Printed by Michael Walfish

```
c11-handout.txt
                                                                                                                                                c11-handout.txt
Oct 16, 15 8:57
                                                                                      Page 1/8
                                                                                                        Oct 16, 15 8:57
                                                                                                                                                                                               Page 2/8
    Handout for CS 3250
                                                                                                         59
   Class 11
                                                                                                            3. Producer/consumer example:
2
                                                                                                         60
    15 October 2015
3
                                                                                                         61
                                                                                                         62
4
    1. Example to illustrate interleavings: say that thread A executes f()
                                                                                                                 "buffer" stores BUFFER_SIZE items
5
                                                                                                         63
    and thread B executes g(). (Here, we are using the term "thread"
                                                                                                         64
                                                                                                                 "count" is number of used slots. a variable that lives in memory
6
                                                                                                                 "out" is next empty buffer slot to fill (if any)
7
    abstractly, to refer to execution contexts that share memory.)
                                                                                                         65
                                                                                                                 "in" is oldest filled slot to consume (if any)
8
                                                                                                         66
                                                                                                                 * /
9
        a.
                                                                                                         67
10
                                                                                                         68
             int x;
                                                                                                                  void producer (void *ignored) {
11
                                                                                                         69
12
                                                                                                         70
                                                                                                                       for (;;) {
             f() \{ x = 1; \}
                                                                                                                           /* next line produces an item and puts it in nextProduced */
                                                                                                         71
13
14
                                                                                                         72
                                                                                                                            nextProduced = means_of_production();
             g() \{ x = 2; \}
15
                                                                                                         73
                                                                                                                           while (count == BUFFER SIZE)
                                                                                                                                ; // do nothing
16
                                                                                                         74
             What are possible values of x after A has executed f() and B has
                                                                                                                           buffer [in] = nextProduced;
17
                                                                                                         75
                                                                                                                           in = (in + 1) % BUFFER_SIZE;
             executed g()?
18
                                                                                                         76
19
                                                                                                         77
                                                                                                                           count++;
                                                                                                                       }
20
        b.
                                                                                                         78
21
             int y = 12;
                                                                                                         79
                                                                                                                  }
22
                                                                                                         80
23
              \begin{array}{c} f() & \left\{ \begin{array}{c} x = y + 1; \\ y = y & 2; \end{array} \right\} \end{array} 
                                                                                                         81
                                                                                                                  void consumer (void *ignored) {
                                                                                                         82
                                                                                                                       for (;;) {
24
25
                                                                                                         83
                                                                                                                           while (count == 0)
             What are the possible values of x?
                                                                                                                               ; // do nothing
26
                                                                                                         84
27
                                                                                                         85
                                                                                                                            nextConsumed = buffer[out];
                                                                                                                           out = (out + 1) % BUFFER_SIZE;
                                                                                                         86
28
        с.
             int x = 0;
29
                                                                                                         87
                                                                                                                            count--;
             \begin{array}{c} f() & \left\{ \begin{array}{c} x = x + 1; \\ g() \end{array} \right\} \\ \left\{ \begin{array}{c} x = x + 2; \end{array} \right\} \end{array}
                                                                                                                            /* next line abstractly consumes the item */
30
                                                                                                         88
                                                                                                                           consume_item(nextConsumed);
31
                                                                                                         89
32
                                                                                                         90
                                                                                                                       }
33
             What are the possible values of x?
                                                                                                         91
                                                                                                         92
34
    2. Linked list example
                                                                                                                 /*
35
                                                                                                         93
                                                                                                         94
                                                                                                                    what count++ probably compiles to:
36
                                                                                                                     regl <-- count
        struct List_elem {
                                                                                                         95
                                                                                                                                             # load
37
38
             int data;
                                                                                                         96
                                                                                                                      reg1 <-- reg1 + 1
                                                                                                                                             # increment register
             struct List_elem* next;
                                                                                                                     count <-- req1
                                                                                                         97
                                                                                                                                             # store
39
        };
                                                                                                         98
40
41
                                                                                                         99
                                                                                                                     what count -- could compile to:
42
        List_elem* head = 0;
                                                                                                        100
                                                                                                                     reg2 <-- count
                                                                                                                                             # load
                                                                                                                     reg2 <-- reg2 - 1
                                                                                                                                             # decrement register
43
                                                                                                        101
        insert(int data) {
44
                                                                                                        102
                                                                                                                     count <-- reg2
                                                                                                                                             # store
             List_elem* l = new List_elem;
                                                                                                                 * /
45
                                                                                                        103
             1->data = data;
46
                                                                                                        104
47
             1->next = head;
                                                                                                         105
                                                                                                                 What happens if we get the following interleaving?
             head = 1;
48
                                                                                                        106
49
                                                                                                        107
                                                                                                                      reg1 <-- count
                                                                                                                     reg1 <-- reg1 + 1
50
                                                                                                        108
51
        What happens if two threads execute insert() at once and we get the
                                                                                                         109
                                                                                                                      reg2 <-- count
         following interleaving?
                                                                                                                      req2 <-- req2 - 1
52
                                                                                                        110
                                                                                                                      count <-- regl
53
                                                                                                        111
         thread 1: 1->next = head
                                                                                                                      count <-- req2
                                                                                                        112
54
         thread 2: 1->next = head
55
                                                                                                        113
         thread 2: head = 1;
56
         thread 1: head = 1;
57
58
```

Printed by Michael Walfish

Oct 16, 15 8	:57 c11–handout.txt	Page 3/8	Oct 16	, 15 8:5	57	c11-handout.txt	Page 4/8
114 4. Prote 115 116 116 117 118 119 120 121 122 123 124 125 125 126 127 128 130 5. How of 131 Here 133 136 136 137 138 139 140 141 142 143 144 144 145 146 144 145 146 147 148 149 150 151 152 153 154 155 155 156 157 158	<pre>cting the linked list Lock list_lock; insert(int data) { List_elem*1 = new List_elem; l->data = data; acquire(&list_lock); l->next = head; // A head = 1; // B release(&list_lock); } an we implement list_lock, acquire(), and release()? e is A BADLY BROKEN implementation: struct Lock { int locked; } void (BROKEN] acquire(Lock *lock) { while (1) { if (lock->locked == 0) {// C lock->locked = 1; // D break; } } void release (Lock *lock) { lock->locked = 0; } what's the problem? Two acquire()s on the same lock on diff both will think they have acquired the lock. This is the sa kind of race that we ere trying to eliminate in inser(). we have made a little progress: now we only need a way to prevent interleaving in one place (acquire()), not for many arbitrary complex sequences of code.</pre>	erent me But	159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 166 187 188 199 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 209	5a. T R d (() (() ()) s } / v v } v v } v v } v v } v v } v v }	<pre>Nest-and-set spinl Relies on atomic i Noing "xchg addr tomically swaps t virtual) memory a nterleaved. One c i) freeze all C ii) temp = *addr iii) *addr = %ea iv) %eax = temp v) un-freeze me * pseudocode */ .nt xchg_val(addr, %eax = value; xchg (*addr), *truct Lock { int locked; * bare-bones vers roid acquire (Lock pushcli(); /* while (1) { if (xchg_val(& break; } roid release(Lock xchg_val(&lock- popcli(); /* Cest-and-test-and- * optimization in roid acquire(Lock* pushcli(); while (xchg_va while (lock } </pre>	<pre>ock nstruction on the CPU. For example, on th , %eax" he contents of %eax with the contents of ddress addr. No other instructions can be an think of xchg like this: PUs' memory activity for address addr mory activity value) { %eax ion of acquire */ *lock) { what does this do? */ lock->locked, 1) == 0) *lock) { >locked, 0; what does this do? */ set lock . acquire; call xchg_val() less frequently lock) { l(&lock->locked, 1) == 1) { k->locked); } </pre>	ie x86, ; ; */
-nday Uctobel	10. 2013	c11-nand	iout.txt				2/4

Oct 16	, 15 8:57	c11-handout.txt	Page 5/8	Oct 16	6, 15 8:57	c11-handout.txt Pag	ge 6/8
210 б.	Ticket locks			258 7.	MCS locks	(a kind of queue lock)	
Oct 16, 210 6. T 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 223 224 225 223 224 225 223 224 225 223 224 225 223 224 225 227 228 229 223 224 225 227 228 229 223 224 225 229 223 224 225 229 223 224 225 229 220 231 232 24 24 24 24 24 24 24 24 24 24 24 24 24	<pre>15 8:57 c11-handout.txt Ticket locks The spinlocks presented above have fairness issues on NUMA machiness (cores closer to the memory containing the 'locked' variable are more likely to succeed in acquiring the lock). Ticket locks address that issue. They rely on an atomic primitive known as "fetch and increment." On the x86, we implement fetch and increment with the XADD instruction, but note that this instruction is not atomic by default, so we need the LOCK prefix. Here's pseudocode: int fetch_and_increment (int* addr) { LOCK: // remember, this is pseudocode int was = *addr: *addr = was + 1; return was; } Here's inline assembly: inline int fetch_and_increment(int *addr) { int was = 1: asm volatile("lock xaddl %1, %0" : "+m" (*addr), "=r" (was) // Output ; "1" (was), "m" (*addr) // Input ; "1" (was); } struct Lock { int current_ticket; int next_ticket; } woid acquire (Lock *lock)</pre>	Page 5/8	Oct 16 258 7. 259 260 261 262 263 264 265 266 267 268 269 270 271 271 271 272 273 274 275 276 277 277 278 279 280 281 283 284 283 284 285 286 285 286 287 283 284 285 286 287 283 284 285 286 287 283 284 285 286 287 283 284 285 286 287 283 284 285 286 287 283 284 285 286 287 281 285 286 287 281 283 284 285 286 287 281 283 284 285 286 287 281 283 284 285 286 287 281 283 284 285 286 287 281 283 284 285 286 287 281 283 284 285 286 287 281 283 284 285 286 287 281 283 284 285 286 287 281 283 284 285 286 287 281 282 283 284 285 286 287 281 282 283 284 285 286 287 277 277 287 287 277 287 277 287 277 287 28	<pre>MCS locks (a kind of queue lock) MCS locks (a kind of queue lock) Ticket locks are fair, as noted above, but they (and baseline spinlocks) have performance issues when there is a lot of contention. These issues fundamenally result from cross-talk among CPUS (which undermines caching and generates traffic on the memory bus). This phenomenon is investigated in depth in the "Scalable Locks are Dangerous" paper. The locks presented below address that issue. These are known as MC locks. Citation: Mellor-Crummey, J. M. and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors, ACM Transactions on Computer Systems, Vol. 9, No. 1, February, 1991, pp.21-65. A. CAS / CMPXCHG Useful operation: compare-and-swap, known as CAS. Says: "atomically check whether a given memory cell contains a given value, and if it does, then replace the contents of the memory cell with this other value; in either case, return the original value in the memory location". On the X86, we implement CAS with the CMPXCHG instruction, but note that this instruction is not atomic by default, so we need the LOCK prefix. Here's pseudocode: int cmpxchg_val(int* addr, int oldval, int newval) { LOCK: // remember, this is pseudocode int was = *addr; if (*addr == oldval) *addr = newval; return was; } </pre>			
244 245 246 247 248 249 250	<pre>int current_ticket; int next_ticket; } void acquire (Lock *lock) { int t = fetch and inc</pre>	rrement (&lock-snewt ticket):		292 293 294 295 296 297 298	r } Here's in	:f (*addr == oldval) *addr = newval; eturn was; lline assembly:	
250 251 252 253 254 255	<pre>while (t != lock->curr } void release (Lock *lock)</pre>	<pre>{</pre>		298 299 300 301 302 303	uint3 v a	<pre>2_t cmpxchg_val(uint32_t* addr, uint32_t oldval, uint32_t ne Lint32_t was; .sm volatile("lock cmpxchg %3, %0"</pre>	wval) {
256 257	}			304 305 306 307	r }	: "cc"); eturn was;	
				308 309	B. The MC	S lock	
				310 311 312 313	Each CPU mean loca CPUs are mode):	has a qnode structure in *local * memory. Here, local can il memory in NUMA machine or its own cache line that other not allowed to cache (i.e., the cache line is in exclusive	
				314 315 316 317 318	typedef s struc bool } qnode;	<pre>struct qnode { t qnode* next; someoneelse_locked;</pre>	
				319 320	typedef o	<pre>mode* lock; // a lock is a pointer to a qnode</pre>	
				321 322 323	The loc or waitin	k itself is literally the *tail* of the list of CPUs holding g for the lock.	
				324 325 326	While w	aiting, a CPU spins on its local "locked" flag.	

Oct 16,	, 15 8:57	c11-handout.txt	Page 7/8	Oct 16, 15 8	8:57 c1 *	1-handout.txt	Page 8/8
326				395 9. Mute:	xes		
327	Here's the code for	or acquire:		396			
328		we have a second s		397 Mot	ivation: all of the aforeme	ntioned locks were called spinlocks	
329	// lockp is a	qnode**. I points to our local qnode.		398 Dec	ause acquire() spins. A mut	ex avoids busy waiting. Usually, in	
330	Voiu acquire(IOCK IOCKP, MIDDE I)		400 USE	i space code, you want to b	e using mutexes, not spiniocks.	
332	I->next =	NULL;		400 401 Spin	nlocks are good for some th	ings, not so great for others. The	
333	qnode* pro	edecessor;		402 mai	n problem is that it *busy	waits*: it spins, chewing up CPU	
334				403 CYC	les. Sometimes this is what	we want (e.g., if the cost of going	3
335	// next l:	ine makes lockp point to I (that is, it se	ts *lockp < I)	404 to	sleep is greater than the c	ost of spinning for a few cycles	
336	// and re	turns the old value of *lockp. Uses atomic	operation	405 wai	ting for another thread or	process to relinquish the spinlock)	
337	// XCHG.	see earlier in handout (or earlier handout	s)	406 But	sometimes this is not at a	Il what we want (e.g., if the lock	
338	// IOT 1mj	plementation of xcng_val.		407 WOU	Id be held for a while: in	those cases, the CPU waiting for the	2
339	predecess	or - ycho val(lockp I): // "A"		408 10C	read or process)	ing instead of funning some other	
340	if (prede	cessor != NULL) { // queue was non-empty		409 0111	cad of process).		
342	I->soi	meoneelse locked = true;		411 Wit	h a mutex, if the lock is n	ot available, the locking thread is	
343	prede	cessor->next = I; // "B"		412 put	to sleep, and tracked by a	queue in the mutex.	
344	while	(I->someoneelse_locked) ; // spin		413			
345	}			414	struct Mutex {		
346	// we hold	d the lock!		415	bool is_held;	/* true if mutex held */	
347	}			416	thread_id owner;	/* thread holding mutex, if locked	d */
348	What's soing	072		417	Lock whit lock:	/* queue of thread itess */	
350	what s going (011:		410	lock wait_iock/	/ a spiniock, as above /	
351	If the lock	is unlocked, then *lockp == NULL.		420	J		
352		· · · · · · · · · · · ·		421	The implementation of mute	x_acquire() and mutex_release() would	ld
353	If the lock	is locked, and there are no waiters, ther	*lockp	422	be something like:		
354	points to the	qnode of the owner		423			
355				424	void mutex_acquire(Mutex *	m) {	
356	If the lock	is locked, and there are waiters, then *J	ockp points	425		· /*	+ /
357	to the ghode a	at the tall of the walter list.		426	acquire(&m=>wait_lock)	, /* we spin to acquire wait_lock	~/
350	Here's the code	for release:		427	while (m-sig held) {	/* someone else has the mutex */	
360	here b the tode	101 1010400		429	##110 (m + 15_Hera) (, bomeone erbe nub ene muten ,	
361	void release()	lock* lockp, qnode* I) {		430	m->waiters.insert(current_thread)	
362	if (!I->n	ext) { // no known successor		431	release(&m->wait_l	ock);	
363	if (c	<pre>mpxchg_val(lockp, I, NULL) == I) { //</pre>	"C"	432			
364	1.	/ swap successful: lockp was pointing to]	, so now	433	/*		
365		/ *lockp == NULL, and the lock is unlocked	. we can	434	* NOTE! Right her	e, mutex_release() could execute. To	0
367	, 	/ go nome now.		430	* on the schedule	r's ready list If we are we	
368	}	ccarii,		437	* shouldn't vield	().	
369	// if	we get here, then there was a timing issu	e: we had	438	*/		
370	// no	known successor when we first checked, bu	t now we	439			
371	// ha	ve a successor: some CPU executed the line	"A"	440	yield_if_we_are_no	t_ready();	
372	// ab	ove. Wait for that CPU to execute line "B'	above.	441	1 (
373	while	(!I->next) ;		442	acquire(&m->wait_1	ock); /* we spin again */	
374	} (/ handin	a the leak off to the next whiter is as a		443	m->walters.remove(current_thread)	
376	// indiuing // inst e	etting that waiter's "someoneelse locked"	flag to false	445	}		
377	I->next->	someoneelse locked = false;	ring to raibe	446	J		
378	}			447	m->is_held = true;	/* we now hold the mutex */	
379	-			448	m->owner = self;		
380	What's going (on?		449			
381				450	release(&m->wait_lock)	;	
382	If I->next	== NULL and *lockp == I, then no one else	15	451	}		
383	walting for th	ne lock. So we set "lockp == NULL.		452	woid muter release (Muter *	m) {	
385	If I->next	== NULL and *lockp != I, then another CPU	is in	453	Void mater_rerease(mater	(iii) (
386	acquire (spec	ifically, it executed its atomic operation	, namely	455	acquire(&m->wait lock)	; /* we spin to acquire wait loc	s */
387	line "A", bef	ore we executed ours, namely line "C"). So	wait for	456	100 1 1 1 1 1 1 1 1 1 1 <u>1</u> 1 0 0 1 1 1		
388	the other CPU	to put the list in a same state, and then	drop	457	m->is_held = false;		
389	down to the n	ext case:		458	m->owner = 0;		
390	TC T	I MITTER Allow on house the set of the set		459	(*	the second s	
391	if I->next	!= NULL, then we know that there is a spir	ning a flog to	460	/* tell scheduler to r	un a waiter */	
392	waiter (the o.	idesi one). Hand it the lock by setting it	S LIAG TO	461	piace_a_waiter_on_read	y_iist(m->waiters);	
393 394	Laise.			462	release(&m-swait lock)	;	
004				464	ICICASC(am >wait_IOCK)	,	
				465	}		
				466			