Stuff

- Lab is due by 5pm today

- Exam 1 next Tues
  - I’ll be out of town so Zhe will give the exam

- New lab and HW assignments after the exam

SKIPPED Power Lecture

- Software perspective on power and energy management
- Mechanisms are provided by the HW people
  - Frequency scaling
  - Voltage scaling
  - Sleep modes
- Analysis of HW + workload can give us ballpark estimate of whether there is a good match
- Policies are up to software
  - But it’s often difficult to balance power, performance, and users’ convenience

Today

- Testing embedded software
  - Kinds of tests
  - When to test
  - How to test
  - Test coverage

- Fact: Most multithreaded Java programs run all of their test cases perfectly well when all locking is removed
    - What does this mean?

Testing

- Testing is the fundamental way that reliable embedded software is created
  - This is why we can build safety-critical applications using buggy compilers!
- However, good testing techniques are neither easy or intuitive
- Lots of basic questions:
  - When to test?
  - Who tests?
  - Where do test cases come from?
  - How to evaluate the result of a test?
  - How much testing is enough?

The Testing Mindset

- Creating good tests for your own software is hard
  - At least three reasons for this
- Microsoft (and other companies) separate testers from developers
  - Different skill sets
- Good testers are adversarial
  - Goal is to break the software
  - This can lead to strained relations between developers and testers
- The best developers truly attempt to break their own code

Kinds of Tests

- **Functionality – testing functional behavior**
- **Interfaces – testing interaction with other systems**
- **Security – test for robustness to intrusion**
- **Standards – check for compliance**
- **Regression**
  - **Testing whether everything works after the system has been changed**
  - **Test cases derived from prior failures**
- **Resources – measuring required resources such as CPU time, memory, network bandwidth**

- **Load and stress – trying to overload the system**

Test Levels

- Hardware unit test
- Hardware integration test
- Software unit test
- Software integration test
- HW/SW integration test
- System test
- Acceptance test
- Field test

Where do tests come from?

- Use cases
- Developer intuition
- Previous failures
- Boundary cases from specification
- Stress tests
- Random inputs
- Directed random / analysis-driven inputs

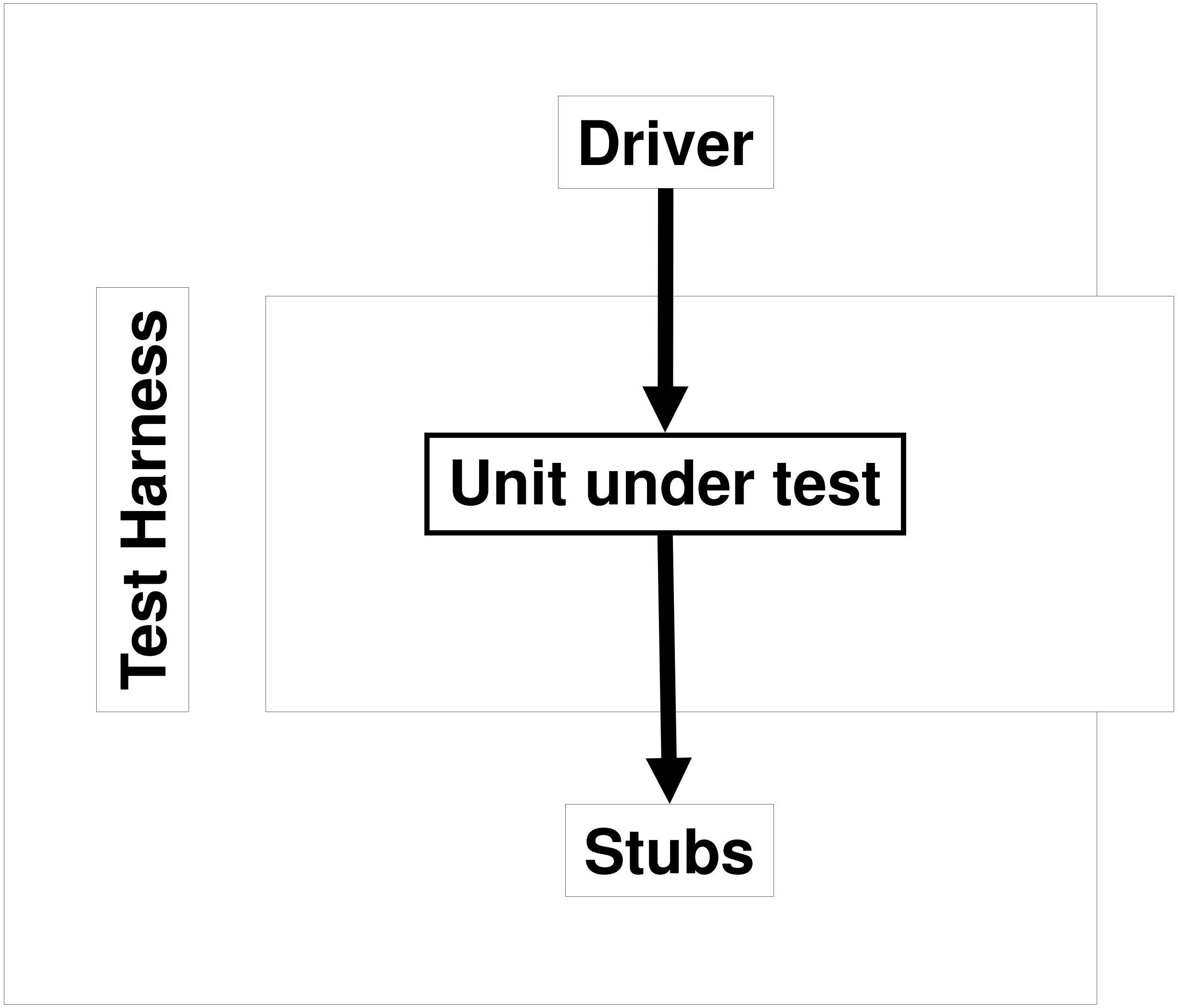
When to Test

- Every combination of kind of test and test level should be run as early as is feasible
- Basic fact: Cost to fix a bug increases greatly as development progresses

Testing by Developers

- Why?
  - Defects cheaper to fix when found earlier
  - High-quality parts make integration possible
  - Defects found late are hard to map back to the source code
  - Some kinds of tests are only possible at the unit level
  - Developers understand the implementation, which can lead to better test cases
- Quality cannot be added at the end of development
  - Has to be there from the start

Unit Testing



Integration Testing Strategies

- Bottom-up
  - Start with low-level modules with few dependencies
  - Exercise them using drivers
- Top-down
  - Overall control structure drives tests
  - Stubs provided for nonexistant modules
  - “Look and feel” of the system established early
- Big-bang
  - Only works for small systems
  - Useful for tightly coupled systems where top-down and bottom-up are difficult

Design for Test

- Term most often used in context of hardware
  - Also applies to software
- How to do this?
  - Lots of assertions for preconditions and postconditions
  - Implement self-tests
  - Provide test scaffolding along with code
  - Expose all interfaces for testing
- Examples – how would you design these for test?
  - Code to set PLL
  - Code responding to an external interrupt source

Test Oracles

- Test oracle – Code that tells us if the system is responding properly to tests
- Some oracles are easy
  - Not working if the software crashes
  - Not working if the software stops responding normally to inputs
  - Not working if an assertion is violated
- Some oracles are very difficult
  - E.g. is the aircraft responding properly to crosswind?
  - Manual interpretation of the specification and test results typically required

Test Coverage

- Coverage metrics try to answer the question: How can we know when to stop testing?
- Example metrics:
  - Function coverage – are all functions executed?
  - Statement coverage – are all statements executed?
  - Branch coverage – is every possible decision executed at every branch?
  - Path coverage – is every path through the code executed?
  - Value coverage – is the full range of every variable covered?
  - Mutation coverage – are all variants of the program covered?
  - Exception coverage – are all exceptions signaled?
- In most cases goal is 100% coverage

Evaluating Coverage Metrics

- Coverage metric must be understood by the user
- Near-complete coverage must be achievable
  - Exceptions require fixing or manual review
- Some action should be taken upon reaching 100% coverage

Coverage of Concurrent SW

- Problem:
  - Traditional test coverage metrics are in terms of sequential software
  - Embedded software is concurrent
- What are some plausible metrics for concurrent software?
  - Interrupt nesting coverage
  - Interrupt preemption coverage
  - Thread preemption coverage
  - Synchronization coverage
    - Each lock “does interesting things”

Stress Testing

- Test system at the limits of (and outside) its load parameters
  - Intuition: This exposes different kinds of problems than regular test cases do
- Examples – how would you stress test:
  - Embedded web server
  - An RTOS
  - A cell phone
- Tricky problem: Thinking of as many sources of stress as possible

Stress Testing for Interrupts

- What bugs are we trying to find?
- How to do it?
  - What if data comes along with the interrupt?
- How to tell when we’re done?

Summary

- Embedded software is only as good as its test cases
  - You should assume any conditions not tested will fail
  - … because they will
- Developers perform early testing of components
  - Requires adversarial mindset
  - Requires wishful thinking to be ruthlessly suppressed
- Integration cannot possibly succeed without reliable components
- Summary:
  - Test early
  - Test often
  - Test creatively