CIA Lab Assignment: Computer Architecture

(version 17.8, 2017/08/30 09:04:16)

Karst Koymans Arno Bakker* Mick Pouw†

Feedback deadline:
September 22, 2017 10:00 CET

Abstract

Exercises to practise basic interaction with the development tools in the Linux operating system and the GNU C compiler.

1 Basic OS interaction

In this first part of the lab exercises we look at the different formats for executables, We use the "`strace`" command to identify system calls in binaries and we look at (parts of) the ELF file format.

1.1 Binaries and scripts

1. Find examples of binaries and five different interpreter scripts (Hint: use the file command in "`/usr/bin`")

1.2 Tracing binaries

Read the "`strace`" man page.

2. Use "`strace`" to find what other system call besides "`stat`" "`zsh`" uses before executing an "`execve`" system call? (Hint: use the "`-c`" option of "`zsh`")

1.3 Stracing strace

3. (Bonus)
   Run "`strace /bin/pwd`" and save the result.

*Arno.Bakker@os3.nl
†Mick.Pouw@os3.nl

Also run "`strace strace /bin/pwd`" and save the result. How are these outputs related? Explain the "2" in "`write(2, ...`".

1.4 ELF format

Read the "`readelf`" man page.
Make sure you make your terminal as wide a possible.

4. Execute "`readelf -Wh <<your favorite ELF binary>>`".
   Match the results for the ELF header with the information on ELF's Wikipedia page. Is this a definitive source for the ELF format? If not, what is?

5. (Bonus) Also inspect the section and program headers, using the command "`readelf -WlS <<your favorite ELF binary>>`".
   (a) Explain which of the two header types (section or program) is used in which context (loader/runtime or linker/relocation)?
   (b) Explain the use of each of the following sections: `.text, .data, .rodata and .bss. (Hint: Consider Type and Flg)`
   `(c) What does the program header with Type INTERP contain?`

6. (Bonus)
   Find the loaded libraries of your shell by executing "`lsof -p $$`".
   Also look at the memory image of your shell by executing "`cat /proc/$$/maps`".
   Why is for instance the libc library mapped into memory multiple times?

2 The gcc compiler

In the lectures we discussed the "Hello OS3!"-program:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  printf("Hello OS3!\n");
  return 2017;
  /* What is the actual exit status? Why? */
}
```

7. Make your own version of the "Hello OS3!"-program, by just only varying the string. Can you come up with a creative text?
   Find all four phases in the compilation by running "`gcc -v -o hello hello.c`".

Now we are going to do all the stages separately in order to inspect the different intermediate steps.

8. Run "`gcc -E hello.c -o hello.i`", producing the file "`hello.i`".
   What are the numbers at the end of the lines in "`hello.i`" supposed to mean?

9. Run "`gcc -S hello.i`", producing the file "`hello.s`"
   Inspect "`hello.s`". Where has "`printf`" gone?
   Remove the newline at the end of the string in "`hello.c`" and look again at the resulting "`hello.s`". Explain.

10. Run "`gcc -c hello.s`", producing the file "`hello.o`".
    Use "`objdump -rd hello.o`" to inspect this relocatable object file.
    Explain why this piece of machine code is (not yet) executable.

11. Run "`gcc -o hello hello.o`", producing the file "`hello`".
    Use "`ltrace ./hello`" to see what library calls your program makes.
    Run "`ltrace -S`" to also see the system calls.
    What is the actual exit code of your program? Can you explain this?
    (Hint: try a "smaller version" of 2017)

Remark: You can generate all the intermediate files at once by using the "`-save-temps`" option of "`gcc`"

3 Inline assembly

In order to experiment with assembly language, it is handy to use a template that enables you to embed your assembly program inside a C program for easy comkmunication of results.

For that purpose we have prepared a template file, called "`template.c`". The macro "`OS3_ASM`" should be filled in with your own assembly language code. The C variables rax, rbx, rcx and rdx are copied into the corresponding registers before the invocation of the assembly language code in "`OS3_ASM`" and back out of the registers into the C variables after the execution of the assembly language code. If you want to use other registers, you have to save them on the stack before use and pop them from the stack afterwards.

```
#define OS3_ASM \
    "shrq $1, %%rax;" \
    "sarq $1, %%rbx;" \
    "shlq $1, %%rcx;" \
    "salq $1, %%rdx;"

#include <inttypes.h>
#include <stdio.h>

int main(void) {

uint64_t rax = 0xFEDCBA9876543210;
uint64_t rbx = 0x76543210;
uint64_t rcx = 0x3210;
uint64_t rdx = 0x10;

printf("Before assembly code...\n");
printf("rax: %016" PRIx64 "\n", rax);
printf("rbx: %016" PRIx64 "\n", rbx);
printf("rcx: %016" PRIx64 "\n", rcx);
printf("rdx: %016" PRIx64 "\n", rdx);
printf("\n");

asm volatile (
    OS3_ASM
    : "+a" (rax), "+b" (rbx), "+c" (rcx), "+d" (rdx)
);

printf("After assembly code...\n");
printf("rax: %016" PRIx64 "\n", rax);
printf("rbx: %016" PRIx64 "\n", rbx);
printf("rcx: %016" PRIx64 "\n", rcx);
printf("rdx: %016" PRIx64 "\n", rdx);

}
```

12. Create assembly language (for OS3_ASM) for calculating the following formulas, where a, b, c and d are (unsigned) 64-bit integers.

    - $a+b$
    - bc
    - b^2-4ac
    - $d^4+d^3+d^2+d+1$ (BONUS: use only 3 multiplications)

13. (Bonus)
    Create assembly language (for OS3_ASM) which uses a simple for loop to calculate something of your choice, as long as it is not Fibonacci or factorial.

3.1 A strange program

The following is the source code for "`inspect.c`".

```
#include <stdio.h>

int main() {
  unsigned long a[4] = {0, 0, 0, 0};
  int i, j;
  char c;

  asm volatile(".byte 15;.byte 162"
               : "+a"(a[0]), "+b"(a[1]), "+d"(a[2]), "+c"(a[3])
  );

  for (i = 1; i < 4 ; i++) {
    for (j = 0; j < 4; j++) {
      c = a[i] >> (8 * j);
      if (c < 32) c = 32;
      if (c > 126) c = 126;
      putchar(c);
    }
  }

  printf("\n");

  return 42;
}
```

To find out what's going on in this program we will use the GNU debugger gdb. A handy cheatsheet for gdb can be downloaded from

`http://csapp.cs.cmu.edu/public/docs/gdbnotes-x86-64.pdf`.

14. Compile the program above with the "`-g`" flag to generate debugging information. Run the program.

    What is the purpose of these bytes (15, 162)? You could look at the object code with a disassembler, but let us run gdb on this program instead.

    Run "`gdb inspect`".
    Set a breakpoint for "`main`": (gdb) break main
    Start the program: (gdb) run
    Execute the next line of code: (gdb) next
    Iterate the last command (by just pressing enter/return) until gdb prints a line containing 15 and 162.
    Inspect the registers RAX-RDX: (gdb) info registers
    Look at the disassembly of the program: (gdb) disassemble
    Execute the next line of code: (gdb) next
    Inspect the registers RAX-RDX: (gdb) info registers
    Display the value of the counter i: (gdb) display i
    Display the value of the counter j: (gdb) display j
    Execute the next line of code: (gdb) next
    Step some more through the for loop and see i and j change
    Continue the program until the end: (gdb) continue

15. Why does the program exit with 052?

3.2 A puzzle for the diehards

Consider the program "`smc.s`":

```
.global _start

mysterybytes:

.byte 0x05, 0x0f, 0x00, 0x00, 0x00, 0x3c, 0xc0, 0xc7, 0x48, 0x6c, 0x33, 0x33, 0x74, 0

_start:

mov $16, %rdx
mov $mysterybytes, %rbx
mov $mystery+15, %rcx
back:
movb (%rbx), %al
movb %al, (%rcx)
inc %rbx
dec %rcx
dec %rdx
jz mystery
jmp back

mystery:

nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
```

16. (Extra bonus)

Compile/assemble the program without standard libs: "`gcc -nostdlib -o smc smc.s`"). Check that it generates a segmentation violation. What is going on?
Find a way to compile this program so that it runs "normally". Mystery question: what is its exit code?