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;
        }
```

    Here's inline assembly:

```
        uint32_t cmpxchg_val(uint32_t* addr, uint32_t oldval, uint32_t newval) {
            uint32_t was;
            asm volatile("lock cmpxchg %3, %0"
                            : "+m" (*addr), "=a" (was)
                            : "a" (oldval), "r" (newval)
                            : "cc");
            return was;
        }
```

B. MCS 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.

    Each CPU has a qnode structure in *local* memory. Here, local can
    mean local memory in NUMA machine or its own cache line that other
    CPUs are not allowed to cache (i.e., the cache line is in exclusive
    mode):

```
    typedef struct qnode {
        struct qnode* next;
        bool someoneelse_locked;
    } qnode;

    typedef qnode* lock;  // a lock is a pointer to a qnode
```

    --The lock itself is literally the *tail* of the list of CPUs holding
    or waiting for the lock.

    --While waiting, a CPU spins on its local "locked" flag. Here's the
    code for acquire:

```
        // lockp is a qnode**. I points to our local qnode.
        void acquire(lock* lockp, qnode* I) {

            I->next = NULL;
            qnode* predecessor;

            // next line makes lockp point to I (that is, it sets *lockp <-- I)
            // and returns the old value of *lockp. Uses atomic operation
            // XCHG. see earlier in handout (or earlier handouts)
            // for implementation of xchg_val.

            predecessor = xchg_val(lockp, I);    // "A"
            if (predecessor != NULL) { // queue was non-empty
                I->someoneelse_locked = true;

                predecessor->next = I;           // "B"
                while (I->someoneelse_locked) ;    // spin
            }
            // we hold the lock!
        }
```

        What's going on?

        --If the lock is unlocked, then *lockp == NULL.

        --If the lock is locked, and there are no waiters, then *lockp
        points to the qnode of the owner

        --If the lock is locked, and there are waiters, then *lockp points
        to the qnode at the tail of the waiter list.

    --Here's the code for release:

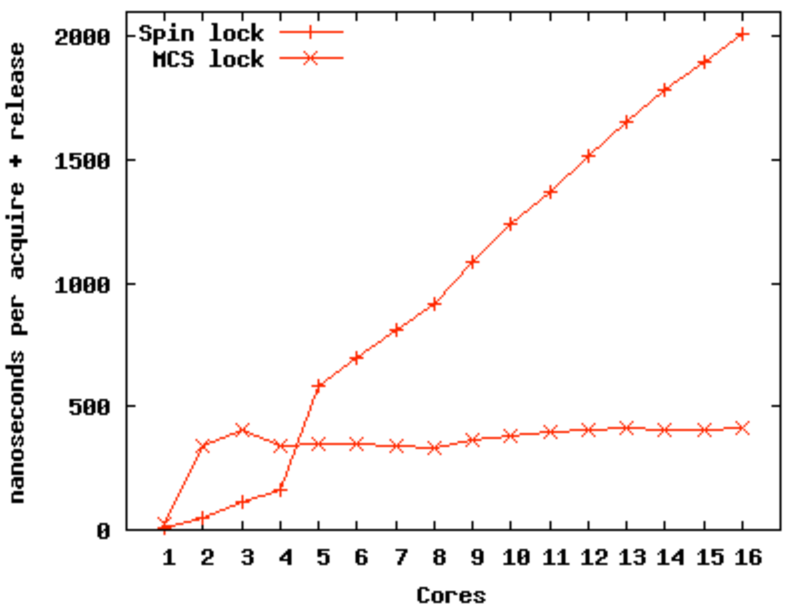
```
        void release(lock* lockp, qnode* I) {
            if (!I->next)   { // no known successor
                if (cmpxchg_val(lockp, I, NULL) == I) {     // "C"
                    // swap successful: lockp was pointing to I, so now
                    // *lockp == NULL, and the lock is unlocked. we can
                    // go home now.
                    return;
                }
                // if we get here, then there was a timing issue: we had
                // no known successor when we first checked, but now we
                // have a successor: some CPU executed the line "A"
                // above. Wait for that CPU to execute line "B" above.
                while (!I->next) ;
            }

            // handing the lock off to the next waiter is as simple as
            // just setting that waiter's "someoneelse_locked" flag to false
            I->next->someoneelse_locked = false;
        }
```

        What's going on?

        --If I->next == NULL and *lockp == I, then no one else is
        waiting for the lock. So we set *lockp == NULL.

        --If I->next == NULL and *lockp != I, then another CPU is in
        acquire (specifically, it executed its atomic operation, namely
        line "A", before we executed ours, namely line "C"). So wait for
        the other CPU to put the list in a sane state, and then drop
        down to the next case:

        --If I->next != NULL, then we know that there is a spinning
        waiter (the oldest one). Hand it the lock by setting its flag to
        false.



Time required to acquire and release a lock on a 16-core AMD machine when varying number of cores contend for the lock. The two lines show Linux kernel spin locks and MCS locks (on Corey). A spin lock with one core takes about 11 nanoseconds; an MCS lock about 26 nanoseconds.

[Reprinted with permission from S. Boyd-Wickizer et al. Corey: An Operating System for Many Cores. Proceedings of Symposium on Operating Systems Design and Implementation (OSDI), December 2008.]

```
Performance v complexity trade−off with locks

/*
 *      linux/mm/filemap.c
 *
 * Copyright (C) 1994−1999  Linus Torvalds
 */

/*
 * This file handles the generic file mmap semantics used by
 * most "normal" filesystems (but you don’t /have/ to use this:
 * the NFS filesystem used to do this differently, for example)
 */
#include <linux/config.h>
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/compiler.h>
#include <linux/fs.h>
#include <linux/aio.h>
#include <linux/capability.h>
#include <linux/kernel_stat.h>
#include <linux/mm.h>
#include <linux/swap.h>
#include <linux/mman.h>
#include <linux/pagemap.h>
#include <linux/file.h>
#include <linux/uio.h>
#include <linux/hash.h>
#include <linux/writeback.h>
#include <linux/pagevec.h>
#include <linux/blkdev.h>
#include <linux/security.h>
#include <linux/syscalls.h>
#include "filemap.h"
/*
 * FIXME: remove all knowledge of the buffer layer from the core VM
 */
#include <linux/buffer_head.h> /* for generic_osync_inode */

#include <asm/uaccess.h>
#include <asm/mman.h>

static ssize_t
generic_file_direct_IO(int rw, struct kiocb *iocb, const struct iovec *iov,
        loff_t offset, unsigned long nr_segs);

/*
 * Shared mappings implemented 30.11.1994. It’s not fully working yet,
 * though.
 *
 * Shared mappings now work. 15.8.1995  Bruno.
 *
 * finished ’unifying’ the page and buffer cache and SMP−threaded the
 * page−cache, 21.05.1999, Ingo Molnar <mingo@redhat.com>
 *
 * SMP−threaded pagemap−LRU 1999, Andrea Arcangeli <andrea@suse.de>
 */

/*
 * Lock ordering:
 *
 *  −>i_mmap_lock               (vmtruncate)
 *    −>private_lock            (__free_pte−>__set_page_dirty_buffers)
 *      −>swap_lock             (exclusive_swap_page, others)
 *        −>mapping−>tree_lock
 *
 *  −>i_mutex
 *    −>i_mmap_lock             (truncate−>unmap_mapping_range)
 *
 *  −>mmap_sem
 *    −>i_mmap_lock
 *      −>page_table_lock or pte_lock   (various, mainly in memory.c)
 *        −>mapping−>tree_lock  (arch−dependent flush_dcache_mmap_lock)
```

```
 *
 *  −>mmap_sem
 *    −>lock_page               (access_process_vm)
 *
 *  −>mmap_sem
 *    −>i_mutex                 (msync)
 *
 *  −>i_mutex
 *    −>i_alloc_sem             (various)
 *
 *  −>inode_lock
 *    −>sb_lock                 (fs/fs−writeback.c)
 *    −>mapping−>tree_lock      (__sync_single_inode)
 *
 *  −>i_mmap_lock
 *    −>anon_vma.lock           (vma_adjust)
 *
 *  −>anon_vma.lock
 *    −>page_table_lock or pte_lock     (anon_vma_prepare and various)
 *
 *  −>page_table_lock or pte_lock
 *    −>swap_lock               (try_to_unmap_one)
 *    −>private_lock            (try_to_unmap_one)
 *    −>tree_lock               (try_to_unmap_one)
 *    −>zone.lru_lock           (follow_page−>mark_page_accessed)
 *    −>zone.lru_lock           (check_pte_range−>isolate_lru_page)
 *    −>private_lock            (page_remove_rmap−>set_page_dirty)
 *    −>tree_lock               (page_remove_rmap−>set_page_dirty)
 *    −>inode_lock              (page_remove_rmap−>set_page_dirty)
 *    −>inode_lock              (zap_pte_range−>set_page_dirty)
 *    −>private_lock            (zap_pte_range−>__set_page_dirty_buffers)
 *
 *  −>task−>proc_lock
 *    −>dcache_lock             (proc_pid_lookup)
 */

/*
 * Remove a page from the page cache and free it. Caller has to make
 * sure the page is locked and that nobody else uses it − or that usage
 * is safe.  The caller must hold a write_lock on the mapping’s tree_lock.
 */
void __remove_from_page_cache(struct page *page)
{
        struct address_space *mapping = page−>mapping;

.............

[point of this item on the handout: fine−grained locking leads to complexity]
```

```
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?


```