### 1. How can we implement lock, acquire(), and release()?

**1a.** Here is an A BADLY BROKEN implementation:

```c
void [BROKEN] acquire(Lock *lock) {
    while (1) {
        if (lock->locked == 0) { // C
            lock->locked = 1;    // D
            break;
        }
    }
}
```

- This implementation is **badly broken** because 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 wasting the lock would waste cycles spinning instead of running some other thread or process).

**1b.** Here’s a way that is correct but only sometimes appropriate:

```c
int xchg_val(addr, value) {
    %eax = value;
    xchg (*addr, %eax);
}
```

- This implementation uses an atomic instruction on the CPU. For example, on the x86, doing
  
  ```c
  *xchg, %eax
  ```
  
  does the following:
  - (i) freeze all CPUs’ memory activity for address `addr`
  - (ii) `temp = *addr`
  - (iii) `*addr = %eax`
  - (iv) `%eax = temp`
  - (v) unfreeze memory activity

  /* pseudocode */

  ```c
  int xchg_val(addr, value) {
      %eax = value;
      xchg (*addr, %eax);
  }
  ```

**1c.** Here’s an object that does not involve busy waiting; it can work as the list_lock mentioned above. Note: the “threads” here can be user-level threads, kernel threads, or threads-inside-kernel. The concept is the same in all cases.

```c
struct Mutex {
    bool is_held;           /* true if mutex held */
    thread_id owner;     /* thread holding mutex, if locked */
    thread_list waiters;    /* queue of thread TCBs */
}
```

- What’s the problem? Two acquire()s on the same lock on different CPUs might both execute line C, and then both execute D. Then both will think they have acquired the lock. This is the same kind of race that we were trying to eliminate in insert(). But we have made a little progress: now we only need a way to prevent interleaving in one place (acquire()), not for many arbitrary complex sequences of code.

**1d.** Here’s a way that is correct but only sometimes appropriate:

```c
void acquire(Mutex *m) {
    acquire(&m->wait_lock);   /* we spin to acquire wait_lock */
    while (m->is_held) {     /* someone else has the mutex */
        m->waiters.insert(current_thread)
        release(&m->wait_lock);
        schedule();   /* run a thread that is on the ready list */
        acquire(&m->wait_lock);   /* we spin again */
        m->is_held = true;     /* we now hold the mutex */
        m->owner = self;
        release(&m->wait_lock);
    }

    void mutex_release(Mutex *m) {
        acquire(&m->wait_lock);   /* we spin to acquire wait_lock */
        while (m->is_held) {     /* someone else has the mutex */
            m->waiters.insert(current_thread)
            release(&m->wait_lock);
            schedule();   /* run a thread that is on the ready list */
            acquire(&m->wait_lock);   /* we spin again */
            m->is_held = false;
            m->owner = 0;
            wake_up_a_waiter(m->waiters); /* select and run a waiter */
            release(&m->wait_lock);
        }
    }
```

### 2. What’s the problem?

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 wasting the lock would waste cycles spinning instead of running some other thread or process).

### Footnote

**NOTE:** Unfortunately, insert() with these locks is correct only if there are some constraints on the order in which the CPU carries out memory reads and writes. For example, if insert() were executed so that the read at A appeared to another processor (and to memory) to be executed before the acquire(), then insert() would be incorrect even with locks.

How do we get the required guarantee? Answer: by ensuring that neither the programmer nor the processor reordered instructions with respect to the acquire().
2. Terminology

To avoid confusion, we will use the following terminology in this course (you will hear other terminology elsewhere):

− A "lock" is an abstract object that provides mutual exclusion
− A "spinlock" is a lock that works by busy waiting, as in 6b
− A "mutex" is a lock that works by having a "waiting" queue and then protecting that waiting queue with atomic hardware instructions, as in 6c. The most natural way to "use the hardware" is with a spinlock, but there are others, such as turning off interrupts, which works if we're on a single CPU machine.

3. Producer/consumer example [also known as bounded buffer]

3a. Recall buggy implementation

```c
/*
 * buffer* stores BUFFER_SIZE items
 * count* is number of used slots. a variable that lives in memory
 * out* is next empty buffer slot to fill (if any)
 * in* is oldest filled slot to consume (if any)
 */

void producer (void *ignored) {
    for (;;) {
        nextProduced = means_of_production();
        while (count == BUFFER_SIZE);
        buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
    }
}

void consumer (void *ignored) {
    for (;;) {
        while (count == 0);
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        consume_item(nextConsumed);
    }
}
```

− Review: what's the problem?
− Answer: count++ and count-- might compile to, respectively:

```c
reg1 <- count  # load
reg1 <- reg1 + 1  # increment register
count <- reg1  # store
```

− Review: why not use instructions like "addl $0x1, _count"?
− Answer: not atomic if there are multiple CPUs.

− Review: so why not use "LOCK addl $0x1, _count"?
− Answer: we could do that here, but LOCK won't save us every time critical sections: locks (or mutexes).
3b. Producer/consumer [bounded buffer] using mutexes

```c
Mutex mutex;

void producer (void *ignored) {
    /* next line produces an item and puts it in nextProduced */
    nextProduced = means_of_production();
    acquire(&mutex);
    while (count == BUFFER_SIZE) {
        release(&mutex);
        yield(); /* or schedule() */
        acquire(&mutex);
    }
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
    release(&mutex);
}

void consumer (void *ignored) {
    for (;;) {
        acquire(&mutex);
        while (count == 0) {
            release(&mutex);
            yield(); /* or schedule() */
            acquire(&mutex);
        }
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        release(&mutex);
        /* next line abstractly consumes the item */
        consume_item(nextConsumed);
    }
}
```

3c. Producer/consumer [bounded buffer] using mutexes and condition variables

```c
Mutex mutex;
Cond nonempty;
Cond nonfull;

void producer (void *ignored) {
    /* next line produces an item and puts it in nextProduced */
    nextProduced = means_of_production();
    acquire(&mutex);
    while (count == BUFFER_SIZE)
        cond_wait(&nonfull, &mutex);
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
    cond_signal(&nonempty, &mutex);
    release(&mutex);
}

void consumer (void *ignored) {
    for (;;) {
        acquire(&mutex);
        while (count == 0)
            cond_wait(&nonempty, &mutex);
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        cond_signal(&nonfull, &mutex);
        release(&mutex);
        /* next line abstractly consumes the item */
        consume_item(nextConsumed);
    }
}
```

Question: why does cond_wait need to both release the mutex and sleep? Why not:

```c
while (count == BUFFER_SIZE) {
    release(&mutex);
    cond_wait(&nonfull);
    acquire(&mutex);
}
```
3d. Producer/consumer (bounded buffer) with semaphores

Semaphore mutex(1); /* mutex initialized to 1 */
Semaphore empty(BUFFER_SIZE); /* start with BUFFER_SIZE empty slots */
Semaphore full(0); /* 0 full slots */

void producer (void *ignored) {
    for (;;) {
        nextProduced = means_of_production();
        sem_down(&empty);
        sem_down(&mutex); /* get exclusive access */
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        sem_up(&mutex);
        sem_up(&full); /* we just increased the # of full slots */
    }
}

void consumer (void *ignored) {
    for (;;) {
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        sem_up(&mutex);
        sem_up(&empty); /* one further empty slot */
        consume_item(nextConsumed);
    }
}

Semaphores *can* (not always) lead to elegant solutions (notice
that the code above is fewer lines than 3c) but they are much
harder to use.

The fundamental issue is that semaphores make implicit (counts,
conditions, etc.) what is probably best left explicit. Moreover,
they *also* implement mutual exclusion.

For this reason, you should not use semaphores. This example is
here mainly for completeness and so you know what a semaphore
is. But do not code with them. Solutions that use semaphores in
this course will receive no credit.

4. Example of a monitor: MyBuffer

class MyBuffer {
    public:
        MyBuffer();
        ~MyBuffer();
        void Enqueue(Item);
        Item Dequeue();
    private:
        int count;
        int in;
        int out;
        Item buffer[BUFFER_SIZE];
        Mutex* mutex;
        Cond* nonempty;
        Cond* nonfull;
    }

void MyBuffer::MyBuffer() {
    in = out = count = 0;
    mutex = new Mutex;
    nonempty = new Cond;
    nonfull = new Cond;
}

void MyBuffer::Enqueue(Item item) {
    mutex.acquire();
    while (count == BUFFER_SIZE)
        cond_wait(&nonfull, &mutex);
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
    sem_up(&mutex);
    sem_up(&empty); /* we just increased the # of full slots */
    mutex.release();
    count++;
}

Item MyBuffer::Dequeue() {
    mutex.acquire();
    while (count == 0)
        cond_wait(&nonempty, &mutex);
    Item ret = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    sem_up(&empty); /* one further empty slot */
    sem_up(&mutex);
    mutex.release();
    return ret;
}
int main(int, char**) {
    MyBuffer buf;
    int dummy;
    tid1 = thread_create(producer, &buf);
    tid2 = thread_create(consumer, &buf);
    thread_join(tid1);
    // never reach this point
    return -1;
}

void producer(void* buf) {
    MyBuffer* sharedbuf = reinterpret_cast<MyBuffer*>(buf);
    for (;;) {
        Item nextProduced = means_of_production();
        sharedbuf->Enqueue(nextProduced);
    }
}

void consumer(void* buf) {
    MyBuffer* sharedbuf = reinterpret_cast<MyBuffer*>(buf);
    for (;;) {
        Item nextConsumed = sharedbuf->Dequeue();
        /* next line abstractly consumes the item */
        consume_item(nextConsumed);
    }
}

Key point: *Threads* (the producer and consumer) are separate from
*shared object* (MyBuffer). The synchronization happens in the
shared object.

5. Readers/writers

state variables:
AR = 0;  // # active readers
AW = 0;  // # active writers
WR = 0;  // # waiting readers
WW = 0;  // # waiting writers
Condition okToRead = NIL;
Condition okToWrite = NIL;
Mutex mutex = FREE;

Database::read() {
    startRead();  // first, check self into the system
    Access Data
doneRead();   // check self out of system
}

Database::startRead() {
    acquire(&mutex);
    while((AW + WR) > 0) { // check if safe to write.
        WW++;
        wait(&okToWrite, &mutex);
        WW--;
    }
    AW++;
    release(&mutex);
}

Database::doneRead() {
    acquire(&mutex);
    AW--;
    if (WW > 0) {
        signal(&okToWrite, &mutex); // give priority to writers
    } else if (WR > 0) {
        broadcast(&okToRead, &mutex);
    }
    release(&mutex);
}

NOTE: what is the starvation problem here?
struct sharedlock {
    int i;
    Mutex mutex;
    Cond c;
};

void AcquireExclusive (sharedlock *sl) {
    acquire(&sl->mutex);
    while (sl->i) {
        wait (&sl->c, &sl->mutex);
        sl->i = -1;
        release(&sl->mutex);
    }
    sl->i = -1;
    release(&sl->mutex);
}

void AcquireShared (sharedlock *sl) {
    acquire(&sl->mutex);
    while (sl->i < 0) {
        wait (&sl->c, &sl->mutex);
    }
    sl->i++;
    release(&sl->mutex);
}

void ReleaseShared (sharedlock *sl) {
    acquire(&sl->mutex);
    if (!--sl->i)
        signal (&sl->c, &sl->mutex);
    release(&sl->mutex);
}

void ReleaseExclusive (sharedlock *sl) {
    acquire(&sl->mutex);
    sl->i = 0;
    broadcast (&sl->c, &sl->mutex);
    release(&sl->mutex);
}

QUESTIONS:
A. There is a starvation problem here. What is it? (Readers can keep
    writers out if there is a steady stream of readers.)
B. How could you use these shared locks to write a cleaner version
    of the code in item 5., above? (Though note that the starvation
    properties would be different.)
13. [12 points] Consider the function doublecheck_alloc() below, which is intended to be invoked from multiple threads on a multiprocessor machine. Its purpose is to avoid a mutex acquisition in the common case that ptr is already initialized. The requirements for this function are:

(i) doublecheck_alloc() must call alloc_foo() no more than once over the whole execution.
(ii) A caller of doublecheck_alloc() must, after the function returns, observe ptr as non-zero.

The machine does not offer sequential consistency. Thus, a processor is not guaranteed to see the memory operations of another processor in program order. However, each of mutex_acquire() and mutex_release() is implemented correctly; in particular, each of them internally contains a memory barrier (mfence on the x86). Recall that mfence ensures that all memory operations before the mfence barrier appear to all processors to have executed before all memory operations after the mfence barrier.

On the other hand, the compiler preserves program order (it does not reorder instructions).

```
struct foo {
    int abc;
    int def;
};
static int ready = 0;
static mutex_t mutex;
static struct foo* ptr = 0;

void doublecheck_alloc(){
    if (!ready) { /* <-- accesses shared variable w/out holding mutex */
        mutex_acquire(&mutex);
        if (!ready) {
            ptr = alloc_foo(); /* <-- sets ptr to be non-zero */
            ready = 1;
        }
        mutex_release(&mutex);
    }
    return;
}
```

The above code certainly violates our coding standards, but this problem is about whether it violates requirements (i) and (ii), above. The questions are given on the next page.
The AMD 16-core system topology. Memory access latency is in cycles and listed before the backslash. Memory bandwidth is in bytes per cycle and listed after the backslash. The measurements reflect the latency and bandwidth achieved by a core issuing load instructions. The measurements for accessing the L1 or L2 caches of a different core on the same chip are the same. The measurements for accessing any cache on a different chip are the same. Each cache line is 64 bytes, L1 caches are 64 Kbytes 8-way set associative, L2 caches are 512 Kbytes 16-way set associative, and L3 caches are 2 Mbytes 32-way set associative.

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:

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

Here’s inline assembly:

```c
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), +t" (newval)) {
        : "m"
    }
    return was;
}
```

B. MCS locks


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):

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

```c
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:

```c
void acquire(lock* lockp, qnode* I) {
    I->next = NULL;
    qnode* predecessor = xchg_val(lockp, I);    // "A"
    if (predecessor != NULL) { // queue was non-empty
        I->someoneelse_locked = true;
        predecessor->next = I;  // "B"
    } else { // if lockp is unlocked
        while (I->someoneelse_locked) ;    // spin
    }
    // we hold the lock!
    return was;
}
```

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:

```c
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.

Performance v complexity trade-off with locks

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/autorelease.h>
include <linux/hash.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);

/*
 * 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/autorelease.h>
# include <linux/hash.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.
 */

/*
 * 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
 *  −>inode_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)
 */

/*
 * 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.
 */

/*
 *  −>mmap_sem
 *  −>lock_page (access_process_vm)
 *  −>mmap_sem
 *  −>page_table_lock or pte_lock (various, mainly in memory.c)
 */

/*
 *  −>i_mutex
 *  −>i_alloc_sem (various)
 *  −>inode_lock
 *  −>fsfs−writeback.c
 *  −>mapping->tree_lock (__sync_single_inode)
 *  −>i_mmap_lock
 *  −>anon_vma.lock (vm_adj)
 *  −>anon_vma.lock
 *  −>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.ldr_lock (follow_page−>mark_page_accessed)
 *  −>zone.ldr_lock (check_pte_range−>isolate_lru_page)
 *  −>private_lock (page_remove_map->set_page_dirty)
 *  −>tree_lock (page_remove_map->set_page_dirty)
 *  −>inode_lock (page_remove_map->set_page_dirty)
 *  −>inode_lock (zap_pte_range->__set_page_dirty)
 *  −>private_lock (zap_pte_range->__set_page_dirty)
 *  −>task->proc_lock
 *  −>dcache_lock (proc_pid_lookup)
 */

/*
 *  −>private_lock
 *  −>task->proc_lock
 */

}