CptS 360 (System Programming)
Unit 4: Debugging

Bob Lewis

School of Engineering and Applied Sciences
Washington State University

Spring, 2022

Motivation

- You're probably going to spend most of your code development time debugging.
- You need to know several different strategies to debug code.
- Debugging is part art, part detective work.

Reference

- `$ info gdb`
  (Be sure the `gdb-doc` package is installed.)
- `$ info ddd`
  (Be sure the `ddd-doc` package is installed.)

Phases of Debugging

- testing
  Does a bug exist?
- stabilization
  Is the bug consistently repeatable?
- localization
  What component (executable - module - function - line) is causing the bug?
- correction
  What are the possible fixes for the bug? Which one is best?
- verification
  Did the fix really work?
- retrospective
  Is there any other part of this program where I might have made the same mistake?

Design for Debugging

- Getting a core dump
- Be sure to set

  ```
  $ ulimit -c unlimited
  ```

  so you can get a core dump.
- "^\\" will force a core dump from a program, if core dumping is enabled.
- "`$ gcore <pid>`", where `<pid>` is a process id, will get a core dump of that process.
- Use *file(1)* to identify core files.

Dealing with Errors

Name some errors that your program can detect.

How should your code indicate an error?

Alternatives:

1. Do nothing.
   What will this lead to?
2. Print an error message (where?) and call *exit(3)*. (With what argument?)
3. Print an error message and call *abort(3)*. (With no argument.)
4. Return an error code to its caller...
   4.1 As the function return.
   4.2 In a global variable.
   4.3 In a pointed-at argument.
5. Raise an exception (in C? maybe...).

Assertions

```
#include <assert.h>
...
assert(n > 0);
```

- stops program and dumps core (for perusal by *gdb(1)*)
- disabled by the compiler flag "`-DNDEBUG`"

(Study the `assert.h` header to see a clever example of *cpp(1)* use.

Simple Debugging Approaches

- Inspection: Look at the code
  - Easy, when it works.
- Instrumentation
  - `printf()` is crude, but effective
  - especially when used with...

Scaffolding

- Traditional:

```
#ifndef NDEBUG
... // debug stuff here ignored if -DNDEBUG is used
#endif
```

  assert(3) invocations are also ignored for "-DNDEBUG".

- I like this better:

```
#if 0 // set to 1 to revert to old code
... // old code
#else
... // old code being debugged *or* new, improved code
#endif
```

  Note how you can enable/disable whole blocks of code by changing one character, or use a `#define` to group blocks.

These are better (and more couth) than "commenting out" code, as comments are unestable. They're also easy to remove.

gdb(1): The GNU Debugger

- How does it work?
- Try "`gdb --tui {program_name}`".
  - use `Ctrl-L` to refresh screen
- GUI wrappers:
  - *ddd(1)*
    data display debugger
  - *kdbg(1)*
    KDE version
  - (others, including many IDEs)

(*Run the change demo – in class only.*)

A Few Useful *gdb(1)* Commands

- `help`
- `info`
- `run`
- `backtrace` (or `where`)
- `print` *expression*
- `display`
- `list` *line/function*
- `break` *line/function [condition]*
- `continue`
- `tbreak` *line/function [condition]*
- `display` *expression*

Commands can be abbreviated to the shortest unique prefix ("b", "tb", "co", "di", "p", etc.)

Profiling

- Compile with "`-pg`".
- Run program (producing "`gmon.out`" file).
- Use *gprof(1)*.

Excellent measurement of where your program is spending time on a function-by-function basis.

There are other tools (e.g. coverage tests).

Tracing library calls

- *strace(1)*
  system call tracing
- *ltrace(1)*
  library call tracing

demo

Memory Debugging

- different from a program debugger
- idea is to detect logical, but non-fatal errors:
  - array bounds errors
  - accessing freed heap memory
  - "memory leaks": allocated memory that nothing points to
  - allocating memory (stack or heap) that is never used

valgrind(1)

`http://valgrind.org`

- pronunciation:
  "val" (as in "value") - "grind" (as in "grinned")
- In Norse mythology, Valgrind was the main entrance to Valhalla.
  - Only those deemed worthy by the guards to the entrance are permitted to enter.
- Based on software emulator of: X86, AMD64, ARM, ARM64, PPC32, PPC64, S390X, or MIPS32 hardware for Linux, Android, or Darwin OSes. (Not available for Windows, but can be used with WSL or Wine.)
- Works like an interpreter (e.g. Java, Python), but interprets object code, not bytecode.

valgrind(1) Syntax

```
$ valgrind --tool=toolName program args
```

where *toolName* is:

- `memcheck`
  a heavy-weight memory checker (the default we'll discuss here)
- `cachegrind`
  a cache usage profiler
- `callgrind`
  is `cachegrind` with call graph production
- `helgrind`
  a data race detector (works with threads, as do the rest)
- `massif`
  a heap profiler
- `lackey`
  a sample you can look at to build own tools yourself that interact with *valgrind(1)* data.

valgrind(1) Advantages and Disadvantages

advantages:

- checks memory usage (at *bit* level!)
- performs detailed profiling (including cache usage)
- has been used on systems with 25 million lines of code (!)
- works in the presence of threads
- works with compiled code (even if it's not optimized)
- works with any language (geared towards C/C++, though)

disadvantage:

- programs are 5-100 times slower
- (so don't use it <u>all the time</u> on long programs!)

When Should You Use *valgrind(1)* For Error Detection?

- all the time for short programs
- in automatic testing
- especially as part of regression testing
- after big changes
- when a bug has been detected
- after a bug has been fixed
- when a bug is suspected
- before a release

When Can You Use *valgrind(1)*?

Any time, e.g.

```
$ valgrind ls -al
```

Notes:

- works on a binary – no source required
- report written to standard error

valgrind(1) Demos

Here are some *valgrind(1)* demos:

- memory leaks:
  `demos/d07_valgrind_mem_leak/valgrind_mem_leak.c`
- invalid read/writes:
  `demos/d08_valgrind_inv_rw/valgrind_inv_rw.c`
- uninitialized variables:
  `demos/d09_valgrind_uninit/valgrind_uninit.c`
- bounds checking:
  `demos/d10_valgrind_bounds_check/valgrind_bounds_check.c`

Under the Hood: How Does *valgrind(1)* Work?

- As above, *valgrind(1)* is a processor emulator.
- For each byte of data (including registers), *valgrind(1)* adds 9 additional bits of information:
  - 8 “V” (“valid”) bits for each bit in the byte.
  - 1 “A” bit indicating that the address of the byte is valid.
- V-bits are checked when:
  - data is used for address generation
  - a control-flow decision is to be made (see below)
- A-bits are:
  - set when memory is allocated
  - cleared when memory is freed

Example: *valgrind(1)* Looks at Uninitialized Values

Assume that we've never set a value for `j` then run

```
for (i = 0; i < 10; i++)
    j += a[i];
if (j == 42)
    printf("Got the answer!\n");
```

valgrind(1) does not complain at the `j` increment, since the undefinedness is not "observable" – it won't affect your program output – but it will complain at the `if` statement.

This is why you get those "uninitialized variable used in conditional" messages from within system I/O or string calls instead of your own code. To see why...

Why not Check Uninitialized Assignments?

Consider this code:

```
struct S {
    int i;
    char c;
} s[2];
...
s[0].i = 42;
s[0].c = 'x';
s[1] = s[0];
```

- With 32-bit `int`s and 8-bit `char`s, `S` structs *could* occupy as little as (Q: How many?) bytes, but compilers make `struct`s word-aligned (see why?), so they "pad" them to 8 bytes.
- At run time, this "padding" intent has been lost. Uninitialized bits are uninitialized bits.
- The assignment "`s[1] = s[0];`" will then copy uninitialized bytes, but it's okay because those bytes probably won't be used in `s[0]` (or `s[1]`). If, by some hack, they are used (in a conditional), they'll be detected.

cachegrind: Cache Optimization

- basic idea: On x86's there are actually two (or three) levels of cache:
  - L1 (fastest)
    - instructions
    - data
  - L2 (next fastest)
    - unified (both instructions and data)
- *cachegrind* (i.e., the *cachegrind* option of *valgrind(1)*) uses the same simulator as *valgrind(1)* to find out the rates of cache hits and misses.
- It can annotate your code with them line-by-line.