
183

```

Jan 29, 13 15:06

I04-handout.txt

Page 5/8

```

184
185 4b. Producer/consumer [bounded buffer] with mutexes and condition variables
186
187     Mutex mutex;
188     Cond nonempty;
189     Cond nonfull;
190
191     void producer (void *ignored) {
192         for (;;) {
193             /* next line produces an item and puts it in nextProduced */
194             nextProduced = means_of_production();
195
196             acquire(&mutex);
197             while (count == BUFFER_SIZE)
198                 cond_wait(&nonfull, &mutex);
199
200             buffer [in] = nextProduced;
201             in = (in + 1) % BUFFER_SIZE;
202             count++;
203             cond_signal(&nonempty, &mutex);
204             release(&mutex);
205         }
206     }
207
208     void consumer (void *ignored) {
209         for (;;) {
210
211             acquire(&mutex);
212             while (count == 0)
213                 cond_wait(&nonempty, &mutex);
214
215             nextConsumed = buffer[out];
216             out = (out + 1) % BUFFER_SIZE;
217             count--;
218             cond_signal(&nonfull, &mutex);
219             release(&mutex);
220
221             /* next line abstractly consumes the item */
222             consume_item(nextConsumed);
223         }
224     }
225
226     Question: why does cond_wait need to both release the mutex and
227     sleep? Why not:
228
229     while (count == BUFFER_SIZE) {
230         release(&mutex);
231         cond_wait(&nonfull);
232         acquire(&mutex);
233     }
234
235

```

Jan 29, 13 15:06

I04-handout.txt

Page 6/8

```

236 4c. Producer/consumer [bounded buffer] with semaphores
237
238     Semaphore mutex(1); /* mutex initialized to 1 */
239     Semaphore empty(BUFFER_SIZE); /* start with BUFFER_SIZE empty slots */
240     Semaphore full(0); /* 0 full slots */
241
242     void producer (void *ignored) {
243         for (;;) {
244             /* next line produces an item and puts it in nextProduced */
245             nextProduced = means_of_production();
246
247             /*
248              * next line diminishes the count of empty slots and
249              * waits if there are no empty slots
250              */
251             sem_down(&empty);
252             sem_down(&mutex); /* get exclusive access */
253
254             buffer [in] = nextProduced;
255             in = (in + 1) % BUFFER_SIZE;
256
257             sem_up(&mutex);
258             sem_up(&full); /* we just increased the # of full slots */
259         }
260     }
261
262     void consumer (void *ignored) {
263         for (;;) {
264
265             /*
266              * next line diminishes the count of full slots and
267              * waits if there are no full slots
268              */
269             sem_down(&full);
270             sem_down(&mutex);
271
272             nextConsumed = buffer[out];
273             out = (out + 1) % BUFFER_SIZE;
274
275             sem_up(&mutex);
276             sem_up(&empty); /* one further empty slot */
277
278             /* next line abstractly consumes the item */
279             consume_item(nextConsumed);
280         }
281     }
282
283     Semaphores *can* (not always) lead to elegant solutions (notice
284     that the code above is fewer lines than 1c) but they are much
285     harder to use.
286
287     The fundamental issue is that semaphores make implicit (counts,
288     conditions, etc.) what is probably best left explicit. Moreover,
289     they *also* implement mutual exclusion.
290
291     For this reason, you should not use semaphores. This example is
292     here mainly for completeness and so you know what a semaphore
293     is. But do not code with them. Solutions that use semaphores in
294     this course will receive no credit.
295

```

Jan 29, 13 15:06

I04-handout.txt

Page 7/8

```

296 5. Example of a monitor: MyBuffer
297
298 // This is pseudocode that is inspired by C++.
299 // Don't take it literally.
300
301 class MyBuffer {
302     public:
303         MyBuffer();
304         ~MyBuffer();
305         void Enqueue(Item);
306         Item = Dequeue();
307     private:
308         int count;
309         int in;
310         int out;
311         Item buffer[BUFFER_SIZE];
312         Mutex* mutex;
313         Cond* nonempty;
314         Cond* nonfull;
315     }
316
317 void
318 MyBuffer::MyBuffer()
319 {
320     in = out = count = 0;
321     mutex = new Mutex;
322     nonempty = new Cond;
323     nonfull = new Cond;
324 }
325
326 void
327 MyBuffer::Enqueue(Item item)
328 {
329     mutex.acquire();
330     while (count == BUFFER_SIZE)
331         cond_wait(&nonfull, &mutex);
332
333     buffer[in] = item;
334     in = (in + 1) % BUFFER_SIZE;
335     ++count;
336     cond_signal(&nonempty, &mutex);
337     mutex.release();
338 }
339
340 Item
341 MyBuffer::Dequeue()
342 {
343     mutex.acquire();
344     while (count == 0)
345         cond_wait(&nonempty, &mutex);
346
347     Item ret = buffer[out];
348     out = (out + 1) % BUFFER_SIZE;
349     --count;
350     cond_signal(&nonfull, &mutex);
351     mutex.release();
352     return ret;
353 }
354

```

Jan 29, 13 15:06

I04-handout.txt

Page 8/8

```

355 int main(int, char**)
356 {
357     MyBuffer buf;
358     int dummy;
359     tid1 = thread_create(producer, &buf);
360     tid2 = thread_create(consumer, &buf);
361
362     // never reach this point
363     thread_join(tid1);
364     thread_join(tid2);
365     return -1;
366 }
367
368 void producer(void* buf)
369 {
370     MyBuffer* sharedbuf = reinterpret_cast<MyBuffer*>(buf);
371     for (;;) {
372         /* next line produces an item and puts it in nextProduced */
373         Item nextProduced = means_of_production();
374         sharedbuf->Enqueue(nextProduced);
375     }
376 }
377
378 void consumer(void* buf)
379 {
380     MyBuffer* sharedbuf = reinterpret_cast<MyBuffer*>(buf);
381     for (;;) {
382         Item nextConsumed = sharedbuf->Dequeue();
383
384         /* next line abstractly consumes the item */
385         consume_item(nextConsumed);
386     }
387 }
388
389 Key point: *Threads* (the producer and consumer) are separate from
390 *shared object* (MyBuffer). The synchronization happens in the
391 shared object.

```