


[TABLE OF CONTENTS](#) [FRONT PAGE](#) [TALKBACK](#) [FAQ](#)


...making Linux just a little more fun!

How main() is executed on Linux

By [Hyouck "Hawk" Kim](#)

Starting

The question is simple: how does linux execute my main()? Through this document, I'll use the following simple C program to illustrate how it works. It's called "simple.c"

```
main()
{
    return(0);
}
```

Build

```
gcc -o simple simple.c
```

What's in the executable?

To see what's in the executable, let's use a tool "objdump"

```
objdump -f simple

simple:      file format elf32-i386
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x080482d0
```

The output gives us some critical information about the executable. First of all, the file is "ELF32" format. Second of all, the start address is "0x080482d0"

What's ELF?

ELF is acronym for Executable and Linking Format. It's one of several object and executable file formats used on Unix systems. For our discussion, the interesting thing about ELF is its header format. Every ELF executable has ELF header, which is the following.

```
typedef struct
{
    unsigned char    e_ident[EI_NIDENT];    /* Magic number and other info */
    Elf32_Half       e_type;                /* Object file type */
    Elf32_Half       e_machine;             /* Architecture */
    Elf32_Word       e_version;             /* Object file version */
    Elf32_Addr       e_entry;               /* Entry point virtual address */
    Elf32_Off        e_phoff;               /* Program header table file offset */
    Elf32_Off        e_shoff;               /* Section header table file offset */
    Elf32_Word       e_flags;               /* Processor-specific flags */
    Elf32_Half       e_ehsize;              /* ELF header size in bytes */
}
```

```
Elf32_Half e_phentsize; /* Program header table entry size */
Elf32_Half e_phnum;     /* Program header table entry count */
Elf32_Half e_shentsize; /* Section header table entry size */
Elf32_Half e_shnum;     /* Section header table entry count */
Elf32_Half e_shstrndx;  /* Section header string table index */
} Elf32_Ehdr;
```

In the above structure, there is "e_entry" field, which is starting address of an executable.

What's at address "0x080482d0", that is, starting address?

For this question, let's disassemble "simple". There are several tools to disassemble an executable. I'll use objdump for this purpose.

```
objdump --disassemble simple
```

The output is a little bit long so I'll not paste all the output from objdump. Our intention is see what's at address 0x080482d0. Here is the output.

```
080482d0 <_start>:
80482d0: 31 ed                xor    %ebp,%ebp
80482d2: 5e                  pop    %esi
80482d3: 89 e1                mov    %esp,%ecx
80482d5: 83 e4 f0             and    $0xffffffff0,%esp
80482d8: 50                  push   %eax
80482d9: 54                  push   %esp
80482da: 52                  push   %edx
80482db: 68 20 84 04 08       push   $0x8048420
80482e0: 68 74 82 04 08       push   $0x8048274
80482e5: 51                  push   %ecx
80482e6: 56                  push   %esi
80482e7: 68 d0 83 04 08       push   $0x80483d0
80482ec: e8 cb ff ff ff      call   80482bc <_init+0x48>
80482f1: f4                  hlt
80482f2: 89 f6                mov    %esi,%esi
```

Looks like some kind of starting routine called "_start" is at the starting address. What it does is clear a register, push some values into stack and call a function. According to this instruction, the stack frame should look like this.

```
Stack Top -----
0x80483d -----
esi -----
ecx -----
0x8048274 -----
0x8048420 -----
edx -----
esp -----
eax -----
```

Now, as you already wonder,we've got a few questions regarding this stack frame.

1. What are those hex values about?
2. What's at address 80482bc, which is called by `_start`?
3. Looks like the assembly instructions doesn't initialize any register with possibly meaningful values. Then who initializes the registers?

Let's answer these questions one by one.

Q1>The hexa values.

If you look at disassembled output from `objdump` carefully, you can answer this question easily.

Here is answer.

0x80483d0 : This is the address of our `main()` function.

0x8048274 : `_init` function.

0x8048420 : `_fini` function `_init` and `_fini` is initialization/finalization function provided by GCC.

Right now, let's not care about these stuffs. And basically, all those hexa values are function pointers.

Q2>What's at address 80482bc?

Again, let's look for address 80482bc from the disassembly output.

If you look for it, the assembly is

```
80482bc:      ff 25 48 95 04 08      jmp     *0x8049548
```

Here `*0x8049548` is a pointer operation.

It just jumps to an address stored at address `0x8049548`.

More about ELF and dynamic linking

With ELF, we can build an executable linked dynamically with libraries.

Here "linked dynamically" means the actual linking process happens at runtime. Otherwise we'd have to build a huge executable containing all the libraries it calls (a "statically-linked executable"). If you issue the command

```
"ldd simple"
```

```
libc.so.6 => /lib/i686/libc.so.6 (0x42000000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

You can see all the libraries dynamically linked with `simple`. And all the dynamically linked data and functions have "dynamic relocation entry".

The concept is roughly like this.

- 1. We don't know actual address of a dynamic symbol at link time. We can know the actual address of the symbol only at runtime.
- 2. So for the dynamic symbol, we reserve a memory location for the actual address.
The memory location will be filled with actual address of the symbol at runtime by loader.
- 3. Our application sees the dynamic symbol indirectly with the momeory location by using kind of pointer operation. In our case, at address 80482bc, there is just a simple jump instruction.
And the jump location is stored at address 0x8049548 by loader during runtime.
We can see all dynamic link entries with objdump command.

```
objdump -R simple

simple:          file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET      TYPE          VALUE
0804954c  R_386_GLOB_DAT  __gmon_start__
08049540  R_386_JUMP_SLOT  __register_frame_info
08049544  R_386_JUMP_SLOT  __deregister_frame_info
08049548  R_386_JUMP_SLOT  __libc_start_main
```

Here address 0x8049548 is called "jump slot", which perfectly makes sense. And according to the table, actually we want to call `__libc_start_main`.

What's `__libc_start_main`?

Now the ball is on libc's hand. `__libc_start_main` is a function in `libc.so.6`. If you look for `__libc_start_main` in glibc source code, the prototype looks like this.

```
extern int BP_SYM (__libc_start_main) (int (*main) (int, char **, char **),
    int argc,
    char * __unbounded * __unbounded ubp_av,
    void (*init) (void),
    void (*fini) (void),
    void (*rtld_fini) (void),
    void * __unbounded stack_end)
__attribute__((noreturn));
```

And all the assembly instructions do is set up argument stack and call `__libc_start_main`. What this function does is setup/initialize some data structures/environments and call our `main()`. Let's look at the stack frame with this function prototype.

Stack Top	-----	
	0x80483d0	main

	esi	argc

	ecx	argv

	0x8048274	_init

	0x8048420	_fini

	edx	_rtlf_fini

	esp	stack_end



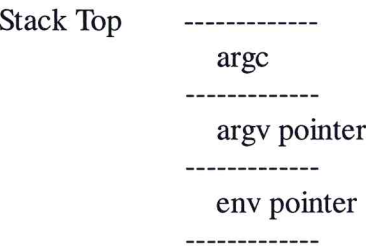
According to this stack frame, esi, ecx, edx, esp, eax registers should be filled with appropriate values before `__libc_start_main()` is executed. And clearly this registers are not set by the startup assembly instructions shown before. Then, who sets these registers? Now I guess the only thing left. The kernel. Now let's go back to our third question.

Q3>What does the kernel do?

When we execute a program by entering a name on shell, this is what happens on Linux.

1. The shell calls the kernel system call "execve" with argc/argv.
2. The kernel system call handler gets control and start handling the system call. In kernel code, the handler is "sys_execve". On x86, the user-mode application passes all required parameters to kernel with the following registers.
 - o ebx : pointer to program name string
 - o ecx : argv array pointer
 - o edx : environment variable array pointer.
3. The generic execve kernel system call handler, which is `do_execve`, is called. What it does is set up a data structure and copy some data from user space to kernel space and finally calls `search_binary_handler()`. Linux can support more than one executable file format such as a.out and ELF at the same time. For this functionality, there is a data structure "struct linux_binfmt", which has a function pointer for each binary format loader. And `search_binary_handler()` just looks up an appropriate handler and calls it. In our case, `load_elf_binary()` is the handler. To explain each detail of the function would be lengthy/boring work. So I'll not do that. If you are interested in it, read a book about it. As a picture tells a thousand words, a thousand lines of source code tells ten thousand words (sometimes). Here is the bottom line of the function. It first sets up kernel data structures for file operation to read the ELF executable image in. Then it sets up a kernel data structure: code size, data segment start, stack segment start, etc. And it allocates user mode pages for this process and copies the argv and environment variables to those allocated page addresses. Finally, argc, the argv pointer, and the environment variable array pointer are pushed to user mode stack by `create_elf_tables()`, and `start_thread()` starts the process execution rolling.

When the `_start` assembly instruction gets control of execution, the stack frame looks like this.



And the assembly instructions gets all information from stack by

```
pop %esi                      <--- get argc
move %esp, %ecx              <--- get argv
```

actually the argv address is the same as the current stack pointer.

And now we are all set to start executing.

What about the other registers?

For esp, this is used for stack end in application program. After popping all necessary information, the `_start` routine simply adjusts the stack pointer (esp) by turning off lower 4 bits from esp register. This perfectly makes sense since actually, to our main program, that is the end of stack. For edx, which is used for `rtld_fini`, a kind of application destructor, the kernel just sets it to 0 with the following macro.

```
#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)
```

The 0 means we don't use that functionality on x86 linux.

About the assembly instructions

Where are all those codes from? It's part of GCC code. You can usually find all the object files for the code at

`/usr/lib/gcc-lib/i386-redhat-linux/XXX` and

`/usr/lib` where XXX is gcc version.

File names are `crtbegin.o`, `crtend.o`, `gcrt1.o`.

Summing up

Here is what happens.

1. GCC build your program with `crtbegin.o/crtend.o/gcrt1.o` And the other default libraries are dynamically linked by default. Starting address of the executable is set to that of `_start`.
2. Kernel loads the executable and setup text/data/bss/stack, especially, kernel allocate page(s) for arguments and environment variables and pushes all necessary information on stack.
3. Control is passed to `_start`. `_start` gets all information from stack setup by kernel, sets up argument stack for `__libc_start_main`, and calls it.
4. `__libc_start_main` initializes necessary stuffs, especially C library(such as malloc) and thread environment and calls our main.
5. our main is called with `main(argv, argv)` Actually, here one interesting point is the signature of main. `__libc_start_main` thinks main's signature as `main(int, char **, char **)` If you are curious, try the following program.

```
main(int argc, char** argv, char** env)
{
    int i = 0;
    while(env[i] != 0)
    {
        printf("%s\n", env[i++]);
    }
    return(0);
}
```


Conclusion

On Linux, our C main() function is executed by the cooperative work of GCC, libc and Linux's binary loader.

References

objdump	"man objdump"
ELF header	/usr/include/elf.h
__libc_start_main	glibc source ./sysdeps/generic/libc-start.c
sys_execve	linux kernel source code arch/i386/kernel/process.c
do_execve	linux kernel source code fs/exec.c
struct linux_binfmt	linux kernel source code include/linux/binfmts.h
load_elf_binary	linux kernel source code fs/binfmt_elf.c
create_elf_tables	linux kernel source code fs/binfmt_elf.c
start_thread	linux kernel source code include/asm/processor.h

Copyright © 2002, Hyouck "Hawk" Kim. Copying license <http://www.linuxgazette.com/copying.html>
Published in Issue 84 of *Linux Gazette*, November 2002

[TABLE OF CONTENTS](#)[FRONT PAGE](#)[TALKBACK](#)[FAQ](#)