

Feb 07, 12 15:10

I07-handout.txt

Page 1/10

```

1 Handout for CS 372H
2 Class 7
3 7 February 2012
4
5 1. Example to illustrate interleavings: say that thread A executes f()
6 and thread B executes g(). (Here, we are using the term "thread"
7 abstractly. This example applies to any of the approaches that fall
8 under the word "thread".)
9
10 a.
11
12     int x;
13
14     f() { x = 1; }
15
16     g() { x = 2; }
17
18     What are possible values of x after A has executed f() and B has
19     executed g()?
20
21 b.
22
23     int y = 12;
24
25     f() { x = y + 1; }
26     g() { y = y * 2; }
27
28     What are the possible values of x?
29
30 c.
31
32     int x = 0;
33     f() { x = x + 1; }
34     g() { x = x + 2; }
35
36     What are the possible values of x?
37
38 2. Linked list example
39
40     struct List_elem {
41         int data;
42         struct List_elem* next;
43     };
44
45     List_elem* head = 0;
46
47     insert(int data) {
48         List_elem* l = new List_elem;
49         l->data = data;
50         l->next = head;
51         head = l;
52     }
53
54     What happens if two threads execute insert() at once and we get the
55     following interleaving?
56
57     thread 1: l->next = head
58     thread 2: l->next = head
59     thread 2: head = l;
60     thread 1: head = l;

```

Feb 07, 12 15:10

I07-handout.txt

Page 2/10

```

61 3. Producer/consumer example:
62
63     /*
64     "buffer" stores BUFFER_SIZE items
65     "count" is number of used slots. a variable that lives in memory
66     "out" is next empty buffer slot to fill (if any)
67     "in" is oldest filled slot to consume (if any)
68     */
69
70     void producer (void *ignored) {
71         for (;;) {
72             /* next line produces an item and puts it in nextProduced */
73             nextProduced = means_of_production();
74             while (count == BUFFER_SIZE)
75                 ; // do nothing
76             buffer [in] = nextProduced;
77             in = (in + 1) % BUFFER_SIZE;
78             count++;
79         }
80     }
81
82     void consumer (void *ignored) {
83         for (;;) {
84             while (count == 0)
85                 ; // do nothing
86             nextConsumed = buffer[out];
87             out = (out + 1) % BUFFER_SIZE;
88             count--;
89             /* next line abstractly consumes the item */
90             consume_item(nextConsumed);
91         }
92     }
93
94     /*
95     what count++ probably compiles to:
96     reg1 <-- count      # load
97     reg1 <-- reg1 + 1   # increment register
98     count <-- reg1     # store
99
100     what count-- could compile to:
101     reg2 <-- count     # load
102     reg2 <-- reg2 - 1  # decrement register
103     count <-- reg2    # store
104
105     */
106     What happens if we get the following interleaving?
107
108     reg1 <-- count
109     reg1 <-- reg1 + 1
110     reg2 <-- count
111     reg2 <-- reg2 - 1
112     count <-- reg1
113     count <-- reg2
114

```

```

115
116 4. Some other examples. What is the point of these?
117
118 [From S.V. Adve and K. Gharachorloo, IEEE Computer, December 1996,
119 66-76. http://rsim.cs.uiuc.edu/~sadve/Publications/computer96.pdf]
120
121 a. Can both "critical sections" run?
122
123     int flag1 = 0, flag2 = 0;
124
125     int main () {
126         tid id = thread_create (p1, NULL);
127         p2 (); thread_join (id);
128     }
129
130     void p1 (void *ignored) {
131         flag1 = 1;
132         if (!flag2) {
133             critical_section_1 ();
134         }
135     }
136
137     void p2 (void *ignored) {
138         flag2 = 1;
139         if (!flag1) {
140             critical_section_2 ();
141         }
142     }
143
144 b. Can use() be called with value 0, if p2 and p1 run concurrently?
145
146     int data = 0, ready = 0;
147
148     void p1 () {
149         data = 2000;
150         ready = 1;
151     }
152     int p2 () {
153         while (!ready) {}
154         use(data);
155     }
156
157 c. Can use() be called with value 0?
158
159     int a = 0, b = 0;
160
161     void p1 (void *ignored) { a = 1; }
162
163     void p2 (void *ignored) {
164         if (a == 1)
165             b = 1;
166     }
167
168     void p3 (void *ignored) {
169         if (b == 1)
170             use (a);
171     }
172
173 d.
174 /* keyword "register" tells compiler to place the variable in a
175 register, not on the stack. So f, g are local to each thread. */
176
177     int flag1 = 0, flag2 = 0;
178
179     int p1 (void *ignored)         int p2 (void *ignored)
180     {                               {
181     { register int f, g;             { register int f, g;
182     flag1 = 1;                       flag2 = 1;
183     f = flag1;                         f = flag2;
184     g = flag2;                         g = flag1;
185     return 2*f + g;                   return 2*f + g;
186     }                               }
187

```

```

188 5. Protecting the linked list.....
189
190     Lock list_lock;
191
192     insert(int data) {
193         List_elem* l = new List_elem;
194         l->data = data;
195
196         acquire(&list_lock);
197
198         l->next = head;           // A
199         head = l;                 // B
200
201         release(&list_lock);
202     }
203
204 6. How can we implement list_lock, acquire(), and release()?
205
206 6a. Here is A BADLY BROKEN implementation:
207
208     struct Lock {
209         int locked;
210     }
211
212     void [BROKEN] acquire(Lock *lock) {
213         while (1) {
214             if (lock->locked == 0) { // C
215                 lock->locked = 1;   // D
216                 break;
217             }
218         }
219     }
220
221     void release (Lock *lock) {
222         lock->locked = 0;
223     }
224
225     What's the problem? Two acquire()s on the same lock on different
226 CPUs might both execute line C, and then both execute D. Then
227 both will think they have acquired the lock. This is the same
228 kind of race that we were trying to eliminate in insert(). But
229 we have made a little progress: now we only need a way to
230 prevent interleaving in one place (acquire()), not for many
231 arbitrary complex sequences of code.
232

```

```

233 6b. Here's a way that is correct but only sometimes appropriate:
234 Use an atomic instruction on the CPU. For example, on the x86,
235 doing
236     "xchg addr, %eax"
237 does the following:
238
239 (i) freeze all CPUs' memory activity for address addr
240 (ii) temp = *addr
241 (iii) *addr = %eax
242 (iv) %eax = temp
243 (v) un-freeze memory activity
244
245 /* pseudocode */
246 int xchg_val(addr, value) {
247     %eax = value;
248     xchg (*addr), %eax
249 }
250
251 struct Lock {
252     int locked;
253 }
254
255 /* bare-bones version of acquire */
256 void acquire (Lock *lock) {
257     pushcli(); /* what does this do? */
258     while (1) {
259         if (xchg_val(&lock->locked, 1) == 0)
260             break;
261     }
262 }
263
264 /* optimization in acquire; call xchg_val() less frequently */
265 void acquire(Lock* lock) {
266     pushcli();
267     while (xchg_val(&lock->locked, 1) == 1) {
268         while (lock->locked) ;
269     }
270 }
271
272 void release(Lock *lock){
273     xchg_val(&lock->locked, 0);
274     popcli(); /* what does this do? */
275 }
276
277 The above is called a *spinlock* because acquire() spins.
278
279 The spinlock above is great for some things, not so great for
280 others. The main problem is that it *busy waits*: it spins,
281 chewing up CPU cycles. Sometimes this is what we want (e.g., if
282 the cost of going to sleep is greater than the cost of spinning
283 for a few cycles waiting for another thread or process to
284 relinquish the spinlock). But sometimes this is not at all what we
285 want (e.g., if the lock would be held for a while: in those
286 cases, the CPU waiting for the lock would waste cycles spinning
287 instead of running some other thread or process).
288
289
290 6c. Here's an object that does not involve busy waiting; it can work
291 as the list_lock mentioned above. Note: the "threads" here
292 can be user-level threads, kernel threads, or threads-inside-kernel.
293 The concept is the same in all cases.
294
295 struct Mutex {
296     bool is_held; /* true if mutex held */
297     thread_id owner; /* thread holding mutex, if locked */
298     thread_list waiters; /* queue of thread TCBS */
299     Lock wait_lock; /* as in 6b */
300 }
301
302 Now, instead of acquire(&list_lock) and release(&list_lock) as
303 above, we'd write, mutex_acquire(&list_mutex) and
304 mutex_release(&list_mutex). The implementation of the latter two
305 would be something like this:

```

```

306     void mutex_acquire(Mutex *m) {
307
308         acquire(&m->wait_lock); /* we spin to acquire wait_lock */
309         while (m->is_held) { /* someone else has the mutex */
310             m->waiters.insert(current_thread)
311             release(&m->wait_lock);
312             schedule(); /* run a thread that is on the ready list */
313             acquire(&m->wait_lock); /* we spin again */
314         }
315         m->is_held = true; /* we now hold the mutex */
316         m->owner = self;
317         release(&m->wait_lock);
318     }
319
320     void mutex_release(Mutex *m) {
321
322         acquire(&m->wait_lock); /* we spin to acquire wait_lock */
323         m->is_held = false;
324         m->owner = 0;
325         wake_up_a_waiter(m->waiters); /* select and run a waiter */
326         release(&m->wait_lock);
327     }
328
329 }
330
331 [Please let me (MW) know if you see bugs in the above.]
332
333 7. NOTE: Unfortunately, insert() with these locks is correct only if
334 there are some constraints on the order in which the CPU carries out
335 memory reads and writes. For example, if insert() were executed so that
336 the read at A appeared to another processor (and to memory) to be
337 executed before the acquire(), then insert() would be incorrect even
338 with locks.
339
340 How do we get the required guarantee? Answer: by ensuring that neither
341 the programmer nor the processor reorders instructions with respect to
342 the acquire().
343
344 8. Terminology
345
346 To avoid confusion, we will use the following terminology in this
347 course (you will hear other terminology elsewhere):
348
349 --A "lock" is an abstract object that provides mutual exclusion
350
351 --A "spinlock" is a lock that works by busy waiting, as in 6b
352
353 --A "mutex" is a lock that works by having a "waiting" queue and
354 then protecting that waiting queue with atomic hardware
355 instructions, as in 6c. The most natural way to "use the hardware"
356 is with a spinlock, but there are others, such as turning off
357 interrupts, which works if we're on a single CPU machine.
358
359

```

Feb 07, 12 15:10

I07-handout.txt

Page 7/10

```

360
361 9. Producer/consumer example [also known as bounded buffer]
362
363 9a. buggy implementation
364
365 /*
366 "buffer" stores BUFFER_SIZE items
367 "count" is number of used slots. a variable that lives in memory
368 "out" is next empty buffer slot to fill (if any)
369 "in" is oldest filled slot to consume (if any)
370 */
371
372 void producer (void *ignored) {
373     for (;;) {
374         /* next line produces an item and puts it in nextProduced */
375         nextProduced = means_of_production();
376         while (count == BUFFER_SIZE)
377             ; // do nothing
378         buffer [in] = nextProduced;
379         in = (in + 1) % BUFFER_SIZE;
380         count++;
381     }
382 }
383
384 void consumer (void *ignored) {
385     for (;;) {
386         while (count == 0)
387             ; // do nothing
388         nextConsumed = buffer[out];
389         out = (out + 1) % BUFFER_SIZE;
390         count--;
391         /* next line abstractly consumes the item */
392         consume_item(nextConsumed);
393     }
394 }
395
396 --Review: what's the problem?
397 --Answer: count++ and count-- might compile to, respectively:
398
399     reg1 <-- count      # load
400     reg1 <-- reg1 + 1   # increment register
401     count <-- reg1     # store
402
403     reg2 <-- count      # load
404     reg2 <-- reg2 - 1   # decrement register
405     count <-- reg2     # store
406
407 --Review: why not use instructions like "addl $0x1, _count"?
408 --Answer: not atomic if there are multiple CPUs.
409
410 --Review: so why not use "LOCK addl $0x1, _count"?
411 --Answer: we could do that here, but LOCK won't save us every time
412
413 --Review: so use general-purpose approach to protecting
414 critical sections: locks (or mutexes).
415
416

```

Feb 07, 12 15:10

I07-handout.txt

Page 8/10

```

417
418 9b. Producer/consumer [bounded buffer] using mutexes
419
420     Mutex mutex;
421
422     void producer (void *ignored) {
423         for (;;) {
424             /* next line produces an item and puts it in nextProduced */
425             nextProduced = means_of_production();
426
427             acquire(&mutex);
428             while (count == BUFFER_SIZE) {
429                 release(&mutex);
430                 yield(); /* or schedule() */
431                 acquire(&mutex);
432             }
433
434             buffer [in] = nextProduced;
435             in = (in + 1) % BUFFER_SIZE;
436             count++;
437             release(&mutex);
438         }
439     }
440
441     void consumer (void *ignored) {
442         for (;;) {
443
444             acquire(&mutex);
445             while (count == 0) {
446                 release(&mutex);
447                 yield(); /* or schedule() */
448                 acquire(&mutex);
449             }
450
451             nextConsumed = buffer[out];
452             out = (out + 1) % BUFFER_SIZE;
453             count--;
454             release(&mutex);
455
456             /* next line abstractly consumes the item */
457             consume_item(nextConsumed);
458         }
459     }
460

```

Feb 07, 12 15:10

I07-handout.txt

Page 9/10

```

461
462 9c. Producer/consumer [bounded buffer] using mutexes and condition
463 variables
464
465     Mutex mutex;
466     Cond nonempty;
467     Cond nonfull;
468
469     void producer (void *ignored) {
470         for (;;) {
471             /* next line produces an item and puts it in nextProduced */
472             nextProduced = means_of_production();
473
474             acquire(&mutex);
475             while (count == BUFFER_SIZE)
476                 cond_wait(&nonfull, &mutex);
477
478             buffer [in] = nextProduced;
479             in = (in + 1) % BUFFER_SIZE;
480             count++;
481             cond_signal(&nonempty, &mutex);
482             release(&mutex);
483         }
484     }
485
486     void consumer (void *ignored) {
487         for (;;) {
488
489             acquire(&mutex);
490             while (count == 0)
491                 cond_wait(&nonempty, &mutex);
492
493             nextConsumed = buffer[out];
494             out = (out + 1) % BUFFER_SIZE;
495             count--;
496             cond_signal(&nonfull, &mutex);
497             release(&mutex);
498
499             /* next line abstractly consumes the item */
500             consume_item(nextConsumed);
501         }
502     }
503
504     Question: why does cond_wait need to both release the mutex and
505     sleep? Why not:
506
507         while (count == BUFFER_SIZE) {
508             release(&mutex);
509             cond_wait(&nonfull);
510             acquire(&mutex);
511         }
512
513

```

Feb 07, 12 15:10

I07-handout.txt

Page 10/10

```

514 9d. Producer/consumer [bounded buffer] with semaphores
515
516     Semaphore mutex(1); /* mutex initialized to 1 */
517     Semaphore empty(BUFFER_SIZE); /* start with BUFFER_SIZE empty slots */
518     Semaphore full(0); /* 0 full slots */
519
520     void producer (void *ignored) {
521         for (;;) {
522             /* next line produces an item and puts it in nextProduced */
523             nextProduced = means_of_production();
524
525             /*
526              * next line diminishes the count of empty slots and
527              * waits if there are no empty slots
528              */
529             sem_down(&empty);
530             sem_down(&mutex); /* get exclusive access */
531
532             buffer [in] = nextProduced;
533             in = (in + 1) % BUFFER_SIZE;
534
535             sem_up(&mutex);
536             sem_up(&full); /* we just increased the # of full slots */
537         }
538     }
539
540     void consumer (void *ignored) {
541         for (;;) {
542
543             /*
544              * next line diminishes the count of full slots and
545              * waits if there are no full slots
546              */
547             sem_down(&full);
548             sem_down(&mutex);
549
550             nextConsumed = buffer[out];
551             out = (out + 1) % BUFFER_SIZE;
552
553             sem_up(&mutex);
554             sem_up(&empty); /* one further empty slot */
555
556             /* next line abstractly consumes the item */
557             consume_item(nextConsumed);
558         }
559     }
560
561     Semaphores *can* (not always) lead to elegant solutions (notice
562     that the code above is fewer lines than 1c) but they are much
563     harder to use.
564
565     The fundamental issue is that semaphores make implicit (counts,
566     conditions, etc.) what is probably best left explicit. Moreover,
567     they *also* implement mutual exclusion.
568
569     For this reason, you should not use semaphores. This example is
570     here mainly for completeness and so you know what a semaphore
571     is. But do not code with them. Solutions that use semaphores in
572     this course will receive no credit.
573

```