Introduction to UNIX assembly programming

Contents

1 Introduction **1**

1.1 Legal blurb . . . 1

1.2 Obtaining this document . . . 1

1.3 Tools you need . . . 1

2 Hello, world! **1**

2.1 System calls . . . 1

2.2 Program layout . . . 2

2.3 Linux . . . 2

2.4 FreeBSD . . . 3

2.5 BeOS . . . 4

2.6 Building an executable . . . 4

3 References **5**

A History **5**

B Acknowledgements **5**

C Endorsements **5**

Abstract

This document is intended to be a tutorial, showing how to write a simple assembly program in several UNIX operating systems on the IA-32 (i386) platform. Included material may or may not be applicable to other hardware and/or software platforms.

This document explains program layout, system call convention, and the build process.

It accompanies the Linux Assembly HOWTO, which may also be of interest, though it is more Linux specific.

1 Introduction

1.1 Legal blurb

1.2 Obtaining this document

The latest version of this document is available from http://asm.sourceforge.net/intro.html If you are reading a few-months-old copy, please check the url above for a new version.

1.3 Tools you need

You will need several tools to play with programs included in this tutorial.

First of all you need the assembler (compiler). As a rule modern UNIX distributions include **as** (or **gas**), but all of the examples here use another assembler -- **nasm** (Netwide Assembler). It comes with full source code, and you can download it from the nasm page, or install it from the ports (or package) system. Compile it, or try to find precompiled binary for your OS; note that several distributions (at least Linux ones) already have **nasm**, check first.

Second, you need a linker -- **ld**, since assembler produces only object code. All distributions with the compilation tools installed will have **ld**.

If you're going to dig in, you should also install include files for your OS, and if possible, kernel source.

Now you should be ready to start, welcome..

2 Hello, world!

Now we will write our program, the old classic "Hello, world" (`hello.asm`). You can download its source and binaries here But before you do, let me explain several basics.

2.1 System calls

Unless a program is just implementing some math algorithms in assembly, it will deal with such things as getting input, producing output, and exiting. For this, it will need to call on OS services. In fact, programming in assembly language is quite the same in different OSes, unless OS services are touched.

There are two common ways of performing a system call in UNIX OS: through the C library (libc) wrapper, or directly.

Using or not using libc in assembly programming is more a question of taste/belief than something practical. Libc wrappers are made to protect programs from possible system call convention changes, and to provide POSIX compatible interface if the kernel lacks it for some call. However, the UNIX kernel is usually more-or-less POSIX compliant -- this means that the syntax of most libc "system calls" exactly matches the syntax of real kernel system calls (and vice versa). But the main drawback of throwing libc away is that one loses several functions that are not just syscall wrappers, like `printf()`, `malloc()` and similar.

This tutorial will show how to use *direct* kernel calls, since this is the fastest way to call kernel service; our code is not linked to any library, does not use ELF interpreter, it communicates with kernel directly.

Things that differ in different UNIX kernels are set of system calls and system call convention (however as they strive for POSIX compliance, there's a lot of common between them).

Note
(Former) DOS programmers might be wondering, "What is a system call?" If you ever wrote a DOS assembly program (and most IA-32 assembly programmers did), you may remember DOS services `int 0x21`, `int 0x25`, `int 0x26` etc.. These are analogous to the UNIX system call. However, the actual implementation is absolutely different, and system calls are not necessarily done via some interrupt. Also, quite often DOS programmers mix OS services with BIOS services like `int 0x10` or `int 0x16` and are very surprised when they fail to perform them in UNIX, since these are not OS services).

2.2 Program layout

As a rule, modern IA-32 UNIXes are 32bit (*grin*), run in protected mode, have a flat memory model, and use the ELF format for binaries.

A program can be divided into sections: `.text` for your code (read-only), `.data` for your data (read-write), `.bss` for uninitialized data (read-write); there can actually be a few other standard sections, as well as some user-defined sections, but there's rare need to use them and they are out of our interest here. A program must have at least `.text` section.

Ok, now we'll dive into OS specific details.

2.3 Linux

System calls in Linux are done through int 0x80. (actually there's a kernel patch allowing system calls to be done via the `syscall` (`sysenter`) instruction on newer CPUs, but this thing is still experimental).

Linux differs from the usual UNIX calling convention, and features a "fastcall" convention for system calls (it resembles DOS). The system function number is passed in `eax`, and arguments are passed through registers, not the stack. There can be up to six arguments in `ebx`, `ecx`, `edx`, `esi`, `edi`, `ebp` consequently. If there are more arguments, they are simply passed though the structure as first argument. The result is returned in `eax`, and the stack is not touched at all.

System call function numbers are in `sys/syscall.h`, but actually in `asm/unistd.h`. Documentation on the actual system calls is in section 2 of the manual pages some documentation is in the 2nd section of manual (for example to find info on `write` system call, issue the command **man 2 write**).

There have been several attempts to write an up-to-date documentation of the Linux system calls, examine URLs in the References section below.

So, our Linux program will look like:

```
section .text
    global _start     ;must be declared for linker (ld)

_start:         ;tell linker entry point

  mov edx,len ;message length
  mov ecx,msg ;message to write
  mov ebx,1 ;file descriptor (stdout)
  mov eax,4 ;system call number (sys_write)
  int 0x80  ;call kernel

  mov eax,1 ;system call number (sys_exit)
  int 0x80  ;call kernel

section .data

msg db  'Hello, world!',0xa ;our dear string
len equ $ - msg     ;length of our dear string
```

Kernel source references:

- arch/i386/kernel/entry.S

- include/asm-i386/unistd.h
- include/linux/sys.h

2.4 FreeBSD

Note
most of this section should apply to other BSD systems (OpenBSD, NetBSD) as well, however the source references may be different.

FreeBSD has the more "usual" calling convention, where the syscall number is in eax, and the parameters are on the stack (the first argument is pushed last). A system call should be done performed through a *function call* to a function containing `int 0x80` and `ret`, not just `int 0x80` itself (kernel expects to find extra 4 bytes on the stack before `int 0x80` is issued). The caller must clean up the stack after the call is complete. The result is returned as usual in `eax`.

There's an alternate way of using `call 7:0` gate instead of `int 0x80`. The end-result is the same, but the `call 7:0` method will increase the program size since you will also need to do an extra `push eax` before, and these two instructions occupy more bytes.

System call function numbers are listed in `sys/syscall.h`, and the documentation on the system calls is in section 2 of the man pages.

Ok, I think the source will explain this better:

```
section .text
    global _start     ;must be declared for linker (ld)

_syscall:
  int 0x80    ;system call
  ret

_start:       ;tell linker entry point

  push  dword len ;message length
  push  dword msg ;message to write
  push  dword 1   ;file descriptor (stdout)
  mov eax,0x4   ;system call number (sys_write)
  call  _syscall  ;call kernel

        ;the alternate way to call kernel:
        ;push eax
        ;call 7:0

  add esp,12    ;clean stack (3 arguments * 4)

  push  dword 0   ;exit code
  mov eax,0x1   ;system call number (sys_exit)
  call  _syscall  ;call kernel

        ;we do not return from sys_exit,
        ;there's no need to clean stack
section .data

msg db  "Hello, world!",0xa ;our dear string
len equ $ - msg     ;length of our dear string
```

Kernel source references:

- i386/i386/exception.s

- i386/i386/trap.c
- sys/syscall.h

2.5 BeOS

Note
if you are building nasm version 0.98 from the source on BeOS, you need to insert `#include "nasm.h"` into `float.h`, and `#include <stdio.h>` into `nasm.h`.

The BeOS kernel also uses the "usual" UNIX calling convention. The difference from the FreeBSD example is that you call `int 0x25`.

For information where to find system call function numbers and other interesting details, examine asmutils, especially the `os_beos.inc` file.

```
section .text
    global _start     ;must be declared for linker (ld)

_syscall:     ;system call
  int 0x25
  ret

_start:       ;tell linker entry point

  push  dword len ;message length
  push  dword msg ;message to write
  push  dword 1   ;file descriptor (stdout)
  mov eax,0x3   ;system call number (sys_write)
  call  _syscall  ;call kernel
  add esp,12    ;clean stack (3 * 4)

  push  dword 0   ;exit code
  mov eax,0x3f  ;system call number (sys_exit)
  call  _syscall  ;call kernel
        ;no need to clean stack
section .data

msg db  "Hello, world!",0xa ;our dear string
len equ $ - msg     ;length of our dear string
```

2.6 Building an executable

Building an executable is the usual two-step process of compiling and then linking. To make an executable out of our `hello.asm` we must do the following:

```
$ nasm -f elf hello.asm   # this will produce hello.o ELF object file
$ ld -s -o hello hello.o  # this will produce hello executable
```

Note
OpenBSD and NetBSD users should issue the following sequence instead (because of `a.out` executable format):

```
$ nasm -f aoutb hello.asm # this will produce hello.o a.out object file
$ ld -e _start -o hello hello.o # this will produce hello executable
```

That's it. Simple. Now you can launch the hello program by entering **./hello**. Look at the binary size -- surprised?

3 References

I hope you enjoyed this journey. If you get interested in assembly programming for UNIX, I strongly encourage you to visit Linux Assembly site for more information, and download the asmutils package, it contains a lot of sample code. For a comprehensive overview of Linux/UNIX assembly programming refer to the Linux Assembly HOWTO.

Thank you for your interest!

A History

B Acknowledgements

I would like to thank:

- DaemonNews people for proofreading and corrections they kindly submitted to me
- Eugene Tsyrklevich for note on NetBSD compile process

C Endorsements

This version of the document is endorsed by Konstantin Boldyshev.

Modifications (including translations) must remove this appendix according to the license agreement.

```
$Id:$
```