Startup state of Linux/i386 ELF binary

(the primary url for this document is http://linuxassembly.org/startup.html)

All information provided here has derived from my own research. So, mistakes and deficiencies could exist.
If you find any -- please contact me.

Contents

---

1. Introduction

The objective of this document is to describe several startup process details and the initial state of the stack & registers of the ELF binary program, for Linux Kernel 2.2.x and 2.0.x on i386.

Portions of material represented here may be applicable to any ELF-based IA-32 OS (FreeBSD, NetBSD, BeOS, etc).

Please note that in general case you can apply this information only to plain assembly programs (gas/nasm); some things described here (stack/registers state) are not true for anything compiled/linked with gcc (C as well as assembly) -- gcc inserts its own startup code which is executed before control is passed to main() function.

Main source and authority of information provided below is Linux Kernel's **fs/binfmt_elf.c** file.
If you want all details of the startup process -- go read it.

All assembly code examples use nasm syntax.

You can download program suite that was used while writing this document at the Linux Assembly (binaries, source).

2. Overview

Every program is executed by means of sys_execve() system call; usually one just types program name at the

shell prompt. In fact a lot of interesting things happen after you press enter. Shortly, startup process of the ELF binary can be represented with the following step-by-step figure:

Function	Kernel file	Comments
shell	...	on user side one types in program name and strikes enter
execve()	...	shell calls libc function
sys_execve()	...	libc calls kernel...
sys_execve()	arch/i386/kernel/process.c	arrive to kernel side
do_execve()	fs/exec.c	open file and do some preparation
search_binary_handler()	fs/exec.c	find out type of executable
load_elf_binary()	fs/binfmt_elf.c	load ELF (and needed libraries) and create user segment
start_thread()	include/asm-i386/processor.h	and finally pass control to program code

Figure 1. Startup process of ELF binary.

Layout of segment created for ELF binary shortly can be represented with *Figure 2*. Yellow parts represent correspondent program sections. Shared libraries are not shown here; their layout duplicates layout of program, except that they reside in earlier addresses.

0x08048000

code	.text section
data	.data section
bss	.bss section
... ...	free space
stack	stack (described later)
arguments	program arguments
environment	program environment
program name	filename of program (duplicated in arguments section)
null (dword)	final dword of zero

0xBFFFFFFF

Figure 2. Segment layout of ELF binary.

Program takes at least two pages of memory (1 page == 4 KB), even if it consists of single sys_exit(); at least one page for ELF data (yellow color), and one for stack, arguments, and environment. Stack is growing to meet .bss; also you can use memory beyond .bss section for dynamic data allocation.

*Note: this information was gathered from **fs/binfmt_elf.c**, **include/linux/sched.h** (task_struct.addr_limit), and core dumps investigated with ultimate binary viewer biew).*

3. Stack layout

Initial stack layout is very important, because it provides access to command line and environment of a program. Here is a picture of what is on the stack when program is launched:

argc	[dword] argument counter (integer)
argv[0]	[dword] program name (pointer)
argv[1] **...** **argv[argc-1]**	[dword] program args (pointers)
NULL	[dword] end of args (integer)
env[0] **env[1]** **...** **env[n]**	[dword] environment variables (pointers)
NULL	[dword] end of environment (integer)

Figure 3. Stack layout of ELF binary.

Here is the piece of source from kernel that proves it:

fs/binfmt_elf.c *create_elf_tables()*

```
...

put_user((unsigned long) argc, --sp);
current->mm->arg_start = (unsigned long) p;
while (argc-- > 0) {
        put_user(p, argv++);
        while (get_user(p++))   /* nothing */
                ;
}
put_user(0, argv);
current->mm->arg_end = current->mm->env_start = (unsigned lo
while (envc-- > 0) {
        put_user(p, envp++);
        while (get_user(p++))   /* nothing */
                ;
}
put_user(0, envp);

...
```

So, if you want to get arguments and environment, you just need to pop then one by one; argc and argv[0] are always present. Here's sample code (quite useless, just shows how to do it):

```
        pop     eax     ;get argument counter
        pop     ebx     ;get our name (argv[0])
.arg:
        pop     ecx     ;pop all arguments
        test    ecx,ecx
        jnz     .arg
.env:                   ;pop all environment vars
        pop     edx
        test    edx,edx
        jnz     .env
```

In fact you can also access arguments and environment in a different way -- directly. This method is based on structure of the user segment of loaded ELF binary: arguments and environment lay consequently at the end of segment (*Figure 2*). So, you can fetch address of first argument from the stack, and then just use it as start address. Arguments and environment variables are null-terminated strings; you need to know who is who, so you have to evaluate start and end of arguments and environment:

```
        pop     eax                             ;get argument counte
        pop     esi                             ;start of arguments
        mov     edi,[esp+eax*4]                 ;end of arguments
        mov     ebp,[esp+(eax+1)*4]             ;start of environmen
```

Second way seems to be more complex, you have to distinguish arguments manually. However it can be more suitable in some cases. Program name also can be fetched by downstepping from 0xBFFFFFFB (0xBFFFFFF-4) address (*Figure 2*).

4. Registers

Or better to say, general registers. Here things go different for Linux 2.0 and Linux 2.2. First I'll describe Linux Kernel 2.0.

4.1 Linux Kernel 2.0

Theoretically, all registers except EDX are undefined on program startup when using Linux 2.0. EDX is zeroed by ELF_PLAT_INIT in **fs/binfmt_elf.c** *create_elf_tables()* . Here is the definition of this macro:

include/asm-i386/elf.h

```
        ...

        /* SVR4/i386 ABI (pages 3-31, 3-32) says that when the progr
           starts %edx contains a pointer to a function which might
           registered using `atexit'.  This provides a mean for the
           dynamic linker to call DT_FINI functions for shared libra
           that have been loaded before the code runs.
```

```
           A value of 0 tells we have no such handler.  */
#define ELF_PLAT_INIT(_r)       _r->edx = 0

        ...
```

Practically, simple investigation shows that other registers have well-defined values. Here we go...

If you will be patient enough and follow the path shown on *Figure 1*, you'll find out that pt_regs structure (that contains register values before system call) is downpassed to *load_elf_binary()* and *create_elf_tables()* in **fs/binfmt_elf.c** COMPLETELY UNCHANGED (I will not cover this chain and appropriate kernel sources here to save space, but do not take my words, go check it :). The only modification is done right before passing control to program code, and was shown above -- EDX is zeroed (note: final *start_thread()* sets only segment & stack registers. Also EAX is always zero too, though I haven't found corresponding kernel source). This means that values of most general registers (EBX, ECX, ESI, EDI, EBP) on program startup are the same as in caller program before sys_execve()! More to say: **one can pass to program any custom values he wants in ESI, EDI and EBP** (certainly by means of direct syscall, not libc execve() function), and called program will receive them (sys_execve() call needs only EBX (program name), ECX (arguments) and EDX (environment) to be set). Conclusion: program gets photo of registers state before sys_execve(). You can use this to hack libc :)

I wrote two simple programs to illustrate state of registers -- *execve* and *regs*. *regs* shows registers state on startup, *execve* executes given program and shows registers before sys_execve() call. You can easily combine them - try running

$ *./execve ./regs*

on Linux 2.0 and you will get the picture of what I'm talking about.

Linux Kernel 2.2

On Linux 2.2 things are much simpler and less interesting -- all general register are zeroed by ELF_PLAT_INIT in *create_elf_tables()*, because ELF_PLAT_INIT is not the same as in Linux 2.0:

include/asm-i386/elf.h

```
#define ELF_PLAT_INIT(_r)       do { \
        _r->ebx = 0; _r->ecx = 0; _r->edx = 0; \
        _r->esi = 0; _r->edi = 0; _r->ebp = 0; \
        _r->eax = 0; \
} while (0)
```

Finally, as visual illustration of this difference, here is partial output of *regs* program both for Linux 2.0 and Linux 2.2:

Linux 2.0 (kernel 2.0.37)

```
EAX     :       0x0
EBX     :       0x80A1928
ECX     :       0x80A1958
EDX     :       0x0
```

```
ESI     :       0x0
EDI     :       0x8049E90
EBP     :       0xBFFFFBC4
ESP     :       0xBFFFFE14
EFLAGS  :       0x282
CS      :       0x23
DS      :       0x2B
ES      :       0x2B
FS      :       0x2B
GS      :       0x2B
SS      :       0x2B
```

Linux 2.2 (kernel 2.2.10)

```
EAX     :       0x0
EBX     :       0x0
ECX     :       0x0
EDX     :       0x0
ESI     :       0x0
EDI     :       0x0
EBP     :       0x0
ESP     :       0xBFFFFB40
EFLAGS  :       0x292
CS      :       0x23
DS      :       0x2B
ES      :       0x2B
FS      :       0x0
GS      :       0x0
SS      :       0x2B
```

In fact you can use this difference to determine quickly what kernel you are running under -- just check whether EBX or ECX are zeroes on startup:

```
        test    ebx,ebx
        jz      .kernel22       ;it is Linux 2.2
.kernel20:                      ;otherwise it is Linux 2.0
        ...

.kernel22:
        ...
```

Also, you probably noticed from *regs* output that FS and GS are not used in Linux 2.2; and they are no longer present in pt_regs structure..

5. Other info

fs/binfmt_elf.c also contains *padzero()* function that zeroes out .bss section of a program; so, every variable contained in .bss section will get value of 0. Once again, you can be sure that uninitialized data will not contain garbage. You can use this issue if you want to initialize any variable(s) with zero -- Linux will do it for you, just place them in .bss section.

6. Summary

Brief summary of things to know about ELF binary startup state:

- .bss section is zeroed out
- on Linux 2.2 all general registers are zeroed out
- on Linux 2.0 EAX and EDX are zeroed out, other contain values before sys_execve() call
- stack contains argc,argv[0 -- (argc-1)] and envp[0 -- n], in that order

7. Contact

Author: Konstantin Boldyshev <konst@linuxassembly.org>

$Id: startup.html,v 1.12 2000/07/11 10:49:34 konst Exp $