6.824: Debugging

Spring 2022, TA: Cel Skeggs (they/them)

Why is it hard to debug distributed systems?

- The algorithms are difficult to comprehend.
  - Details are easy to misunderstand or misinterpret, leaving your code with subtle bugs.
- Distributed systems are highly parallel, with many possible interleavings.
  - You have to consider all of the ways different threads and machines can interact.
- The activities in a distributed system are often not cleanly separated.
  - It can be hard to isolate erroneous behavior from the vast quantities of correct behavior occurring around it.

Objectives for lab correctness

- A distributed system used by a million users that functions correctly 99.9% of the time will still be broken for a thousand users.
  - Real world distributed systems need to be thoroughly debugged. Issues that appear small in testing will often be amplified into massive problems in production.
- **You will learn the most if you refine your code until all the bugs are gone.**
  - Many bugs only occur rarely. You will want to do both extensive testing and analysis.
- We only give you credit for a test case if you pass it every time.
  - We give you the full set of test cases. There are no secret or trick tests.
  - However: our testing machine has 80 cores. It tends to catch lots of obscure bugs!
  - Some test cases are designed to be easier than others. A slightly buggy implementation is still likely to pass many of the tests – just not all of them.

What are the labs like in 6.824?

- You are likely to spend more time debugging than writing code.
- But debugging is more about **quality** than **time spent**.
  - Some students report having made no progress after spending a dozen hours debugging, but this isn't what we want.
  - **If you aren't making progress, stop and try a new approach!**
- If you run out of ideas and get stuck, please come to office hours.
  - **We do not want you to suffer alone.** We will help you figure out the right next steps!

Demonstration: Fixing a bug in Lab 1

(We will return to the slides in a moment.)

Where do bugs come from?

- Mistakes and typos in the code you wrote. (Common.)
- Hardware glitches and compiler bugs. (Rare.)
- The worst bugs often come from **incorrect or incomplete mental models**.
  - Strange behaviors in systems can emerge from **unexpected interactions** of different components.
  - “A system is [something that has] the ability to fail in a way humans find surprising.” - https://twitter.com/SwiftOnSecurity/status/1484962240465932304
  - If you don’t quite realize something about how a system works, you may write incorrect code for it, and **you might not realize that the code is incorrect, even when you look at it.**

Example: Incomplete mental model

```
func (c *Coordinator) AssignTask(a *Args, r *Reply) error {
    if !c.AllMapTasksDone() {
        task, nextTimeout := c.PickMapTask()
        if task != nil {
            *r = Reply{ Mode: Map, Task: task }
        } else {
            *r = Reply{ Mode: Sleep, WakeAt: nextTimeout }
        }
    } else {
        task, nextTimeout := c.PickReduceTask()
        if task != nil {
            *r = Reply{ Mode: Reduce, Task: task }
        } else {
            *r = Reply{ Mode: Sleep, WakeAt: nextTimeout }
        }
    }
    return nil
}
```

Terminology

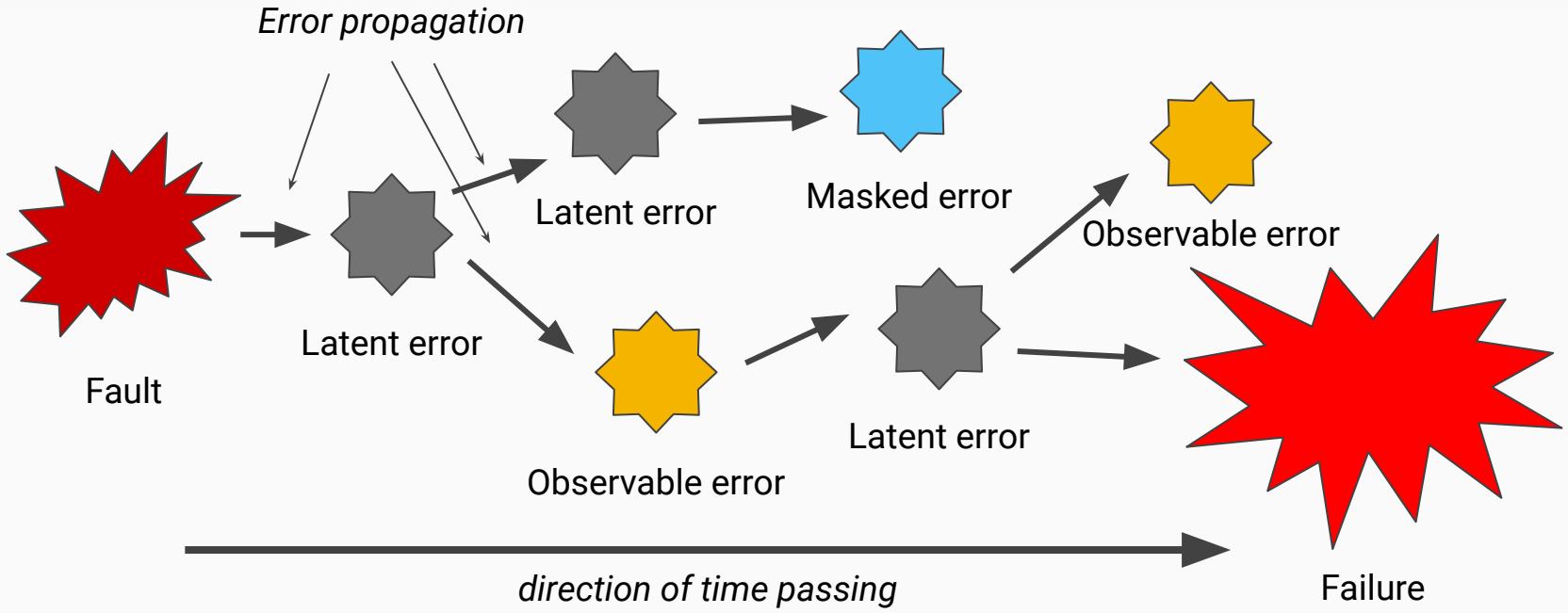
- **FAULT:** The **original underlying cause** of an error.
  - Such as a typo, incorrect logic, a wrong assumption, or a hardware glitch.
- **ERROR:** Any specific state in the system **that is not what it should be**.
  - Such as a variable with the wrong value or a function getting called when it shouldn't.
- **FAILURE:** The **final visible malfunction** of a system.
  - Such as a crash or a failure of a test case.

We debug because we've observed a **failure** and want to find the **fault**, so that we can fix it and prevent the failure from happening again. We can find the fault by tracing backwards through the series of **errors** that led to the failure.

Types of errors

- **LATENT ERROR:** an error where something is **invisibly wrong**.
  - These errors may silently propagate and lead to further errors.
- **MASKED ERROR:** an error that becomes **unimportant by happenstance**.
  - From the demo earlier: imagine if our reduce worker ignored any intermediate files it couldn't find, and it just so happened that the map tasks that weren't done executing didn't contain any key/value pairs.
  - It would still have been an error to assign the reduce task early, but it wouldn't matter!
- **OBSERVABLE ERROR:** an error that has been **surfaced** to you.
  - An observable error is apparent in the output of a program, possibly as an error message, an unexpected message, the lack of an expected message, or incorrect data in an intermediate file.

The error model



Example of the error model

In our timeout example, we had a series of things go wrong:

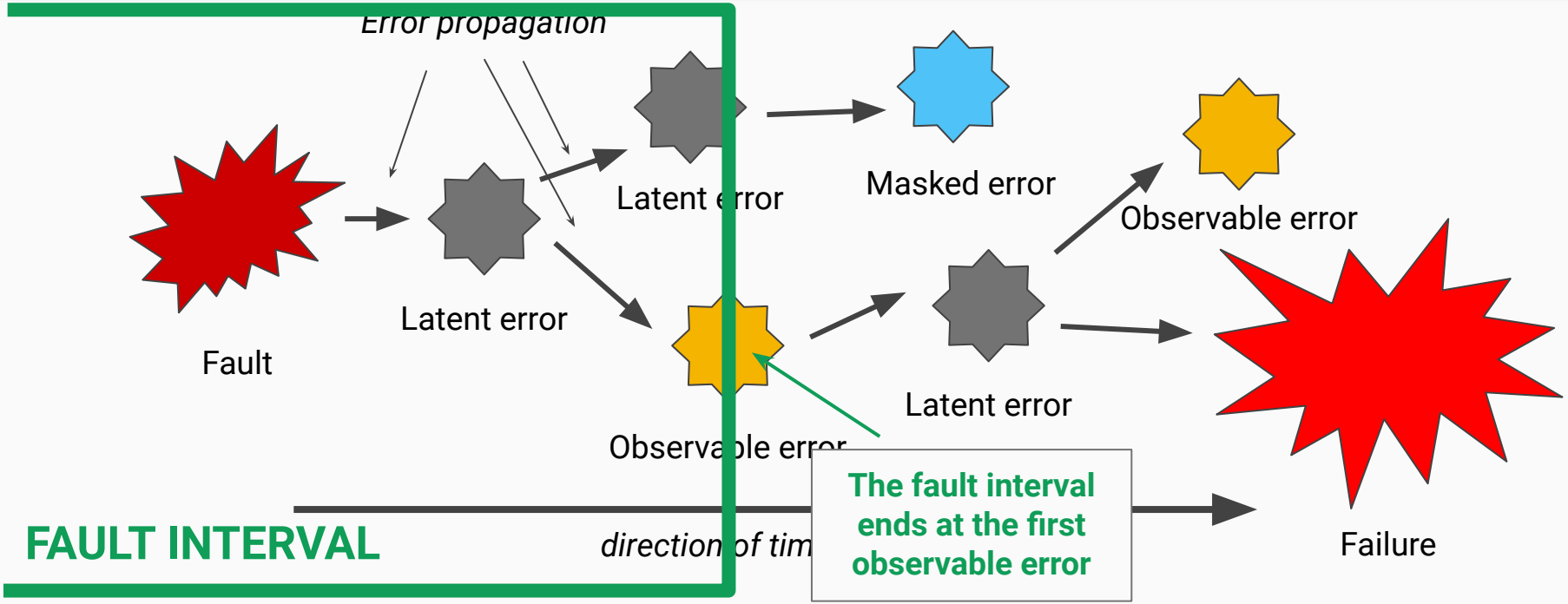
1. **The fault:** the coordinator code had a flaw in how it assigned its timeouts.
2. **Latent error:** the coordinator assigned too long of a timeout to reducer #2.
3. **Latent error:** reducer #2 received too long of a timeout.
4. **Latent error:** reducer #2 did not wake up promptly after map completion.
5. **Latent error:** reducer #2 did not request a task after map completion.
6. **Latent error:** reducer #2 did not execute a reduce task.
7. **Latent error:** reducer #2 did not create a “mr-worker-XXX” temporary file.
8. **Latent error:** reducer #1 did not observe a second “mr-worker-XXX” file.
9. **Observable error:** “mr-out-X” indicated that one reduce task ran at a time.
10. **Failure:** the test script reported that the test case had the wrong value.

We could zoom in further if we wanted! When one error leads to another, there is always a finer-grained error between the two. This applies recursively, all the way down to the voltages within the processor.

Methodical debugging: fault intervals

- A **fault interval** is an interval of time in which a fault *must* have occurred.
- Initial fault interval: [ran the test script, observed a failure)
- The fault *must* have occurred in this interval.
- To find it, we shrink the length of the interval until there's only a single line of code that could be at fault.
  - Pushing the start forward can only be an approximation: just because an intermediate state *appears* error-free, that doesn't mean it is!
  - Pulling the end backward is easier: it is always the moment of the earliest observed error.
- Every time we observe an error inside the fault interval, we can shrink it!

The error model



How do we narrow down the fault interval?

- We add **instrumentation** to turn **latent errors** inside the fault interval into **observable errors**.
- Instrumentation: the parts of your code that let you detect particular errors.
- Examples:
  - **Assertions** and **validations** that check a piece of state, ensure that it has the expected value, and panic (or at least print) if it's wrong.
  - **Print statements** that display values, so that they can be manually analyzed.
- Also, consider using your own **test cases** to validate certain slices of code.
  - These can help you identify shorter fault intervals, which will be easier to debug!
  - We provide the grading set, and you are welcome to copy them as a basis for your own!
  - You can augment our **opaque testing** with **clear-box testing**, by examining internal state.

Code examples of instrumentations

```
func LoadReduceKeyValues(taskId int, files []string) (kvs []KeyValue) {
    for _, filename := range files {
        log.Printf("[REDUCE %d] Intermediate filename: %s",
                   taskId, filename)
        contents, err := ioutil.ReadFile(filename)
        if err != nil {
            log.Fatalf("[REDUCE %d] Error reading %s: %v",
                       taskId, filename, err)
        }
        newKVs := ParseKeysValues(contents)
        if len(newKVS) == 0 {
            log.Printf("[REDUCE %d] Warning: file %s was empty!?",
                       taskId, filename)
        }
        kvs = append(kvs, newKVs...)
    }
    return kvs
}
```

Print statement

Assertion

Validation

Message: what does this instrumentation indicate?

Detail: what would be valuable to know in this situation?

Trigger: when does this instrumentation apply?

Context: which piece of code is running, and what task is it working on?

Code example for a test case (error handling omitted for clarity)

```
func TestLoadReduceKeyValues(t *testing.T) {
    temp := ioutil.TempDir("", "test_directory")
    defer os.RemoveAll(temp)
    file1, file2 := path.Join(temp, "mr-0-0"), path.Join(temp, "mr-7-0")
    kvs1 := []KeyValue {
        { Key: "A", Value: "X"   },
        { Key: "B", Value: "123" },
    }
    ioutil.WriteFile(file1, EncodeKeysValues(kvs1), 0644)
    kvs2 := []KeyValue {
        { Key: "C", Value: "Y"   },
    }
    ioutil.WriteFile(file2, EncodeKeysValues(kvs2), 0644)
    allKVs := LoadReduceKeyValues(0, []string{ file1, file2 })
    expected := append(kvs1, kvs2...)
    if !areKVsEqual(allKVs, expected) {
        t.Fatalf("Incorrect output: expected %v but got %v!",
    }
}
```

Preparation

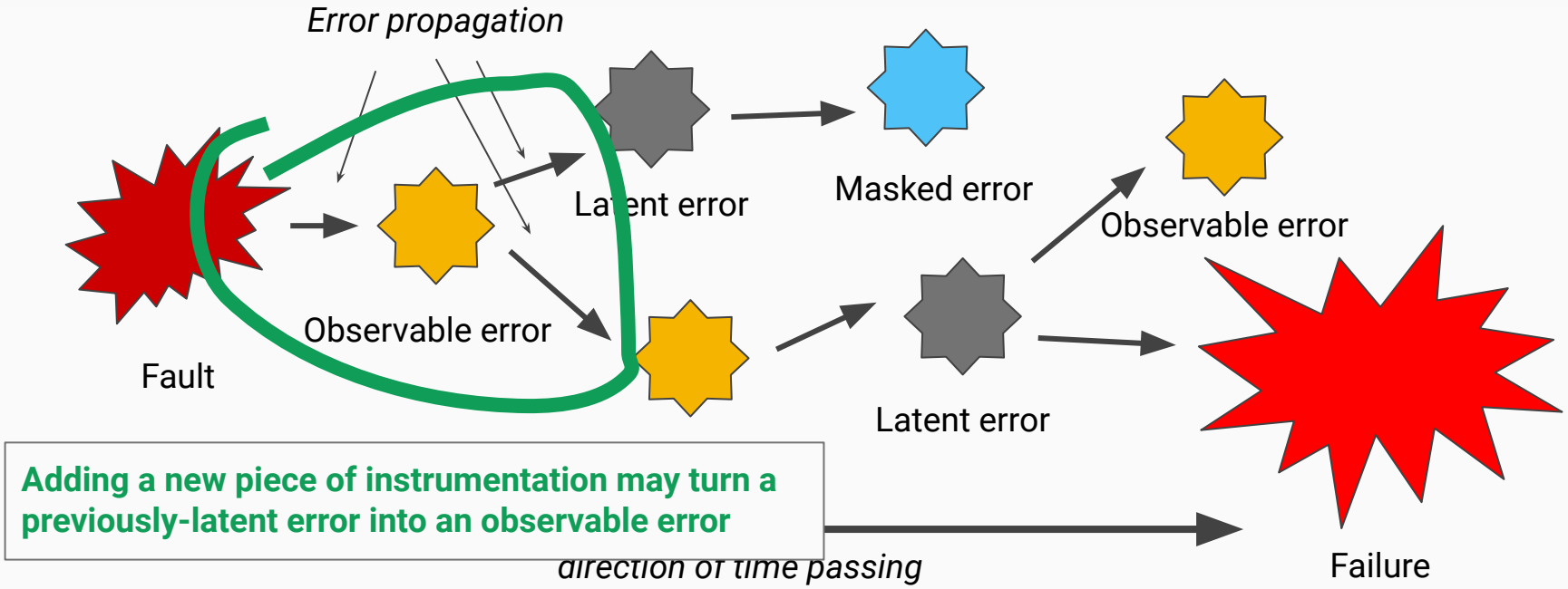
Cleanup

Test Inputs

Execution

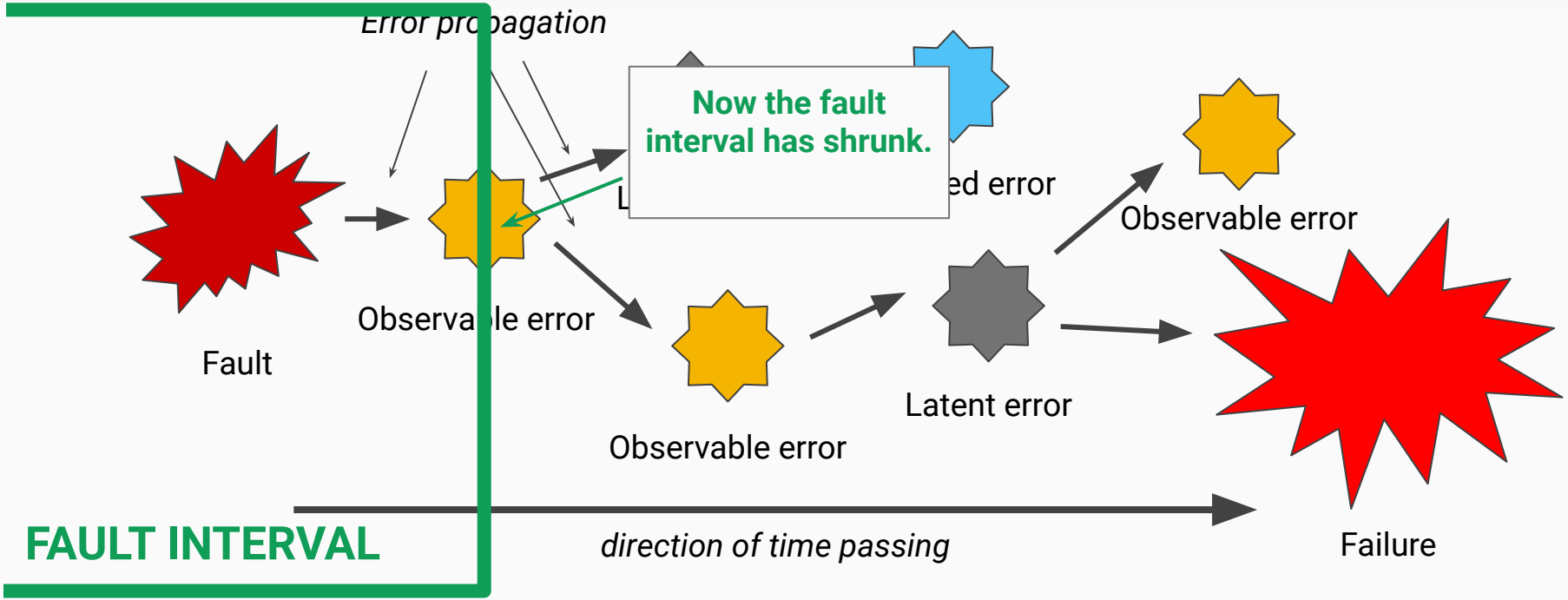
Verification

The error model



Adding a new piece of instrumentation may turn a previously-latent error into an observable error

The error model



The debugging “main loop”

1. Identify the earliest observable error (which is the end of the fault interval).
2. Formulate a **hypothesis** about the **most proximate cause** of the error.
3. Add instrumentation to test that hypothesis and run your code again.
4. **If false:** go back to step 2 and come up with a new hypothesis.
5. **If true:** you have a new earliest observable error. Continue from step 1!

Tips on following the main loop

- If you're struggling to make progress, start writing down notes!
  - Write down every time you find an earlier first observable error.
  - Write down your hypotheses, and their results.
- If your hypotheses aren't turning out to be true, make them more specific!
  - Eventually, you end up at the level of "why does this variable equal X instead of Y?"

Optimization: error intervals

- Situation: We have a variable that holds a value that it should not hold.
  - For example: a reduce task local variable holds the filename of a map task input.
- We can often diagnose a value error by **tracing the value backward**.
  - We can identify the start of the interval: the place where the input is specified.
  - We can identify the end of the interval: the place where the wrong output is produced.
- We can apply **binary search** to repeatedly bisect the flow of the variable.
  - Based on your mental model, identify a point about halfway between the start and end of the interval. Then, add instrumentation to determine what value it has at that point.
  - If it has the correct value, then the error starts in the second half – otherwise it starts in the first half!
- This quickly lets us discover where the incorrect value is introduced!

A note on multiple faults

- There can be more than one fault in a system.
  - This is especially common with new code that hasn't been debugged yet.
  - But it can also happen when working with existing code that had masked faults.
- It can be hard to tell if an error is the result of one fault or another.
  - This doesn't need to change our approach: we still keep debugging, identifying faults one at a time and fixing them, until our code stops failing.
- Remember: **if you fix a fault, and your code still fails, that doesn't necessarily mean you were wrong about the fault!** There might be multiple faults involved!
  - The question to ask if this happens is – does this fault still lead to an error? And if not, what other source might the error have?

Tips on building instrumentation

- You need to pick the right locations to add new instrumentation, and pick the right information for that instrumentation to report.
  - You can and should use your knowledge about your system to speed this up, but you must always validate your assumptions by actually running your code!
  - Remember: if you have a bug, there is likely *something* wrong with your mental model!
- Consider different approaches to solve different problems.
  - **Assertions and validations** are effective when it is easy to check if something is correct, because they only provide a signal when tripped, and don't obscure other information.
  - **Print statements** are applicable to many situations, but can easily become overwhelming.
  - **Test cases** can help you reduce the length of the program's execution.

Dealing with intermittent bugs

- In our labs, you are likely to encounter bugs that only appear rarely.
  - There are lots of ways for the executions of your code to be interleaved.
- You may have trouble tracking down bugs that only show up intermittently.
- Solution: **run your code *many* times**, until it fails at least once, while **printing out everything that could be relevant**.
- Afterwards, use tools to help you review different **slices** of the output.
  - Use tools like grep, or your own tools, to search the output to answer specific questions.
- You can make lots of progress debugging from just a single execution. You only have to rerun when you want to add new information to your output!
  - The more intermittent the bug, the more information you probably want to include!

Logging to a file

```
$ go test -race
```
This displays the output on your screen directly.

```
$ go test -race &> output.log
```
This puts all the output into “output.log” so that you can review it later.

```
$ go test -race |& tee output.log
```
This puts the output into the log AND displays it on your screen.

```
$ grep "an important string" output.log
```
This searches through your log for only lines containing “an important string.” This can help you avoid scrolling through irrelevant information.

Using format strings

- It can be complicated to read multi-line print messages!
  - ```
    fmt.Println(“SOMETHING HAPPENED”);
    fmt.Println(my_variable);
    fmt.Println(my_other_variable);
    ```
- Consider using format strings to print concise single-line messages
  - ```
    log.Printf(“[MODULE] Thing happened. MV=%v, MOV=%v\n”,
               my_variable, my_other_variable);
    ```

Questions to ask about print statements

- How much detail do I need?
  - More details makes instrumentation more broadly useful. Less helps avoid distraction!
- Can I make it easy to adjust the focus and detail level later?
  - You can define a constant for each module, and set it to true or false based on whether you want that module to print output. You can also filter out irrelevant detail with a tool!
- What format should my print statements take?
  - You may have to scroll through 1000s of lines of output. What format is best for *you*?
  - Consider text colors, columns, consistent formats, code words, and symbols!

Print statements may have timing effects

- Unfortunately, sometimes printing lots of information can actually mask or unmask bugs in your code!
  - This is because print statements can be slow, and so they can affect timing.
- Dealing with this requires some creativity.
  - You might write your logging messages to an array, and print them out in the background.
  - You might focus on using assertions and validations, which can execute faster in the common case.
  - You might try to inject or eliminate some other source of variability to compensate.
  - Or you might switch to trying to track down why inserting a delay on a certain line leads to success or failure!

Further Advice

Design for debugging

- You will spend lots of time debugging in 6.824, so you should consider *debuggability* as a primary goal when you write your code!
- Some ideas you might consider:
  - Using assertions *everywhere* in your code to keep errors from snowballing.
  - Building abstractions around debugging, such as by having all RPCs go through a single method, and always printing out the request and reply every time.
  - Printing out specially-formatted logs, so that you can filter them or put them into columns.
  - Minimizing the number of goroutines that you run, and the number of times you use locks + channels, so that there are fewer possible interleavings of your code.
- Be creative! Writing debuggable code can be difficult, so be prepared to revise your approach as the semester goes on!

Understand whose code is in scope

- **All code is in scope for debugging**.
- You may need to read, understand, and instrument **any piece of code**.
  - Yes, even if we provided it!
- If you're failing a test case and don't know why, read that test case!
- If you don't understand a piece of code we provided – **ask!**

Tips on avoiding pitfalls

- Once you understand the code you've written, continuing to reread it to try to find bugs may not be effective!
- Just because a piece of code looks like it's correct doesn't mean it is!
- Just because you wrote a piece of code recently doesn't mean it's the most likely place to find a fault!
- If you make changes to your code before you're confident that you understand what's wrong with it, you might make it worse!

Tweaking timeouts

- In the Raft labs, we ask you to decide on timeouts for certain behaviors.
  - Like elections and heartbeats.
- There are a wide range of timeouts that will let you pass the test cases.
  - You may need to try a few options!
- However: timeouts also impact the execution flow of a test case.
  - Changing timeouts can cause unrelated errors to become masked or unmasked!
- If you're spending a lot of time fiddling with timeouts, you're probably on the wrong track!
  - Oftentimes, this is simply obscuring or unobscuring the underlying fault.

Ask questions when you can't make progress!

- Our Piazza is a great place to ask questions when you are stuck on labs.
- When you ask, please remember to be **specific**. Let us know what you've **observed**, what you've **tried**, what you've **learned**, and **where you got stuck!**
  - (If you're posting an error message, send us a screenshot so we can see the exact output!)
- We will do our best to answer questions, even if they are vague, but often all we can do with a vague question is provide a vague answer!
  - Your code is unique, which means that the ways in which your code can fail are unique.
  - There is no canonical list of things that you could have done wrong for us to suggest!
- Remember that we run office hours every weekday. If you are so stuck that you don't know what to ask, you may find office hours more valuable!

Wrap-up

- Debugging is challenging, but **you can learn to do it well**.
- Follow a **methodical** process to debug.
- **Experiment** with new approaches to debugging!
- Most importantly: **if you're spending hours debugging and don't get anywhere, that means you should try a different approach!**

Questions?

Further Resources

- Blog post from a former TA: Debugging by Pretty Printing
  - If you don't already know how to effectively filter down and view very large quantities of output from a program, please read this!
- The lab guidance page: Lab guidance
  - There are many great tips here!
- Office hours!
  - Helping you learn to debug is part of our job, and we are happy to help!