Debugging

• finding problems
  – the scientific method
  – verification, assessment, triage
  – confident problem identification
  – debugging tools
• fixing problems
  – root cause analysis
• psychological issues in debugging
  – how to become a better debugger
• bug reports and tracking

What makes debugging hard?

• programs are not as simple as theorems
  – much larger, nowhere nearly as well designed
• complex causes
  – faults may have very complex preconditions
    • interactions between components & people
    • relative timing of complex events
  – some causal elements may seem unrelated
• complex cause/symptom relationships
  – symptoms result from a cascade of events
  – symptoms may occur long after cause
  – symptoms may appear in unrelated areas

the scientific debugging process

1. verify the bug
2. find the bug
3. fix the bug

when it doesn’t make sense ...

• get more information
  – identify more failure (and non-failure) cases
  – use tracers or debuggers
  – instrumentation to catch errors sooner
  – try everything on McConnell’s brute force list
• write down all the facts you know
  – spread them out on the wall or a white board
• ask for help
  – more eyes and brains in the analysis
  – suggestions for additional instrumentation

information - program output

• programs often produce progress output
  – files processed, actions, events received
  – may have to be enabled w/verbose options
• many programs have diagnostic options
  – enable more detailed activity traces
  – request dumps of internal tables
• this can help us understand the problem
  – what is happening when the problem occurs
  – interesting state associated with the event

debugger stack traces
(information - back-traces)
• history of all active sub-routine calls
  – an interpreted trace back through stack
  • list of all currently stacked subroutine calls
  • often includes values of parameters
  • may include values of all automatic variables
• tells us, in general, what program is doing
  – e.g. invalid address, zero-divide
• can also be obtained from a debugger

Sample system call trace (strace) output
open("/opt/gnome2/lib/libc.so.6", O_RDONLY) = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY) = 3
old_mmap(NULL, 65928, PROT_READ, MAP_PRIVATE, 3, 0) = 0xbf596000
close(3)
open("/lib/tls/libc.so.6", O_RDONLY) = 3
read(3, "\177ELF\1\2\1\3\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0`ht\000", 512) = 512

system call tracing

information - run in debugger
• debuggers have many valuable features
  – source level (vs machine language)
  – code break points (stop when it gets here)
  – single-stepping (line or instruction at a time)
  – data watch points (stop when it touches this)
  – view and change variables at any time
• invaluable for brute force debugging
  – when you aren’t sure what you’re looking for
  – run to an interesting point, stop, look around

information – instrumentation
• there isn’t always enough information
  – program produces little diagnostic output
  – stack/execution traces are useless
  • detected failure occurs long after the actual error
  – execution in debugger is not practical
  • don’t know what code to look at
  • suspect code is called very frequently
• add new instrumentation to program
  – log all conceivably interesting events
  – attempt to reproduce failures w/new version
  – may change symptoms of the failure

information – forensic analysis
• some problems can be hard to recreate
  – distributed apps often have timing problems
  – pointer errors can have indeterminate results
• some problems have leave little evidence
  – failure may occur long after the original error
  – most of the evidence is long since gone
• we must infer cause from remaining clues
  – often requires CSI-like detective-work
  – such systems often include diagnostic logs
  – the ultimate fall-back: complete core dump
the scientific debugging process

1. verify the bug
2. find the bug
3. fix the bug

reviewing fixes
- original code clearly needed more review
  - original developer did not get it right
  - original reviewers did not find the error
  - original test cases did not find the error
- bug fixes almost always require review
  - correctness of code is clearly not obvious
  - bug fixers are often not original developer
  - they often have less familiarity with the code
  - they often have less programming experience
- how did this get past us before?

Root Cause Analysis
- some bugs may be essentially random
  - software is complex, people are fallible
- many bugs turn out not to be random
  - people keep repeating the same mistakes
  - inadequate training, tools & methodology
- after a problem has been found and fixed
  - identify the root cause of the defect
    - understand how we made and failed to find it
  - do statistical studies of root causes
    - identify clusters, find ways to eliminate them

Psychological Issues
- We assume what we think we know
  - we do not see our work as it actually is
  - rather, we see it as we intended it to be
- We assume we are better than we are
  - we believe in our abilities and methodology
  - we don’t like to believe ourselves error-prone
  - we suspect problems come from elsewhere
- These blind us to many hypotheses
  - this blindness impairs our debugging ability

Inconceivability
- It is possible that the bug isn’t your fault
  - operating systems are not perfect
    - they do contain bugs, especially new releases
    - compilers sometimes generate incorrect code
      - I’ve personally tracked down a few of these
    - computers can even mis-execute instructions
      - in 35 years, I have encountered one instance
  - these may be conclusions you come to
    - they should never be assumptions you make

Becoming a Better Debugger
- Keep an open mind
  - there are none so blind as those who will not see
- Learn from your programming mistakes
  - what did you do wrong?
  - why didn’t you notice it sooner?
- Learn from your debugging mistakes
  - what clues were there, but you missed them?
  - what dead-ends did you wind up following?
- Learn from others
  - questions they ask, details they notice
  - tools and techniques do they use
### Key Attributes of a Bug Report

- **ID** (assigned automatically)
- **Title** (one line description of bug)
- **Status**
  - current state, severity, priority, owner
- **Description**
  - suspected component at fault
  - platform, symptoms, how to cause
  - diagnosis, work-arounds, and fix location
- **History**
  - log of all associated operations & comments

### Bug Report Life Cycle

(simplified typical model)

- New
- Unconfirmed
- Assigned
- Resolved
- Fixed
- Closed
- Reopen
- NonReproducible
- Duplicate (of XXX)
- Not A Bug (reason)
- WillNotFix (reason)

### Good Bug Reports

- Clearly describe the problem
  - what should have happened, what did happen
- Clearly describe the impact
  - consequences to the affected users
- Clearly describe the affected systems
  - what platforms, what versions of what software
- Clearly describe how to cause the problem
  - ideally with a relatively simple test case
  - developing minimal failure cases is work
- Dispassionate, separate facts from opinions

### Bug Tracking Systems

- List of open tasks for developers
  - what work needs to be done
  - communication between developers & users
- Current status of product/development
  - what known problems are there (#, severity)
  - what is the status of each
- Support database
  - known problems and work arounds
- Project management database
  - defect detection rates, fix rates
  - number of problems discovered
  - regression and not-a-bug rates

### Bug Triage and Priority

- in emergency, sort patients into 3 groups
  - those who can wait a few hours
  - those who will die no matter what
  - those who need immediate attention
- bugs are prioritized in a similar fashion
  - disastrous bugs, **must** fix ASAP
    - they render the product unacceptable
  - serious bugs, **should** fix before shipment
    - they significantly compromise value of the product
  - minor bugs, fixes can be deferred to later

### For Next Lecture

- McConnell, chapter 29
  - overview of basic integration strategies
- Kampe, Integration Strategy
  - integration, architecture, testing, and schedule
- Kampe, Solaris Train Model
  - incremental integration for existing products
- Fowler, Continuous Integration
  - good advice for a more rational process
- Wikipedia, Test Driven Development
  - introduction to a useful agile development practice
- Kampe, Test Harnesses
  - introduction to a general class of testing tools
- Sourceforge: CUnit (just skim)
  - an instance of a noble family of testing harnesses
Supplementary Slides

---

**bug verification**

- reproduce the reported error
  - find a test case that reliably causes error
    - it is difficult to fix a problem one cannot observe
  - confirm that we observe reported behavior
    - we may have misunderstood the report
  - enable a preliminary assessment
    - does this, indeed, appear to be an error

- verify that program behavior is wrong
  - problem may be user-error or documentation
  - user may have unreasonable expectations

---

**minimal failure cases**

- some failures are complex or subtle
  - failure occurs after millions of operations
  - failure depends on environmental factors
  - failure isn’t always in the same place
- find a simple case that fails solidly
  - isolate the contributing factors
  - find minimal combination that fails reliably
- this makes problem easier to reproduce, and easier to debug