Handout for CS 372H

Class 12
25 February 2010

1. 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) {
    int was = *addr;
    if (*addr == oldval)
        *addr = newval;
    return was;
}
```

Here's inline assembly:

```assembly
asm volatile("lock cmpxchg %3, %0" : "m" (*addr), "=a" (was), "r" (newval), "cc")
```

2. 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;
```

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;
    // 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
    }
    // we hold the lock!
    return;
}
```

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 inline assembly:

```assembly
uint32_t cmpxchg_val(uint32_t* addr, uint32_t oldval, uint32_t newval) {
    uint32_t was;
    if (cmpxchg_val(lockp, I, NULL) == I) {     // "C"
        uint32_t was;
        if (*addr == oldval)
            *addr = newval;
        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" 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:
    // we hold the lock!
    return;
}
```

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.
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.

3. 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);

4. 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);
  while (navailable < nwanted) {
    wait(&cond_n, &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?
* 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]