Reproducing Problems

Andreas Zeller
The First Task

- Once a problem is reported (or exposed by a test), some programmer must fix it.
- The first task is to *reproduce* the problem.
Why reproduce?

• **Observing the problem.** Without being able to reproduce the problem, one cannot observe it or find any new facts.

• **Check for success.** How do you know that the problem is actually fixed?
A Tough Problem

• Reproducing is one of the toughest problems in debugging.

• One must
  • recreate the environment in which the problem occurred
  • recreate the problem history – the steps that lead to the problem
Reproducing the Environment

<table>
<thead>
<tr>
<th>Where to reproduce?</th>
<th>Chances of Success</th>
<th>Costs</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>+</td>
<td>--</td>
</tr>
<tr>
<td>Developer</td>
<td>0</td>
<td>+</td>
</tr>
</tbody>
</table>
Iterative Reproduction

• Start with your environment

• While the problem is not reproduced, adapt more and more circumstances from the user’s environment

• Iteration ends when problem is reproduced (or when environments are “identical”)

• Side effect: Learn about failure-inducing circumstances
Setting up the Environment

- Millions of configurations
- Testing on dozens of different machines
- All needed to find & reproduce problems
Virtual Machines

System Summary

<table>
<thead>
<tr>
<th>Processors (2)</th>
<th>Memory (1.5 G)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual Machines</td>
<td>19 %</td>
</tr>
<tr>
<td>Other</td>
<td>20 %</td>
</tr>
<tr>
<td>System Total</td>
<td>39 %</td>
</tr>
</tbody>
</table>

Virtual Machines (8)

<table>
<thead>
<tr>
<th>HB</th>
<th>Display Name</th>
<th>Up</th>
<th>% CPU</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Windows XP Professional</td>
<td>6 hours</td>
<td>7</td>
<td>301.0 M</td>
</tr>
<tr>
<td></td>
<td>Windows 2000 Cluster Node 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Windows 2000 Cluster Node 1</td>
<td>Suspended</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>WinNT IIS Web Server</td>
<td>Suspended</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Novell NetWare 6.5</td>
<td>22 hours</td>
<td>3</td>
<td>157.0 M</td>
</tr>
<tr>
<td></td>
<td>Windows Server 2003</td>
<td>35 hours</td>
<td>8</td>
<td>176.0 M</td>
</tr>
<tr>
<td></td>
<td>Red Hat Enterprise Linux 3</td>
<td>35 hours</td>
<td>1</td>
<td>260.0 M</td>
</tr>
<tr>
<td></td>
<td>SuSE Linux Enterprise Server 8</td>
<td>Suspended</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Download VMware Virtual Machine Console: Windows (exe) | Linux (rpm) | Linux (tar.gz)

Copyright © 1998-2004 VMware, Inc. All rights reserved.
Protected by one or more of U.S. Patent Nos. 6,397,242 and 6,496,847; patents pending.
Reproducing Execution

- After reproducing the environment, we must reproduce the execution
- Basic idea: Any execution is determined by the input (in a general sense)
- Reproducing input → reproducing execution!
Program Inputs

- Randomness
- Operating System
- Communication
- Schedules
- User Interaction
- Physics
- Data
- Debugging Tools
Program Inputs
Data

• Easy to transfer and replicate
• Caveat #1: Get all the data you need
• Caveat #2: Get only the data you need
• Caveat #3: Privacy issues
Program Inputs

User Interaction → Data → Program
User Interaction

Input Sources

Record

Replay
Recorded Interaction

send_xevents key H @400,100
send_xevents wait 376
send_xevents key T @400,100
send_xevents wait 178
send_xevents key T @400,100
send_xevents wait 214
send_xevents key P @400,101
send_xevents wait 537
send_xevents keydn Shift_L @400,101
send_xevents wait 218
send_xevents key “;” @400,101
send_xevents wait 167
send_xevents keyup Shift_L @400,101
send_xevents wait 1556
send_xevents click 1 @428,287
send_xevents wait 3765
Program Inputs

- Communication
- User Interaction
- Data
Communication

• General idea: Record and replay like user interaction

• Bad impact on performance

• Alternative #1: Only record since last checkpoint (= reproducible state)

• Alternative #2: Only record “last” transaction
Randomness

• Program behaves different in every run
• Based on random number generator
  • Pseudo-random: save seed (and make it configurable)
  • Same applies to time of day
• True random: record + replay sequence
Program Inputs

- Randomness
- Operating System
- Communication
- User Interaction
- Data

Program
Operating System

• The OS handles *entire* interaction between program and environment

• Recording and replaying OS interaction thus makes entire program run reproducible
# include <string>
#include <iostream>
using namespace std;

string secret_password = "secret";

int main()
{
    string given_password;
    cout << "Please enter your password: ";
    cin >> given_password;
    if (given_password == secret_password)
        cout << "Access granted."
    else
        cout << "Access denied."
}
Traced Interaction

$ c++ -o password password.C
$ strace ./password 2> LOG
Enter your password: secret
Access granted.
$ cat LOG
...
write(1, "Please enter your password: ", 28) = 28
read(0, "secret\n", 1024) = 7
write(1, "Access granted.\n", 16) = 16
exit_group(0) = ?
How Tracing works

Program

Tracer

Kernel
Replaying Traces

Program → Tracer

Tracer → Trace File

Kernel
Challenges

- Tracing creates *lots* of data
- Example: Web server with 10 requests/sec
  A trace of 10 k/request means 8GB/day
- All of this must be *replayed* to reproduce the failure (alternative: *checkpoints*)
- Huge performance penalty!
Program Inputs

- Randomness
- Operating System
- Communication
- Schedules
- User Interaction
- Data
Accessing Passwords

Thread A
- open (".htpasswd")
- read (...)
- modify (...)
- write (...)
- close (...)

Thread B
- open (".htpasswd")
- read (...)
- modify (...)
- write (...)
- close (...)

.htpasswd file
Lost Update

Thread A
- open(".htpasswd")
- read(…)
- read(…)
- modify(…)
- write(…)
- close(…)

A’s updates get lost!

Thread B
- open(".htpasswd")
- read(…)
- modify(…)
- write(…)
- close(…)

A’s updates get lost!
Reproducing Schedules

• Thread changes are induced by a scheduler
• It suffices to record the schedule (i.e. the moments in time at which thread switches occur) and to replay it
• Requires deterministic input replay
Constructive Solutions

- Lock resource before writing
- Check resource update time before writing
- ... or any other synchronization mechanism
Program Inputs

Randomness  Operating System

Communication  Schedules

User Interaction

Data

Physics
Physical Influences

- Static electricity
- Alpha particles \textit{(not} cosmic rays)\textit{)}
- Quantum effects
- Humidity
- Mechanical failures + real bugs

Rare and hard to reproduce
Program Inputs

- Randomness
- Operating System
- Communication
- Schedules
- User Interaction
- Physics
- Data
- Debugging Tools
A Heisenbug

- Code fails outside debugger only

```c
int f() {
    int i;
    return i;
}
```

In program: returns random value

In debugger: returns 0
More Bugs

• Heisenbug
  • A bug that disappears when you attempt to find it
• Bohr Bug
  • Repeats reliably and consistently; opposite of Heisenbug
• Mandelbug
  • Bug that’s difficult to find/fix because of complexity/unpredictability
• Schrödinbug
  • The bug manifests only after someone reads the source and realizes the code never should have worked in the first place. Then the code stops working until fixed.
Isolating Units

- Capture + replay *unit* instead of program
- Needs an *unit control layer* to monitor input
Isolated Units

- **Databases.** Replay only the interaction with the database.

- **Compilers.** Record + replay intermediate data structures rather than the entire front-end.

- **Networking.** Record + replay communication calls.
interface Map {
    void add(string key, int value);
    void del(string key);
    int lookup(string key);
};

// create a log that looks like Map calls!

class MapLogger implements Map {
    Map theMap;
    MapLogger(Map m) { theMap = m; }
    void add(string key, int value) {
        System.out.println("map.add(" + key + ", " +
        value + ");"); //
    }
    // etc
}
More Interaction

- Variables (hard to detect)
- Other units (break dependency if needed)
- Time (record + replay, too)
Mock Objects

• A Mock Object simulates an original object

• Its implementation tells how to react on specific calls (i.e. returning other mock objects)

• Can be combined with recording, too!