Week 5: GDB Debugging

Debugging

- finding and eliminating errors from programs

Process

1. Reproduce the bug
2. Simplify program input
3. Use a debugger to track down the origin of the problem
4. Fix the problem

Debugger

- a program that is used to run and debug other (target) programs
- advantages:
    - programmer can:
        - step through source code line by line
        - interact with and inspect program at run-time
        - if program crashes, debugger outputs where and why it crashed

GDB - GNU Debugger

- debugger for several languages
    - C, C++, Java, Objective-C, etc.
- allows you to inspect what the program is doing at a certain piont during execution
- logical errors and segmentation faults are easier to find with the help of gdb

Using GDB

1. Compile program

- **normally**: `$ gcc [flags] <source files> -o <output file>`
- **debugging**: `$ gcc [other flags] -g <source files> -o <output file>`
    - -g option enables built-in debugging support

2. Specify program to debug

```
$ gdb <executable>
# or
```

```
$ gdb
(gdb) file <executable>
```

3. Run program

- `(gdb) run [optional-arguments]`

4. In GDB interactive shell

- Tab to Autocomplete
- up-arrow arrows to recall history
- `help [command]` to get more info about a command

5. Exit GDB

- `(gdb) quit`

Run-Time Errors

Segmentation fault

```
Program received signal SIGSEGV, Segmentation fault. 0x0000000000400524 in function
(arr=0x7fffc902a270, r1=2, c1=5, r2=4, c2=6) at file.c:12
```

- line number where it crashed and parameters to the function that caused the error

Logic Error

- program will run and exit successfully

Setting Breakpoints

- used to stop running program at specific point

```
(gdb) break file1.c:6
# Program will pause when it reaches line 6 of file1.c

(gdb) break my_function
# Program will pause at top of my_function every time it is called

(gdb) break [position] if expression
# Program will pause at position only when expression evaluates to true
```

- you can view a list of all breakpoints via
    - `(gdb) info|i breakpoints|break|br|b`

Deleting, Disabling, and Ignoring Breakpoints

```
(gdb) delete [bp_number | range]
# Deletes specified breakpoint or range of breakpoints

(gdb) disable [bp_number | range]
# Temporarily disables a breakpoint or range of breakpoints

(gdb) enable [bp_number | range]
# Restores disabled breakpoints

# If no arguments are provided to above commands, all breakpoints are affected

(gdb) ignore bp_number iterations
# Instructs GDB to pass over a breakpoint without stopping a certain number of times
```

Displaying Data

```
(gdb) print [/format] expression
# Prints value of specified expression in the specified format
```

Formats

- d: decimal notation (default format for integers)
- x: hexadecimal
- o: octal
- t: binary

Resuming Execution After a Break

- when a program stops at a breakpoint, there are 4 possible kinds of gdb operations
    - **(c)ontinue**: debugger will continue executing until next breakpoint
    - **(s)tep**: debugger will continue to next source line
    - **(n)ext**: debugger will continue to next source line in the current stack frame (function)
    - **(f)inish**: debugger will resume execution until current function returns
        - execution stops immediately after program flow returns to function's caller
        - function's return value and line containing next statement are displayed

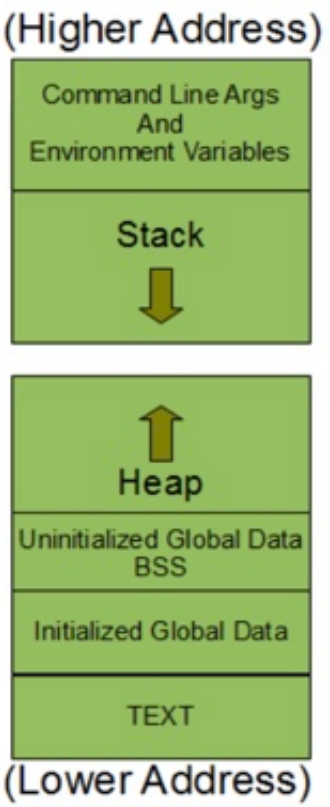
Watchpoints

- watch/observe changes to variables

```
(gdb) watch my_var
# sets a watchpoint on my_var
# debugger will stop the program when the value of my_var changes
# old and new values will be printed

(gdb) rwatch expression
# debugger stops the program whenever the program reads the value of any object involved in the evaluation of "expression"
```

Process Memory Layout

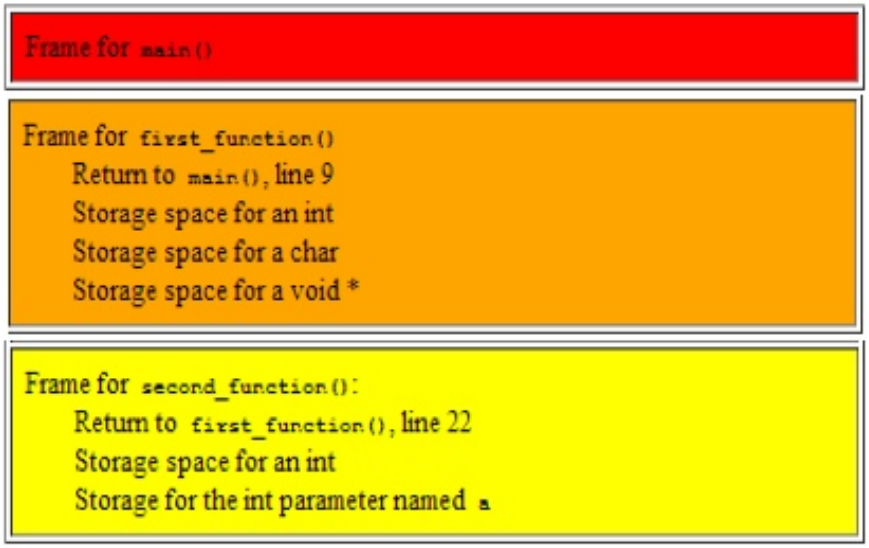


- **TEXT segment**
  - contains machine instruction to be executed
- **global variables**
  - initialized
  - uninitialized
- **heap segment**
  - dynamic memory allocation
  - malloc, free
- **stack segment**
  - push frame: function invoked
  - pop frame: function returned
  - stores
    - local variables
    - return address, registers, etc.
- **command line arguments and environment variables**

Stack Info

- a program is made up of one or more functions that interact by calling each other
- every time a function is called, an area of memory is set aside for it
    - area is caled a **stack frame** and holds the following crucial info
        - storage space for all local variables
        - memory address to return to when the called function returns/ends
        - arguments, or parameters of the called function
- each function call gets its own stack frame
    - all stack frames make up the **call stack**

```
1   #include <stdio.h>
2   void first_function(void);
3   void second_function(int);
4
5   int main(void)
6   {
7      printf("hello world\n");
8      first_function();
9      printf("goodbye goodbye\n");
10
11     return 0;
12  }
13
14
15  void first_function(void)
16  {
17     int imidate = 3;
18     char broiled = 'c';
19     void *where_prohibited = NULL;
20
21     second_function(imidate);
22     imidate = 10;
23  }
24
25
26  void second_function(int a)
27  {
28     int b = a;
29  }
```



Analyzing the Stack in GDB

`(gdb) backtrace|bt`

- shows the call trace (call stack)
- without function calls:
    - #0 main() at program.c:10
    - one frame on the stack, numbered 0, and it belongs to main()
- after call to function display()
    - #0 display (z=5, zptr=0xbffffb34) at program.c:15 #1 0x08048455 in main() at program.c:10

- two stack frames
  - frame 1 belonging to main()
  - frame 0 belonging to display()
- each fram listing gives
  - the arguments to that function
  - the line number that's currently being executed within that frame

```
(gdb) info frame
# displays info about the current stack frame, including its return address and saved register values

(gdb) info locals
# list the local variables of the function corresponding to the stack frame, with their current values

(gdb) info args
# list the argument values of the corresponding function call
```

Other Useful Commands

```
(gdb) info functions
# list all functions in the program

(gdb) list
# lists source code lines around the current line
```

Lab 5

- Download old version of coreutils with buggy ls program
  - Untar, configure, make
- Bug: ls -t mishandles files whose time stamps are very far in the past. It seems to act as if they are in the future

```
$ touch -d '1918-11-11 11:00 GMT' wwi-armistice
$ touch now
$ sleep 1
$ touch now1
$ ls -lt wwi-armistice now now1
```

Output:

-rw-r--r-- 1 eggert eggert 0 Nov 11 1918 wwi-armistice

-rw-r--r-- 1 eggert eggert 0 Feb 5 15:57 now1

-rw-r--r-- 1 eggert eggert 0 Feb 5 15:57 now

- **Reproduce the Bug**
  - Follow steps on lab web page
- **Simplify input**
  - Run ls with –l and –t options only
- **Debug**
  - Use gdb to figure out what's wrong
  - $ gdb ./ls
  - (gdb) run –lt wwi-armistice now now1

  (run from the directory where the compiled ls lives)
- **Patch**
  - Construct a patch "lab5.diff" containing your fix
  - It should contain a ChangeLog entry followed by the output of diff -u

- **Reproduce the Bug**
  - Follow steps on lab web page
- **Simplify input**
  - Run ls with –l and –t options only
- **Debug**
  - Use gdb to figure out what's wrong
  - $ gdb ./ls
  - (gdb) run –lt wwi-armistice now now1

  (run from the directory where the compiled ls lives)
- **Patch**
  - Construct a patch "lab5.diff" containing your fix
  - It should contain a ChangeLog entry followed by the output of diff -u

- Don't forget to answer all questions! (lab5.txt)
- Make sure not to submit a reverse patch! (lab5.diff)
- "Try to reproduce the problem in your home directory, instead of the $tmp directory. How well does SEASnet do?"
  - Timestamps represented as seconds since Unix Epoch
  - SEASnet NFS filesystem has unsigned 32-bit time stamps
  - Local File System on Linux server has signed 32-bit time stamps
  - If you touch the files on the NFS filesystem it will return timestamp around 2054
  - => files have to be touched on local filesystem (df –l)
- Use "info functions" to look for relevant starting point
- Compiler optimizations: -O2 -> -O0
  - ./configure CFLAGS="…-O0"