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 l09 handout 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
    }
    I->next = predecessor;  // "C"
    if (predecessor != NULL) { // queue was non-empty
        predecessor->next = I;
        I->someoneelse_locked = true;
        predecessor = xchg_val(lockp, I);  // "A"
    }
    while (!I->someoneelse_locked);  // spin
    // we hold the lock!
}

Here’s pseudocode:
int cmpxchg_val(int* addr, int oldval, int newval) {
    if (*addr == oldval)
        *addr = newval;
    return *addr;
}

Here’s inline assembly:
	asm volatile("lock cmpxchg %3, %0"
          : "m" (*addr), ":=a" (was)
          : "a" (oldval), "r" (newval)
          : "cc");

typedef qnode* lock;  // a lock is a pointer to a qnode
typedef struct qnode {
    struct qnode* next;
    bool someoneelse_locked;
} qnode;

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
        uint32_t was;
        if (cmpxchg_val(lockp, I, NULL) == I) {     // "C"
            asm volatile("lock cmpxchg %3, %0"
                      : "m" (*addr), ":=a" (was)
                      : "a" (oldval), "r" (newval)
                      : "cc");
            return;
        }
        return was;
    }
    // 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.
3. Some examples related to sequential consistency

a. What might p2 return if run concurrently with p1?

```c
int data = 0, ready = 0;

void p1 () {
    data = 2000;
    ready = 1;
}

void p2 () {
    while (!ready) {}
    return data;
}
```

[answer depends on the memory model given *by* the hardware *to* the software. if the model is sequential consistency, then the code does what you expect. but if not, then p2 can return 0.]

b. Can both "critical sections" run?

```c
int flag1 = 0, flag2 = 0;

void p1 (void *ignored) {
    flag1 = 1;
    if (!flag2) {
        critical_section_1 ();
    }
}

void p2 (void *ignored) {
    flag2 = 1;
    if (!flag1) {
        critical_section_2 ();
    }
}
```

[answer again depends on the memory model. if there's no sequential consistency, both "critical sections" can run.]
*/ 
* 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]
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.

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.