MIT 6.005
Debugging
Concepts

• Validation
  – Purpose is to uncover problems and increase confidence
  – Combination of reasoning and testing

• Debugging
  – Finding out why a program is not functioning as intended

• Defensive programming
  – Programming with validation and debugging in mind

• Testing ≠ debugging
  – test: reveals existence of problem
  – debug: pinpoint location+cause of problem
Defense in Depth

• Don’t write code with bugs
• During development, make anything unexpected immediately visible to minimize distance between bug and symptom
• Last resort is debugging
  – Needed when effect of bug is distant from cause
  – Design experiments to gain information about bug
    • Fairly easy, in a program with good modularity, representation hiding, specs, unit tests etc.
    • Much harder with a poor design
Correctness

• Think before you code, especially when debugging is going to be hard
  – Concurrency
  – Difficult instrumentation environments
  – Program must meet timing deadlines

• Modularity
  – Divide program into chunks that are easy to understand
  – Use abstract data types with well-defined interfaces
  – Use defensive programming; avoid rep exposure

• Specification
  – Write specs for all modules, so that an explicit, well-defined contract exists between each module and its clients
Basic Concepts In Debugging

- Fault – bug in the program
- Activation – when code with fault executes
- Error – damage to program caused by activation of fault
- Failure – error propagates to the point where it causes the program to observably violate its specification
- Debugging starts when failure is observed
  - Unit testing
  - Integration testing
  - In the field
Kinds of Bugs

• Quick, easy bugs (few minutes)
• Medium bugs (hours)
• Hard bugs (small number of days)
• Really Bad bugs (many days to never)
• Look for bugs in this order!
• Different debugging strategies for each
Debugging Hints

• Hope for a quick bug, take a first quick shot
  – Look at backtrace
  – Look at code where you think there might be a problem, put a few print statements in
  – Try to get lucky
• Make the first shot quick! Don’t get sucked in!
• Recompile/reboot everything before take next shot
• Look for medium bug with next shot
  – Use print statements
  – Design an organized print strategy
  – Legible, easy to read error messages
• Make the medium shot medium! Don’t get sucked in!
Bad Print Strategy

class Point {
    int x,y;
    Point(int a, int b) { x = a; y = b; System.out.print(a); }
    move(int a, int b) {
        x += a; y += b;
        System.out.print(a); System.out.print(x);
    }
    move(int a) {
        x += a;
        System.out.print(x); System.out.print(a);
        System.out.print(y);
    }
}
}
Better Print Strategy (How Good Depends on Context)

class Point {
    int x,y;
    void print(String b, String e) {
        String s = b + "<" + x + "," + "y" + ">"+ e;
        System.out.print(s);
    }
    Point(int a, int b) {
        x = a; y = b; this.print("new Point: ", ",\n");
    }
    void move(int a, int b) {
        this.print(">> Point move(" + a + "," + b + ") ", ",\n");
        x += a; y += b;
        this.print("<< (" + a + "," + b + ") ", ",\n");
    }
    void move(int a) {
        this.print("-> Point move(" + a + ") ", ",\n");
        x += a;
        this.print("<- (" + a + ") ", ",\n");
    }
}

class Point {
    int x, y; static boolean debug = false;
    void print(String b, String e) {
        String s = b + "<" + x + "," + "y" + ">" + e;
        System.out.print(s);
    }
    Point(int a, int b) {
        x = a; y = b; if (debug) this.print("new Point: ", "\n");
    }
    void move(int a, int b) {
        if (debug) this.print(">> Point move(" + a + "," + b + ") ", "\n");
        x += a; y += b;
        if (debug) this.print("<< (" + a + "," + b + ") ", "\n");
    }
    void move(int a) {
        if (debug) this.print("-> Point move(" + a + ") ", "\n");
        x += a;
        if (debug) this.print("<- (" + a + ") ", "\n");
    }
}
So You Have A Hard Bug

• Rebuild system from scratch and reboot
• Explain bug to a friend
  (engage different part of brain)
• Make sure it is a bug – program may be working correctly and you don’t realize it!
• Minimize input required to exercise bug
• Add checks to program
  – Minimize distance between error and detection
  – Use binary search to narrow down possible locations
• Use logs to record events in history
Reducing Input Size Example

boolean substr(String s, String b)
    returns false for
    s = “The wworld is ggreat! Liffe is wwonderful! I am so vvery happy all of the ttime!”
    b = “very happy”
even though “very happy” is a substring of s
Wrong approach: try to trace the execution of substr for this case
Right approach: try to reduce the size of the test case
Reducing Input Size

substr("I am so vvery happy all of the ttime!", "very happy") == false
substr("very happy all of the ttime!", "very happy") == true
substr("I am so vvery happy", "very happy") == false
substr("I am so vvery happy", "happy") == true
substr("I am so vvery happy", "very") == false
substr("I am so vvery happy", "ve") == false
substr("vvery happy", "ve") == false
substr("vvery happy", "v") == true
substr("vvery", "ve") == false
substr("vve", "ve") == false
substr("ve", "ve") == true
boolean substr(String s, String b) {
    int i=0, j=0;
    for (i = 0; i < s.length; i++) {
        if (b[j] == s[i]) j++; else j = 0;
        if (j == b.length) return true;
    }
    return false;
}
Inserting Checks

• Insert checks galore with an intelligent checking strategy
  – Precondition checks
  – Consistency checks
  – Bug-specific checks

• Goal: stop the program as close to bug as possible
  – If check fails, infinite loop
  – Use debugger to see where you are, explore program a bit
Checking For Preconditions

static int find(a, k) {
   //: requires exists i <= a.length. a[i] == k
   int i = 0;
   while (i < a.length) {
      if (a[i] == k) break;
      i++;
   }
   assert(i < a.length, "Precondition Violated");
   return i;
}

Precondition violated? Get an assertion!
Downside of Assertions

static int sum(Integer a[], List<Integer> l) {
    int s = 0;
    for (e:l) {
        assert(e < a.length, "Precondition Violated");
        s = s + a[e];
    }
    return s;
}

Assertion not checked until use data
Bug occurs when bad index inserted into list
May be a long distance between fault activation and error detection
Data Structure Consistency Checks

static void check(Integer a[], List<Integer> l) {
    for (e:l) {
        assert(e < a.length, "Inconsistent Data Structure");
    }
}

Perform check after all updates to minimize distance between bug occurrence and bug detection

Can also write a single procedure to check ALL data structures, then scatter calls to this procedure throughout code
Bug-Specific Checks

```java
static void check(Integer a[], List<Integer> l) {
    for (e:l) {
        assert(e != 1234, "Inconsistent Data Structure");
    }
}

Bug shows up as 1234 in list
Check for that specific condition
```
Using Checks to Find Bugs

public static void main(String args[]) {
    a(); check(); b(); check(); c(); check(); d();
}

public static void b() {
    foo(); check(); bar(); check(); baz();
}
Checks In Production Code

• Should you include assertions and checks in production code? Arguments:
  – Yes: stop program if check fails - don’t want to take chance program will do something wrong
  – No: may need program to keep going, maybe bug does not have such bad consequences
  – Correct answer depends on context!
Ariane 5 Failure

• Primary computer runs computation, integer overflow exception
• Shut down primary, switch to backup
• Same code, same bug, same overflow
• Shut down backup
• Rocket flies without control…
• Kicker – value with overflow not used!
Dealing With Errors In General

• Potential alternatives
  – Stop computer (but maybe you need functionality)
  – Keep going (but maybe computer does something wrong and bad)
  – Throw exception, let caller handle it (but probably caller has no idea what to do)
Logging Events

• Often you would like to have some indication of past when a check fails

• Design a logging infrastructure
  – Dump events to a file (strings)
  – Events have consistent format to enable efficient searches
  – Sometimes (usually for timing reasons) must keep lot in memory, not on disk
  – Circular logs to avoid resource exhaustion
Searching for bugs

• Take advantage of modularity
  – Start with everything, take away pieces until bug goes away
  – Start with nothing, add pieces back in until bug appears

• Take advantage of modular reasoning
  – Trace through program, viewing intermediate results

• Can use binary search to speed things up
  – Bug happens somewhere between first and last statement
  – So can do binary search on that ordered set of statements
  – Works great with data structure corruption bugs
Make Bug Repeatable

• Sequential, deterministic program – bug is repeatable

• Distributed program
  – Bug may be difficult to repeat because of timing in message delivery
  – Build a message log/replay infrastructure where you control timing of replay

• Shared-memory parallel program
  – No really good mechanism
  – Try to repeat environment in which you saw bug
  – Try schedulers that do strange task orders
  – Use event logs to capture key events
Debugging In Harsh Environments

• Harsh environments
  – Bug is nondeterministic, difficult to reproduce
  – Can’t print or use debugger
  – Can’t change timing of program
    (or bug has to do with timing)

• Build an event log (circular buffer)
• Log events during execution of program as it runs at speed
• When detect error, stop program and examine logs
Regression Testing

• Whenever you find and fix a bug
  – Add a test for it
  – Re-run all your tests
• Why this is a good idea
  – Often reintroduce old bugs while fixing new ones
  – Helps to populate test suite with good tests
  – If a bug happened once, it could well happen again
  – If a bug shows up again, probably in code you just modified
• Run regression tests as frequently as you can afford to (at least every night), automate process
My Nastiest Bugs

• Inadvertent side effect on data structure stored in a backtracking stack – caused search to miss a case
• Potential buffer overflow (unchecked) in message passing system on hypercube
• Race condition on two returns from suspend – resume returns before suspend, stack collision
• Some wild pointer errors in C
when the going gets really tough

• Debug for a day or two
• Fill your head with information
• Then
  – Trade latency for efficiency (or get both)
  – And walk away for a while
when the going gets really tough

• Walk away
  – Trade latency for efficiency – sleep, go surfing, go hiking!
  – One good reason to start early
• Infrastructure problems
  – Look at generated code for compiler bug/turn off optimizations
  – Port to a new system, see if bug still there
• Rewrite part of system that appears to be at fault with simplicity in mind, new design approach
• Start building serious debugging infrastructure for that bug
• Live with the bug
  – automatically detect and recover from bug
  – automatically reboot frequently
• Bail on the project…