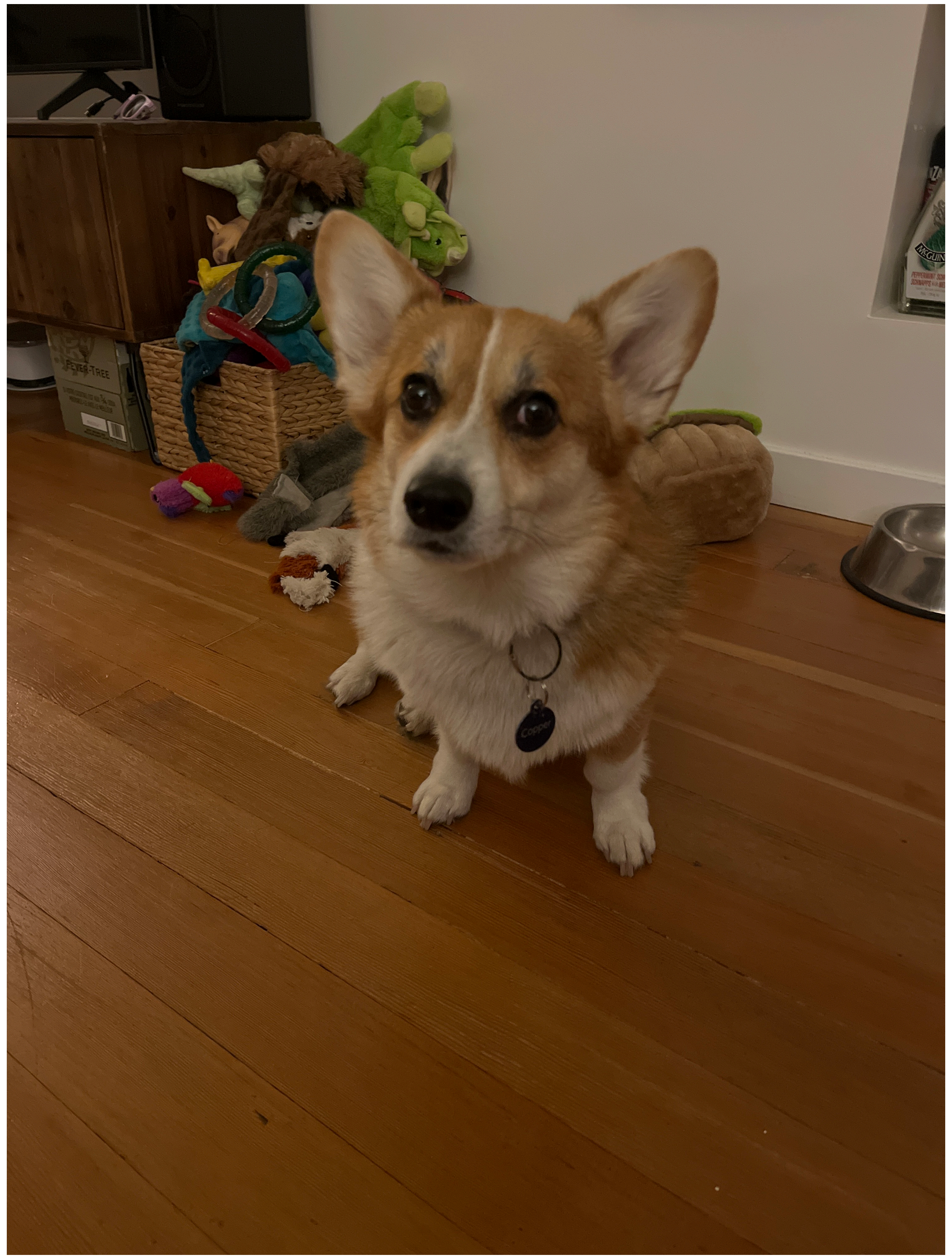
CMPT 383

Lecture 16: Concurrency



Anders Miltner

Concurrency

- Very difficult
- Causes complex bugs
- Bugs are often hard to diagnose
  - Not easily reproducible
- Concurrency is hard

Why is concurrency hard?

- Shared, mutable state
- Read/write conflicts
- Race conditions
- And more!

Shared State?

- Rust’s type system prevents it!
- Can’t share state at the same time, unless it’s immutable
  - And if it’s immutable, who cares!
- So everything should be quite easy?

Spawning threads!

```
fn concurrency() {
    use std::thread
    let handle = thread::spawn(|| {
        println!(“Spawned!”);
    });
    println!(“Main thread!”);
    let res = handle.join();
    println(“Thread done!”);
}
```

- First two print statements can occur in any order
- Last print statement must occur at the end

Quick Aside — Rust Lambdas

```
(|| {
        println!(“No Arg!”);
    })

(|x| {
       println!(“Arg {:?},x);
    })
```

Rust Threads?

- thread::spawn returns a JoinHandle
- JoinHandle.join() returns Result<T,E>
- Result<T,E> = Ok(T) | Err(E)
- Ok(T) if the thread finishes normally — .unwrap() will get T out
- Err(E) if the thread errors out
- Can leak threads if you don’t remember to join them
    - Not protected by the type system!

Parallelism Dangers?

```
fn my_fun() -> JoinHandle<()> {
    let v = vec![1,2,3];
    let handle = thread::spawn(|| {
        println!("{:?}",&v);
    });
    return handle
}
```

What if the thread runs after v is freed?

closure may outlive the current function, but it borrows `v`, which is owned by the current function

Ownership Challenges

- A closure describes the variables used in a lambda that refer to the outer scope
- The closure borrows, but compiler can’t prove the reference will always be valid
    - Would actually be always valid if it was a ‘static reference
- We must instead take ownership of values in the closure

New Syntax!

```
fn my_fun() -> JoinHandle<()> {
    let v = vec![1,2,3];
    let handle = thread::spawn(move || {
        println!(“{:?}”,&v);
    });
    return handle
}
```

Almost there

- Main issue: we *want* to have shared, mutable state
- Otherwise concurrency is much less powerful
  - Not as dynamic

Message Passing!

- Built-in queue for passing messages between threads
- std::sync::mpsc
  - Multi-producer, single consumer queue
- std::sync::spmc
  - Single-producer, multie consumer queue

Let’s focus on mpsc

- mpsc::channel()
  - Returns (Sender<T>,Receiver<T>)
  - Because mp, Sender can be cloned, creating multiple ways of sending
  - Because sc, Receiver cannot be cloned, only one way of receiving
- Whatever thread has Receiver gets messages from all threads with Sender

MPSC Example

```
use std::sync::mpsc;
let (sender, receiver) = mpsc::channel();
for i in 0..3 {
    let snd = sender.clone();
    thread::spawn(move || {
        snd.send(i * 10).unwrap();
    });
}
for i in 0..3 {
    println!("{:?}", receiver.recv());
}
```

Ok(0)
Ok(20)
Ok(10)

Other ways of sharing data

- Arc<Mutex<T>> another option
- Message passing is often good style
  - Not always