

- 1. Last time
 - 2. Performance issues
 - 3. Programmability issues
 - 4. Mutexes and interleavings
-

2. Performance issues

(a) Invoking spinlocks/mutexes can be expensive
MCS locks

(b) Coarse locks limit available parallelism...

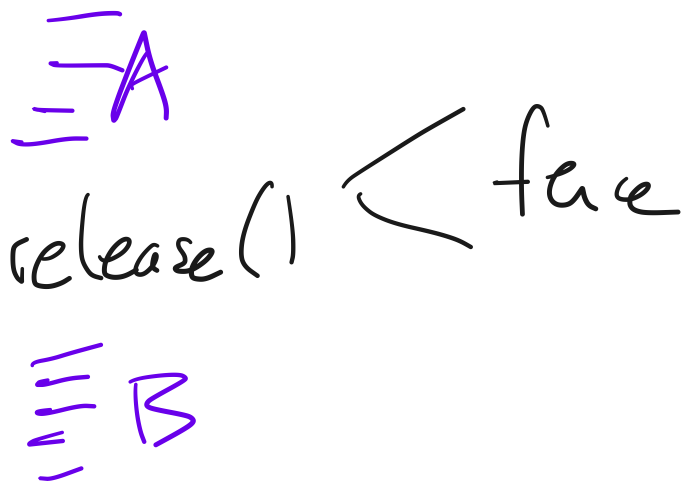
(c) ... but fine-grained locking leads to complexity
and hence bugs

3. Programmability issues

- loss of modularity printf("...");

- what's the fundamental problem?

4. mutexes and interleavings



A ^{stuff} happens before B stuff
although ^{it} is A

□ w/in B,

"stuff" can happen out of order from the perspective of other cores, assuming they could view the operations separately.

Feb 13, 23 6:36

handout06.txt

Page 1/7

```

1 CS 202, Spring 2023
2 Handout 6 (Class 7)
3
4 1. Simple deadlock example
5
6     T1:
7         acquire(mutexA);
8         acquire(mutexB);
9
10        // do some stuff
11
12        release(mutexB);
13        release(mutexA);
14
15     T2:
16         acquire(mutexB);
17         acquire(mutexA);
18
19        // do some stuff
20
21        release(mutexA);
22        release(mutexB);
23

```

Feb 13, 23 6:36

handout06.txt

Page 2/7

```

24 2. More subtle deadlock example
25
26     Let M be a monitor (shared object with methods protected by mutex)
27     Let N be another monitor
28
29     class M {
30     private:
31         Mutex mutex_m;
32
33         // instance of monitor N
34         N another_monitor;
35
36         // Assumption: no other objects in the system hold a pointer
37         // to our "another_monitor"
38
39     public:
40         M();
41         ~M();
42         void methodA();
43         void methodB();
44     };
45
46     class N {
47     private:
48         Mutex mutex_n;
49         Cond cond_n;
50         int navailable;
51
52     public:
53         N();
54         ~N();
55         void* alloc(int nwanted);
56         void free(void*);
57     }
58
59     int
60     N::alloc(int nwanted) {
61         acquire(&mutex_n);
62         while (navailable < nwanted) {
63             wait(&cond_n, &mutex_n);
64         }
65
66         // peel off the memory
67
68         navailable -= nwanted;
69         release(&mutex_n);
70     }
71
72     void
73     N::free(void* returning_mem) {
74         acquire(&mutex_n);
75
76         // put the memory back
77
78         navailable += returning_mem;
79
80         broadcast(&cond_n, &mutex_n);
81
82         release(&mutex_n);
83     }
84
85

```

Feb 13, 23 6:36

handout06.txt

Page 3/7

```

86 void
87 M::methodA() {
88
89     acquire(&mutex_m);
90
91     void* new_mem = another_monitor.alloc(int nbytes);
92
93     // do a bunch of stuff using this nice
94     // chunk of memory n allocated for us
95
96     release(&mutex_m);
97 }
98
99 void
100 M::methodB() {
101
102     acquire(&mutex_m);
103
104     // do a bunch of stuff
105
106     another_monitor.free(some_pointer);
107
108     release(&mutex_m);
109 }
110
111 QUESTION: What's the problem?
112

```

Monday February 13, 2023

handout06.txt

Feb 13, 23 6:36

handout06.txt

Page 4/7

```

113 3. Locking brings a performance vs. complexity trade-off
114
115 /*
116  *      linux/mm/filemap.c
117  *
118  * Copyright (C) 1994-1999 Linus Torvalds
119  */
120
121 /*
122  * This file handles the generic file mmap semantics used by
123  * most "normal" filesystems (but you don't /have/ to use this:
124  * the NFS filesystem used to do this differently, for example)
125  */
126 #include <linux/export.h>
127 #include <linux/compiler.h>
128 #include <linux/dax.h>
129 #include <linux/fs.h>
130 #include <linux/sched/signal.h>
131 #include <linux/uaccess.h>
132 #include <linux/capability.h>
133 #include <linux/kernel_stat.h>
134 #include <linux/gfp.h>
135 #include <linux/mm.h>
136 #include <linux/swap.h>
137 #include <linux/mman.h>
138 #include <linux/pagemap.h>
139 #include <linux/file.h>
140 #include <linux/uio.h>
141 #include <linux/hash.h>
142 #include <linux/writeback.h>
143 #include <linux/backing-dev.h>
144 #include <linux/pagevec.h>
145 #include <linux/blkdev.h>
146 #include <linux/security.h>
147 #include <linux/cpuset.h>
148 #include <linux/hugetlb.h>
149 #include <linux/memcontrol.h>
150 #include <linux/cleancache.h>
151 #include <linux/shmem_fs.h>
152 #include <linux/rmap.h>
153 #include "internal.h"
154
155 #define CREATE_TRACE_POINTS
156 #include <trace/events/filemap.h>
157
158 /*
159  * FIXME: remove all knowledge of the buffer layer from the core VM
160  */
161 #include <linux/buffer_head.h> /* for try_to_free_buffers */
162
163 #include <asm/mman.h>
164
165 /*
166  * Shared mappings implemented 30.11.1994. It's not fully working yet,
167  * though.
168  *
169  * Shared mappings now work. 15.8.1995 Bruno.
170  *
171  * finished 'unifying' the page and buffer cache and SMP-threaded the
172  * page-cache, 21.05.1999, Ingo Molnar <mingo@redhat.com>
173  *
174  * SMP-threaded pagemap-LRU 1999, Andrea Arcangeli <andrea@suse.de>
175  */
176
177 /*
178  * Lock ordering:
179  *
180  * ->i_mmap_rwsem          (truncate_pagecache)
181  * ->private_lock        (__free_pte->__set_page_dirty_buffers)
182  * ->swap_lock           (exclusive_swap_page, others)
183  * ->i_pages lock
184  *
185  * ->i_mutex

```

2/4

```

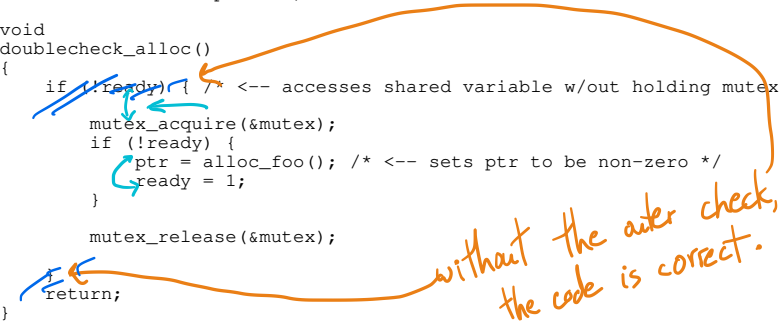
186 *   ->i_mmap_rwsem          (truncate->unmap_mapping_range)
187 *
188 *   ->mmap_sem
189 *   ->i_mmap_rwsem
190 *     ->page_table_lock or pte_lock  (various, mainly in memory.c)
191 *     ->i_pages lock              (arch-dependent flush_dcache_mmap_lock)
192 *
193 *   ->mmap_sem
194 *     ->lock_page              (access_process_vm)
195 *
196 *   ->i_mutex                (generic_perform_write)
197 *     ->mmap_sem              (fault_in_pages_readable->do_page_fault)
198 *
199 *   bdi->wb.list_lock        (fs/fs-writeback.c)
200 *     sb_lock                (fs/fs-writeback.c)
201 *     ->i_pages lock         (__sync_single_inode)
202 *
203 *   ->i_mmap_rwsem
204 *     ->anon_vma.lock        (vma_adjust)
205 *
206 *   ->anon_vma.lock
207 *     ->page_table_lock or pte_lock  (anon_vma_prepare and various)
208 *
209 *   ->page_table_lock or pte_lock
210 *     ->swap_lock            (try_to_unmap_one)
211 *     ->private_lock         (try_to_unmap_one)
212 *     ->i_pages lock         (try_to_unmap_one)
213 *     ->zone_lru_lock(zone)  (follow_page->mark_page_accessed)
214 *     ->zone_lru_lock(zone)  (check_pte_range->isolate_lru_page)
215 *     ->private_lock         (page_remove_rmap->set_page_dirty)
216 *     ->i_pages lock         (page_remove_rmap->set_page_dirty)
217 *     bdi.wb->list_lock      (page_remove_rmap->set_page_dirty)
218 *     ->inode->i_lock         (page_remove_rmap->set_page_dirty)
219 *     ->memcg->move_lock      (page_remove_rmap->lock_page_memcg)
220 *     bdi.wb->list_lock      (zap_pte_range->set_page_dirty)
221 *     ->inode->i_lock         (zap_pte_range->set_page_dirty)
222 *     ->private_lock         (zap_pte_range->__set_page_dirty_buffers)
223 *
224 *   ->i_mmap_rwsem
225 *     ->tasklist_lock        (memory_failure, collect_procs_ao)
226 */
227
228 static int page_cache_tree_insert(struct address_space *mapping,
229                                  struct page *page, void **shadowp)
230 {
231     struct radix_tree_node *node;
232     .....
233
234 [the point is: fine-grained locking leads to complexity.]
235

```

```

236 4. Cautionary tale
237
238 Consider the code below:
239
240 struct foo {
241     int abc;
242     int def;
243 };
244 static int ready = 0;
245 static mutex_t mutex;
246 static struct foo* ptr = 0;
247
248 void
249 doublecheck_alloc()
250 {
251     if (!ready) { /* <-- accesses shared variable w/out holding mutex */
252         mutex_acquire(&mutex);
253         if (!ready) {
254             ptr = alloc_foo(); /* <-- sets ptr to be non-zero */
255             ready = 1;
256         }
257     }
258
259     mutex_release(&mutex);
260
261     return;
262 }
263
264
265 This is an example of the so-called "double-checked locking pattern."
266 The programmer's intent is to avoid a mutex acquisition in the common
267 case that 'ptr' is already initialized. So the programmer checks a flag
268 called 'ready' before deciding whether to acquire the mutex and
269 initialize 'ptr'. The intended use of doublecheck_alloc() is something
270 like this:
271
272 void f() {
273     doublecheck_alloc();
274     ptr->abc = 5;
275 }
276
277 void g() {
278     doublecheck_alloc();
279     ptr->def = 6;
280 }
281
282 We assume here that mutex_acquire() and mutex_release() are implemented
283 correctly (each contains memory barriers internally, etc.). Furthermore,
284 we assume that the compiler does not reorder instructions.
285
286 NEVERTHELESS, on multi-CPU machines that do not offer sequential
287 consistency, doublecheck_alloc() is broken. What is the bug?
288
289 -----
290
291 Unfortunately, double-checked initialization (or double-checked locking
292 as it's sometimes known) is a common coding pattern. Even some
293 references on threads suggest it! Still, it's broken.
294
295 While you can fix it (in C) by adding another barrier (exercise:
296 where?), this is not recommended, as the code is tricky to reason about.
297 One of the points of this example is to show you why it's so important
298 to protect global data with a mutex, even if "all" one is doing is
299 reading memory, and even if the shortcut looks harmless.
300

```



Feb 13, 23 6:36

handout06.txt

Page 7/7

```
301 Finally, here are some references on this topic:
302
303 --http://www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf
304 explores issues with this pattern in C++
305
306 --The "Double-Checked Locking is Broken" Declaration:
307 http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html
308
309 --C++11 provides a way to implement the pattern correctly and
310 portably (again, using memory barriers):
311 https://preshing.com/20130930/double-checked-locking-is-fixed-in-cpp11/
```