

Debugging on Linux

Print Debugging

Most often, debugging is done via print...

...But that scales poorly

GDB

A Debugger (such as GDB) can help here

What is GDB?

GDB is a tool to inspect your program while it is running. It allows:

- ▶ to stop program execution at any point
- ▶ step through the program
- ▶ print the value of variables
- ▶ modify variables

Usage

Before running a program in GDB, it needs to be compiled with debug symbol.

- ▶ Available compiler flags are `-g`, `-g3`, `-ggdb3`
- ▶ `-ggdb3` gives the most debug information

The program can then be run under gdb: `gdb ./a.out`

An Example...

```
1  #include<stdio.h>
2  #include<stdbool.h>
3  #include<tgmath.h>
4
5  bool is_prime(int number) { /* snip */ }
6
7  int
8  main(int argc, char** argv) {
9      int number;
10     scanf("%d", &number);
11     if (is_prime(number))
12         printf("%d: prime\n", number);
13     else
14         printf("%d: not prime\n", number);
15     return 0;
16 }
```

An Example...

```
$ gcc -ggdb3 simple.c -lm
$ gdb ./a.out
GNU gdb (GDB) 8.3.1
Copyright (C) 2019 Free Software Foundation, Inc.
...snip...
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./a.out...
(gdb)
```

The Start Command

`start` starts program execution and stops it at the beginning of `main`

```
(gdb) start
Temporary breakpoint 1 at 0x11c7: file simple.c, line 16.
Starting program: /home/dcm/misc/debugging-on-linux/code/gdb-examples/a.out

Temporary breakpoint 1, main (argc=1, argv=0x7fffffffdf18) at simple.c:16
16 main(int argc, char** argv) {
(gdb)
```

The Step Command

`step` executes the next line. If there's a function, it will step into it.

```
(gdb) start
Temporary breakpoint 1 at 0x11c7: file simple.c, line 16.
Starting program: /home/dcm/misc/debugging-on-linux/code/gdb-examples/a.out

Temporary breakpoint 1, main (argc=1, argv=0x7fffffffdf18) at simple.c:16
16  main(int argc, char** argv) {
(gdb) step
18  scanf("%d", &number);
(gdb)
```

The Next Command

`next` executes the next line. If there's a function, **DO NOT** step into it.

```
(gdb) start
Temporary breakpoint 1 at 0x11c7: file simple.c, line 16.
Starting program: /home/dcm/misc/debugging-on-linux/code/gdb-examples/a.out

Temporary breakpoint 1, main (argc=1, argv=0x7fffffffdf18) at simple.c:16
16  main(int argc, char** argv) {
(gdb) step
18  scanf("%d", &number);
(gdb) next
1234567
19  if (is_prime(number))
(gdb)
```

The Print Command

`print` prints the contents of variables. It also allows calling functions.

```
(gdb) step
18     scanf("%d", &number);
(gdb) next
1234567
19     if (is_prime(number))
(gdb) print number
$1 = 1234567
(gdb) print is_prime(number)
$2 = false
(gdb)
```

The List Command

`list` lists the 10 lines of source code surrounding the current one

```
(gdb) next
1234567
19     if (is_prime(number))
(gdb) list
14
15     int
16     main(int argc, char** argv) {
17         int number;
18         scanf("%d", &number);
19         if (is_prime(number))
20             printf("%d: prime\n", number);
21         else
22             printf("%d: not prime\n", number);
23         return 0;
(gdb)
```

The Break Command

`break` adds a break point, which will stop program execution when reached

```
(gdb) list
14
15  int
16  main(int argc, char** argv) {
17      int number;
18      scanf("%d", &number);
19      if (is_prime(number))
20          printf("%d: prime\n", number);
21      else
22          printf("%d: not prime\n", number);
23      return 0;
(gdb) break 22
Breakpoint 2 at 0x55555555214: file simple.c, line 22.
(gdb)
```

The Continue Command

`continue` continues program execution until either a breakpoint is hit or the program exits

```
(gdb) break 22
Breakpoint 2 at 0x555555555214: file simple.c, line 22.
(gdb) continue
Continuing.

Breakpoint 2, main (argc=1, argv=0x7fffffffdf08) at simple.c:22
22     printf("%d: not prime\n", number);
(gdb)
```

The Quit Command

quit quits GDB, terminating the program

```
(gdb) continue
Continuing.

Breakpoint 2, main (argc=1, argv=0x7fffffffdf08) at simple.c:22
22     printf("%d: not prime\n", number);
(gdb) quit
A debugging session is active.

    Inferior 1 [process 7431] will be killed.

Quit anyway? (y or n) y
$
```

Text User Interface

GDB can also be run with a TUI: `gdb -tui ./a.out`

Text User Interface

```
simple.c
1
2 #include<stdio.h>
3 #include<stdbool.h>
4 #include<tgmath.h>
5
6 bool
7 is_prime(int number) {
8     int max = ((int) sqrt((double) number)) + 1;
9     for (int i = 2; i < max; i++)
10         if (!(number % i))
11             return false;
12     return true;
13 }
14
15 int
16 main(int argc, char** argv) {
17     int number;
18     scanf("%d", &number);
19     if (is_prime(number))
20         printf("%d: prime\n", number);
21     else
22         printf("%d: not prime\n", number);
23     return 0;
24 }
```

native process 23777 In: main L19 PC: 0x555555551ee
Type 'apropos word' to search for commands related to 'word'.
Reading symbols from ./a.out...
(gdb) start
Temporary breakpoint 1 at 0x11c7: file simple.c, line 16.
Starting program: /home/dca/misc/debugging-on-linux/code/gdb-examples/a.out
Temporary breakpoint 1, main (argc=1, argv=0x7fffffff18) at simple.c:15
(gdb) start
The program being debugged has been started already.
Start it from the beginning? (y or n) n
Program not restarted.
(gdb) n
(gdb) n
(gdb) █

Quick Warning

The following is slide code and output, therefore all source code and program output is heavily shortend.

Valgrind

- ▶ Collection of usefull debugging tools
 - ▶ Memcheck
 - ▶ Helgrind
- ▶ Works on any executable
 - ▶ Doesn't require recompilation
 - ▶ Output gets more readable with debug symbols!

How to use Valgrind?

- ▶ (optionally) Recompile the program with debug symbols
- ▶ Choose a tool
- ▶ Run the program under valgrind: `valgrind --tool=$TOOL ./a.out`

Memcheck

- ▶ finds common memory errors
 - ▶ Use after free
 - ▶ Use of uninitialised values
 - ▶ Memory leaks
 - ▶ ...and many more
- ▶ Usage: `valgrind [--leak-check=full] ./a.out`

Use After Free

Use After Free Usage of a pointer after it has been `free()`'d

Use After Free

```
1  #include <stdlib.h>
2
3  int
4  main(int argc, char **argv) {
5      int *ip;
6      ip = malloc(sizeof(int));
7      free(ip);
8      *ip = 3;
9      return 0;
10 }
```

Use After Free

```
==4474== Memcheck, a memory error detector
..snip..
==4474== Invalid write of size 4
==4474==    at 0x109176: main (uaf.c:8)
==4474==    Address 0x4a86040 is 0 bytes inside a block of size 4 free'd
==4474==    at 0x48399AB: free (vg_replace_malloc.c:540)
==4474==    by 0x109171: main (uaf.c:7)
==4474==    Block was alloc'd at
==4474==    at 0x483877F: malloc (vg_replace_malloc.c:309)
==4474==    by 0x109161: main (uaf.c:6)
==4474==
..snip..
```

Uninitialized Values

Uninitialized Values Use of a variable or memory before it has been initialized

Uninitialized Values

```
1  #include <stdio.h>
2
3  int
4  main(int argc, char **argv) {
5      int i;
6      if (i)
7          printf("Hui\n");
8      else
9          printf("Pfui\n");
10     return 0;
11 }
```

Uninitialized Values

```
==7096== Memcheck, a memory error detector
..snip..
==7096== Conditional jump or move depends on uninitialised value(s)
==7096==    at 0x10914C: main (ui.c:6)
==7096==
Pfui
==7096==
..snip..
```

Memory Leaks

Memory Leak A piece of memory is `malloc()`'d, but never `free()`'d

Memory Leaks

```
1 #include <stdlib.h>
2
3 int
4 main(int argc, char **argv) {
5     int *p;
6     p = malloc(sizeof(int));
7     return 0;
8 }
```

Memory Leaks

```
==7219== Memcheck, a memory error detector
..snip..
==7219== HEAP SUMMARY:
==7219==      in use at exit: 4 bytes in 1 blocks
==7219==    total heap usage: 1 allocs, 0 frees, 4 bytes allocated
==7219==
==7219== 4 bytes in 1 blocks are definitely lost in loss record 1 of 1
==7219==    at 0x483877F: malloc (vg_replace_malloc.c:309)
==7219==    by 0x109151: main (ms.c:6)
==7219==
==7219== LEAK SUMMARY:
==7219==    definitely lost: 4 bytes in 1 blocks
==7219==    indirectly lost: 0 bytes in 0 blocks
==7219==    possibly lost: 0 bytes in 0 blocks
==7219==    still reachable: 0 bytes in 0 blocks
==7219==    suppressed: 0 bytes in 0 blocks
..snip..
```

Helgrind

- ▶ finds common threading problems
 - ▶ Potential lock order inversions
 - ▶ Race conditions
- ▶ Has problems with lock-free data structures/algorithms
- ▶ Usage: `valgrind --tool=helgrind ./a.out`

Race Condition

Race Condition Several Threads try to modify the same variable at the same time yielding unexpected results

Race Condition

```
1  #include <pthread.h>
2
3  volatile int inc;
4
5  void *
6  thread(void *arg __attribute__((unused))) {
7      for (int i = 0; i < 65536; i++)
8          inc++;
9      return NULL;
10 }
11
12 int
13 main(int argc, char **argv) {
14     pthread_t t1, t2;
15     pthread_create(&t1, NULL, thread, NULL);
16     pthread_create(&t2, NULL, thread, NULL);
17     /* snip */
18 }
```

Race Condition

```
==7454== Helgrind, a thread error detector
..snip..
==7454== Possible data race during read of size 4 at 0x10C03C by thread #3
==7454== Locks held: none
==7454==   at 0x10915A: thread (rc.c:8)
==7454==   by 0x483F876: mythread_wrapper (hg_intercepts.c:389)
==7454==   by 0x48A84CE: start_thread (in /usr/lib/libpthread-2.30.so)
==7454==   by 0x49C02D2: clone (in /usr/lib/libc-2.30.so)
==7454==
==7454== This conflicts with a previous write of size 4 by thread #2
==7454== Locks held: none
==7454==   at 0x109163: thread (rc.c:8)
==7454==   by 0x483F876: mythread_wrapper (hg_intercepts.c:389)
==7454==   by 0x48A84CE: start_thread (in /usr/lib/libpthread-2.30.so)
==7454==   by 0x49C02D2: clone (in /usr/lib/libc-2.30.so)
==7454== Address 0x10c03c is 0 bytes inside data symbol "inc"
..snip..
```

Lock Order Inversion

Lock Order Inversion Two threads try to acquire two locks in opposite orders

Lock Order Inversion

```
6 void *
7 thread1(void *arg __attribute__((unused))) {
8     pthread_mutex_lock(&lock1);
9     pthread_mutex_lock(&lock2); //Lock-Order-Inversion between here...
10    pthread_mutex_unlock(&lock2);
11    pthread_mutex_unlock(&lock1);
12    return NULL;
13 }
14
15 void *
16 thread2(void *arg __attribute__((unused))) {
17    pthread_mutex_lock(&lock2);
18    pthread_mutex_lock(&lock1); // ... and here!
19    pthread_mutex_unlock(&lock1);
20    pthread_mutex_unlock(&lock2);
21    return NULL;
22 }
```

Lock Order Inversion

```
==8617== Thread #3: lock order "0x10C080 before 0x10C0C0" violated
==8617==
==8617== Observed (incorrect) order is: acquisition of lock at 0x10C0C0
==8617==   at 0x483CC3F: mutex_lock_WRK (hg_intercepts.c:909)
==8617==   by 0x1091C3: thread2 (loi.c:17)
..snip..
==8617==   followed by a later acquisition of lock at 0x10C080
==8617==   at 0x483CC3F: mutex_lock_WRK (hg_intercepts.c:909)
==8617==   by 0x1091CF: thread2 (loi.c:18)
..snip..
==8617== Required order was established by acquisition of lock at 0x10C080
==8617==   at 0x483CC3F: mutex_lock_WRK (hg_intercepts.c:909)
==8617==   by 0x109180: thread1 (loi.c:8)
..snip..
==8617==   followed by a later acquisition of lock at 0x10C0C0
==8617==   at 0x483CC3F: mutex_lock_WRK (hg_intercepts.c:909)
==8617==   by 0x10918C: thread1 (loi.c:9)
```

Sanitizer

- ▶ Error checking compiled into binary
 - ▶ Thread Sanitizer
 - ▶ Address Sanitizer
 - ▶ Undefined Behaviour Sanitizer
- ▶ needs recompilation!
- ▶ Largely covers the same things as valgrind
- ▶ Available for both GCC and LLVM

How to use Sanitizer?

- ▶ Recompile the program with the Sanitizer enabled
 - ▶ Needs to be enabled everywhere
 - ▶ Usage: `gcc -ggdb3 -fsanitize=$SANITIZER`
- ▶ run the resulting binary

Sanitizer - Address

- ▶ Checks for all kinds of address violations
- ▶ Also includes leak detector
- ▶ Similar to valgrind's Memcheck
 - ▶ Can detect some bugs memcheck can't
 - ▶ Can't detect some bugs memcheck can
 - ▶ Provides more information
- ▶ Usage `gcc -fsanitize=address`

Use After Free

```
1  #include <stdlib.h>
2
3  int
4  main(int argc, char **argv) {
5      int *ip;
6      ip = malloc(sizeof(int));
7      free(ip);
8      *ip = 3;
9      return 0;
10 }
```

Use After Free

```

==4299==ERROR: AddressSanitizer: heap-use-after-free on address 0x602000000010
    at pc 0x5630686c61d9 bp 0x7ffda2c5a820 sp 0x7ffda2c5a810
WRITE of size 4 at 0x602000000010 thread T0
    #0 0x5630686c61d8 in main /home/dcm/misc/debugging-on-linux/code/bugtype/
        uaf.c:8

0x602000000010 is located 0 bytes inside of 4-byte region [0x602000000010,0
    x602000000014)
freed by thread T0 here:
    #1 0x5630686c61a1 in main /home/dcm/misc/debugging-on-linux/code/bugtype/
        uaf.c:7

previously allocated by thread T0 here:
    #1 0x5630686c6191 in main /home/dcm/misc/debugging-on-linux/code/bugtype/
        uaf.c:6

```

Use After Free

```
SUMMARY: AddressSanitizer: heap-use-after-free /home/dcm/misc/debugging-on-  
linux/code/bugtype/uaf.c:8 in main
```

```
Shadow bytes around the buggy address:
```

```
0x0c047fff7fb0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
0x0c047fff7fc0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
0x0c047fff7fd0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
0x0c047fff7fe0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
0x0c047fff7ff0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
=>0x0c047fff8000: fa fa[fd]fa fa  
0x0c047fff8010: fa  
0x0c047fff8020: fa  
0x0c047fff8030: fa  
0x0c047fff8040: fa  
0x0c047fff8050: fa fa
```

Memory Leak

```
1 #include <stdlib.h>
2
3 int
4 main(int argc, char **argv) {
5     int *p;
6     p = malloc(sizeof(int));
7     return 0;
8 }
```

Memory Leaks

```
=====
==4545==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 4 byte(s) in 1 object(s) allocated from:
    #0 0x7f9b5971faca in __interceptor_malloc /build/gcc/src/gcc/libsanitizer/
        asan/asan_malloc_linux.cc:144
    #1 0x55d960688171 in main /home/dcm/misc/debugging-on-linux/code/bugtype/
        ms.c:6
    #2 0x7f9b5946f152 in __libc_start_main (/usr/lib/libc.so.6+0x27152)

SUMMARY: AddressSanitizer: 4 byte(s) leaked in 1 allocation(s).
```

Buffer Overflow

Buffer Overflow Reading or writing memory outside of an allocated buffer

Buffer Overflow

```
1  int
2  main(int argc, char **argv) {
3      int arr[10];
4      arr[11] = 1;
5      return 0;
6  }
```

Buffer Overflow

```
==4464==ERROR: AddressSanitizer: stack-buffer-overflow on address 0
    x7ffe66905e4c at pc 0x564ce591325c bp 0x7ffe66905dd0 sp 0x7ffe66905dc0
WRITE of size 4 at 0x7ffe66905e4c thread T0
    #0 0x564ce591325b in main /home/dcm/misc/debugging-on-linux/code/bugtype/
        bo.c:5
    #1 0x7f0a22d6b152 in __libc_start_main (/usr/lib/libc.so.6+0x27152)
    #2 0x564ce59130ad in _start (/home/dcm/misc/debugging-on-linux/code/
        bugtype/a.out+0x10ad)

Address 0x7ffe66905e4c is located in stack of thread T0 at offset 92 in frame
    #0 0x564ce5913188 in main /home/dcm/misc/debugging-on-linux/code/bugtype/
        bo.c:3

This frame has 1 object(s):
    [48, 88) 'arr' (line 4) <== Memory access at offset 92 overflows this
        variable
```

Sanitizer - Thread

- ▶ Detects common threading problems
- ▶ Similar to valgrind's Helgrind
- ▶ Always run programs with `TSAN_OPTIONS=second_deadlock_stack=1`
 - ▶ `$ TSAN_OPTIONS=second_deadlock_stack=1 ./a.out`
- ▶ Usage `gcc -fsanitize=thread`

Lock Order Inversion

```
6 void *
7 thread1(void *arg __attribute__((unused))) {
8     pthread_mutex_lock(&lock1);
9     pthread_mutex_lock(&lock2); //Lock-Order-Inversion between here...
10    pthread_mutex_unlock(&lock2);
11    pthread_mutex_unlock(&lock1);
12    return NULL;
13 }
14
15 void *
16 thread2(void *arg __attribute__((unused))) {
17    pthread_mutex_lock(&lock2);
18    pthread_mutex_lock(&lock1); // ... and here!
19    pthread_mutex_unlock(&lock1);
20    pthread_mutex_unlock(&lock2);
21    return NULL;
22 }
```

Lock Order Inversion

```
WARNING: ThreadSanitizer: lock-order-inversion (potential deadlock) (pid=7033)
Cycle in lock order graph: M9 (0x55fc4d1620a0) => M10 (0x55fc4d1620e0) => M9
```

```
Mutex M10 acquired here while holding mutex M9 in thread T1:
```

```
#1 thread1 /home/dcm/misc/debugging-on-linux/code/bugtype/loi.c:9
```

```
Mutex M9 previously acquired by the same thread here:
```

```
#1 thread1 /home/dcm/misc/debugging-on-linux/code/bugtype/loi.c:8
```

```
Mutex M9 acquired here while holding mutex M10 in thread T2:
```

```
#1 thread2 /home/dcm/misc/debugging-on-linux/code/bugtype/loi.c:18
```

```
Mutex M10 previously acquired by the same thread here:
```

```
#1 thread2 /home/dcm/misc/debugging-on-linux/code/bugtype/loi.c:17
```

Race Condition

```
1  #include <pthread.h>
2
3  volatile int inc;
4
5  void *
6  thread(void *arg __attribute__((unused))) {
7      for (int i = 0; i < 65536; i++)
8          inc++;
9      return NULL;
10 }
11
12 int
13 main(int argc, char **argv) {
14     pthread_t t1, t2;
15     pthread_create(&t1, NULL, thread, NULL);
16     pthread_create(&t2, NULL, thread, NULL);
17     /* snip */
18 }
```

Race condition

```
WARNING: ThreadSanitizer: data race (pid=11607)
  Read of size 4 at 0x559e8e885074 by thread T2:
    #0 thread /home/dcm/misc/debugging-on-linux/code/bugtype/rc.c:8

  Previous write of size 4 at 0x559e8e885074 by thread T1:
    #0 thread /home/dcm/misc/debugging-on-linux/code/bugtype/rc.c:8

  Location is global 'inc' of size 4 at 0x559e8e885074 (a.out+0x000000004074)

  Thread T2 (tid=11610, running) created by main thread at:
    #1 main /home/dcm/misc/debugging-on-linux/code/bugtype/rc.c:16

  Thread T1 (tid=11609, finished) created by main thread at:
    #1 main /home/dcm/misc/debugging-on-linux/code/bugtype/rc.c:15
```

Sanitizer - Undefined Behaviour

- ▶ Detects undefined behaviour in C
- ▶ Doesn't cover undefined behaviour related to memory
- ▶ Usage `gcc -fsanitize=undefined`

Undefined Bitshifts

Undefined Bitshift Bitshifts larger than the width of the integer type

Undefined Bitshifts

```
1  int
2  main(int argc, char **argv) {
3      int i = 32;
4      int j = 0xCAFFEE;
5      j = j << i;
6      return 0;
7  }
```

Undefined Bitshifts

```
ud.c:5:9: runtime error: shift exponent 32 is too large for 32-bit type 'int'
```

Strace

- ▶ Shows all system calls made by a program
- ▶ Allows debugging of everything File related
 - ▶ Which files are used by a program?
 - ▶ Where does a program search for files?
 - ▶ What does a program do with a file?
- ▶ Usage: `strace ./a.out`

Some important Linux Systemcalls

System Call	Function	Return Value
<code>openat</code>	Opens a file	A filedescriptor for the file
<code>close</code>	Close a file	0
<code>read</code>	Reads from a file	Number of bytes read
<code>write</code>	Writes to a files	Number of bytes written
<code>mmap</code>	Maps a file into memory	Pointer to memory
<code>socket</code>	Creates a new socket	A filedescriptor for the socket
<code>clone</code>	Spawn new thread/process	ID of the newly create Process/Thread
<code>execve</code>	Load new Program	0
<code>exit_group</code>	Exit	N/A

An Example...

```
1 #include <stdio.h>
2
3 int
4 main(int argc, char **argv) {
5     printf("Hello World!");
6     return 0;
7 }
```

An Example...

```
$ strace ./a.out > /dev/null
execve("./a.out", ["./a.out"], 0x7ffee24755b0 /* 47 vars */) = 0
brk(NULL)                                = 0x55fd00b9c000
--snip--
fstat(1, {st_mode=S_IFCHR|0666, st_rdev=makedev(0x1, 0x3), ...}) = 0
ioctl(1, TCGETS, 0x7fff3fb75ac0)         = -1 ENOTTY (Inappropriate ioctl for
    device)
brk(NULL)                                = 0x55fd00b9c000
brk(0x55fd00bbd000)                       = 0x55fd00bbd000
write(1, "Hello World!", 12)             = 12
exit_group(0)                             = ?
+++ exited with 0 +++
```

Filtering output

- ▶ strace allows for filtering output by syscall
- ▶ Syntax: `strace -e $ARGUMENT`
 - ▶ ARGUMENT should be a comma seperated list of systemcalls
 - ▶ if a syscall is prefixed by `!`, it means all systemcalls but it

An Example...

```
$ strace -e write ./a.out  
write(1, "Hello World!", 12Hello World!)      = 12  
+++ exited with 0 +++
```

Files associated with File descriptors

- ▶ Enabled by `-y`
- ▶ Will show the file associated with each file descriptor
- ▶ `/dev/pts/*` is standart output

An Example

```
1  #include <stdio.h>
2
3  int
4  main(int argc, char **argv) {
5      FILE *f = fopen("/tmp/test", "w+");
6      fprintf(f, "Hello\n");
7      fclose(f);
8      printf("Done!\n");
9      return 0;
10 }
```

An Example

```
$ strace -y -e openat,write,close ./a.out >/dev/null
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3</etc/ld.so.cache>
close(3</etc/ld.so.cache>) = 0
openat(AT_FDCWD, "/usr/lib/libc.so.6", O_RDONLY|O_CLOEXEC) = 3</usr/lib/libc
-2.30.so>
close(3</usr/lib/libc-2.30.so>) = 0
openat(AT_FDCWD, "/tmp/test", O_RDWR|O_CREAT|O_TRUNC, 0666) = 3</tmp/test>
write(3</tmp/test>, "Hello\n", 6) = 6
close(3</tmp/test>) = 0
write(1</dev/null>, "Done!\n", 6) = 6
+++ exited with 0 +++
```

Thank you for your attention!

Question?