EE485: Introduction to Environment and Tools for Modern Software Development
Spring 2021

Lecture 6: Debugging Techniques

KAIST EE

Acknowledgement: content borrowed from Chapter 5 of
“The Practice of Programming” by Brian Kernighan & Rob Pike
Complementary to the “Debugging” Lecture in EE209

Debugging

- Good programmers know debugging takes as much time as writing the original code.
  - Some people enjoy debugging – solving a puzzle!
  - Others don't – can't submit my homework code
- Good strategy for debugging - write the code to ***avoid*** bugs
  - Some of the popular strategies
    - Good design
    - Good coding style
    - Checking the boundary condition
    - Assertions
    - Sanity checks in the code
    - Defensive programming
    - Well-designed interfaces
    - Limited global data
    - Checking tools
  - Prevention of bugs is a virtue!

Language Features May Help or Not

- Language features that prevent bugs.
  - Range checking of array subscripts
  - Restricted pointers (or no pointers)
  - Garbage collection
  - String data types
  - Strong type checking

- Language features that are prone to generate bugs
  - Global variables
  - `goto` statements
  - Unrestricted pointers
  - Automatic type conversions

Bugs

- What if my program crashes or prints nonsense or hangs forever?
  - Novice programmers? blame compilers, libraries, anything other than my code.
  - Experienced programmers? most problems are likely to be their faults.

- Most bugs are simple
  - Infer the bug with the stack trace, debugging output, etc.
  - Requires backward reasoning: how it happened?
  - Fixing bugs is a like solving a mystery (who's the murderer?)

Easy Bugs – Common Bugs

(1) Look for common patterns

```
int main()
{
 int n;
 double d = 3.14;
 printf("%d %f\n", d, n);
 scanf("%d", n);
 return 0;
}
```

```
printf("%d %f", n, d);
```

```
scanf("%d", &n);
```

- Easy to detect these errors by monitoring compiler warnings (-Wall)

```
$ gcc test.c -Wall
test.c:8:11: warning: format '%d' expects argument of type 'int', but argument 2 has type 'double' [-Wformat=]
  printf("%d %f\n", d, n);
        ~^
        %f
test.c:8:14: warning: format '%f' expects argument of type 'double', but argument 3 has type 'int' [-Wformat=]
  printf("%d %f\n", d, n);
          ~^
          %d
test.c:9:10: warning: format '%d' expects argument of type 'int *', but argument 2 has type 'int' [-Wformat=]
  scanf("%d", n);
       ~^
test.c:8:2: warning: 'n' is used uninitialized in this function [-Wuninitialized]
  printf("%d %f\n", d, n);
  ^~~~~~~~~~~~~~~~~~~~~~~
```

Easy Bugs- Program Hangs Forever or Crashes

(2) Program hangs or crashes? Get the stack trace & where!

```
$ gdb yourprog
(gdb) run
^C
Program received signal SIGINT, Interrupt.
(gdb) where
#0 __GI___libc_read (fd=0, buf=0x8402470, nbytes=512) at ../sysdeps/unix/sysv/linux/read.c:27
#1 _IO_new_file_underflow (fp=0x7fffff3eba00 <_IO_2_1_stdin_>) at fileops.c:531
#2 __GI__IO_default_uflow (fp=0x7fffff3eba00 <_IO_2_1_stdin_>) at genops.c:380
#3 _IO_vfscanf_internal (s=<>, format=<>, argptr=argptr@entry=0x7fffffffee220, errp=errp@entry=0x0)
at vfscanf.c:630
#4  __isoc99_scanf (format=<optimized out>) at isoc99_scanf.c:37
#5  main () at test.c:9
```

You need to type Ctrl+C to stop the program

Program hangs waiting for input

Similarly, you can easily find out where the program crashes

```
$ gdb yourprog
(gdb) run
Program received signal SIGSEGV, Segmentation fault.
0x00007fffff06f8c2 in _IO_vfscanf_internal (s=<…>, format=<…>, argptr=…, errp=…) at vfscanf.c:1898
(gdb) where
#0  _IO_vfscanf_internal (s=<…>, format=<…>, argptr=…, errp=…) at vfscanf.c:1898
#1  __isoc99_scanf (format=<optimized out>) at isoc99_scanf.c:37
#2  main () at test.c:9
```

Debugging with Core Dump

- Sometimes, your program does *not* crash *only* when it runs with gdb.
  - Why? running with `gdb` incurs some overhead, which makes the program avoid the crashing course.
- Run the program without `gdb` and make it crash with a core dump
  - Core dump: a binary file that contains the program state at the time of crash

```
$ ./myprog
…
Segmentation fault (core dumped)
```

    - Why called “core”? “core” used to mean “memory” in the past
  - Novice programmers: well, I don’t see any file that looks like a “core dump” file!
- Need to set a shell variable before running the application
  - To enable storing core dump – default option is to disable core dump

```
$ ulimit -a
…
core file size          (blocks, -c) 0
…
```

Core dump file size is set to 0; disabled!

Debugging with Core Dump

- Enable core dump file: set its file size large enough
  - How large? Depends on your program size, but 10MB is enough for most cases.

```
$ ulimit -c 10000000
$ ulimit -a
…
core file size          (blocks, -c) 10000000
…
```

- How to run `gdb` with core dump file?

```
$ ./myprog
…
Segmentation fault (core dumped)
$ ls core*
core
$ gdb myprog core
Core was generated by `./myrpog'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  _IO_vfscanf_internal (s=<>, format=<>, argptr=…, errp=…)  at vfscanf.c:1899
(gdb) where
#0  _IO_vfscanf_internal (s=<>, format=<>, argptr=…, errp=…)  at vfscanf.c:1899
#1 __isoc99_scanf (format=<optimized out>) at isoc99_scanf.c:37
#2 main () at test.c:9
```

Other Ways to Deal with Bugs

(3) Read before typing – resist the urge to start typing

- Hastily changing the code to see if it fixes the code would waste time.
- That could introduce new bugs without fixing the original ones.
- Take a break, and return - what you currently see may be what you meant, not what you wrote!

(4) Don’t make the same mistake twice

- Copy and paste the boilerplate code from somewhere else.
- And you forgot the code has a problem.

(5) Debug it now, not later

- Mars Pathfinder malfunctioned once every day – could have been fixed before launch as the engineers knew the problem (but forgot to fix!)

(6) Explain your code to someone else

- While explaining the problem, you might realize what the problem is!

The First Bug on Mars

In 1971, the USSR delivered the first planetary rovers on skis to Mars, whose task was to puncture the surface with a rod (housing a dynamic penetrometer and a radiation densitometer) to see if Mars was solid or liquid dusty. The first probe crashed on November 27; the second soft-landed on December 2 but didn't manage to get out of the "shell" of the lander, so that attempt didn't count.

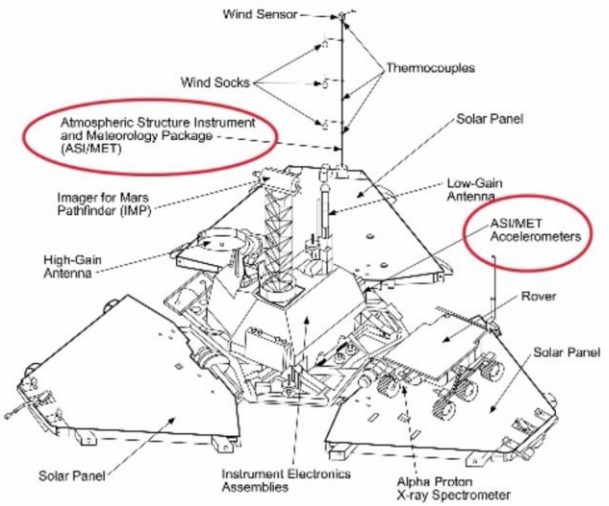
> This article was originally published in Russian on habrahabr.ru. The original and translated versions are posted on our website with the permission of the author.

25 years later

On July 4, 1997, the U.S. probe arrived at Mars and brought a "sojourner" with the first bug.

Priority inversion

Priority inversion occurs when two or more threads with different priorities start competing for CPU resources.



The lander was carrying a radiation-hardened IBM Risc 6000 Single Chip (Rad6000 SC) 20 MIPS CPU with 128 Mbytes of RAM and 6 Mbytes of EEPROM. The operating system used was VxWorks.

Dealing with Hard Bugs

- Sometimes you have no clue on what's going on!

(1) Make the bug reproducible

- Bugs that do not manifest often are hard to debug
- See if you can construct input/parameters to reproduce the bug "reliably"

(2) Divide and conquer

- Binary search to find the problem with minimal input
- Remove the half of the input and see if the problem persists – otherwise test with the remaining half. Repeat this until you come up with the minimal input.

(3) Study the numerology of failures

- See if the bug has some numerical pattern
- Say copy & paste loses a character at a random location. See if the location has some pattern (e.g., the distance of missing characters is 1023 characters long)
- Then, find the source code that's related to 1023 (like 1024). Overwrite to 1024-th place in a string array that is erased by "null"?

Dealing with Hard Bugs

(4) Display output to localize your search

- Display more information (e.g., print “can’t get here” , if it should be impossible to reach here – you can set up a breakpoint to inspect further)
- Display message in a compact fixed format – easy to find a pattern by eye or `grep` (e.g., format the value in the same way – `%x` or %p in C/C++)

(5) Add self-checking code

- Write a check function to see whether some code leads into a weird state
- Leave the check code even after debugging (e.g., `#ifdef DEBUGX … #endif`)

```
check(“before suspect”);
/* … suspect code … */
check(“after suspect”);
```

```
void check(char *s) {
     if (var1 > var2) {
          printf(“%s: var1 %d var2 %d\n”, s, var1, var2);
          fflush(stdout);
          abort();
     }
}
```

What If Nothing Works?

- What if your logic is plain wrong, but you don't realize it's wrong?
  - Debugger (gdb) might help to elucidate the errors in your mental model.

- Misconception
  - Operator precedence, wrong operator, wrong indentation

```
if (x & 1 == 0) // why always false?
    …
```

```
while ((c = getchar()) != EOF)
    if (c = '\n') break;
```

```
for (i=0; i < n; i++); // extra code
   a[i++] = 0;         // i++?
```

  - Local variable that hides a global name (or global var. in local scope)
  - Bugs that do not manifest often are hard to debug
  - See if you can construct input/parameters to reproduce the bug "reliably"

Non-reproducible Bugs

- Non-deterministic manifestation of bugs
  - Not likely to be a bug in your algorithm
  - Your code might be using information that changes each time the program runs
- Check whether all variables have been initialized
  - Some variable picks up random value, which ends up producing wrong behavior
- Bugs disappear if debugging code is added?
  - Likely to be a problem with dynamic memory allocation
  - Writing outside the allocated memory – the behavior could change if you add debugging code (like printf())

Non-reproducible Bugs

- Crashes in the middle of nowhere?
  - Far away from anything that could be wrong
  - Most likely problem is overwriting memory that isn't used until much later
  - Or return an address of a local variable as a pointer – recipe for delayed disaster!

```
char * msg(int n, char *s) {
    char buf[100];
    sprintf(buf, "error %d: %s\n", n, s);
     return buf;
}
```

  - Anything wrong with the following code?

```
for (p = listp; p != NULL; p = p->next)
    free(p);
```

Homework 6

- This homework is open-ended, and there's no "right" answer.
  - As long as you submit the homework with relevant comments, you'll get full credit.

- Homework description:
  - If you take EE209, share with us your debugging experience for assignment 2. Did you use gdb? If so, briefly explain why you used it and if it was useful to debug it with gdb? Point out any techniques in Lecture 5 or Lecture 6 that were useful in doing assignment 2 of EE209. If you did not use gdb, explain why. Explain any non-gdb techniques that were useful to debug your assignment code.

  - If you do not take EE209, please point out what techniques in Lecture 5/6 would be most useful to you, and explain why. If you have any recent C programming/debugging experience, explain what was the most difficult in terms of debugging. Any relevant story is fine.

  - Write your comments into a text file (named YourStudentID.txt) and submit it to KLMS