

# 实例分析 1: 进程

## 1 第零部分: 初识 xv6

### 1.0.1 基础题

1. 安装相应工具链, 在 qemu 中运行 xv6。并在操作系统中运行 ls 命令。
2. 阅读 ls.c。请回答: 代码中的 read() 函数和 printf() 函数哪个是系统调用? 它们的函数声明在哪里, 函数定义在哪里?

read() 是系统调用, 函数声明在 user/user.h 中, 定义在 user.pl 中, 其中有一段内联汇编代码, 将 sys\_read() 系统调用的系统调用号送给 a7, 然后调用 ecall 指令, 进入内核态执行系统调用。printf() 不是系统调用, 函数声明在 user/user.h 中, 定义在 user/printf.c 中。

3. 阅读 usys.S, 查阅 RISC-V 相关知识, 请问 ecall 指令的功能是什么?(如果找不到 usys.S, 请自我反思一下, 不要阅读静态代码)

ecall 指令是 RISC-V 中的一个特权指令, 用于从用户态切换到内核态, 触发异常处理程序, 执行系统调用。a7 寄存器中存放系统调用号, a0-a6 寄存器中存放系统调用的参数。

### 1.0.2 进阶题

阅读 xv6 项目相关的 Makefile 文件、mkfs.c 文件和链接器相关文件, 分析 xv6 把内核和用户态程序编译链接的整个过程。并思考一个问题, xv6 运行的时候, ls 可以看到里面有一个 README 文件, 我在 xv6 操作系统把它删除, 为什么本地项目的 README 文件仍存在 (反之如此)?

xv6 的 Makefile 文件中定义了一系列的变量, 包括编译器、编译选项、链接器等。在编译 xv6 时, 首先编译 bootblock, 然后编译 kernel, 最后编译用户态程序。编译用户态程序时, 首先编译 mkfs, 然后编译 init, 最后编译其他用户态程序。编译用户态程序时, 需要链接器将用户态程序和内核链接在一起, 生成 ELF 文件。在编译 xv6 时, 会生成一个文件系统的镜像文件, 包括 bootblock、kernel、用户态程序等。在运行 xv6 时, 会将这个镜像文件加载到内存中, 然后运行 bootblock, bootblock 会加载 kernel, kernel 会加载用户态程序。因此, xv6 运行的时候, ls 可以看到里面有一个 README 文件, 如果在 xv6 操作系统中删除 README 文件, 本地项目的 README 文件仍然存在, 是因为本地项目的 README 文件和 xv6 的文件系统镜像文件是两个不同的文件, 删除 xv6 的文件系统镜像文件中的 README 文件并不会影响本地项目的 README 文件。

## 2 第四部分: 进程的运行

### 2.0.1 基础题

1. proc.c 中 uchar initcode[] 的诡异的二进制数据的含义是什么? 为什么 userinit() 函数要把它拷口到 usr page 中? 它和 user/initCode.S 有什么联系?

```
// a user program that calls exec("/init")
// assembled from ../user/initcode.S
// od -t xC ../user/initcode
uchar initcode[] = {
    0x17, 0x05, 0x00, 0x00, 0x13, 0x05, 0x45, 0x02,
    0x97, 0x05, 0x00, 0x00, 0x93, 0x85, 0x35, 0x02,
    0x93, 0x08, 0x70, 0x00, 0x73, 0x00, 0x00, 0x00,
    0x93, 0x08, 0x20, 0x00, 0x73, 0x00, 0x00, 0x00,
    0xef, 0xf0, 0x9f, 0xff, 0x2f, 0x69, 0x6e, 0x69,
    0x74, 0x00, 0x00, 0x24, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00
};
```

这一些二进制数据实际上拼起来是一个用户程序,调用 `exec("/init")`。

`userinit()` 函数将这个用户程序拷贝到用户态的内存中,然后调用 `exec()` 函数执行这个用户程序,是为了在系统启动时执行一个初始的用户程序,从而进入用户模式。

下面展示出 `userinit()` 函数的代码:

```
// Set up first user process.
void
userinit(void)
{
    struct proc *p;

    p = allocproc();
    initproc = p;

    // allocate one user page and copy initcode's instructions
    // and data into it.
    uvmfirst(p->pagetable, initcode, sizeof(initcode));
    p->sz = PGSIZE;

    // prepare for the very first "return" from kernel to user.
    p->trapframe->epc = 0; // user program counter
    p->trapframe->sp = PGSIZE; // user stack pointer

    safestrcpy(p->name, "initcode", sizeof(p->name));
    p->cwd = namei("/");

    p->state = RUNNABLE;

    release(&p->lock);
```

```
}
```

initcode[] 和 user/initCode.S 是一样的,都是一个用户程序,调用 `exec("/init")`。

## 2. 第一个进程启动后在用户态执行的程序是什么? 这个程序执行了哪个系统调用?

第一个进程启动后在用户态执行的程序是 `user/init.c`,代码如下:

```
// init: The initial user-level program

#include "kernel/types.h"
#include "kernel/stat.h"
#include "kernel/spinlock.h"
#include "kernel/sleeplock.h"
#include "kernel/fs.h"
#include "kernel/file.h"
#include "user/user.h"
#include "kernel/fcntl.h"

char *argv[] = { "sh", 0 };

int
main(void)
{
    int pid, wpid;

    if(open("console", O_RDWR) < 0){
        mknod("console", CONSOLE, 0);
        open("console", O_RDWR);
    }
    dup(0); // stdout
    dup(0); // stderr

    for(;;){
        printf("init: starting sh\n");
        pid = fork();
        if(pid < 0){
            printf("init: fork failed\n");
            exit(1);
        }
        if(pid == 0){
            exec("sh", argv);
            printf("init: exec sh failed\n");
            exit(1);
        }
    }
}
```

```
for(;;){
    // this call to wait() returns if the shell exits,
    // or if a parentless process exits.
    wpid = wait((int *) 0);
    if(wpid == pid){
        // the shell exited; restart it.
        break;
    } else if(wpid < 0){
        printf("init: wait returned an error\n");
        exit(1);
    } else {
        // it was a parentless process; do nothing.
    }
}
}
```

这个程序执行了 `open()`、`mknod()`、`dup()`、`fork()`、`exec()`、`wait()`、`exit()` 等系统调用。

`open()` 系统调用用于打开文件，`mknod()` 系统调用用于创建设备文件，`dup()` 系统调用用于复制文件描述符，`fork()` 系统调用用于创建子进程，`exec()` 系统调用用于执行程序，`wait()` 系统调用用于等待子进程退出，`exit()` 系统调用用于退出进程。

上面的程序是一个无限循环，不断创建子进程执行 `sh` 程序，如果 `sh` 程序退出，则重新创建一个 `sh` 进程。

3. 在 `exec.c` 的 `exec()` 中，用到的 `struct elfhdr` 数据结构，其中 `magic`，`phnum`，`phoff` 等字段的作用是什么？以及 `struct proghdr` 的数据结构，其中 `vaddr`，`memsz`，`filesz` 等字段的作用是什么？

`struct elfhdr` 数据结构是 ELF 文件头，用于描述 ELF 文件的基本信息，`magic` 字段用于标识 ELF 文件，`phnum` 字段表示 ELF 文件中的程序头的数量，`phoff` 字段表示 ELF 文件中的程序头表的偏移量。

`struct proghdr` 数据结构是程序头，用于描述 ELF 文件中的程序段的信息，`vaddr` 字段表示程序段的虚拟地址，`memsz` 字段表示程序段在内存中的大小，`filesz` 字段表示程序段在文件中的大小。

4. 在 `exec()` 中，如何确定并设置待运行的程序的 PC 值和栈指针？

`exec()` 函数会读取 ELF 文件头，以获取程序入口点的地址，然后将这个地址设置为新进程的 PC 值。

在加载完程序段后，`exec()` 函数会为用户栈分配内存，并将命令行参数拷贝到栈中。然后，计算并设置栈指针(SP)。计算过程如下：

```
// Push argument strings, prepare rest of stack in ustack.
for(argc = 0; argv[argc]; argc++) {
    if(argc >= MAXARG)
        goto bad;
    sp -= strlen(argv[argc]) + 1;
```

```
    sp -= sp % 16; // riscv sp must be 16-byte aligned
    if(sp < stackbase)
        goto bad;
    if(copyout(pagetable, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
        goto bad;
    ustack[argc] = sp;
}
ustack[argc] = 0;

// push the array of argv[] pointers.
sp -= (argc+1) * sizeof(uint64);
sp -= sp % 16;
if(sp < stackbase)
    goto bad;
if(copyout(pagetable, sp, (char *)ustack, (argc+1)*sizeof(uint64)) < 0)
    goto bad;
```

5. `exec()` 执行完以后,返回的地址是什么? 为什么?

`exec()` 函数执行完以后,返回的地址是新程序的入口点地址 `elf.entry`,因为它会用新程序替换当前进程的地址空间,并开始执行新程序。

6. 在 `user/init.c` 中调用了 `fork()` 函数创建子进程。请问在调用 `fork()` 系统调用后,是父进程先返回还是子进程先返回?

子进程先返回,因为父进程会 `wait()` 子进程,等待子进程退出。

7. 在调用 `fork()` 系统调用后,子进程是如何从 `RUNNABLE` 转换到 `RUNNING` 状态的?

实际上,子进程是由操作系统调度器调度的,当子进程创建后,`fork()` 函数会将子进程的状态设置为 `RUNNABLE`,然后由操作系统调度器调度,将子进程从 `RUNNABLE` 状态转换到 `RUNNING` 状态。

8. 对于父进程和子进程,`fork()` 返回的 `pid` 相同么? 为什么?

父进程和子进程是两个不同的进程,`pid` 当然不同。

9. `wait` 系统调用的功能?

`wait()` 系统调用用于等待子进程退出,如果子进程已经退出,则 `wait()` 会立即返回,否则 `wait()` 会阻塞当前进程,直到子进程退出。

## 2.0.2 进阶题

请结合代码详细分析 Linux 中 `elf` 文件格式 (利用 `readelf` 命令),以及链接和加载的机制。

在 `kernel/elf.h` 中定义了 `ELF` 文件头的数据结构 `elfhdr` 和程序头的数据结构 `proghdr`,代码如下:

---

```
// Format of an ELF executable file

#define ELF_MAGIC 0x464C457FU // "\x7FELF" in little endian

// File header
struct elfhdr {
    uint magic; // must equal ELF_MAGIC
    uchar elf[12];
    ushort type;
    ushort machine;
    uint version;
    uint64 entry;
    uint64 phoff;
    uint64 shoff;
    uint flags;
    ushort ehsize;
    ushort phentsize;
    ushort phnum;
    ushort shentsize;
    ushort shnum;
    ushort shstrndx;
};

// Program section header
struct proghdr {
    uint32 type;
    uint32 flags;
    uint64 off;
    uint64 vaddr;
    uint64 paddr;
    uint64 filesz;
    uint64 memsz;
    uint64 align;
};

// Values for Proghdr type
#define ELF_PROG_LOAD 1

// Flag bits for Proghdr flags
#define ELF_PROG_FLAG_EXEC 1
#define ELF_PROG_FLAG_WRITE 2
#define ELF_PROG_FLAG_READ 4
```

---

elf 文件格式的示意图如下所示：

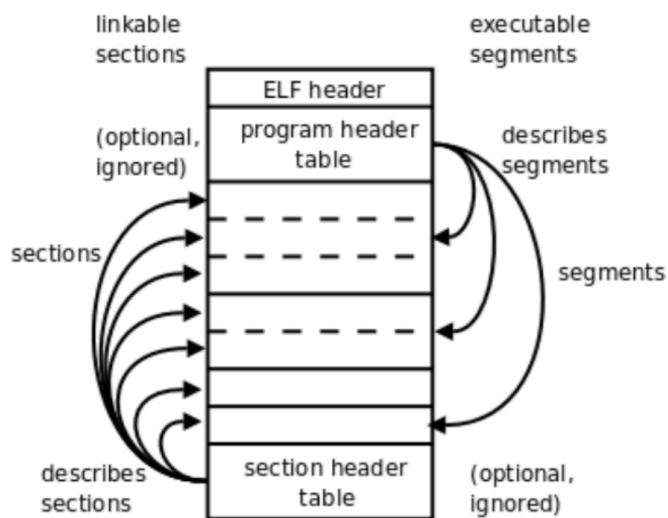


图 P1-4: ELF 文件结构

## ELF Object File Format

- Elf header
  - Word size, byte ordering, file type (.o, exec, .so), machine type, etc.
- Program (Segment) header table
  - Page size, virtual addresses memory segments (sections), segment sizes.
- .text section 写死的虚地址
  - Code
- .rodata section
  - Read only data: jump tables, ...
- .data section
  - Initialized global variables
- .bss section
  - Uninitialized global variables
  - “Block Started by Symbol”
  - “Better Save Space”
  - Has section header but occupies no space

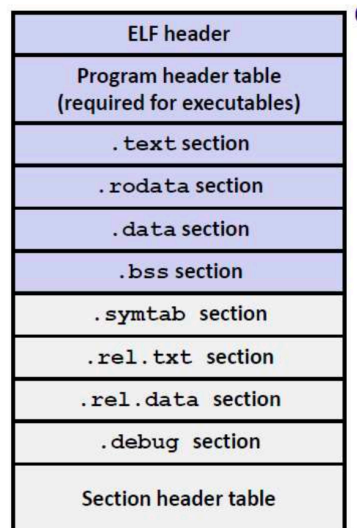


图 1. ELF 文件格式

链接是将多个目标文件和库文件组合成一个可执行文件的过程。链接可以分为两种类型：静态链接和动态链接。

静态链接(Static Linking): 在编译时, 将所有需要的库函数和目标文件组合到一个单一的可执行文件中。

动态链接(Dynamic Linking):在运行时,将库函数动态加载到内存中,并链接到可执行文件。

加载是加载器将可执行文件从磁盘加载到内存中,并准备执行的过程。包括以下步骤:

读取可执行文件头:加载器首先读取可执行文件的头部信息,以确定文件格式、入口点、段信息等。分配内存:根据可执行文件的段信息,加载器在内存中分配相应的空间,包括代码段、数据段、堆和栈。加载段:将可执行文件的各个段(如代码段、数据段)加载到分配的内存区域中。动态链接(如果使用动态链接):加载器在运行时加载动态库,并解析符号表,将库函数链接到可执行文件中。设置入口点:加载器将程序计数器(PC)设置为可执行文件的入口点地址。开始执行:加载器将控制权交给可执行文件,开始执行程序。