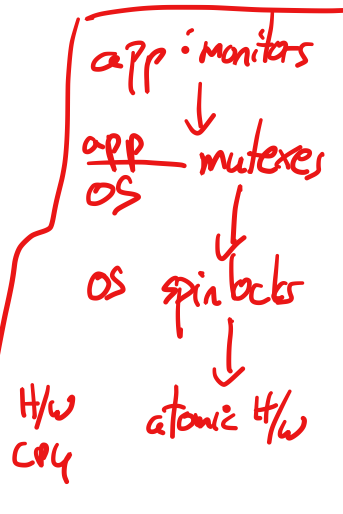
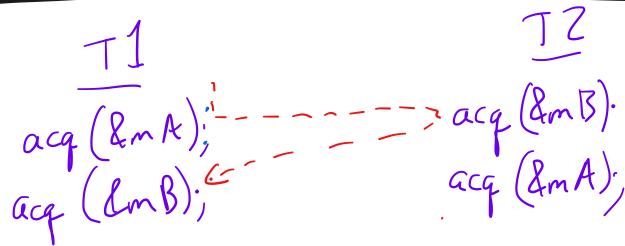


- 01. Last time
- 02. Implementation of locks, spinlocks, mutexes
- 03. Deadlock
- 04. Other progress issues
- 05. Performance issues
- 06. Programmability issues
- 07. Mutexes and interleavings
- 08. Questions

Next time



### 3. Deadlock



release (&mB);  
release (&mA);

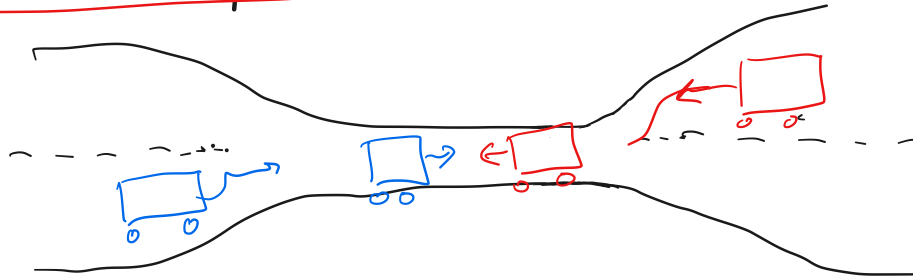
Happens when all four of these conditions are present:

- i. mutual exclusion
- ii. hold and wait
- iii. no pre-emption
- iv. circular wait

What can we do about deadlock?

- (a) ignore it
- (b) detect + recover
- (c) avoid algorithmically

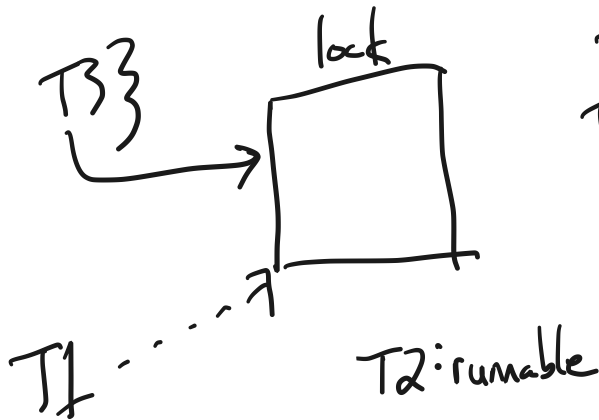
- (d) negate one of the 4 conditions
  - (e) static/dynamic detection tools
- 



#### 4. Other progress issues

Starvation

Priority inversion



T1: highest  
T2: medium  
T3: lowest

Assume: highest-prio runnable thread runs.

---

#### 5. Performance issues + tradeoffs

locks/interlocks have a perf cost

- spinlocks/mutexes

- coarse-grained locking limits parallelism...

∴ fine-grained locking leads to complexity, bugs

---

## 2. Implementation of spinlocks and mutexes

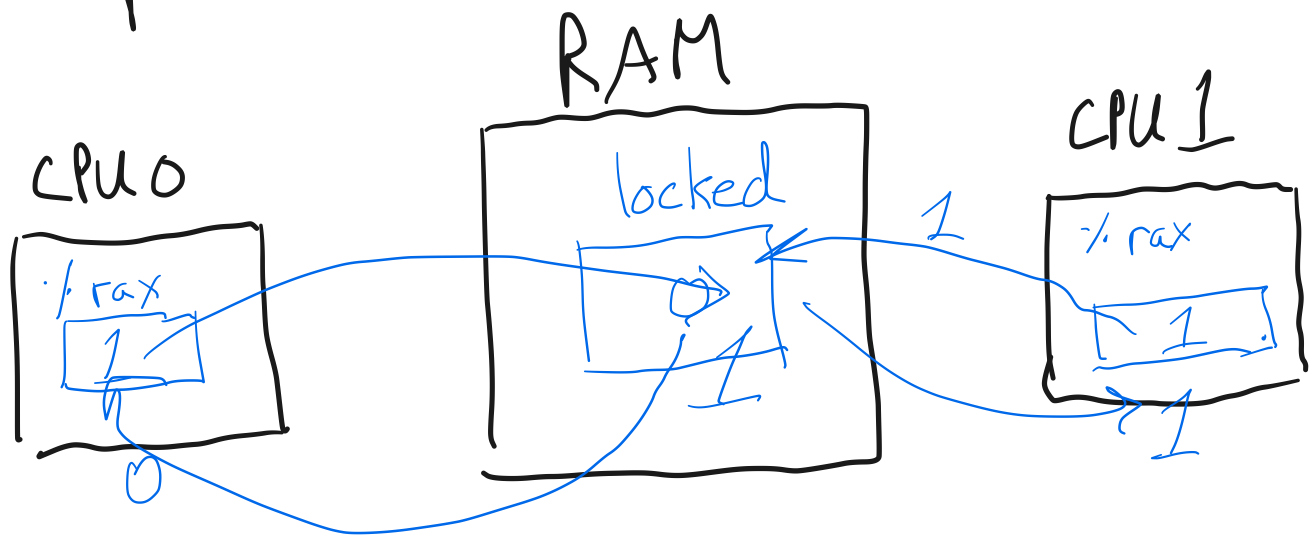
high-level interface: lock()/unlock()

how to provide?

(a) Peterson's algorithm → busy waiting, static bound

(b) disable interrupts

(c) spinlocks



(d) mutexes: spinlock + a queue

(a) material

- textbook has an implementation
- handout has another

Feb 08, 23 0:04

handout05.txt

Page 1/4

```

1 CS 202, Spring 2023
2 Handout 5 (Class 6)
3
4 The previous handout demonstrated the use of mutexes and condition
5 variables. This handout demonstrates the use of monitors (which combine
6 mutexes and condition variables).
7
8 1. The bounded buffer as a monitor
9
10 // This is pseudocode that is inspired by C++.
11 // Don't take it literally.
12
13 class MyBuffer {
14     public:
15         MyBuffer();
16         ~MyBuffer();
17         void Enqueue(Item);
18         Item = Dequeue();
19     private:
20         int count;
21         int in;
22         int out;
23         Item buffer[BUFFER_SIZE];
24         Mutex* mutex;
25         Cond* nonempty;
26         Cond* nonfull;
27     }
28
29 void
30 MyBuffer::MyBuffer()
31 {
32     in = out = count = 0;
33     mutex = new Mutex;
34     nonempty = new Cond;
35     nonfull = new Cond;
36 }
37
38 void
39 MyBuffer::Enqueue(Item item)
40 {
41     mutex.acquire();
42     while (count == BUFFER_SIZE)
43         cond_wait(&nonfull, &mutex);
44
45     buffer[in] = item;
46     in = (in + 1) % BUFFER_SIZE;
47     ++count;
48     cond_signal(&nonempty, &mutex);
49     mutex.release();
50 }
51
52 Item
53 MyBuffer::Dequeue()
54 {
55     mutex.acquire();
56     while (count == 0)
57         cond_wait(&nonempty, &mutex);
58
59     Item ret = buffer[out];
60     out = (out + 1) % BUFFER_SIZE;
61     --count;
62     cond_signal(&nonfull, &mutex);
63     mutex.release();
64     return ret;
65 }
66

```

Feb 08, 23 0:04

handout05.txt

Page 2/4

```

67
68 int main(int, char**)
69 {
70     MyBuffer buf;
71     int dummy;
72     tid1 = thread_create(producer, &buf);
73     tid2 = thread_create(consumer, &buf);
74
75     // never reach this point
76     thread_join(tid1);
77     thread_join(tid2);
78     return -1;
79 }
80
81 void producer(void* buf)
82 {
83     MyBuffer* sharedbuf = reinterpret_cast<MyBuffer*>(buf);
84     for (;;) {
85         /* next line produces an item and puts it in nextProduced */
86         Item nextProduced = means_of_production();
87         sharedbuf->Enqueue(nextProduced);
88     }
89 }
90
91 void consumer(void* buf)
92 {
93     MyBuffer* sharedbuf = reinterpret_cast<MyBuffer*>(buf);
94     for (;;) {
95         Item nextConsumed = sharedbuf->Dequeue();
96
97         /* next line abstractly consumes the item */
98         consume_item(nextConsumed);
99     }
100 }
101
102 Key point: *Threads* (the producer and consumer) are separate from
103 *shared object* (MyBuffer). The synchronization happens in the
104 shared object.
105

```

Feb 08, 23 0:04

handout05.txt

Page 3/4

```

106 2. This monitor is a model of a database with multiple readers and
107 writers. The high-level goal here is (a) to give a writer exclusive
108 access (a single active writer means there should be no other writers
109 and no readers) while (b) allowing multiple readers. Like the previous
110 example, this one is expressed in pseudocode.

```

```

111 // assume that these variables are initialized in a constructor
112 state variables:
113     AR = 0; // # active readers
114     AW = 0; // # active writers
115     WR = 0; // # waiting readers
116     WW = 0; // # waiting writers
117
118     Condition okToRead = NIL;
119     Condition okToWrite = NIL;
120     Mutex mutex = FREE;
121
122 Database::read() {
123     startRead(); // first, check self into the system
124     Access Data
125     doneRead(); // check self out of system
126 }
127
128 Database::startRead() {
129     acquire(&mutex);
130     while((AW + WW) > 0){
131         WR++;
132         wait(&okToRead, &mutex);
133     }
134     WR--;
135     AR++;
136     release(&mutex);
137 }
138
139 Database::doneRead() {
140     acquire(&mutex);
141     AR--;
142     if (AR == 0 && WW > 0) { // if no other readers still
143         signal(&okToWrite, &mutex); // active, wake up writer
144     }
145     release(&mutex);
146 }
147
148 Database::write(){ // symmetrical
149     startWrite(); // check in
150     Access Data
151     doneWrite(); // check out
152 }
153
154 Database::startWrite() {
155     acquire(&mutex);
156     while ((AW + AR) > 0) { // check if safe to write.
157         // if any readers or writers, wait
158         WW++;
159         wait(&okToWrite, &mutex);
160     }
161     WW--;
162     AW++;
163     release(&mutex);
164 }
165
166 Database::doneWrite() {
167     acquire(&mutex);
168     AW--;
169     if (WW > 0) {
170         signal(&okToWrite, &mutex); // give priority to writers
171     } else if (WR > 0) {
172         broadcast(&okToRead, &mutex);
173     }
174     release(&mutex);
175 }
176
177 NOTE: what is the starvation problem here?
178

```

Feb 08, 23 0:04

handout05.txt

Page 4/4

```

179
180 3. Shared locks
181
182 struct sharedlock {
183     int i;
184     Mutex mutex;
185     Cond c;
186 };
187
188 void AcquireExclusive (sharedlock *sl) {
189     acquire(&sl->mutex);
190     while (sl->i) {
191         wait (&sl->c, &sl->mutex);
192     }
193     sl->i = -1;
194     release(&sl->mutex);
195 }
196
197 void AcquireShared (sharedlock *sl) {
198     acquire(&sl->mutex);
199     while (sl->i < 0) {
200         wait (&sl->c, &sl->mutex);
201     }
202     sl->i++;
203     release(&sl->mutex);
204 }
205
206 void ReleaseShared (sharedlock *sl) {
207     acquire(&sl->mutex);
208     if (!--sl->i)
209         signal (&sl->c, &sl->mutex);
210     release(&sl->mutex);
211 }
212
213 void ReleaseExclusive (sharedlock *sl) {
214     acquire(&sl->mutex);
215     sl->i = 0;
216     broadcast (&sl->c, &sl->mutex);
217     release(&sl->mutex);
218 }
219
220 QUESTIONS:
221 A. There is a starvation problem here. What is it? (Readers can keep
222 writers out if there is a steady stream of readers.)
223 B. How could you use these shared locks to write a cleaner version
224 of the code in the prior item? (Though note that the starvation
225 properties would be different.)

```

```

1 Implementation of spinlocks and mutexes
2
3 1. Here is a BROKEN spinlock implementation:
4
5     struct Spinlock {
6         int locked;
7     }
8
9     void acquire(Spinlock *lock) {
10        while (1) {
11            if (lock->locked == 0) { // A
12                lock->locked = 1;    // B
13                break;
14            }
15        }
16    }
17
18    void release (Spinlock *lock) {
19        lock->locked = 0;
20    }
21
22    What's the problem? Two acquire()s on the same lock on different
23    CPUs might both execute line A, and then both execute B. Then
24    both will think they have acquired the lock. Both will proceed.
25    That doesn't provide mutual exclusion.
26

```

```

27
28 2. Correct spinlock implementation
29
30     Relies on atomic hardware instruction. For example, on the x86-64,
31     doing
32         "xchg addr, %rax"
33     does the following:
34
35     (i) freeze all CPUs' memory activity for address addr
36     (ii) temp <-- *addr
37     (iii) *addr <-- %rax
38     (iv) %rax <-- temp
39     (v) un-freeze memory activity
40
41     /* pseudocode */
42     int xchg_val(addr, value) {
43         %rax = value;
44         xchg (*addr), %rax
45     }
46
47     /* bare-bones version of acquire */
48     void acquire (Spinlock *lock) {
49         pushcli(); /* what does this do? */
50         while (1) {
51             if (xchg_val(&lock->locked, 1) == 0)
52                 break;
53         }
54     }
55
56     void release(Spinlock *lock){
57         xchg_val(&lock->locked, 0);
58         popcli(); /* what does this do? */
59     }
60
61     /* optimization in acquire; call xchg_val() less frequently */
62     void acquire(Spinlock* lock) {
63         pushcli();
64         while (xchg_val(&lock->locked, 1) == 1) {
65             while (lock->locked) ;
66         }
67     }
68
69     The above is called a *spinlock* because acquire() spins. The
70     bare-bones version is called a "test-and-set (TAS) spinlock"; the
71     other is called a "test-and-test-and-set spinlock".
72
73     The spinlock above is great for some things, not so great for
74     others. The main problem is that it *busy waits*: it spins,
75     chewing up CPU cycles. Sometimes this is what we want (e.g., if
76     the cost of going to sleep is greater than the cost of spinning
77     for a few cycles waiting for another thread or process to
78     relinquish the spinlock). But sometimes this is not at all what we
79     want (e.g., if the lock would be held for a while: in those
80     cases, the CPU waiting for the lock would waste cycles spinning
81     instead of running some other thread or process).
82
83     NOTE: the spinlocks presented here can introduce performance issues
84     when there is a lot of contention. (This happens even if the
85     programmer is using spinlocks correctly.) The performance issues
86     result from cross-talk among CPUs (which undermines caching and
87     generates traffic on the memory bus). If we have time later, we will
88     study a remediation of this issue (search the Web for "MCS locks").
89
90     ANOTHER NOTE: In everyday application-level programming, spinlocks
91     will not be something you use (use mutexes instead). But you should
92     know what these are for technical literacy, and to see where the
93     mutual exclusion is truly enforced on modern hardware.
94

```



Feb 08, 23 0:04

spinlock-mutex.txt

Page 3/3

95 3. Mutex implementation

96  
97  
98  
99  
100  
101

The intent of a mutex is to avoid busy waiting: if the lock is not available, the locking thread is put to sleep, and tracked by a queue in the mutex. The next page has an implementation.

Feb 08, 23 0:12

fair-mutex.c

Page 1/1

```

1 #include <sys/queue.h>
2
3 typedef struct thread {
4     // ... Entries elided.
5     STAILQ_ENTRY(thread_t) qlink; // Tail queue entry.
6 } thread_t;
7
8 struct Mutex {
9     // Current owner, or 0 when mutex is not held.
10    thread_t *owner;
11
12    // List of threads waiting on mutex
13    STAILQ(thread_t) waiters;
14
15    // A lock protecting the internals of the mutex.
16    Spinlock splock; // as in item 1, above
17 };
18
19 void mutex_acquire(struct Mutex *m) {
20
21    acquire(&m->splock);
22
23    // Check if the mutex is held; if not, current thread gets mutex and returns
24    if (m->owner == 0) {
25        m->owner = id_of_this_thread;
26        release(&m->splock);
27    } else {
28        // Add thread to waiters.
29        STAILQ_INSERT_TAIL(&m->waiters, id_of_this_thread, qlink);
30
31        // Tell the scheduler to add current thread to the list
32        // of blocked threads. The scheduler needs to be careful
33        // when a corresponding sched_wakeup call is executed to
34        // make sure that it treats running threads correctly.
35        sched_mark_blocked(&id_of_this_thread);
36
37        // Unlock spinlock.
38        release(&m->splock);
39
40        // Stop executing until woken.
41        sched_swch();
42
43        // When we get to this line, we are guaranteed to hold the mutex. This
44        // is because we can get here only if context-switched-TO, which itself
45        // can happen only if this thread is removed from the waiting queue,
46        // marked "unblocked", and set to be the owner (in mutex_release()
47        // below). However, we might have held the mutex in lines 39-42
48        // (if we were context-switched out after the spinlock release(),
49        // followed by being run as a result of another thread's release of the
50        // mutex). But if that happens, it just means that we are
51        // context-switched out an "extra" time before proceeding.
52    }
53 }
54
55 void mutex_release(struct Mutex *m) {
56    // Acquire the spinlock in order to make changes.
57    acquire(&m->splock);
58
59    // Assert that the current thread actually owns the mutex
60    assert(m->owner == id_of_this_thread);
61
62    // Check if anyone is waiting.
63    m->owner = STAILQ_GET_HEAD(&m->waiters);
64
65    // If so, wake them up.
66    if (m->owner) {
67        sched_wakeone(&m->owner);
68        STAILQ_REMOVE_HEAD(&m->waiters, qlink);
69    }
70
71    // Release the internal spinlock
72    release(&m->splock);
73 }

```

Handwritten annotations in purple:

- At line 1: `#include <sys/queue.h>` is annotated with `mutex acq:`.
- At line 3: `typedef struct thread {` is annotated with `spinlock acq`.
- At line 8: `struct Mutex {` is annotated with `spinlock rel.`.
- At line 10: `thread_t *owner;` is annotated with `C.S.` (Context Switch).
- At line 19: `void mutex_acquire(struct Mutex *m) {` is annotated with `mutex rel:`.
- At line 21: `acquire(&m->splock);` is annotated with `spinlock acq`.
- At line 24: `if (m->owner == 0) {` is annotated with `spinlock rel.`.

Feb 13, 23 6:36

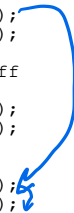
handout06.txt

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

```

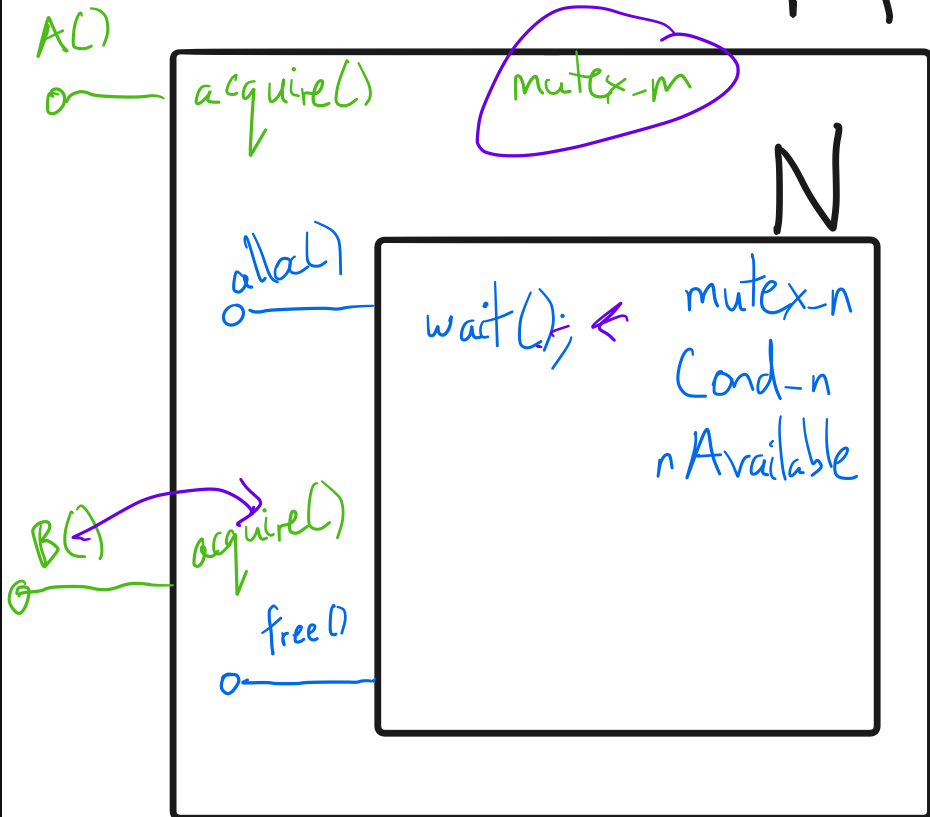
```

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

```

M

N



```

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

```

Feb 13, 23 6:36

handout06.txt

Page 5/7

```

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

```

Feb 13, 23 6:36

handout06.txt

Page 6/7

```

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
253             mutex_acquire(&mutex);
254             if (!ready) {
255                 ptr = alloc_foo(); /* <-- sets ptr to be non-zero */
256                 ready = 1;
257             }
258
259             mutex_release(&mutex);
260
261         }
262         return;
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/
```