Crash Course on UNIX and Systems Tools

Day 5 --- More Project Structure and Debugging

Project Structure: *Example makefile*

Here's where we are … let's improve it

```
CC:=gcc
CFLAGS:=-c
LIBS=:-lm

myprog: lib.o test.o
    $(CC) $^ $(LIBS) -o $@

lib.o: lib.c
    $(CC) $(CFLAGS) $< -o $@

test.o: test.c
    $(CC) $(CFLAGS) $< -o $@

clean:
    rm -f lib.o test.o myprog

.PHONY: clean
```

Project Structure: *Makefiles*

- Let's move our headers and sources into their own directories --- good convention (**`./src`** , **`./include`**)
  - Need to make sure that this is still valid: **`#include "lib.h"`**
- Use '**`-I./myheaders`**' to offer header files to each source

```
gcc -c -I./include lib.c -lm -o lib.o
```

- In the makefile --- simply add the flag at the end of **`CFLAGS`**

Project Structure: *Makefiles*

```
SRC:=./src
INC:=./include

CC:=gcc
CFLAGS:=-c -I$(INC)
LIBS=:-lm

myprog: lib.o test.o
	$(CC) $^ $(LIBS) -o $@

lib.o: $(SRC)/lib.c
	$(CC) $(CFLAGS) $< -o $@

test.o: $(SRC)/test.c
	$(CC) $(CFLAGS) $< -o $@

clean:
	rm -f lib.o test.o myprog

.PHONY: clean
```

Project Structure: *Makefiles*

- Still not that modularized … what if we have more `.o` to build?
  - If every `.o` is built in the same way, have one modular rule!
- Use ‘`%`’ to create a **pattern rule** that targets many files
  - ‘`%`’ is a **pattern substitution character** --- “`%.c`” matches all `.c` sources
  - Be careful about targets that can match multiple rules

Project Structure: *Makefiles*

```
SRC:=./src
INC:=./include

CC:=gcc
CFLAGS:=-c -I$(INC)
LIBS:=-lm

myprog: lib.o test.o
	$(CC) $^ $(LIBS) -o $@

%.o: $(SRC)/%.c
	$(CC) $(CFLAGS) $< -o $@

clean:
	rm -f lib.o test.o myprog

.PHONY: clean
```

Project Structure: *Makefiles*

- Still not that modularized … what about the dependencies?
  - Need to handle growing number of dependencies to build “**`myprog`**”
- Fetch all the sources at once using the **`wildcard`** function
  - **`SRCS:=$(wildcard $(SRC)/*.c)`**
  - Achieve wildcard function in a variable declaration
  - Here, **`*.c`** matches for all C sources, and the **wildcard** function will fetch these files (perform the expansion) and space-separate each match

Project Structure: *Makefiles*

- We’re not done --- notice that the dependencies are **`.o`**, and we fetched **`.c`** files (those **`.o`** files don’t exist yet)

```
myprog: lib.o test.o
```

- Create a list of **`.o`** deps. --- let’s use the **`patsubst`** function
  - **`OBJS=$(patsubst $(SRC)/%.c,%.o,$(SRCS))`**
    `$(patsubst to_match,to_gen,where_to_match_from)`
  - Generates a space-separated string via a pattern and search and replace

Project Structure: *Makefiles*

```
NAME:=myprog
SRC:=./src
INC:=./include

SRCS:=$(wildcard $(SRC)/*.c)
OBJS:=$(patsubst $(SRC)/%.c,%.o,$(SRCS))

CC:=gcc
CFLAGS:=-c -I$(INC)
LIBS:=-lm
```

```
$(NAME): $(OBJS)
	$(CC) $^ $(LIBS) -o $@

%.o: $(SRC)/%.c
	$(CC) $(CFLAGS) $< -o $@

clean:
	rm -rf $(OBJS) $(NAME)

.PHONY: clean
```

Debugging: *Running* `myprog`

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

- **Segmentation fault** --- we’re accessing/using memory that we should not have

What now?

Debugging: *gdb*

- Use a debugger to help you understand the problem --- **gdb**
- First --- compile with **symbols** and **debug information** (-g)

```
gcc -c -g -I./include lib.c -o lib.o
```

Debugging: **`gdb`**

- Run your program with gdb (with arguments, use the alternative)
  - *Please note in subsequent screenshots --- my paths may not always the same as yours --- try to locate the right file*

```
$ gdb myprog
Reading symbols from myprog...done.
(gdb)
```

Debugging: *gdb*

- **r** to run the program

```
(gdb) r
Starting program: /root/sandbox/project_structure/myprog

Program received signal SIGSEGV, Segmentation fault.
0x00005555555546c2 in recursive_factorial (
    n=<error reading variable: Cannot access memory at address
0x7fffff7feffc>) at src/lib.c:9
9        {

(gdb)
```

Debugging: *gdb*

- **`bt`** to see a **backtrace** (i.e. the calls that led to this position)

```
(gdb) bt
#0  0x00005555555546c2 in recursive_factorial (
    n=<error reading variable: Cannot access memory at address 0x7fffff7feffc>) at src/lib.c:9
#1  0x00005555555546d6 in recursive_factorial (n=-54407990) at src/lib.c:13
#2  0x00005555555546d6 in recursive_factorial (n=1287167194) at src/lib.c:13
#3  0x00005555555546d6 in recursive_factorial (n=1090044298) at src/lib.c:13
#4  0x00005555555546d6 in recursive_factorial (n=1785087258) at src/lib.c:13
```

… and keeps going …

Debugging: *gdb*

- **`list`** to see the erroring location in the source code

```
(gdb) list
4
5           /*
6            * Recursive factorial calculation
7            */
8           int recursive_factorial(int n)
9           {
10              /*
11               * Recurse
12               */
13              return recursive_factorial(n * (n - 1));
(gdb)
```

Debugging: *`gdb`*

The **conclusion**? We had the following problems:

- An **infinite recursion** caused by **`recursive_factorial`** (as seen from the stack trace using **`bt`**)
    - Why a **seg fault**? --- Infinite recursion will “smash the stack” as arguments and return addresses keep piling onto the stack (this is a 213 concept)
- A solution? Add the **base case**!

Debugging: *gdb*

```
/*
 * Recursive factorial calculation
 */
int recursive_factorial(int n)
{
    /*
     * Handle the base case
     */
    if (n == 0) { return 1; }


    /*
     * Recurse
     */
    return recursive_factorial(n * (n - 1));
}
```

Debugging: **gdb**

- Let’s try it again!

```
$ gdb myprog
Reading symbols from myprog...done.
(gdb) r
Starting program: /root/sandbox/crash/day3/intermediate/myprog

Program received signal SIGSEGV, Segmentation fault.
0x00005555555546c2 in recursive_factorial (
    n=<error reading variable: Cannot access memory at address 0x7fffff7feffc>) at src/lib.c:9
9       {
```

Same problem again … let’s examine more carefully in **gdb**

Debugging: *gdb*

- Let's use a breakpoint --- the program pauses once it reaches a point that we specify from the code. We can examine there
  - Break at **`lib.c:19`** (according to the stack trace from **`bt`**)

```
$ gdb myprog
Reading symbols from myprog...done.
(gdb) break lib.c:19
Breakpoint 1 at 0x6d2: file src/lib.c, line 19.
(gdb) r
Starting program: /root/sandbox/crash/day3/intermediate/myprog

Breakpoint 1, recursive_factorial (n=10) at src/lib.c:19
19          return recursive_factorial(n * (n - 1));
(gdb) 
```

Debugging: *gdb*

- We've hit the breakpoint (the program's execution has reached the line that we've specified)
  - Specifies current function (**`recursive_factorial`**), the specific arguments for this invocation (**`n=10`**), and the line in the source code

```
$ gdb myprog
Reading symbols from myprog...done.
(gdb) break lib.c:19
Breakpoint 1 at 0x6d2: file src/lib.c, line 19.
(gdb) r
Starting program: /root/sandbox/crash/day3/intermediate/myprog

Breakpoint 1, recursive_factorial (n=10) at src/lib.c:19
19          return recursive_factorial(n * (n - 1));
(gdb) █
```

Debugging: **gdb**

- We can visualize the code again with **`list`**

```
Breakpoint 1, recursive_factorial (n=10) at src/lib.c:19
19              return recursive_factorial(n * (n - 1));
(gdb) list
14
15
16              /*
17               * Recurse
18               */
19              return recursive_factorial(n * (n - 1));
20          }
21
22
23          /*
```

Debugging: `gdb`

- We have some info from the breakpoint --- what now?
- **Step** through the code to understand **how it's executing**
  - **`step`** (or **`s`**) will execute until the next line of source code and pause
  - Can also step **into a function** using **`s`** if a certain line has a function call
  - Other instructions like **`next`** (or **`n`**) will stay in the current function and step **over** function calls in code

Debugging: `gdb`

- In this case, we should **step into** since the function is **recursive**
- The following slide shows this process using **`s`** and **`bt`** to show where we are as we step into each recursive call

Debugging: *gdb*

```
Breakpoint 1, recursive_factorial (n=10) at src/lib.c:19
19          return recursive_factorial(n * (n - 1));
(gdb) bt
#0  recursive_factorial (n=10) at src/lib.c:19
#1  0x0000555555554773 in main (argc=1, argv=0x7fffffffe308) at src/test.c:8
(gdb) s
recursive_factorial (n=90) at src/lib.c:13
13          if (n == 0) { return 1; }
(gdb) bt
#0  recursive_factorial (n=90) at src/lib.c:13
#1  0x00005555555546e3 in recursive_factorial (n=10) at src/lib.c:19
#2  0x0000555555554773 in main (argc=1, argv=0x7fffffffe308) at src/test.c:8
(gdb) s

Breakpoint 1, recursive_factorial (n=90) at src/lib.c:19
19          return recursive_factorial(n * (n - 1));
(gdb) s
recursive_factorial (n=8010) at src/lib.c:13
13          if (n == 0) { return 1; }
(gdb) bt
#0  recursive_factorial (n=8010) at src/lib.c:13
#1  0x00005555555546e3 in recursive_factorial (n=90) at src/lib.c:19
#2  0x00005555555546e3 in recursive_factorial (n=10) at src/lib.c:19
#3  0x0000555555554773 in main (argc=1, argv=0x7fffffffe308) at src/test.c:8
(gdb)
```

Debugging: `gdb`

- By stepping and looking at each recursive call, we see an interesting pattern with the value of `n`:

```
(gdb) bt
#0  recursive_factorial (n=8010) at src/lib.c:13
#1  0x00005555555546e3 in recursive_factorial (n=90) at src/lib.c:19
#2  0x00005555555546e3 in recursive_factorial (n=10) at src/lib.c:19
#3  0x0000555555554773 in main (argc=1, argv=0x7fffffffe308) at src/test.c:8
(gdb)
```

- We're calculating a factorial --- the values of **`n` should be decrementing** --- the issue must be with **which value is passed** into each recursive call

Debugging: *`gdb`*

- Revisiting the source code (see previous slides), we see that the "`n *`" in Line 19 is in the completely **wrong place**

```
Breakpoint 1, recursive_factorial (n=10) at src/lib.c:19
19              return recursive_factorial(n * (n - 1));
(gdb) list
14
15
16              /*
17               * Recurse
18               */
19              return recursive_factorial(n * (n - 1));
20          }
21
22
23          /*
```

Debugging: *gdb*

The fix!

```c
/*
 * Recursive factorial calculation
 */
int recursive_factorial(int n)
{
    /*
     * Handle the base case
     */
    if (n == 0) { return 1; }


    /*
     * Recurse
     */
    return n * recursive_factorial(n - 1);
}
```

Debugging: ***`gdb`***

- The program runs, but the output is *suspicious* (see **`main`** in **`test.c`**)

`$ ./myprog`

`3628800` ← 10! according to **`recursive_factorial`**

`0` ← 10! according to **`iterative_factorial`**

`0.000000` ← √(10!) according to **`sqrt_of_factorial`** and **`iterative_factorial`**

Debugging: *gdb*

- The output should match --- seems like a problem in **`iterative_factorial`**

```
$ ./myprog
3628800
0
0.000000
```

- ← 10! according to **`recursive_factorial`**
- ← 10! according to **`iterative_factorial`**
- ← √(10!) according to **`sqrt_of_factorial`** and **`iterative_factorial`**

Debugging: *gdb*

- Let's run under **gdb** and break at **iterative_factorial**

```
$ gdb myprog
Reading symbols from myprog...done.
(gdb) break iterative_factorial
Breakpoint 1 at 0x6ec: file src/lib.c, line 31.
(gdb) r
Starting program: /root/sandbox/crash/day3/intermediate/myprog
3628800

Breakpoint 1, iterative_factorial (n=10) at src/lib.c:31
31          int result = 0;
(gdb)
```

Debugging: *`gdb`*

- Since the return value of **`iterative_factorial`** is 0, we need to see how the factorial is calculated in each step
- We can **print** out a variable from the source code as we step through it

Debugging: *gdb*

- We can use **`n`** to step, and seems like **`result`** is always 0
- Must mean that we initialized **`result`** poorly --- should be 1

```
Breakpoint 1, iterative_factorial (n=10) at src/lib.c:31
31          int result = 0;
(gdb) n
32          for (int i = 2 ; i <= n ; i++)
(gdb) n
34              result *= i;
(gdb) n
32          for (int i = 2 ; i <= n ; i++)
(gdb) print result
$2 = 0
```

Debugging: *gdb*

The fix!

```c
/*
 * Iterative factorial calculation
 */
int iterative_factorial(int n)
{
    /*
     * Iteratively compute using a loop
     */
    int result = 1;
    for (int i = 2 ; i <= n ; i++)
    {
        result *= i;
    }


    return result;
}
```

Debugging: `valgrind`

- Another common tool to debug and profile is **`valgrind`** --- which specializes in **checking memory usage**
- Very useful to understand:
    - Stack overflows
    - Memory corruption
    - Memory leaks
    - Null pointer dereferences, invalid writes/reads

Debugging: **`valgrind`**

- Run your executable (with args, flags, etc.) but place "**`valgrind`**" at the front of the command (**`valgrind -v`** for verbose checks)

```
$ valgrind ./myprog
==2305== Memcheck, a memory error detector
==2305== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2305== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==2305== Command: ./myprog
==2305==
```

Debugging: `valgrind`

- If we revert changes back to the buggy library and run under **`valgrind`**, we can see a detailed report of the **stack overflow**:

```
==2361== Stack overflow in thread #1: can't grow stack to 0x1ffe801000
==2361==
==2361== Process terminating with default action of signal 11 (SIGSEGV)
==2361==  Access not within mapped region at address 0x1FFE801FF8
==2361== Stack overflow in thread #1: can't grow stack to 0x1ffe801000
==2361==    at 0x1086CD: recursive_factorial (lib.c:19)
==2361==  If you believe this happened as a result of a stack
==2361==  overflow in your program's main thread (unlikely but
==2361==  possible), you can try to increase the size of the
==2361==  main thread stack using the --main-stacksize= flag.
==2361==  The main thread stack size used in this run was 8388608.
==2361== Stack overflow in thread #1: can't grow stack to 0x1ffe801000
==2361==
==2361== Process terminating with default action of signal 11 (SIGSEGV)
==2361==  Access not within mapped region at address 0x1FFE801FF0
==2361== Stack overflow in thread #1: can't grow stack to 0x1ffe801000
==2361==    at 0x4A2A650: _vgnU_freeres (in /usr/lib/valgrind/vgpreload_core-amd64-linux.so)
==2361==  If you believe this happened as a result of a stack
==2361==  overflow in your program's main thread (unlikely but
==2361==  possible), you can try to increase the size of the
==2361==  main thread stack using the --main-stacksize= flag.
==2361==  The main thread stack size used in this run was 8388608.
```