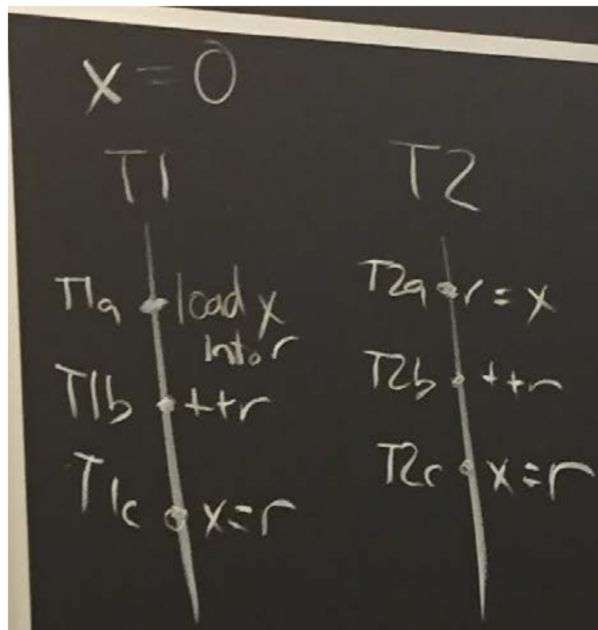
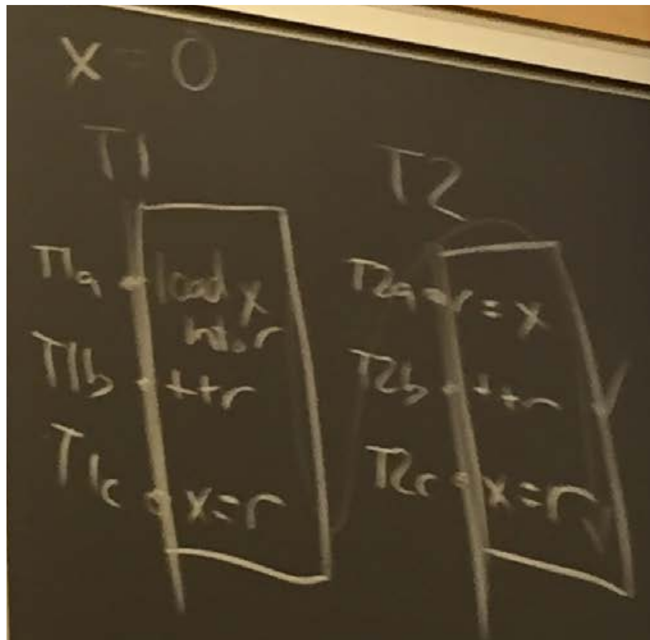
Computer Science 61
Scribe Notes
Tuesday, November 25, 2014 (aka the day before Thanksgiving Break)

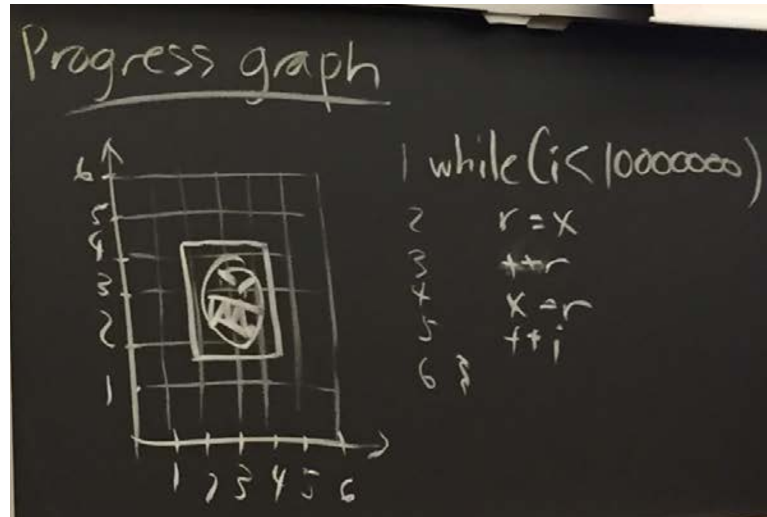
- Problem Set 6 Released!
  - People have fun with it
    - Make Games
      - Snake Game
    - Hack JavaScript
  - Due Wed., last day of reading period (not next wednesday)
    - Classwide extension for all problem sets and regrades up until reading period
- Mutual Exclusion
  - Service server: serves information about services
  - Tried version where every new connection created new process
    - Ran out of processes
  - Reduced overhead by using threads instead of processes
    - One thread per connection
    - Attacker could still run the system out of threads
    - Applied a limit to the number of threads that could be created
      - If too many connections, wait until one of the connections exits
      - Polling mechanism; bad utilization
        - "while (n_connection_threads > 100) {sched_yield()};
      - To improve, added mutual exclusion
  - **Mutual Exclusion**: At most thread's instruction pointer can be in one region at a time. That region is called the **critical region**.
  - Example in incr.c
    - Each thread adds 10 million to shared pointer stored on the stack
    - main() creates threads
      - pthread_join() is a lot like waitpid
        - Arguements that waitpid accepted don't apply to pthread_join()
    - threadfunc(void* arg)
      - *arg* passes a pointer to a variable that is on the main thread's stack
      - Is passing a pointer to the stack safe?
        - Not always
        - If you create a stack variable in a function that returns, it is possible for that variable to go out of scope which means you would be modifying memory you should no longer have access to
        - Not a problem in our program since main function lasts at least as long as threads last

- What do we expect the total to be?
  - Turns out it is often 40 million but is sometimes less
    - The four threads each add 10 x 10^6, which results in a sum anywhere between 10 x 10^6 and 40 x 10^6
  - Let's try it on a machine with a lot of cores
    - About 11 million
- What's going on?
  - One thread says "I'm going to take this value and add to it."
  - Another thread says the same thing, pulling the same initial value.
  - Second thread's storing of the value overwrites first thread's progress incrementing.
  - But there's one instruction, right?
    - This operation accesses memory, and operations that access memory are rarely actually **atomic** (indivisible)
    - When we write ++mem; it might be translated into assembly as a single instruction, but what the actual processor does is a three-step procedure
      - Loads %eax into a register (maybe a register that we can't name -- internal)
      - Increments register
      - Writes that register value back to memory
  - What is an order of operations that gives x=2?
    - 

    - Given the steps
      - x = 0
      - T1a => load x into r
      - T1b => ++r

                    - T1c => x = r
                    - T2a => r = x
                    - T2b => ++r
                    - T2c => x = r
                - Options that work:
                    - T1a, T1b, T1c, T2a, T2b, T2c
                    - T2a, T2b, T2c, T1a, T1b, T1c,
                - Turns out that the only orders that work are all of the 1 steps followed by the 2 steps or vice versa
                    - Other orders load 0 into r (as x has not been set to the updated value yet)
        - What we need here is mutual exclusion!
            - If T1 and T2 were critical regions, we would eliminate the problem
            - 

    - Textbook uses something called a **progress graph**

            - 

            - One thread's instruction pointer is one one axis; the other's is on the other axis
            - The critical region on a progress graph is a region that should never be entered.
            - We don't want both threads to have their pointers in 2, 3, or 4 at the same time.
            - Professor Kohler finds diagrams easier than progress graphs to logic about more than two threads at a time
        - How could we implement mutual exclusion without hardware?
            - Use a lock L
                - Locking L
                    - if L is not locked, then lock it and return
                    - otherwise, try again
                - Unlocking L
                    - mark L as unlocked
                - Represent unlocked as 0, locked as 1
                    - Locking L
                        - while (L==1) {do nothing}; set L=1; return;
                    - Unlocking L
                        - L=0; return;
                - The problem with this plan is that it just multiplies the issue!
                    - This lock itself suffers a synchronization issue!
            - Is this even something that's possible to do?
                - **Lamport's Bakery Algorithm**
                    - Makes the assumption that memory reads and writes happen in a well-defined order
                    - Not actually true due to caches
                        - Multicore systems only work if the same data can be cached in multiple caches at the same time

            - On the x86 system, at most one processor can write to the cacheline at a time
                - cacheline bounces between processors
                - slow process because it requires coordination between processors
                    - “Are you writing x?”
                    - “Yes, I am.”
                - This protocol isn’t necessarily engaged all of the time. A **store buffer** exists which allows a register to hold on to some stored values and reconcile them only when the store buffer is flushed.
                    - Only flushed when full or when told to.
                - In an ideal world where sequential instructions exist, it is not possible to get both r1 and r2 equal to 0
                - On an x86, you could get 0, 0
            - This is an issue of **memory models**
                - Fascinating and difficult subject essential to multiprocessors
- Can’t actually make a lock without hardware
    - Intel’s got our back
        - What memory operations need to be atomic to allow us to ensure locks work properly?
            - Checking L and setting L need to be atomic
                - “while (L == 1) {do nothing}; L=1;”
            - When a lock is in place, only one processor can write to or read from a particular cacheline
        - Intel has given us an assembly code prefix called “lock” that implements mutual exclusion
            - “lock addl” ensures that load and store happen in one atomic step
            - When we use this lock, we find that we are guaranteed a sum of 40 million, even on the multicore machine
                - Takes 10 times the amount of time because of all of the locking and unlocking
                - Of course, if we over optimize we get the correct result, 40 million, all of the time without any calculation taking place even with the incorrect code. Our compiler solves the problem and compiles to a

program that just returns 40 million without doing anything.

- How do we write this in c code?
  - gcc gives us certain macros that will compile down to atomic assembly code
    - "__sync_fetch_and_add()" for example
      - Note the double underscore at the beginning
    - see atomic_threadfunc() in lecture code
  - What does our new code look like?
    - Lock
      - while (__snyc_fetch_and_add(x, 1) > 1) {--1;}
    - Unlock
      - Also need to change this code to prevent L=0 from interleaving between if (result>1) and --L; in the lock thread
      - Instead, atomically decrement L in unlock function instead of setting it equal to 0
- Digression: Creating a random number using parallelism?
  - Cool idea!
  - Poor utilization
  - Attacker on same computer could control random number by taking up a lot of the cores
    - Only a small number of cores left to work on the cacheline; therefore, reduced randomness
- Could we do this example with code that uses one atomic instruction instead of four?
  - Yes, but we need a new atomic instruction
    - cmpxchg(&m, expected, desired)
      - "**Compare and Exchange**"
      - in a single step, changes *m* to *desired* only if it matches *expected*
  - New code for lock and unlock
    - Lock
      - while (cmpxchg(&L, 0, 1 != 0){/*spin*/}
    - Unlock
      - L=0;

                - Our new instruction ensures that lock is only 1 or 0 at a given time: it never enters the 2 intermediary
                    - As a result, we can write L=0; in Unlock instead of L--;
            - Compare and Exchange is the basis of any efficient locking and unlocking system
                - You could even use compare and exchange to derive the atomic increment/decrement instruction that we were using before

- Let me tell you a story: Debian and SSH key disaster
    - Debian security advisory: predictable random number generator
    - When you want to log into a secure location, you create an SSH key
    - Debian is a free vendor of Linux
    - All vendors make changes to Linux
        - Try to change branding
        - Try to fix bugs
            - Might run through valgrind to find places where programs are accessing uninitialized memory
            - Might comment out the part of a program that accesses uninitialized memory
            - Debian did that, not realizing that it was data that the kernel was requesting for the purpose of creating random data
    - As a result, if you created an SSH key on a Debian machine, you got one of 10,000 possible keys.
        - There were only 10,000 possible keys IN THE WORLD
    - Github had to crawl its entire database and throw out all of the Debian keys