Linux操作系统分析
Chapter 9-2 Linux中程序的执行

陈香兰（xlanchen@ustc.edu.cn）

计算机应用教研室@计算机学院
嵌入式系统实验室@苏州研究院
中国科学技术大学
Fall 2014

November 4, 2014

Outline

操作系统是如何通过可执行文件的内容建立进程的执行上下文的？

- 可执行文件的格式
  - 程序以可执行文件的形式存放在磁盘上
- 库
  - 可供很多程序使用的一些例程的集合
  - 静态库 vs 共享库
- 命令行参数、环境变量等
  - She11提示符下输入
  - 从she11继承而来，用户可修改

Outline

可执行文件

- 可执行文件是一个普通的文件，它描述了如何初始化一个新的进程上下文
- Linux中：Fork + execve
  - 例如：shell程序中执行一个命令

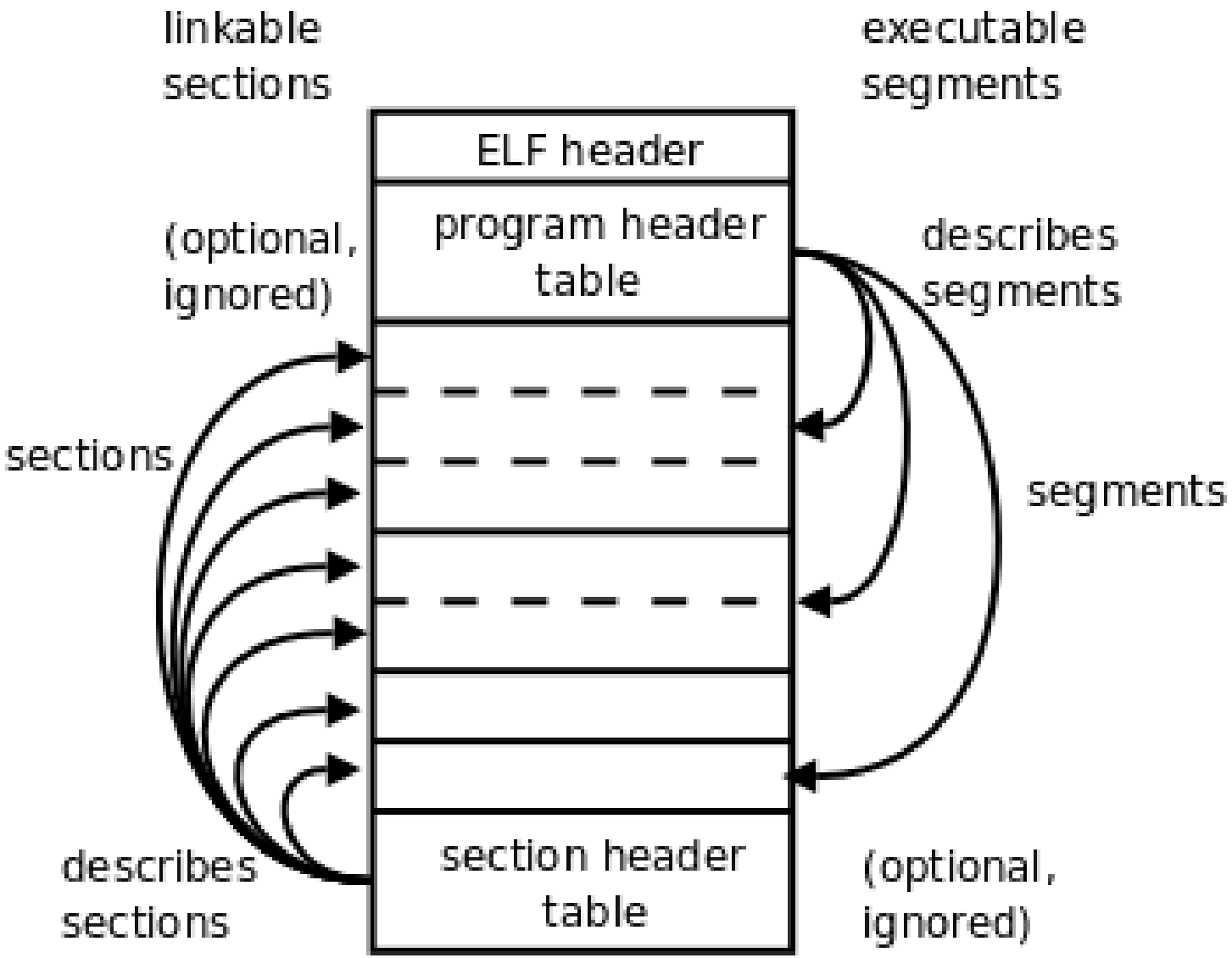
可执行文件的格式

- Linux标准的可执行格式
  - ELF：Executable and Linking Format
    查看ELF格式可执行文件xxx的头部信息：

    ```
    readelf -h xxx
    ```

  - 旧版的可执行文件格式
    - ★ a.out：Assembler OUT put format
      目前，Linux中a.out被ELF取代
      测试：可以使用readelf -h查看a.out可执行文件的header信息
- 其他
  - MS-DOS的exe文件
  - UNIX BSD的COFF文件

ELF文件格式


linkable
sections
executable
segments
ELF header
program header
table
(optional,
ignored)
describes
segments
sections
segments
describes
sections
section header
table
(optional,
ignored)

查看ELF格式可执行文件的头部信息

- 以“void main(void){}”为例：

ubuntu-14.04：readelf -h hello

```
ELF 头：
Magic：  7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
Class:                             ELF32
Data:                              2's complement, little endian
Version:                           1 (current)
OS/ABI:                            UNIX - System V
ABI Version:                       0
Type:                              EXEC (可执行文件)
Machine:                           Intel 80386
Version:                           0x1
入口点地址：                       0x80482f0
程序头起点：                       52 (bytes into file)
Start of section headers:          4424 (bytes into file)
标志：                             0x0
本头的大小：                       52 (字节)
程序头大小：                       32 (字节)
Number of program headers:         9
节头大小：                 40 (字节)
节头数量：                 30
字符串表索引节头：         27
```

Outline

Linux对可执行文件格式的管理

- 可执行文件格式的描述符：linux_binfmt

```
/*
 * This structure defines the functions that are used to load the binary formats that
 * linux accepts.
 */
struct linux_binfmt {
  struct list_head lh;
  struct module *module;
  int (*load_binary)(struct linux_binprm *, struct pt_regs * regs);
  int (*load_shlib)(struct file *);
  int (*core_dump)(long signr, struct pt_regs *regs,
                   struct file *file, unsigned long limit);
  unsigned long min_coredump; /* minimal dump size */
  int hasvdso;
};
```

- load_binary: 通过读取存放在可执行文件中的信息为当前进程建立一个新的进程上下文
- load_shlib: 动态的把一个共享库绑定到一个已经在运行的进程
- core_dump：把当前进程的上下文保存到名为core的文件中

Linux对可执行文件格式的管理

- 例如，ELF格式的描述符elf_format

fs/binfmt_elf.c

```
static struct linux_binfmt elf_format = {
    .module = THIS_MODULE,
    .load_binary = load_elf_binary,
    .load_shlib = load_elf_library,
    .core_dump = elf_core_dump,
    .min_coredump = ELF_EXEC_PAGESIZE,
    .hasvdso = 1
};
```

Linux对可执行文件格式的管理

- 可执行文件格式的注册和注销：

include/linux/binfmts.h

```
extern int register_binfmt(struct linux_binfmt *);
extern void unregister_binfmt(struct linux_binfmt *);
```

- 可执行文件格式的链表

fs/exec.c

```
static LIST_HEAD(formats);
static DEFINE_RWLOCK(binfmt_lock);
```

- 在系统启动时，所有编译进内核的可执行格式都被注册
  - 查看Linux 2.6.26中看到的可执行文件格式
- 在系统运行过程中，也可以注册一个新的可执行文件格式

Linux对可执行文件格式的管理

- Linux通过可执行文件的扩展名或者存放在文件前128字节的magic数来识别文件格式

fs/exec.c

```
/*
* cycle the list of binary formats handler, until one recognizes the image
*/
int search_binary_handler(struct linux_binprm *bprm,struct pt_regs *regs)
{...}
```

- 文件扩展名
  - Exe
  - Bat
  - ...

Outline

Exec函数家族

man execl

```
#include <unistd.h>
extern char **environ;
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ..., char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[], char *const envp[]);
```

- 用一个指定的可执行文件所描述的上下文代替进程的上下文
- 相关系统服务例程：
  - sys_execve() executes a new program.n

Exec函数家族

arch/x86/kernel/process_32.c

```
asmlinkage int sys_execve(struct pt_regs regs) {
    int error;
    char * filename;

    filename = getname((char __user *) regs.bx);
    error = PTR_ERR(filename);
    if (IS_ERR(filename))
        goto out;
    error = do_execve(filename,
                      (char __user * __user *) regs.cx,
                      (char __user * __user *) regs.dx,
                      &regs);
    if (error == 0) {
        /* Make sure we don't return using sysenter.. */
        set_thread_flag(TIF_IRET);
    }
    putname(filename);
out:
    return error;
}
```

Exec函数家族

fs/exec.c

```
int do_execve(char * filename,
        char __user *__user *argv,
        char __user *__user *envp,
        struct pt_regs * regs) {
    ...
    file = open_exec(filename);
    ...
    retval = bprm_mm_init(bprm);
    ...
    retval = copy_strings_kernel(1, &bprm->filename, bprm);
    if (retval < 0) goto out;
    retval = copy_strings(bprm->envc, envp, bprm);
    if (retval < 0) goto out;
    retval = copy_strings(bprm->envc, envp, bprm);
    if (retval < 0) goto out;
    retval = search_binary_handler(bprm,regs);
    ...
}
```

Exec函数家族

```
/*
 * Create a new mm_struct and populate it with a temporary stack
 * vm_area_struct. We don't have enough context at this point to set the stack
 * flags, permissions, and offset, so we use temporary values. We'll update
 * them later in setup_arg_pages().
 */
int bprm_mm_init(struct linux_binprm *bprm) {
    int err;
    struct mm_struct *mm = NULL;
    bprm->mm = mm = mm_alloc();
    err = -ENOMEM;
    if (!mm) goto err;
    err = init_new_context(current, mm);
    if (err) goto err;
    err = __bprm_mm_init(bprm);
    if (err) goto err;
    return 0;
err:
    if (mm) {
        bprm->mm = NULL;
        mmdrop(mm);
    }
    return err;
}
```

Exec函数家族

fs/exec.c

```
/*
* cycle the list of binary formats handler, until one recognizes the image
*/
int search_binary_handler(struct linux_binprm *bprm,struct pt_regs *regs) {
    ...
    for (try=0; try<2; try++) {
        read_lock(&binfmt_lock);
        list_for_each_entry(fmt, &formats, lh) {
            int (*fn)(struct linux_binprm *, struct pt_regs *) = fmt->load_binary;
            if (!fn) continue;
            if (!try_module_get(fmt->module)) continue;
            read_unlock(&binfmt_lock);
            retval = fn(bprm, regs);
            ...
}
```

Exec函数家族

- 以elf格式的文件为例（fn=load_elf_binary）：

fs/binfmt_elf.c

```
static int load_elf_binary(struct linux_binprm *bprm, struct pt_regs *regs) {
    ...
    /* First of all, some simple consistency checks */
    if (memcmp(loc->elf_ex.e_ident, ELFMAG, SELFMAG) != 0)
    goto out;
    ...
}
```

  - 调用set_brk，elf_map，do_mmap，建立各线性区

程序段和进程的线性区

- 在逻辑上，Unix程序的线性地址空间被划分为各种段（segment）
  - 正文段，text
  - 数据段，data
  - Bss段
  - 堆栈段
- 在mm_struct中都有对应的字段

```
unsigned long start_code, end_code, start_data, end_data;
unsigned long start_brk, brk, start_stack;
unsigned long arg_start, arg_end, env_start, env_end;
```

- 此外，还有共享库和文件的映射，映射在其他线性区
- 参阅/proc/1/maps了解init进程的线性区

命令行参数和shell环境

- 用户使用shell来执行某个程序时，可以指定命令行参数。
  例如：

$$ls\ -l\ /usr/bin$$

  列出/usr/bin下的目录信息
- Shell本身不限制命令行参数的个数，
  命令行参数的个数受限于命令自身。
  例如

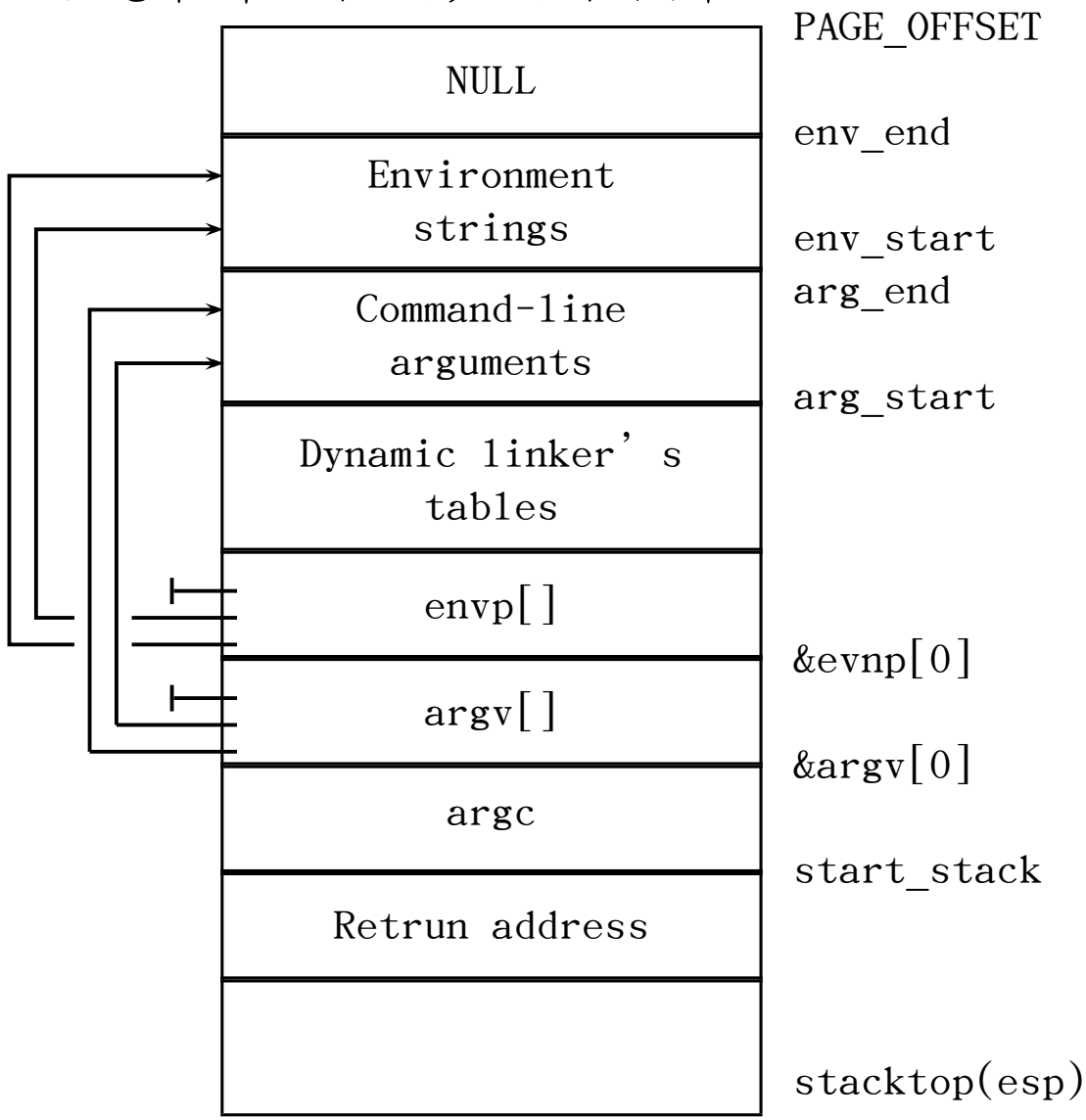
$$int\ main(int\ argc,\ char\ *argv[])$$

  又如

$$int\ main(int\ argc,\ char\ *argv[],\ char\ *envp[])$$

命令行参数和shell环境

- 命令行参数和环境串都放在用户态堆栈中



库

- 源文件→目标文件→可执行文件
- 最小的程序也会利用到C库。例如

```
void main(void) { }
```

1. 要为main的执行建立执行上下文
2. 在进程结束时，杀死进程（在main的最后插入exit()）

- 其他库
  - libm，包含浮点操作的基本函数
  - libX11，所有X11窗口系统图形接口的基本底层函数
  - ...

静态链接 vs 动态链接

- 静态库
- 动态链接：共享库
- Gcc的-static选项指明使用静态库

举例

- 对于

```
void main(void) { }
```

1. 使用gcc -g -static参数编译
2. 使用readelf -h获得可执行文件头，查看可执行程序的入口地址。例如：

入口点地址：0x8048d2a

举例

3. 使用objdump -D反汇编，查看入口地址所对应的程序是什么？（_start）。例如：

```
08048d2a <_start>:
 8048d2a:  31 ed                xor %ebp,%ebp
 8048d2c:  5e                   pop %esi
 8048d2d:  89 e1                mov %esp,%ecx
 8048d2f:  83 e4 f0             and $0xfffffff0,%esp
 8048d32:  50                   push %eax
 8048d33:  54                   push %esp
 8048d34:  52                   push %edx
 8048d35:  68 d0 95 04 08       push $0x80495d0
 8048d3a:  68 30 95 04 08       push $0x8049530
 8048d3f:  51                   push %ecx
 8048d40:  56                   push %esi
 8048d41:  68 44 8e 04 08       push $0x8048e44
 8048d46:  e8 05 01 00 00       call 8048e50 <__libc_start_main>
 8048d4b:  f4                   hlt
 8048d4c:  66 90                xchg %ax,%ax
 8048d4e:  66 90                xchg %ax,%ax
```

举例

4. 从返回编代码中找到从入口_start到main的调用路径

```
08048e44 <main>:
 8048e44:  55                    push %ebp
 8048e45:  89 e5                 mov %esp,%ebp
 8048e47:  5d                    pop %ebp
 8048e48:  c3                    ret
 8048e49:  66 90                 xchg %ax,%ax
 8048e4b:  66 90                 xchg %ax,%ax
 8048e4d:  66 90                 xchg %ax,%ax
 8048e4f:  90                    nop
```

- HOW?

```
08048e50 <__libc_start_main>:
 ...
 8049016:  ff 54 24 60           call *0x60(%esp)
 804901a:  89 04 24              mov %eax,(%esp)
 804901d:  e8 1e 57 00 00        call 804e740 <exit>
 ...
```

举例

5. 使用gdb来跟踪执行
   - 在_start处设置断点，查看esp的值（验证堆栈位置在3G附近）

Outline

小结

1. 可执行文件及其格式
2. Linux对可执行文件格式的管理
3. 可执行文件的执行
4. 小结和作业

作业

- 按照ppt中所举的例子void main(void) {}，查看反汇编代码，若main函数有参数，则参数是如何传递给main的？

Thanks !

The end.