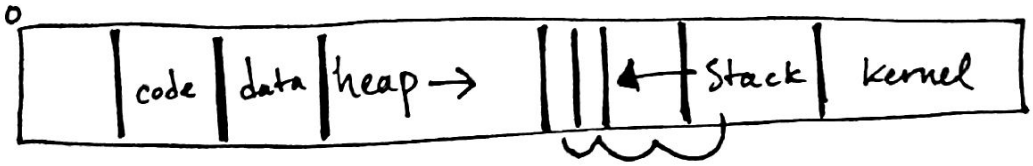
TODAY: Threads, Synchronization

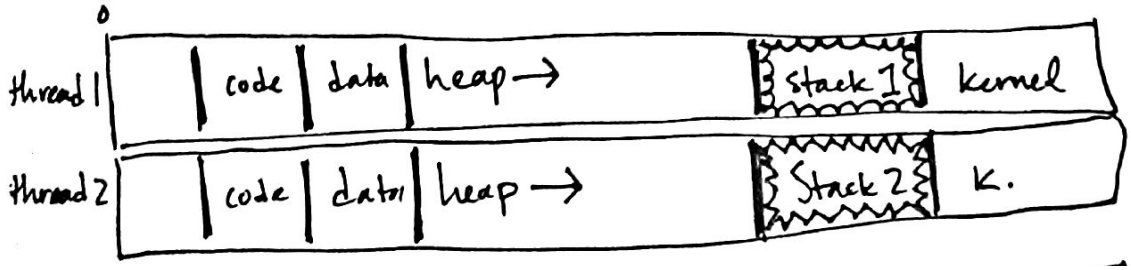
- Pset 5! AT LONG LAST! "Adversarial network pong"—handling dropped packets, server delays, overloads with connection attempts, etc. Exercises in networking, concurrency.

THREADS: (abstract CPUs)

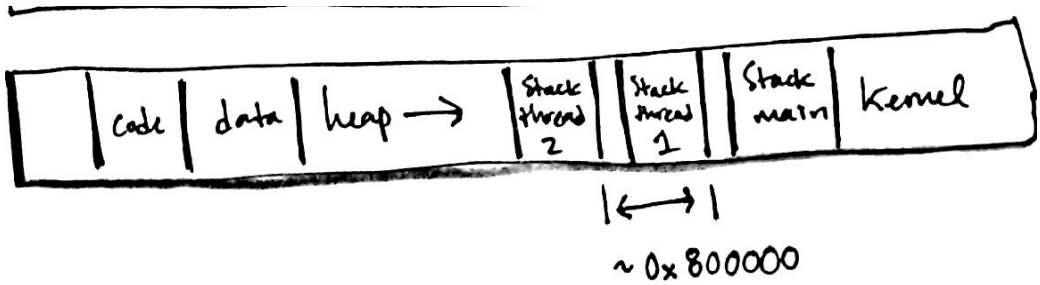
A normal picture of memory:



Threads with different virtual memory spaces (everything except stacks are shared):



Memory with threads' stack underneath each other:



- serviceserver07 (which uses threads) doesn't hold up to the blaster.
  THE ERROR: Cannot allocate memory. **THE PROBLEM**: **Every thread has its own stack of local variables with automatic storage duration** (lifetime of variable is lifetime of containing function). Stacks store activation records: when functions call other functions, all records remain on the same stack.

- Q: What would happen if threads shared a stack?
  All kinds of crazy errors. (We'll get to locks later today).

- Q: Where do we STORE all of these thread stacks? Virtual memory? Not quite.
  How can we find out for ourselves where all these stacks are? Let's try printing out all of the stack pointers. Trying this with serviceserver07 yields two

addresses (0xb6f0b35c and 0xb770c35c). Normal stack pointers are 0xbfff____, but these two are lower.

- Q: Running this several times, we get different addresses now and then. Why?
  Security! We don't want it to be easy to predict the stack address. There's a little bit of randomness involved: address space layout randomization.

- Q: The two stack addresses are ~0x800000 apart: why is there such a big difference?
  We need room for deeply recursive functions.

- Q: What if stack 1 hits stack 2? Stack overflow! Same for main stack hitting stack 1, or any of the stacks hitting the heap, etc.

- NB: The "out of memory" error does NOT mean that the machine has no more physical memory here. Rather, it means that the machine ran out of space for stacks in the address space. Because so much space is allocated for each stack, we can't fit that many stacks on a 32-bit machine.

- NB: Eddie refers to his laptop as "Mr. Brick," and his shoes as "nonexistent."

- Q: What's the advantage of this memory layout (with stacks underneath each other) vs. VM? Let's look at serviceserver07. The call to
  pthread_create(&t, NULL, connection_thread, (void *) f);
  passes f to connection_thread. Each thread is meant to handle a specific file. If threads had different virtual memory spaces, you wouldn't be allowed to pass addresses on the MAIN stack as arguments.

- Q: Where are all of the stack pointers stored? (A fabulous question!) In Linux, they're all stored in the kernel.

- Q: Again, why can't we use VM? Live code demo: create a struct with a file pointer and its descriptor (an integer), and alter functionality to accommodate this. This ONLY works when the threads are sharing their overall memory space.

- An unplanned segue into the awful, awful world of…

SYNCHRONIZATION: The *art* of writing correct multithreaded code.

- Q: What does correctness mean in this context? (No bugs, to be sure).
  Given two functions:

```
void t1(void *arg) {
      int *p = (int *) arg;
      (*p)++;
      return 0;
}
```

```
void t2(void *arg) {
      int *p = (int *) arg;
      (*p)--;
      return 0;
}
```

```
main(…){
      int x = 0;
      pthread_create(…, t1, &x);
      pthread_create(…, t2, &x);
      //wait for threads to exit
      printf("%d\n", x);
}
```

We would EXPECT each function to execute once. We should expect the print statement to yield zero.
THE ISSUE: This program has NO SEMANTICS=>NASAL DEMONS.
This program might yield any of the following: 0, 1, -1.
This is what we call a…

RACE CONDITION: A bug dependent on *scheduling order*.

- NB: Eddie's laptop is now "Mr. Laptop." This might be a step up.
  Mr. Laptop has 48 cores, allegedly.

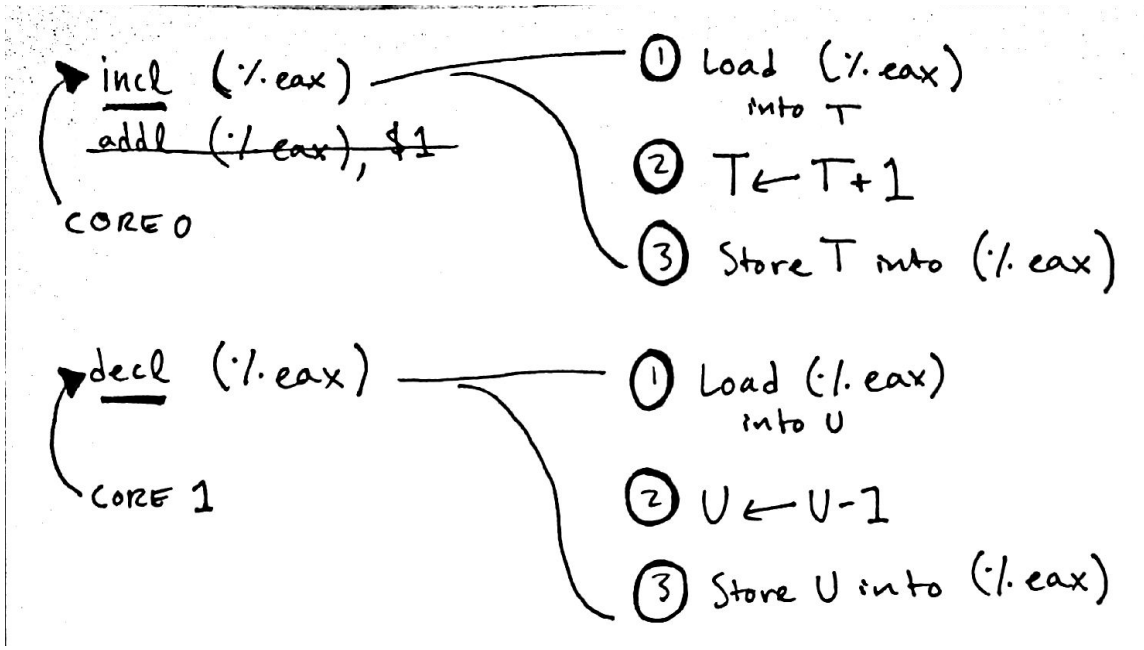
- serviceserver10 keeps a global unsigned thread_count that is incremented and then decremented with each call to connection_thread() (the decrement occurring after fclose() is called).

- Q: What if we increment and decrement unnecessary times? (Here, 1000 times).
  Logically, this *should* make no difference. (Here, we delegated tasks to another file to prevent the compiler from optimizing this away.)

- Running serviceblasters create 400 threads, so we expect thread_count to be 400. Let's use gdb to see how thread_count is holding up. It's MASSIVE. 3071486790, to be exact.

---------- BREAK! ----------

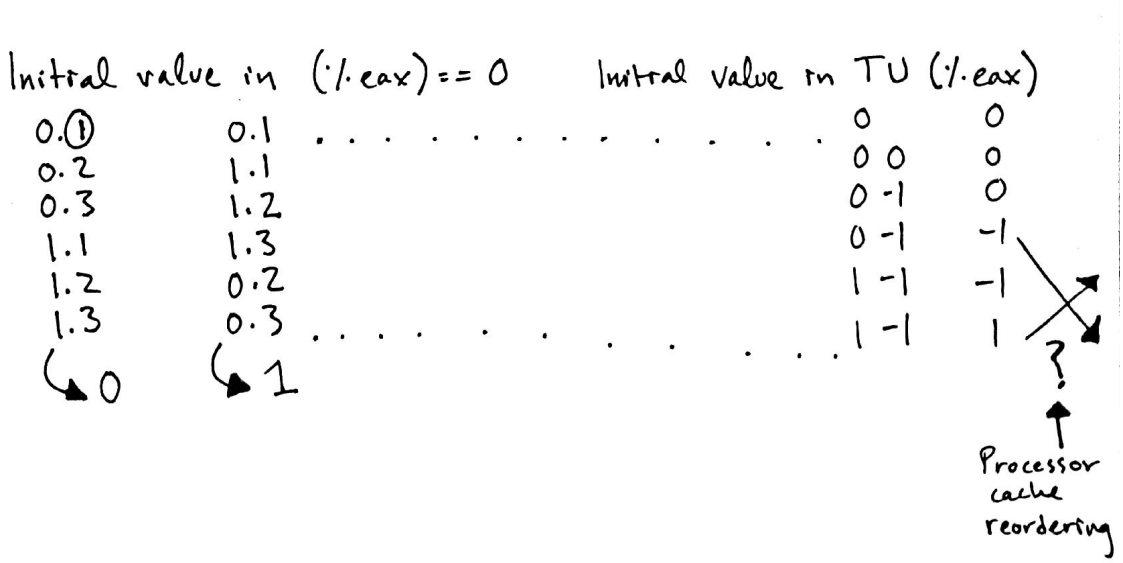
- Q: What's going on when we increment a variable in one thread and decrement it in another? Why is the result insane? How does the *processor* handle increments and decrements?



- At the machine level: incl (%eax) (address applied as an indirect argument).
  Equivalently, addl (%eax), $1. However, this instruction can't be done in a single *atomic* step; it's actually *three* fragments!

1. Load (%eax) into some temporary space (register T)
2. Increment T. T <- T + 1
3. Store T into (%eax)

- If you have TWO cores, one (core 0) implementing addl and the other (core 1) implementing decl…
  1. Load (%eax) into some temporary space (register U)
  2. Decrement U. U <- U - 1
  3. Store U into (%eax)



- Let's assume that the initial value in (%eax) == 0
  One interleaving of steps, (0.1, 0.2, 0.3, 1.1, 1.2, 1.3), will give us 0, as expected. Another, (0.1, 1.1, 1.2, 1.3, 0.2, 0.3), yields 1. It could also yield -1!
  Writing instructions in a line makes the assumption that the cores' caches don't interfere with each other. This assumption is false. Processor caches are not write-through caches; they're not immediately coherent. The writes to -1 and 1 might be reordered in the processor cache!

- When we access the same memory location in multiple cores, with not all accesses being reads, we open ourselves to a world of *pain*.

- **Problem**: Race conditions—simultaneous access to shared variables where at least one access is a write (all reads would be safe).

- **Solution**: **CRITICAL SECTION**: A set of instructions so that no race condition occurs UNLESS two or more cores are executing that instruction at the same time.

- Trivial example: compute all 0._ instructions before 1._ instructions? NOT ENOUGH!
  This protects 0 against interference from 1, but not vice versa. Critical section: ALL SIX INSTRUCTIONS!

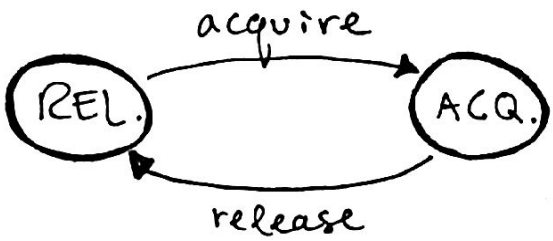
- Q: How can we ensure correct execution? How can we enforce a critical section?

LOCKS & MUTUAL EXCLUSION

Mutual exclusion (mutex): Prevents multiple threads or cores from executing a critical section simultaneously.

Lock: Object providing mutual exclusion.

- Given two threads, one incrementing and the other decrementing, how can we provide mutex? What if we had an object that we could acquire and release around an execution?
  - acquire: Blocks until the lock is released; IN ONE ATOMIC STEP set to acquired state.
  - release: Puts the lock in release state.



- A lock has two states. the acquire operation moves from RELEASE to ACQUIRE; the release operation moves from ACQUIRE to RELEASE.

- Q: Why does this work? Why does this enforce a critical section?
  Once the winning thread has acquired the lock, it can execute the critical section and release the lock. At that point, another thread is free to acquire the lock.

- Q: How are these states implemented? (Another fabulous question!)
  (typedef int mutex_t)
  0 == RELEASED,
  1 == ACQUIRED.

An (incorrect) attempt at acquire:

```
void acquire(mutex_t *k){
      int x;
      while (k != 0)
            /*do nothing */;
      *k = 1;
}
```

```
void release (mutex_t *k){
      *k = 0;
}
```

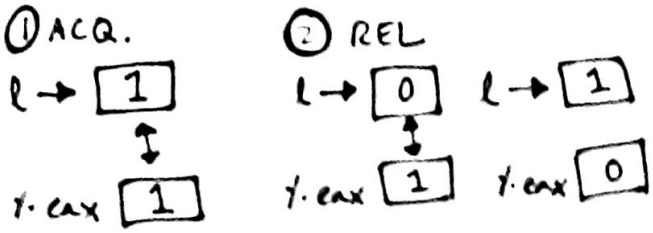
This has its own critical section!!!

- Q: What if we had a *swap* that could be performed in one *atomic* step?
  (ATOMIC: No intermediate state can be observed. Strictly one step.)

Correct Solution:

```
void acquire(mutex_t *k) {
      while (swap(k, 1))
      /*do nothing*/;
}
```

Code Explanation:
- In x86, "swap" is called lock xchgl
- int swap(int *x, int y) exchanges value in *x with y, returns previous *x.



1. In the ACQUIRED state, k -> 1
   A call to swap(1, 1) does nothing; the lock remains in the acquired state
2. In the RELEASED state, k -> 0
   A call to swap sets the lock to the acquired state. This atomic exchange is the key to mutual exclusion.

- Q: In a machine with 48 cores, what needs to happen to make sure lock xchgl is truly atomic? The machine has to go to every core and inquire as to whether it's changed the desired value recently (the MESI protocol: SLOW).

- Introduces us to an awful situation: the more correct the code is, the slower it gets!
  Mutual exclusion yields low utilization… solution next time.