```
CS5600, Handout week 7.a

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);
    }

    void
     M::methodB() {

       acquire(&mutex_m);

       // do a bunch of stuff

       another_monitor.free(some_pointer);

       release(&mutex_m);
     }

QUESTION: What's the problem?
```

```
112
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
```



```
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
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
```

```
Implementation of spinlocks and mutexes

1. Here is a BROKEN spinlock implementation:

   struct Spinlock {
       int locked;
   }

   void acquire(Spinlock *lock) {
       while (1) {
           if (lock->locked == 0) { // A
               lock->locked = 1; // B
               break;
           }
       }
   }

   void release (Spinlock *lock) {
       lock->locked = 0;
   }

What's the problem? Two acquire()s on the same lock on different
CPUs might both execute line A, and then both execute B. Then
both will think they have acquired the lock. Both will proceed.
That doesn't provide mutual exclusion.


```



```


2. Correct spinlock implementation

Relies on atomic hardware instruction. For example, on the x86-64,
doing
       "xchg addr, %rax"
does the following:

       (i) freeze all CPUs' memory activity for address addr
       (ii) temp <-- *addr
       (iii) *addr <-- %rax
       (iv) %rax <-- temp
       (v) un-freeze memory activity

       /* pseudocode */
       int xchg_val(addr, value) {
           %rax = value;
           xchg (*addr), %rax
       }

    /* bare-bones version of acquire */
    void acquire (Spinlock *lock) {
        pushcli(); /* what does this do? */
        while (1) {
            if (xchg_val(&lock->locked, 1) == 0)
                break;
        }
    }

    void release(Spinlock *lock){
        xchg_val(&lock->locked, 0);
        popcli(); /* what does this do? */
    }

    /* optimization in acquire; call xchg_val() less frequently */
    void acquire(Spinlock* lock) {
        pushcli();
        while (xchg_val(&lock->locked, 1) == 1) {
            while (lock->locked) ;
        }
    }

    The above is called a *spinlock* because acquire() spins. The
    bare-bones version is called a "test-and-set (TAS) spinlock"; the
    other is called a "test-and-test-and-set spinlock".

    The spinlock above is great for some things, not so great for
    others. The main problem is that it *busy waits*: it spins,
    chewing up CPU cycles. Sometimes this is what we want (e.g., if
    the cost of going to sleep is greater than the cost of spinning
    for a few cycles waiting for another thread or process to
    relinquish the spinlock). But sometimes this is not at all what we
    want (e.g., if the lock would be held for a while: in those
    cases, the CPU waiting for the lock would waste cycles spinning
    instead of running some other thread or process).

    NOTE: the spinlocks presented here can introduce performance issues
    when there is a lot of contention. (This happens even if the
    programmer is using spinlocks correctly.) The performance issues
    result from cross-talk among CPUs (which undermines caching and
    generates traffic on the memory bus). If we have time later, we will
    study a remediation of this issue (search the Web for "MCS locks").

    ANOTHER NOTE: In everyday application-level programming, spinlocks
    will not be something you use (use mutexes instead). But you should
    know what these are for technical literacy, and to see where the
    mutual exclusion is truly enforced on modern hardware.


```

3. Mutex implementation

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.

```
#include <sys/queue.h>

typedef struct thread {
    // ... Entries elided.
    STAILQ_ENTRY(thread_t) qlink; // Tail queue entry.
} thread_t;

struct Mutex {
    // Current owner, or 0 when mutex is not held.
    thread_t *owner;

    // List of threads waiting on mutex
    STAILQ(thread_t) waiters;

    // A lock protecting the internals of the mutex.
    Spinlock splock; // as in item 1, above
};

void mutex_acquire(struct Mutex *m) {

    acquire(&m->splock);

    // Check if the mutex is held; if not, current thread gets mutex and returns
    if (m->owner == 0) {
        m->owner = id_of_this_thread;
        release(&m->splock);
    } else {
        // Add thread to waiters.
        STAILQ_INSERT_TAIL(&m->waiters, id_of_this_thread, qlink);

        // Tell the scheduler to add current thread to the list
        // of blocked threads. The scheduler needs to be careful
        // when a corresponding sched_wakeup call is executed to
        // make sure that it treats running threads correctly.
        sched_mark_blocked(&id_of_this_thread);

        // Unlock spinlock.
        release(&m->splock);

        // Stop executing until woken.
        sched_swtch();

        // When we get to this line, we are guaranteed to hold the mutex. This
        // is because we can get here only if context-switched-TO, which itself
        // can happen only if this thread is removed from the waiting queue,
        // marked "unblocked",/ and set to be the owner (in mutex_release()
        // below). However, we might actually have held the mutex in lines 141-144
        // (if we were context-switched out after the spinlock release(),
        // followed by being run as a result of another thread's release of the
        // mutex). But if that happens, it just means that we are
        // context-switched out an "extra" time before proceeding.
    }
}

```



```
void mutex_release(struct Mutex *m) {
    // Acquire the spinlock in order to make changes.
    acquire(&m->splock);

    // Assert that the current thread actually owns the mutex
    assert(m->owner == id_of_this_thread);

    // Check if anyone is waiting.
    m->owner = STAILQ_GET_HEAD(&m->waiters);

    // If so, wake them up.
    if (m->owner) {
        sched_wakeone(&m->owner);
        STAILQ_REMOVE_HEAD(&m->waiters, qlink);
    }

    // Release the internal spinlock
    release(&m->splock);
}
```