Debuggers and Compile Tools

Comp-206 : Introduction to Software Systems
Lecture 17

Alexandre Denault
Computer Science
McGill University
Fall 2006

Bug

- A software bug is an error, flaw, mistake, failure, or fault in a computer program that prevents it from working as intended, or produces an incorrect result.
- Bugs can exist at different levels
  - Design
  - Source Code
- Bugs have severities
  - Some bugs simply crash an application.
  - Some bugs cause loss of data.
  - Some bugs cause loss of money.
  - The worst bugs cause loss of life.

Origin or Bugs

In 1946, when Hopper was released from active duty, she joined the Harvard Faculty at the Computation Laboratory where she continued her work on the Mark II and Mark III. Operators traced an error in the Mark II to a moth trapped in a relay, coining the term bug. This bug was carefully removed and taped to the log book September 9th 1945. Stemming from the first bug, today we call errors or glitch's [sic] in a program a bug.

Well known bugs

- Y2K – date overflow
- Ariane 5 Flight 501 – conversion overflow
- MIM-104 Patriot bug – clock drift
- Therac-25 – multiple causes

Debugging

- Debugging is the act of finding the source of a bug and fixing it.
- The hardest part of debugging is finding the problem.
  - This becomes exponentially difficult when the source is very large.
- Just analyzing the code is not always enough to find bugs.
  - You need to run the application.
- Debuggers are tools that help the debugging process.

Debuggers

- Debuggers allow a programmer to run the application in a different mode.
- When running under this different mode (“debug mode”), many new features are available to the programmer.
  - If an application crashes, the programmer can see what line caused the fatal operation. He can also consult the content of the memory.
  - A programmer can an application, line-by-line. At each line, he can consult the content of the memory.
- Most programming language have their debugger.

Why not printf?

- Sometimes you need a lot of printf()'s, and it can get tedious putting them in and taking them out.
- A symbolic debugger can do an awful lot that printf() can't.
  - halt the program temporarily,
  - list source code,
  - print to the datatype of a variable
  - jump to an arbitrary line of code
- You can use a symbolic debugger on a process that has already crashed and died.

Introduction to GDB

- GDB is a debugger which is part of the Free Software Foundation's GNU operating system.
- GDB can be used to debug C, C++, Objective-C, Fortran, Java and Assembly programs.

Using GDB

- First, you must compile the executable with the –g flag

  gcc HelloWorld.c -g -o HelloWorld
- The program is then compiled with additional information (such as the source code).
- This additional information is needed by the debugger.

Starting GDB

- Any executable compiled with the –g flag can be used with gdb.

```
gdb HelloWorld
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
  Copyright 2003 Free Software Foundation, Inc.
  GDB is free software, covered by the GNU
  General Public License, and you are welcome to
  change it and/or distribute copies of it under
  certain conditions. Type "show copying" to see
  the conditions. There is absolutely no
  warranty for GDB. Type "show warranty" for
  details. This GDB was configured as "i386-
  redhat-linux-gnu"...
(gdb)
```

Getting Help

- When using GDB, you can always type in the “help” command to get a list of available commands.

```
(gdb) help
```

- You can also get help on a specific command by typing “help” and the name of that command.

```
(gdb) help breakpoints
Making program stop at certain points.
```

Looking at the Source code

- You can use the *list* command to display the source code you are debugging.
- List either takes a filename and a line number, or a function name.

```
(gdb) list main.c:10
5
6       int main (int argc, char **argv) {
7
8               library* mylibrary = createLibrary(20);
9
10              loadLibrary("lib.txt", mylibrary);
11
12              addBookToLibrary(mylibrary, createBook("Lotr", "Tolkien", 300));
13              addBookToLibrary(mylibrary, createBook("Harry_Potter", "Rowing",
   50));
14              addBookToLibrary(mylibrary, createBook("C_Prog", "Kerning", 100));
```

- You can change the size of a list using the *set listsize* command.

Breakpoints

- A breakpoint is a stop or a pause place in a program for debugging purposes.
- To properly debug an application, we need to place a breakpoint before the code we want to observe.
- This can be done using the *break* command.

```
(gdb) b main.c:8
Breakpoint 1 at 0x8048534: file main.c, line 8.
```

- You can also set breakpoints using the function name.

```
(gdb) b main
Breakpoint 2 at 0x8048545: file main.c, line 8.
```

- You can remove a breakpoint using the *delete* command and list all breakpoints using the *info breakpoints* command.

Running an Application

- First, we need to start the application.
- This can be done using the *run* command.

```
(gdb) run
Starting program:
   /home/user/adenau/cs206/debug/library/library


Breakpoint 1, main () at main.c:8
8                library* mylibrary = createLibrary(20);
(gdb)
```

- The application will run until hit its a break point.
- You can continue running the application using *continue.*

Step-by-Step

- You can also run an application line-by-line.
- Both the *step* and *next* command allow you to do this.
- However, if you encounter a function call,
  - Step will step into that function call.
  - Next will continue over the code in the function call.

Where am I?

- At any moment, even after a crash, you can use the *where* command to produce a backtrace. (stacktrace)

```
(gdb) where
#0  createBook (title=0x804a218 "Lotr", author=0x804a110
   "Tolkien",
    pages=300) at book.c:8
#1  0x080487fe in loadLibrary (filename=0x8048aa8
   "lib.txt",
    myLibrary=0x804a008) at file.c:20
#2  0x08048567 in main () at main.c:10
(gdb)
```

What's that value?

- You can use the print command to print out the value of a variable.

```
(gdb) print newBook
$1 = (book *) 0x8048b01
```

- You can also use the display command to repetitivly display the value of a variable at each step.

```
(gdb) display newBook
1: newBook = (book *) 0x8048b01
(gdb) step
9     newBook->title = (char*)malloc(strlen(title)+1);
1: newBook = (book *) 0x804a488
```

- You can use the undisplay command to cancel the display of a variable.

Changing a variable

- You can use the *set* command to change the value of a variable.

```
(gdb) print mylibrary
$2 = (library *) 0x804a008
(gdb) set mylibrary = 0
(gdb) print mylibrary
$3 = (library *) 0x0
(gdb)
```

DDD

- GNU DDD is a graphical front-end for command-line debuggers such as
  - GDB,
  - DBX,
  - JDB,
  - the Python debugger,
  - Etc
- DD can do four main kinds of things :
  - Start your program.
  - Make your program stop on specified conditions.
  - Examine what has happened.
  - Change things in your program.

Lint

- Originally, lint was a tool that flagged suspicious and non-portable constructs (maybe bugs?) in C.
- Now, the name is used for any tools that reports suspicious behavior.
- Suspicious behavior include
  - variables being used before being set
  - conditions that are always true/false
  - calculations whose result is likely to be outside the range of values representable in the type used

Unacceptably Slow

- An application that is unacceptably slow is considered slow by some.
- Thus, the application needs to be optimized.
  - The code of the application needs to be improved so the application can accomplish the same tasks with less resources.
  - The behavior of the application should remain unchanged.
- Before optimizing, we need to find “what is slow”.

Profilers

- Profilers are dynamic performance analysis tools
  - As opposed to lint which are static analysis tools.
- The record data on the operation of an application.
- This data is often used to determine which part of an application needs optimization.
  - Speed, memory, etc.