```
Handout for CS 439
Class 6
5 February 2013

1. Simple deadlock example

    T1:
        acquire(mutexA);
        acquire(mutexB);

        // do some stuff

        release(mutexB);
        release(mutexA);

    T2:
        acquire(mutexB);
        acquire(mutexA);

        // do some stuff

        release(mutexA);
        release(mutexB);
```



```
2. More subtle deadlock example

    Let M be a monitor (shared object with methods protected by mutex)
    Let N be another monitor

    class M {
        private:
            Mutex mutex_m;

            // instance of monitor N
            N another_monitor;

            // Assumption: no other objects in the system hold a pointer
            // to our "another_monitor"

        public:
            M();
            ~M();
            void methodA();
            void methodB();
    };

    class N {
        private:
            Mutex mutex_n;
            Cond cond_n;
            int navailable;

        public:
            N();
            ~N();
            void* alloc(int nwanted);
            void  free(void*);
    }

    int
    N::alloc(int nwanted) {
        acquire(&mutex_n);
        while (navailable < nwanted) {
            wait(&cond_n, &mutex_n);
        }

        // peel off the memory

        navailable −= nwanted;
        release(&mutex_n);
    }

    void
    N::free(void* returning_mem) {

        acquire(&mutex_n);

        // put the memory back

        navailable += returning_mem;

        broadcast(&cond_n, &mutex_n);

        release(&mutex_n);
    }

    void
    M::methodA() {

        acquire(&mutex_m);

        void* new_mem = another_monitor.alloc(int nbytes);

        // do a bunch of stuff using this nice
        // chunk of memory n allocated for us

        release(&mutex_m);
```

```
98      }
99
100     void
101     M::methodB() {
102
103         acquire(&mutex_m);
104
105         // do a bunch of stuff
106
107         another_monitor.free(some_pointer);
108
109         release(&mutex_m);
110     }
111
112     QUESTION: What's the problem?
113
```



```
114 3. Performance vs. complexity trade−off with locks
115
116 /*
117  *      linux/mm/filemap.c
118  *
119  * Copyright (C) 1994−1999  Linus Torvalds
120  */
121
122 /*
123  * This file handles the generic file mmap semantics used by
124  * most "normal" filesystems (but you don't /have/ to use this:
125  * the NFS filesystem used to do this differently, for example)
126  */
127 #include <linux/config.h>
128 #include <linux/module.h>
129 #include <linux/slab.h>
130 #include <linux/compiler.h>
131 #include <linux/fs.h>
132 #include <linux/aio.h>
133 #include <linux/capability.h>
134 #include <linux/kernel_stat.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/pagevec.h>
144 #include <linux/blkdev.h>
145 #include <linux/security.h>
146 #include <linux/syscalls.h>
147 #include "filemap.h"
148 /*
149  * FIXME: remove all knowledge of the buffer layer from the core VM
150  */
151 #include <linux/buffer_head.h> /* for generic_osync_inode */
152
153 #include <asm/uaccess.h>
154 #include <asm/mman.h>
155
156 static ssize_t
157 generic_file_direct_IO(int rw, struct kiocb *iocb, const struct iovec *iov,
158         loff_t offset, unsigned long nr_segs);
159
160 /*
161  * Shared mappings implemented 30.11.1994. It's not fully working yet,
162  * though.
163  *
164  * Shared mappings now work. 15.8.1995  Bruno.
165  *
166  * finished 'unifying' the page and buffer cache and SMP−threaded the
167  * page−cache, 21.05.1999, Ingo Molnar <mingo@redhat.com>
168  *
169  * SMP−threaded pagemap−LRU 1999, Andrea Arcangeli <andrea@suse.de>
170  */
171
172 /*
173  * Lock ordering:
174  *
175  *  ->i_mmap_lock               (vmtruncate)
176  *    ->private_lock            (__free_pte->__set_page_dirty_buffers)
177  *      ->swap_lock             (exclusive_swap_page, others)
178  *        ->mapping->tree_lock
179  *
180  *  ->i_mutex
181  *    ->i_mmap_lock             (truncate->unmap_mapping_range)
182  *
183  *  ->mmap_sem
184  *    ->i_mmap_lock
185  *      ->page_table_lock or pte_lock   (various, mainly in memory.c)
186  *        ->mapping->tree_lock  (arch−dependent flush_dcache_mmap_lock)
```

```
187  *
188  *  ->mmap_sem
189  *    ->lock_page               (access_process_vm)
190  *
191  *  ->mmap_sem
192  *    ->i_mutex                 (msync)
193  *
194  *  ->i_mutex
195  *    ->i_alloc_sem             (various)
196  *
197  *  ->inode_lock
198  *    ->sb_lock                 (fs/fs−writeback.c)
199  *    ->mapping->tree_lock      (__sync_single_inode)
200  *
201  *  ->i_mmap_lock
202  *    ->anon_vma.lock           (vma_adjust)
203  *
204  *  ->anon_vma.lock
205  *    ->page_table_lock or pte_lock     (anon_vma_prepare and various)
206  *
207  *  ->page_table_lock or pte_lock
208  *    ->swap_lock               (try_to_unmap_one)
209  *    ->private_lock            (try_to_unmap_one)
210  *    ->tree_lock               (try_to_unmap_one)
211  *    ->zone.lru_lock           (follow_page->mark_page_accessed)
212  *    ->zone.lru_lock           (check_pte_range->isolate_lru_page)
213  *    ->private_lock            (page_remove_rmap->set_page_dirty)
214  *    ->tree_lock               (page_remove_rmap->set_page_dirty)
215  *    ->inode_lock              (page_remove_rmap->set_page_dirty)
216  *    ->inode_lock              (zap_pte_range->set_page_dirty)
217  *    ->private_lock            (zap_pte_range->__set_page_dirty_buffers)
218  *
219  *  ->task->proc_lock
220  *    ->dcache_lock             (proc_pid_lookup)
221  */
222
223 /*
224  * Remove a page from the page cache and free it. Caller has to make
225  * sure the page is locked and that nobody else uses it − or that usage
226  * is safe.  The caller must hold a write_lock on the mapping's tree_lock.
227  */
228 void __remove_from_page_cache(struct page *page)
229 {
230         struct address_space *mapping = page->mapping;
231
232 ..............
233
234 [point of this item on the handout: fine−grained locking leads to complexity]
```