Executing user mode programs that perform I/O on the x86 model

Soumava Ghosh
The University of Texas at Austin

Agenda

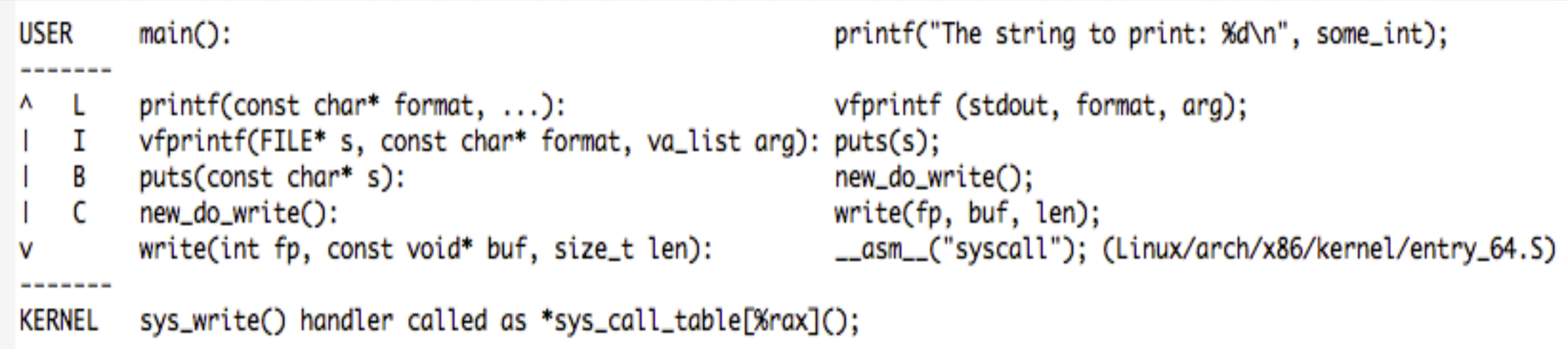
- Overview of programs that perform I/O
- Linking, loading and the x86 model
- Modifying programs to perform I/O on the x86 model
- Interpreting and loading binaries on the x86 model
- Demo

Programs performing I/O

- Most user mode programs are interactive.
- Simplest interactive programs perform console or file input/output (I/O).
- Generally achieved in C by invoking the functions printf, scanf, gets, puts, fread, fwrite etc.
- These are library functions provided by LibC – the C standard library.

Tracing I/O functions

- Let's say the user's code called the printf() function from main(). Here is what happens internally:



```
USER     main():                                      printf("The string to print: %d\n", some_int);
-------
^   L    printf(const char* format, ...):             vfprintf (stdout, format, arg);
|   I    vfprintf(FILE* s, const char* format, va_list arg): puts(s);
|   B    puts(const char* s):                         new_do_write();
|   C    new_do_write():                              write(fp, buf, len);
v        write(int fp, const void* buf, size_t len):  __asm__("syscall"); (Linux/arch/x86/kernel/entry_64.S)
-------
KERNEL   sys_write() handler called as *sys_call_table[%rax]();
```

- The above sequence of method calls ultimately culminates in the execution of the 'write' kernel routine.

Why do we need LibC?

- The C standard library provides a higher level of abstraction to the user than the system calls exposed by the OS.

- Ease of use:
  - writing/reading variable values at runtime
  - buffered I/O via LibC buffers results in better performance.

- Platform and architecture independent interface

Building the binary

- Programs compiled using GCC by default link with the C Standard library.

- Linking could be:
    - dynamic: system loader loads the libraries at desired addresses in the process' address space at runtime.
    - static: binary is built such that it contains the library code – self sufficient binary, but has a larger size depending on the libraries it links against. (-*static* flag)

- X86 model requires static binaries only. Loading libraries at runtime is not supported at present.

LibC linked binaries and the x86 model

- Upon static linking with LibC, an increase of ~900KB is generally observed.
- Simple I/O operations require too many machine instructions, would probably take hours to execute on the x86 model.
- Use of segmented registers in LibC machine code – not yet supported in the x86 model.
- Static compilation with LibC is not the way to go.
- Q: How to execute programs that perform I/O on the x86 model without using LibC?

Removing the LibC dependency

- For example, as shown earlier, printf invokes the write system call to display output – the user’s program could do the same directly without using printf.

- How to execute the write system call?
  - Cannot be called directly – it is a kernel mode routine.
  - User mode code requires to indicate to the OS that a change in privilege level is required.
  - Means to achieve this:
    - INT 80H: the historical assembly instruction to interrupt and invoke a system call.
    - SYSCALL/SYSRET or SYSENTER/SYSEXIT: Modern fast system call assembly instructions.

Re-writing the program

- The only parts of the program that need to be rewritten are the ones which invoke the LibC I/O functions.
- Printf/Scanf format strings: Need to be implemented. If not generic, something specific to the program is good too.
- The hard part: writing assembly code to invoke the appropriate system routines.
- LibC generates code for the _start() entry point for every executable. In the absence of LibC, a _start needs to be provided to execute the program on a real machine.

Inline assembly for system calls

System call signature:

```
size_written = write(file_desc, buffer,
                     num_bytes_to_write);
```

Inline Assembly equivalent:

```
asm volatile
(
  "mov $1, %%rax\n\t”     // System call number (__NR_write = 1, unistd.h)
  "mov $1, %%rdi\n\t"     // First parameter in RDI (stdout = 1)
  "mov %1, %%rsi\n\t"     // Second parameter in RSI (buffer)
  "mov %2, %%rdx\n\t"     // Third parameter in RDX (num_bytes_to_write)
  "syscall”
    : "=a"(size_written)  // Output (=) to be stored in size_written
    : "g"(buffer), "g"(num_bytes_to_write)
    : "%rdi", "%rsi", "%rdx", "%rcx", "%r11”
);
```

The _start function

- Easy to write _start if there are no command line arguments to the program.

```
void _start() {
   main();
   asm (
     "mov $60, %rax;”        // The ‘exit’ system call
     "xor %rdi, %rdi;”       // The parameter (status) set to 0
     "syscall");
}
```

- If command line arguments are present, some stack pointer math is required to pop them from the correct location.

Putting it all together

- Replace printf with code constructing the string followed by assembly code calling the write system call.
- Replace scanf with assembly code calling the read system call followed by code parsing the input.
- Add the _start entry point to the program.
- Compile with the '*-nostdlib*' flag to prevent linking with LibC.

Loading the binary on the x86 model

- The SDLF (Simple Dumb Loader Format) reader has been recently developed for the Darwin and Linux platforms.
- These loaders interpret the binary as per the standard formats (mach-o for Darwin and ELF for Linux) and write the bytes to the appropriate memory locations in the x86 stobj.
- We still require to consult with ObjDump (a tool that produces the machine code dump of an object file) to decide the halt address for the x86 model.

SDLF Usage

- Path to books:

```
x86/x86-byte-mem/tools/model-validation/cosim/sdlf/elf
x86/x86-byte-mem/tools/model-validation/cosim/sdlf/mach-o
```

- Binary interpretation functions:

```
(X86ISA::file-read <file_name> |sdlf| |state|)
(X86ISA::elf-file-read <file_name> |elf| |state|)
```

- Section loading functions:

```
(X86ISA::load-text-section |{sdlf, elf}| |x86|)
(x86ISA::load-data-section |{sdlf, elf}| |x86|)
(x86ISA::load-rodata-section |elf| |x86|)
```

Demo

- Modification of the Micro SAT solver to execute on the x86 model.

- Reads a file test.cnf (of a particular format) and writes ‘Satisfiable’ or ‘Not Satisfiable’ to the command line as an end result of the execution.

- LibC functions replaced: fopen, fclose, printf, scanf.