Fe	eb 17, 11 16:22	I10-handout.txt	Page 1/9	Feb 17	7, 11 16	6:22	l10-handout.txt	Page 2/9
1 2 3	Handout for CS 372H Class 10 17 February 2011			49 50 51	2b. H t	Here's Use an doing	a way that is correct but only sometimes appropriate: atomic instruction on the CPU. For example, on the x86	,
4 5	1. Protecting the linked	list		52 53	c	does th	"xchg addr, %eax" e following:	
6 7 8	Lock list_lock;			54 55 56		(i) f (ii) t	reeze all CPUs' memory activity for address addr emp = *addr	
9 10 11	insert(int data) List_elem* l l->data = da	{ = new List_elem; ta;		57 58 59		(iii) * (iv) % (v) u	addr = %eax eax = temp n-freeze memory activity	
12 13 14	acquire(&lis	t_lock);		60 61 62	,	/* pseu int xch	docode */ g_val(addr, value) {	
15 16 17	l->next = he head = l;	ad; // A // B		63 64 65		%ea: xch }	x = value; g (*addr), %eax	
18 19	release(&lis	t_lock);		66 67	ŝ	struct	Lock {	
20	2. How can we implement	list_lock, acquire(), and release()?		69		}		
23 23 24 25	2a. Here is A BADLY :	BROKEN implementation:		70 71 72 73	7	/* bare void ac pushc	-bones version of acquire */ quire (Lock *lock) { li(); /* what does this do? */	
26 27 28	int locked; }			74 75 76		while if b	(1) { (xchg_val(&lock->locked, 1) == 0) reak:	
29 30 31	void [BROKEN] ac while (1) { if (lock->lo	quire(Lock *lock) { cked == 0) { // C		77 78 79		}		
32 33 34	lock->lock break; }	ed = 1; // D		80 81 82	7	/* optin void ac pus	<pre>mization in acquire; call xchg_val() less frequently * quire(Lock* lock) { hcli();</pre>	/
35 36 37	}			83 84 85		whi }	<pre>le (xchg_val(&lock->locked, 1) == 1) { while (lock->locked) ;</pre>	
38 39	void release (Lo lock->locked =	ck *lock) { 0;		86 87		}		
40 41 42	What's the proble	em? Two acquire()s on the same lock or	different CPUs	88 89 90	1	void re xchg popc	:lease(Lock *lock){ _val(&lock->locked, 0); li(); /* what does this do? */	
43 44 45	might both execu think they have were trying to e	might both execute line C, and then both execute D. Then think they have acquired the lock. This is the same kind were trying to eliminate in insert(). But we have made a	ind of race we le a little	91 92 93		} The abo	ve is called a *spinlock* because acquire() spins.	
46 47 48	(acquire()), not	for many arbitrary complex sequences	of code.	94 95 96	τ	Ünfortu CPU car	nately, insert() with these locks is correct only if e ries out memory reads and writes in program order. Fo	ach r
				97 98 99	e t	example that it would b	:, if the CPU were to execute insert() out of order so did the read at A before the acquire(), then insert() is incorrect even with locks. Many modern processors	
				100 101 102	6	execute So we m "SFENCE	<pre>: memory operations out of order to increase performanc way have to use special instructions ("lock", "LFENCE", ", "MFENCE") to tell the CPU not to re-order memory</pre>	e!
				103 104 105	a t	operati also ge the ord	ons past acquire()s and release()s. The compiler may merate instructions in orders that don't correspond to ler of the source code lines, so we have to worry about	
				106 107 108	t	chat to volatil	o. One way around this is to make the asm instructions e.	
				109 110 111 112 113	1 0 1 2	Moral o concurr how loa archite	of the above paragraph: if you're implementing a ency primitive, read the processor's documentation abo ds and stores get sequenced (chapter 8 in current ecture manual).	ut
				114 115 116	(The spin others. chewing	nlock above is great for some things, not so great for The main problem is that it *busy waits*: it spins, up CPU cycles. Sometimes this is what we want (e.g.,	if
				117 118 119 120 121	ו נ ע ע ע	for a	ew cycles waiting for another thread or process to ish the spinlock). But sometimes this is not at all wh .g., if the lock would be held for a while: in those the CPU waiting for the lock would waste cycles spinni	at we

```
110-handout.txt
                                                                                                                                       110-handout.txt
Feb 17, 11 16:22
                                                                                Page 3/9
                                                                                                 Feb 17, 11 16:22
                                                                                                                                                                                 Page 4/9
            instead of running some other thread or process).
122
                                                                                                 185
                                                                                                    4. Producer/consumer example [also known as bounded buffer]
123
                                                                                                 186
124
                                                                                                 187
        2c. Here's an object that does not involve busy waiting; it can work as the list_lock mentioned in \#1, above. Note: the "threads" here
                                                                                                        4a. buggy implementation (from last time)
125
                                                                                                 188
126
                                                                                                 189
127
        can be user-level threads, kernel threads, or threads-inside-kernel.
                                                                                                 190
                                                                                                          "buffer" stores BUFFER SIZE items
128
        The concept is the same in all cases.
                                                                                                 191
                                                                                                         "count" is number of used slots. a variable that lives in memory
129
                                                                                                 192
            struct Mutex {
                                                                                                          "out" is next empty buffer slot to fill (if any)
130
                                                                                                 193
                 bool is_held;
                                           /* true if mutex held */
                                                                                                         "in" is oldest filled slot to consume (if any)
131
                                                                                                 194
                 thread_id owner;
                                           /* thread holding mutex, if locked */
132
                                                                                                 195
133
                 thread_list waiters;
                                           /* queue of thread TCBs */
                                                                                                 196
                Lock wait lock;
                                           /* as in 2b */
                                                                                                          void producer (void *ignored) {
134
                                                                                                 197
                                                                                                 198
                                                                                                               for (;;) {
135
                                                                                                 199
                                                                                                                   /* next line produces an item and puts it in nextProduced */
136
137
            Now, instead of acquire(&list_lock) and release(&list_lock) as
                                                                                                 200
                                                                                                                   nextProduced = means_of_production();
            in #1, we'd write, mutex acquire(&list mutex) and
                                                                                                                   while (count == BUFFER SIZE)
138
                                                                                                 201
            mutex_release(&list_mutex). The implementation of the latter two
                                                                                                                       ; // do nothing
139
                                                                                                 202
140
            would be something like this:
                                                                                                 203
                                                                                                                   buffer [in] = nextProduced;
                                                                                                                   in = (in + 1) % BUFFER_SIZE;
141
                                                                                                 204
            void mutex_acquire(Mutex *m) {
                                                                                                 205
                                                                                                                   count++;
142
143
                                                                                                 206
144
                 acquire(&m->wait_lock); /* we spin to acquire wait_lock */
                                                                                                 207
                                           /* someone else has the mutex */
                while (m->is_held) {
145
                                                                                                 208
146
                     m->waiters.insert(current thread)
                                                                                                 209
                                                                                                          void consumer (void *ignored) {
                     release(&m->wait_lock);
                                                                                                              for (;;) {
147
                                                                                                 210
148
                     schedule(); /* run a thread that is on the ready list */
                                                                                                 211
                                                                                                                   while (count == 0)
                     acquire(&m->wait_lock); /* we spin again */
                                                                                                                      ; // do nothing
149
                                                                                                 212
                                                                                                                   nextConsumed = buffer[out];
150
                                                                                                 213
                                           /* we now hold the mutex */
151
                m->is held = true;
                                                                                                 214
                                                                                                                   out = (out + 1) % BUFFER SIZE;
                m->owner = self;
152
                                                                                                 215
                                                                                                                   count --;
153
                release(&m->wait lock);
                                                                                                 216
                                                                                                                   /* next line abstractly consumes the item */
                                                                                                                   consume_item(nextConsumed);
154
                                                                                                 217
155
                                                                                                 218
            void mutex_release(Mutex *m) {
156
                                                                                                 219
157
                                                                                                 220
                                             /* we spin to acquire wait lock */
                 acquire(&m->wait_lock);
                                                                                                 221
                                                                                                         --Review: what's the problem?
158
159
                m->is_held = false;
                                                                                                         --Answer: count++ and count-- might compile to, respectively:
                                                                                                 222
                m \rightarrow owner = 0;
160
                                                                                                 223
                 wake_up_a_waiter(m->waiters); /* select and run a waiter */
                                                                                                             reg1 <-- count
                                                                                                                                   # load
161
                                                                                                 224
                release(&m->wait_lock);
162
                                                                                                 225
                                                                                                             reg1 <-- reg1 + 1
                                                                                                                                   # increment register
163
                                                                                                 226
                                                                                                             count <-- regl
                                                                                                                                   # store
164
                                                                                                 227
                                                                                                                                   # load
165
                                                                                                 228
                                                                                                             reg2 <-- count
            [Please let me (MW) know if you see bugs in the above.]
                                                                                                             reg2 <-- reg2 - 1
166
                                                                                                 229
                                                                                                                                   # decrement register
                                                                                                             count <-- reg2
167
                                                                                                 230
                                                                                                                                   # store
168
                                                                                                 231
   3. Terminology
                                                                                                         --Review: why not use instructions like "addl $0x1, _count"?
169
                                                                                                 232
170
                                                                                                 233
                                                                                                         --Answer: not atomic if there are multiple CPUs.
171
        To avoid confusion, we will use the following terminology in this
                                                                                                 234
172
        course (you will hear other terminology elsewhere):
                                                                                                 235
                                                                                                         --Review: so why not use "LOCK addl $0x1, _count"?
                                                                                                         --Answer: we could do that here, but LOCK won't save us every time
                                                                                                 236
173
        --A "lock" is an abstract object that provides mutual exclusion
174
                                                                                                 237
                                                                                                         --Review: so use general-purpose approach to protecting
175
                                                                                                 238
        --A "spinlock" is a lock that works by busy waiting, as in 6b
                                                                                                         critical sections: locks (or mutexes).
176
                                                                                                 239
177
                                                                                                 240
        --A "mutex" is a lock that works by having a "waiting" queue and
178
                                                                                                 241
179
        then protecting that waiting queue with atomic hardware
        instructions, as in 6c. The most natural way to "use the hardware"
180
        is with a spinlock, but there are others, such as turning off
181
        interrupts, which works if we're on a single CPU machine.
182
183
184
```

Printed by Michael Walfish

Feb	17, 11 16:22	l10-handout.txt	Page 5/9	Feb 17	, 11 16:22	l10-handout.txt	Page 6/9
Feb 242 243 244 246 247 248 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 266 267 268 269 260 261 262 263 264 265 266 267 268 269 260 261 262 263 264 265 266 267 268 269 260 261 262 263 264 265 266 267 268 269 260 261 262 263 264 265 266 267 268 269 260 261 262 263 264 265 266 267 268 269 260 261 262 263 264 265 266 267 268 269 260 261 267 268 269 260 261 267 268 269 260 261 267 268 269 260 261 267 268 269 260 261 267 268 269 260 261 269 260 261 262 263 264 265 266 267 268 267 277 278 277 277 278 277 277 278 280 277 278 280 277 278 280 261 277 278 268 269 260 261 267 268 269 260 261 267 268 269 269 260 261 269 260 261 269 260 261 269 260 261 269 260 261 269 260 261 269 260 261 277 273 274 275 276 277 278 279 280 271 277 278 280 281 277 278 282 282 283 284 285 284 285 277 278 279 280 277 278 282 282 282 283 284 283 284 285 283 284 285 283 284 285 283 284 285 283 284 285 283 284 285 283 284 285 283 284 285 285 285 285 285 285 285 285 285 285	<pre>17, 11 16:22 4b. Producer/consume Mutex mutex; void producer (voi for (;;) { /* next li nextProduc acquire(&m while (cou release yield() acquire(&m while (cout++; release(&m } } void consumer (voi for (;;) { acquire(&m while (cou release(&m while (cou release(&m while (cou release(&m out = (out count; release(&m /* next li consume_it } }</pre>	<pre>l10-handout.txt rr [bounded buffer] using mutexes dd *ignored) { ne produces an item and puts it in nextProduced red = mens_of_production(); nutex); nutex); nutex); if = nextProduced; 1) % BUFFER_SIZE; nutex); dd *ignored) { mutex); int == 0) {</pre>	Page 5/9	Feb 17 286 287 288 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305 306 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323 324 325 326 327 328 329 330 331 332 333 34 356 377	<pre>,1116:22 4c. Producer/ variables Mutex mu Cond nom Cond nom void pro for } void con for } Question: sleep? Wr while r c } </pre>	<pre>/// Intervent and the set of the set of</pre>	dition xtProduced */
				331 332 333 334 336 336 337 338	sleep? Wh while r c a }	<pre>hy not: e (count == BUFFER_SIZE) { release(&mutex); cond_wait(&nonfull); acquire(&mutex);</pre>	
	5. 5. h m		14.0				0/5

Printed by Michael Walfish

```
110-handout.txt
Feb 17, 11 16:22
                                                                              Page 7/9
        4d. Producer/consumer [bounded buffer] with semaphores
339
340
                                             /* mutex initialized to 1 */
341
            Semaphore mutex(1);
            Semaphore empty(BUFFER_SIZE); /* start with BUFFER_SIZE empty slots */
342
                                             /* 0 full slots */
343
            Semaphore full(0);
344
345
            void producer (void *ignored) {
                 for (;;)
346
                     /* next line produces an item and puts it in nextProduced */
347
                     nextProduced = means_of_production();
348
349
350
                     /*
                     * next line diminishes the count of empty slots and
351
352
                     * waits if there are no empty slots
                      */
353
354
                     sem_down(&empty);
                     sem down(&mutex); /* get exclusive access */
355
356
357
                     buffer [in] = nextProduced;
                     in = (in + 1) % BUFFER_SIZE;
358
359
360
                     sem_up(&mutex);
361
                     sem_up(&full);
                                      /* we just increased the # of full slots */
                 }
362
363
364
             void consumer (void *ignored) {
365
                 for (;;) {
366
367
368
                      /*
                      * next line diminishes the count of full slots and
369
370
                      * waits if there are no full slots
                      * /
371
                     sem down(&full);
372
                     sem_down(&mutex);
373
374
                     nextConsumed = buffer[out];
375
376
                     out = (out + 1) % BUFFER_SIZE;
377
                     sem_up(&mutex);
378
379
                     sem_up(&empty); /* one further empty slot */
380
                     /* next line abstractly consumes the item */
381
382
                     consume_item(nextConsumed);
383
                 }
             }
384
385
            Semaphores *can* (not always) lead to elegant solutions (notice
386
387
            that the code above is fewer lines than 1c) but they are much
388
            harder to use
389
            The fundamental issue is that semaphores make implicit (counts,
390
391
            conditions, etc.) what is probably best left explicit. Moreover,
            they *also* implement mutual exclusion.
392
393
394
            For this reason, you should not use semaphores. This example is
            here mainly for completeness and so you know what a semaphore
395
396
            is. But do not code with them. Solutions that use semaphores in
            this course will receive no credit.
397
398
```

Fe	b 17, 11 16:22	l10-handout.txt	Page 8/9
399	5. Example of a monitor	: MyBuffer	
400			
401	<pre>// This is pseudoco</pre>	ode that is inspired by C++.	
402	// Don't take it l:	iterally.	
403			
404	class MyBuffer {		
405	public:		
406	MyBuffer();		
407	~MyBuffer();		
408	void Enqueue(It	cem);	
409	Item = Dequeue	();	
410	private		
411	int count;		
412	int in;		
413	int out;		
414	Item buller[BU	(FER_SIZE);	
415	Mulex* mulex;		
410	Cond* nonfull:		
417	l cond nonruir,		
410	J		
420	void		
421	MvBuffer::MvBuffer		
422	{		
423	in = out = cour	nt = 0;	
424	mutex = new Mut	cex;	
425	nonempty = new	Cond;	
426	nonfull = new (Cond;	
427	}		
428			
429	void		
430	MyBuffer::Enqueue([tem item)	
431	{		
432	mutex.acquire();	
433	while (count ==	= BUFFER_SIZE)	
434	cond_wait(8	anonfull, &mutex);	
435	buffoulin1 - ii		
436	buller[III] = II	Leur	
437	III = (III + I) +	BUFFER_SIZE,	
438	acond gignal (sn)	nompty (mutox):	
439	mutex release(:	
440	}		
442	J		
443	Ttem		
444	MvBuffer::Dequeue(
445	{		
446	mutex.acquire());	
447	while (count ==	= 0)	
448	cond_wait(8	anonempty, &mutex);	
449			
450	Item ret = bufi	[er[out];	
451	out = (out + 1)) % BUFFER_SIZE;	
452	count;		
453	cond_signal(&no	onfull, &mutex);	
454	mutex.release() i	
455	return ret;		
456	J		
407			

```
I10-handout.txt
Feb 17, 11 16:22
                                                                                  Page 9/9
         int main(int, char**)
458
459
             MyBuffer buf;
460
             int dummy;
461
             tid1 = thread_create(producer, &buf);
462
             tid2 = thread_create(consumer, &buf);
463
             thread_join(tid1);
464
465
             // never reach this point
466
467
             return -1;
468
        }
469
        void producer(void* buf)
470
471
             MyBuffer* sharedbuf = reinterpret_cast<MyBuffer*>(buf);
472
473
             for (;;) {
                 /* next line produces an item and puts it in nextProduced */ Item nextProduced = means_of_production();
474
475
476
                 sharedbuf->Engueue(nextProduced);
477
478
         }
479
480
        void consumer(void* buf)
481
             MyBuffer* sharedbuf = reinterpret_cast<MyBuffer*>(buf);
482
             for (;;) {
483
                 Item nextConsumed = sharedbuf->Dequeue();
484
485
                 /* next line abstractly consumes the item */
486
487
                 consume_item(nextConsumed);
             }
488
489
490
491
        Key point: *Threads* (the producer and consumer) are separate from
        *shared object* (MyBuffer). The synchronization happens in the
492
493
        shared object.
```