Because of the duck-related excitement, we didn't have time in lecture to cover the slides about implementing `acquire()` and `release()` in a lot of detail. So here are some notes!

🦆🦆🦆🦆🦆

In lecture, we went through two big examples of applying locks: to bounded buffers and to filesystems. To really believe that all of this works, though, we should think about how one might actually implement `acquire()` and `release()`. Here's a simple strawman design:

First, we're going to treat a lock as a flag. The flag is true (1) when a piece of code is holding the lock, and false (0) otherwise. Our implementation of `release()`, then, is easy; we just set the flag to zero.

```
release(lock):
    lock = 0
```

What about `acquire()`? It's not safe to acquire the lock if another piece of code is holding it (`lock != 0`). If that's the case, we're going to have our function sit and do nothing. Once it's safe to acquire the lock, we'll go ahead and set the flag to 1.

```
acquire(lock):
    while lock != 0:
        do nothing
    lock = 1
```

But you all should know better than to think that our first attempt will work. Imagine this scenario:

- Two CPUs run `acquire(lock)`
- Both find that lock = 0 (e.g., one does the check, then the other CPU takes over and does the same check before anyone has set the flag to 1)
- Both set the flag to 1 and start executing the code after `acquire()`, since they both have the lock.

This is bad! This is a race condition. It's exactly what we don't want to happen. We need the check on the flag's value, as well as setting it equal to 1, to be an atomic action. In other words, we need locks to implement locks. This is a similar problem to virtual memory, where we needed a way to get the page table's physical address, or DNS, where we needed a way to get the IP address of the root server.

🦆🦆🦆🦆🦆

And we're going to solve this problem in the same way we did those: with a little bit of hardware support. Here's how the x86 architecture handles this. It uses at atomic exchange instruction, XCHG, that looks like this:

```
XCHG reg, addr
    temp <- mem[addr]
    mem[addr] <- reg
    reg <- temp
```

All ("all") that this function is doing is swapping the values stored in `reg` with the value stored in memory at address `addr`. The code that you see above is not inherently atomic (why would it be, it's three lines with seemingly no protection surrounding them). But there is a controller responsible for managing access to memory. We can partition the overall system memory across multiple controllers such that each memory location is the responsibility of one controller. To perform an atomic operation — such as XCHG — a CPU must obtain permission from the controller. The controller ensures that only one processor has permission at a time (processors request permission by sending messages to the controller). This means that there is a piece of hardware that prevents more than one CPU from executing this exchange operation at a time; it *cannot* be interrupted.

Now let's use XCHG to implement `acquire()`:

```
acquire(lock):
    do:
        r <= 1
        XCHG r, lock
    while r == 1
```

How does this work? If `lock` = 0, after XCHG, `r` will be 0, and the loop will terminate. Otherwise, another program is holding `lock`, and we need to keep trying. When we succeed, we will have atomically installed 1 into `lock`, so others will block.