实例分析报告
xv6 exec

小组成员：丁元浩 张倍宁 宋嘉程
报告人：丁元浩

exec函数简介

```
int exec(char*, char**);
```

功能：使另一个可执行程序替换当前的进程

特点：

- 源进程完全由新程序代换，而新程序则从其main函数开始执行
- 调用exec并不创建新进程，所以前后的进程ID不变
- 在原进程中已经打开的文件描述符，在新进程中仍将保持打开，在原进程中已打开的目录流都将在新进程中被关闭。

用途：

- 当进程不需要再往下继续运行时，调用exec函数族中的函数让自己得以延续下去。
- 如果当一个进程想执行另一个可执行程序时，可以使用fork函数先创建一个子进程，然后通过子进程来调用exec函数从而实现可执行程序的功能。

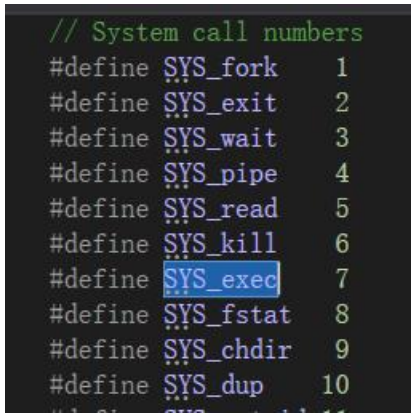
PART 01

用户态exec定义

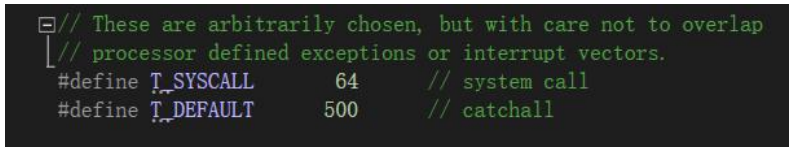
用户态exec定义

usys.S

```
#include "syscall.h"
#include "traps.h"

#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret

SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
SYSCALL(kill)
SYSCALL(exec)
SYSCALL(open)
SYSCALL(mknod)
SYSCALL(unlink)
SYSCALL(fstat)
SYSCALL(link)
SYSCALL(mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
```



syscall.h

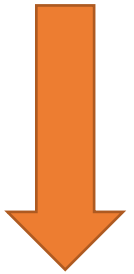


traps.h

用户态exec定义

```
#include "syscall.h"
#include "traps.h"

#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret

SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
SYSCALL(kill)
SYSCALL(exec)
SYSCALL(open)
SYSCALL(mknod)
SYSCALL(unlink)
SYSCALL(fstat)
SYSCALL(link)
SYSCALL(mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
```

```
.globl exec;
exec:
movl $SYS_exec, % eax;
int $T_SYSCALL;
ret
```



```
.globl exec;
exec:
movl $7, % eax;
int 0x40;
ret
```

与一般函数的区别

- 涉及cpu状态的切换
- 用户态调用exec，内核态执行exec

syscall

```
.globl exec;
exec:
movl $7, % eax;
int 0x40;
ret
```

用户态→内核态

```
10  int
11  exec(char *path, char **argv)
12  {
13    char *s, *last;
14    int i, off;
15    uint argc, sz, sp, ustack[3+MAXARG+1];
16    struct elfhdr elf;
17    struct inode *ip;
18    struct proghdr ph;
19    pde_t *pgdir, *oldpgdir;
20    struct proc *curproc = myproc();
21
22    begin_op();
23
24    if((ip = namei(path)) == 0){
25      end_op();
26      cprintf("exec: fail\n");
27      return -1;
28    }
29    ilock(ip);
30    pgdir = 0;
31
32    // Check ELF header
33    if(readi(ip, (char*)&elf, 0, sizeof(elf)) != sizeof(elf))
34      goto bad;
35    if(elf.magic != ELF_MAGIC)
36      goto bad;
```

PART 02

xv6中exec系统调用

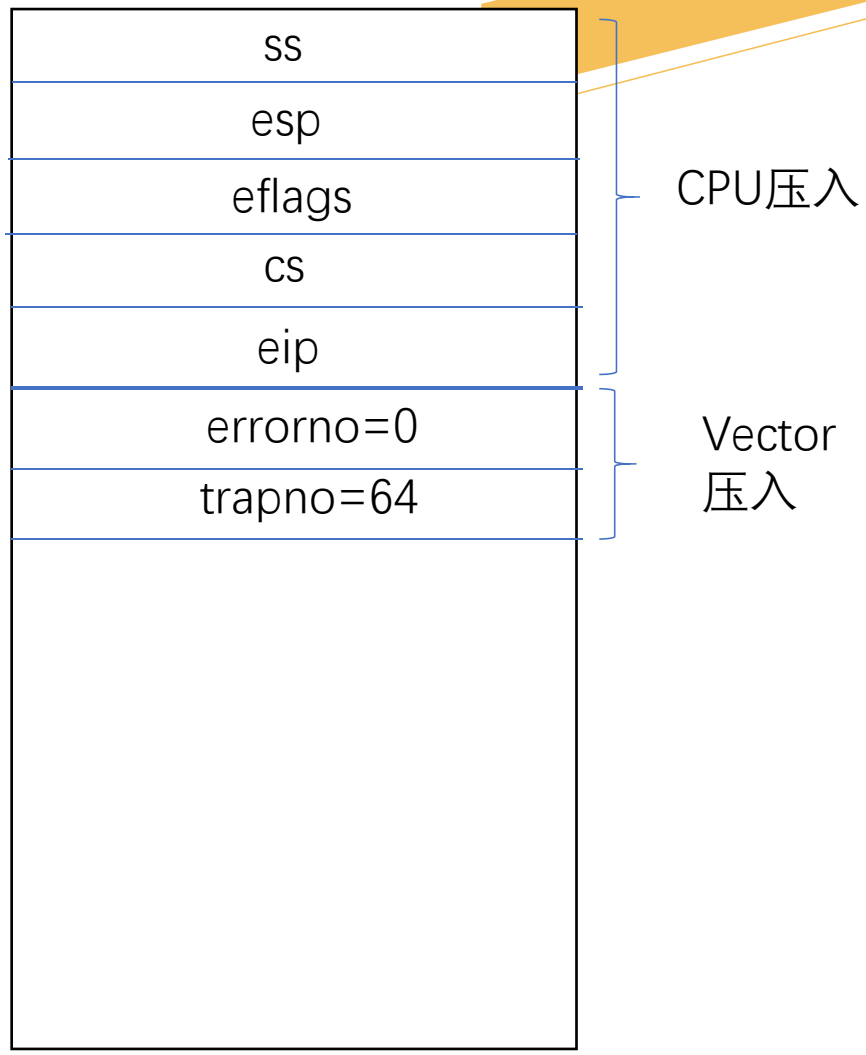
系统调用

```
.globl exec;
exec:
movl $7, % eax;
int 0x40;
ret
```

系统调用

```
# Generate vectors.S, the trap/interrupt entry points.
# There has to be one entry point per interrupt number
# since otherwise there's no way for trap() to discover
# the interrupt number.

print "# generated by vectors.pl - do not edit\n";
print "# handlers\n";
print ".globl alltraps\n";
for(my $i = 0; $i < 256; $i++){
    print ".globl vector$i\n";
    print "vector$i:\n";
    if(!($i == 8 || ($i >= 10 && $i <= 14) || $i == 17)){
        print "  pushl \$0\n";
    }
    print "  pushl \$$i\n";
    print "  jmp alltraps\n";
}

print "\n# vector table\n";
print ".data\n";
print ".globl vectors\n";
print "vectors:\n";
for(my $i = 0; $i < 256; $i++){
    print "  .long vector$i\n";
}
```

```
.globl vector64
vector64:
  pushl $0
  pushl $64
  jmp alltraps
```



通过vectors.pl编译得到vectors.S

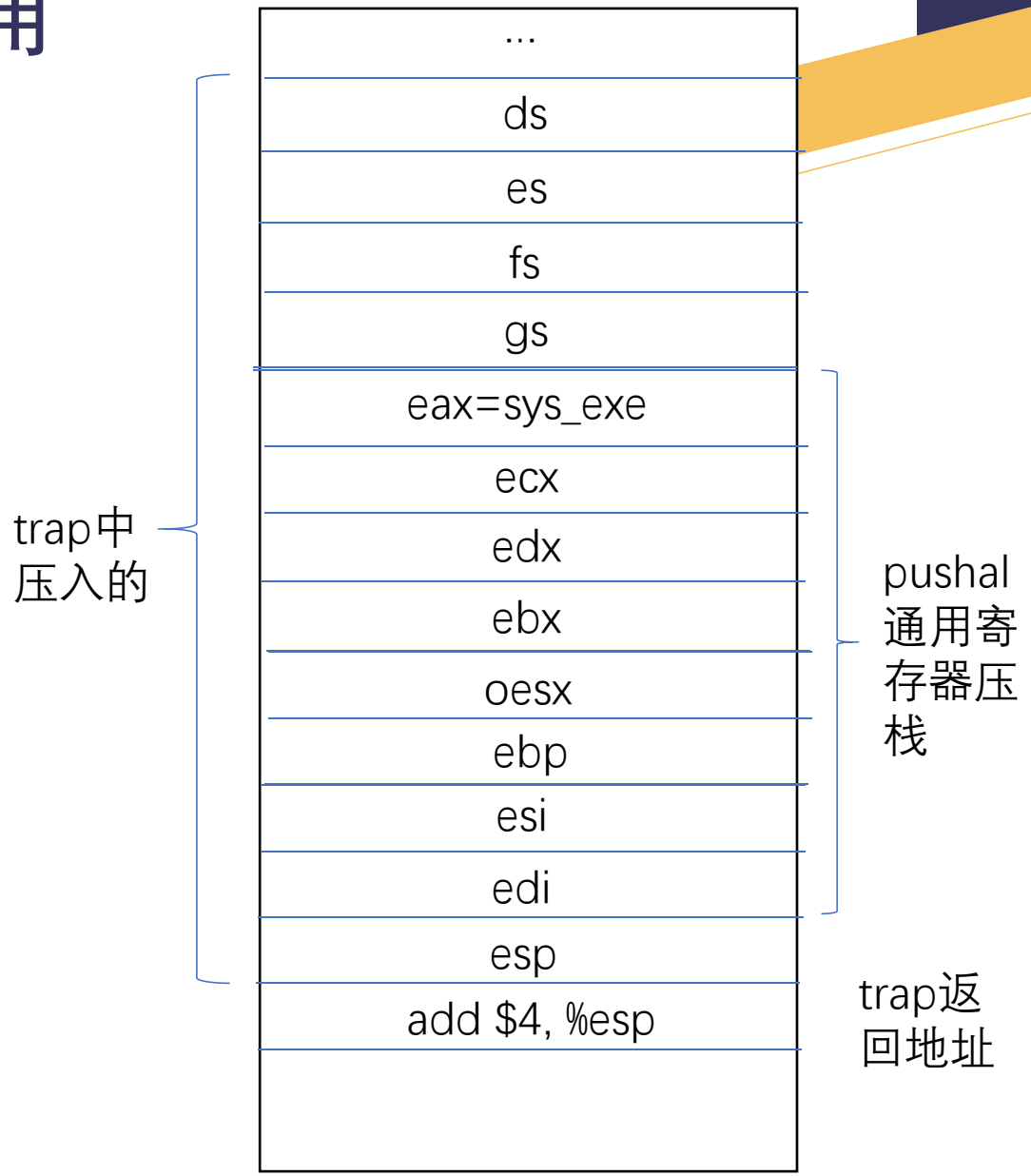
系统调用

```
#include "mmu.h"

  # vectors.S sends all traps here.
.globl alltraps
alltraps:
  # Build trap frame.
  pushl %ds
  pushl %es
  pushl %fs
  pushl %gs
  pushal

  # Set up data segments.
  movw $(SEG_KDATA<<3), %ax
  movw %ax, %ds
  movw %ax, %es

  # Call trap(tf), where tf=%esp
  pushl %esp
  call trap
  addl $4, %esp
```

……

```
# Call trap(tf), where tf=%esp
pushl %esp
call trap
addl $4, %esp
```

trapasm.S



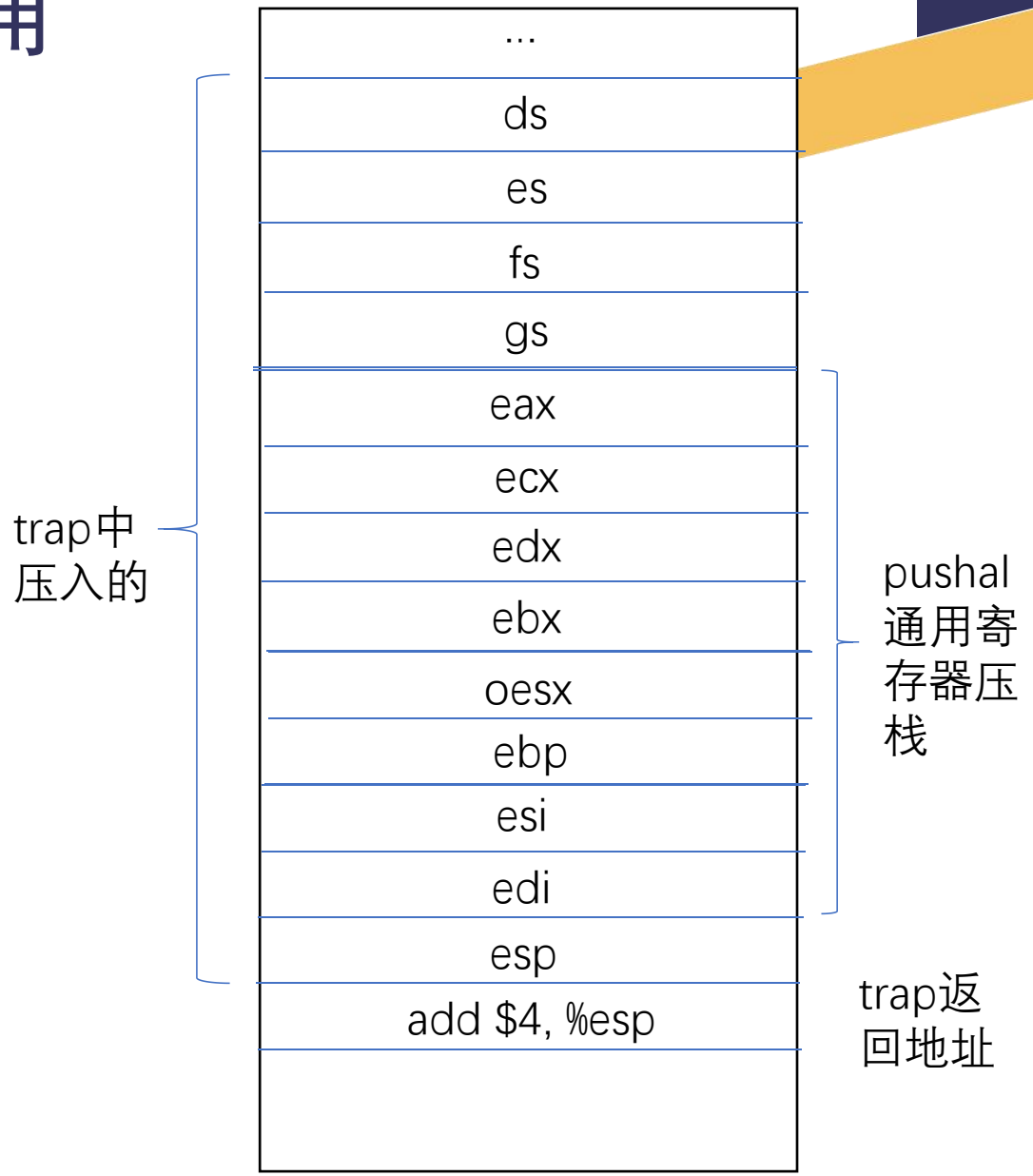
系统调用

```
# Call trap(tf), where tf=%esp
pushl %esp
call trap
addl $4, %esp
```

trapasm.S

```
//PAGEBREAK: 41
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }
```

trap.c

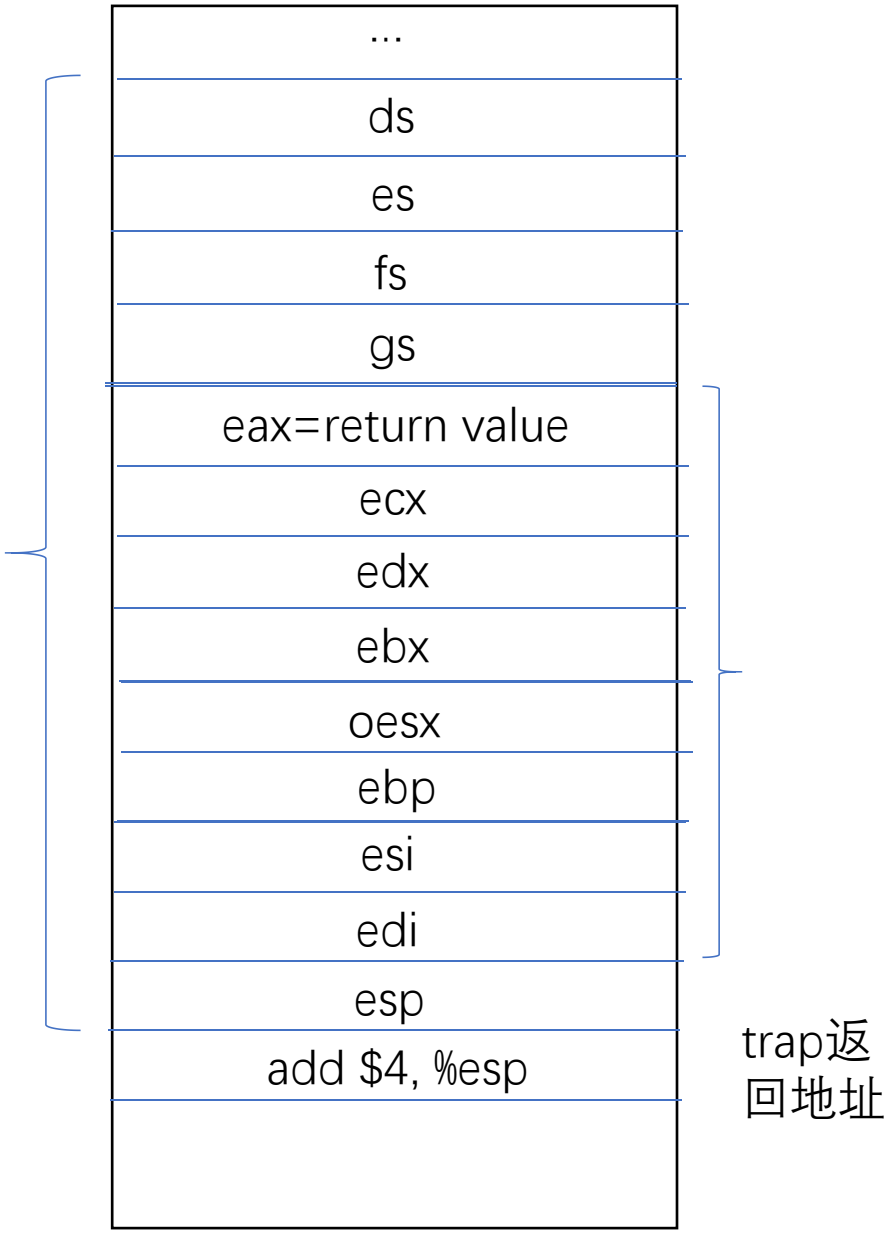


系统调用

```
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
```

```
void
syscall(void)
{
  int num;
  struct proc *curproc = myproc();

  num = curproc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->tf->eax = syscalls[num]();
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            curproc->pid, curproc->name, num);
    curproc->tf->eax = -1;
  }
}
```

syscall.c



系统调用

```
int
sys_exec(void)
{
  char *path, *argv[MAXARG];
  int i;
  uint uargv, uarg;

  if(argstr(0, &path) < 0 || argint(1, (int*)&uargv) < 0){
    return -1;
  }
  memset(argv, 0, sizeof(argv));
  for(i=0;; i++){
    if(i >= NELEM(argv))
      return -1;
    if(fetchint(uargv+4*i, (int*)&uarg) < 0)
      return -1;
    if(uarg == 0){
      argv[i] = 0;
      break;
    }
    if(fetchstr(uarg, &argv[i]) < 0)
      return -1;
  }
  return exec(path, argv);
}
```

sysfile.c

```
if(argstr(0, &path) < 0 || argint(1, (int*)&uargv) < 0){
```

```
for(i=0;; i++){
  if(i >= NELEM(argv))
    return -1;
  if(fetchint(uargv+4*i, (int*)&uarg) < 0)
    return -1;
  if(uarg == 0){
    argv[i] = 0;
    break;
  }
```

```
  if(fetchstr(uarg, &argv[i]) < 0)
    return -1;
}
```

系统调用

```
int
sys_exec(void)
{
  char *path, *argv[MAXARG];
  int i;
  uint uargv, uarg;

  if(argstr(0, &path) < 0 || argint(1, (int*)&uargv) < 0){
    return -1;
  }
  memset(argv, 0, sizeof(argv));
  for(i=0;; i++){
    if(i >= NELEM(argv))
      return -1;
    if(fetchint(uargv+4*i, (int*)&uarg) < 0)
      return -1;
    if(uarg == 0){
      argv[i] = 0;
      break;
    }
    if(fetchstr(uarg, &argv[i]) < 0)
      return -1;
  }
  return exec(path, argv);
}
```

sysfile.c

```
int
fetchint(uint addr, int *ip)
{
  struct proc *curproc = myproc();

  if(addr >= curproc->sz || addr+4 > curproc->sz)
    return -1;
  *ip = *(int*)(addr);
  return 0;
}
```

fetchint函数:取addr处数据保存在ip所指位置。

```
// Fetch the nth 32-bit system call argument.
int
argint(int n, int *ip)
{
  return fetchint((myproc()->tf->esp) + 4 + 4*n, ip);
}
```

argint函数

argint函数

- 用途：获取栈上第n个syscall参数
- 第一个参数含义：栈上第n个参数的地址
- myproc()->tp->esp指向用户栈，因为syscall是用户态程序调用系统功能时使用的
- myproc()->tp->esp+4+4*n其中“+4”：栈顶值为过程调用的返回地址，“+4*n”：第n个参数偏移量

```
// Fetch the nth 32-bit system call argument.
int
argint(int n, int *ip)
{
  return fetchint((myproc()->tf->esp) + 4 + 4*n, ip);
}
```

系统调用

```
int
exec(char *path, char **argv)
{
  char *s, *last;
  int i, off;
  uint argc, sz, sp, ustack[3+MAXARG+1];
  struct elfhdr elf;
  struct inode *ip;
  struct proghdr ph;
  pde_t *pgdir, *oldpgdir;
  struct proc *curproc = myproc();

  begin_op();

  if((ip = namei(path)) == 0){
    end_op();
    cprintf("exec: fail\n");
    return -1;
  }
  ilock(ip);
  pgdir = 0;

  // Check ELF header
  if(readi(ip, (char*)&elf, 0, sizeof(elf)) != sizeof(elf))
    goto bad;
  if(elf.magic != ELF_MAGIC)
    goto bad;

  if((pgdir = setupkvm()) == 0)
    goto bad;
```

要执行文件，首先要拿到这个文件的inode，所以首先是根据路径获取inode并加锁

这一段是使用inode把这个文件的elf读出并执行相应的检查操作

建立新的页目录

系统调用

```
// Load program into memory.
sz = 0;
for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
  if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
    goto bad;
  if(ph.type != ELF_PROG_LOAD)
    continue;
  if(ph.memsz < ph.filesz)
    goto bad;
  if(ph.vaddr + ph.memsz < ph.vaddr)
    goto bad;
  if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
    goto bad;
  if(ph.vaddr % PGSIZE != 0)
    goto bad;
  if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
    goto bad;
}
iunlockput(ip);
end_op();
ip = 0;

// Allocate two pages at the next page boundary.
// Make the first inaccessible.  Use the second as the user stack.
sz = PGROUNDUP(sz);
if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
  goto bad;
clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
sp = sz;
```

将每个程序读到页目录对应的内存空间，这里在每次循环主要有三部分：读取磁盘、分配内存、加载到内存空间，分别对应着这三个函数。至此，代码段和数据段就加载完成了

接下来是栈的加载，这里在已经分配的大小sz基础上又向后紧接着分配了两个page，第一个是不可访问的guard page，防止用户栈越界，第二个是真正的用户栈段：

系统调用

```
// Push argument strings, prepare rest of stack in ustack.
for(argc = 0; argv[argc]; argc++) {
  if(argc >= MAXARG)
    goto bad;
  sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
  if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
    goto bad;
  ustack[3+argc] = sp;
}
ustack[3+argc] = 0;

ustack[0] = 0xffffffff;  // fake return PC
ustack[1] = argc;
ustack[2] = sp - (argc+1)*4;  // argv pointer

sp -= (3+argc+1) * 4;
if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
  goto bad;

// Save program name for debugging.
for(last=s=path; *s; s++)
  if(*s == '/')
    last = s+1;
safestrcpy(curproc->name, last, sizeof(curproc->name));
```

在用户栈存储程序启动时的参数，将栈空间设为进 入程序 main( ) 函数执行的状态，设置 sp 指针：

注意这里这个ustack并不是真正的栈指针，sp才是，这个ustack只是临时创建方便结构输入的，接下来会把它copy到用户栈上：

系统调用

```
// Save program name for debugging.
for(last=s=path; *s; s++)
  if(*s == '/')
    last = s+1;
safestrcpy(curproc->name, last, sizeof(curproc->name));

// Commit to the user image.
oldpgdir = curproc->pgdir;
curproc->pgdir = pgdir;
curproc->sz = sz;
curproc->tf->eip = elf.entry;  // main
curproc->tf->esp = sp;
switchuvm(curproc);
freevm(oldpgdir);
return 0;

bad:
if(pgdir)
  freevm(pgdir);
if(ip){
  iunlockput(ip);
  end_op();
}
return -1;
}
```

获取debug信息

这里是更新进程信息、切换页表、释放原进程的所有内存，其中调用调用 switchuvm( ) 函数，使硬件按照新的页表进行用户地址空间的映射；调用 freevm( ) 函数，释放之前的页表映射的物理内存

最后是错误处理，如果exec执行过程中出错，那么就会回滚，这时候控制权还能够重新返回给调用exec的进程：

系统调用

```
curproc->tf->eip = elf.entry;  // main
curproc->tf->esp = sp;
switchuvm(curproc);
freevm(oldpgdir);
return 0;
```

exec.c

```
    return -1;
  }
  return exec(path, argv);
}
```

sysfile.c

.....

```
trapret:
  popal
  popl %gs
  popl %fs
  popl %es
  popl %ds
  addl $0x8, %esp  # trapno and errcode
  iret
```

trapasm.S

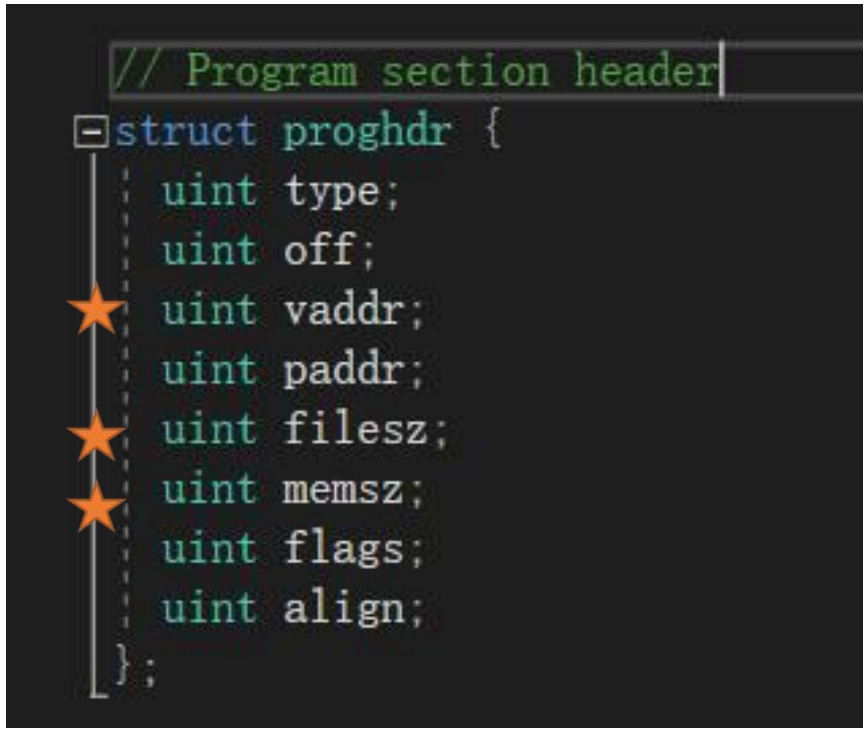
逐层返回

内核态->用户态
最终返回到exec所启动的程序的入口地址，开始执行该程序

PART 03

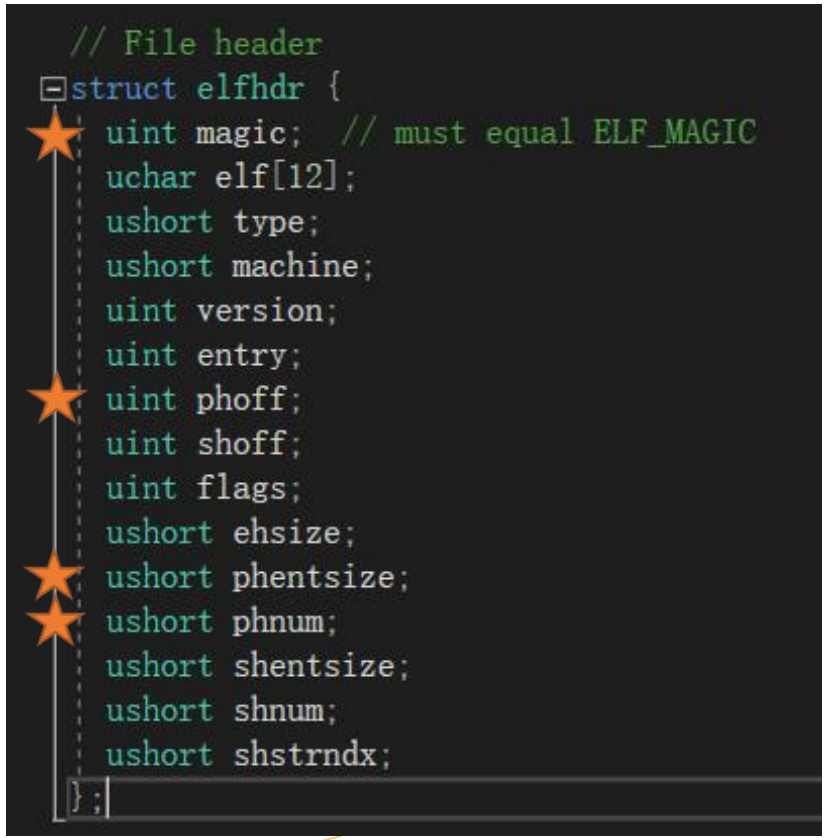
exec.c中的数据结构

elf.h



- vaddr 代表 segment 将要加载在虚存中的起始地址
- filesz 代表 segment 在 ELF 文件中的大小
- memsz 代表 segment 被加载到内存所需的大小

elf.h



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

- magic 代表 ELF 文件开头的 4 字节 magic number， 用于核对文件是否为 ELF 文件
- phoff 代表第一个 Program Header Table 在 ELF 文件中 的偏移位置
- phentsize 代表每一个 Program Header Table 的大小
- phnum 代表 ELF 文件中 Program Header Table 的数量 每个 Program Header Table 对应描述一个segment

PART 04

Linux中elf文件格式，以及链接和加载的机制。

ELF文件头格式

定义于/usr/include/elf.h：

```
typedef struct
{
  unsigned char e_ident[EI_NIDENT];     /* Magic number and other info */
  Elf64_Half    e_type;                 /* Object file type */
  Elf64_Half    e_machine;              /* Architecture */
  Elf64_Word    e_version;              /* Object file version */
  Elf64_Addr    e_entry;                /* Entry point virtual address */
  Elf64_Off     e_phoff;                /* Program header table file offset */
  Elf64_Off     e_shoff;                /* Section header table file offset */
  Elf64_Word    e_flags;                /* Processor-specific flags */
  Elf64_Half    e_ehsize;               /* ELF header size in bytes */
  Elf64_Half    e_phentsize;            /* Program header table entry size */
  Elf64_Half    e_phnum;                /* Program header table entry count */
  Elf64_Half    e_shentsize;            /* Section header table entry size */
  Elf64_Half    e_shnum;                /* Section header table entry count */
  Elf64_Half    e_shstrndx;             /* Section header string table index */
} Elf64_Ehdr;
```

readelf命令解析结果

使用readelf命令读取可执行文件的elf头：

```
hollowdaze@ubuntu:~/tryal$ readelf -h test
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x400600
  Start of program headers:          64 (bytes into file)
  Start of section headers:          6864 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         9
  Size of section headers:           64 (bytes)
  Number of section headers:         31
  Section header string table index: 28
```

ELF文件的运行

为什么linux可以运行ELF文件?
内核对所支持的每种可执行的程序类型都有个struct linux_binfmt的数据结构，其定义如下:
(in include/linux/binfmts.h)

```
/*
 * 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 *);
        int (*load_shlib)(struct file *);
        int (*core_dump)(struct coredump_params *cprm);
        unsigned long min_coredump;     /* minimal dump size */
};
```

所有的linux_binfmt对象都处于一个链表中，当我们执行一个可执行程序的时候，内核会尝试遍历所有注册的linux_binfmt对象，对其调用load_binary方法来尝试加载，直到加载成功为止。具体到ELF文件而言，加载方法便是load_elf_binary。

load_elf_binary

简单来说，流程可以分为：

1. 读取并检查目标可执行程序的头信息, 检查完成后加载目标程序的程序头表

2. 如果动态加载则读取并检查解释器的头信息, 检查完成后加载解释器的程序头表

3. 装入目标程序的段segment, 这些才是目标程序二进制代码中的真正可执行映像

4. 填写程序的入口地址(如果有解释器则填入解释器的入口地址, 否则直接填入可执行程序的入口地址)

5. create_elf_tables填写目标文件的参数环境变量等必要信息

6. start_kernel宏准备进入新的程序入口

动态加载

从编译/链接和运行的角度看，应用程序和库程序的链接加载有两种方式：

静态加载：把需要用到的库函数的目标代码（二进制）代码从程序库中抽取出来，链接进应用软件的目标映像中；

动态加载：库函数的代码并不进入应用软件的目标映像，应用软件在编译/链接阶段并不完成跟库函数的链接，而是把函数库的映像也交给用户，到启动应用软件目标映像运行时才把程序库的映像也装入用户空间（并加以定位），再完成应用软件与库函数的连接。

动态加载中，程序在被内核加载到内存，内核跳到用户空间后并不是直接执行目标程序的，而是先把控制权交到用户空间的解释器，由解释器加载运行用户程序所需要的动态库（比如libc等等），然后控制权才会转移到用户程序。例如不加static标签的gcc编译

动态加载

内核工作:
1.读取ELF文件头部等，得知各段或节的地址及标识，把可加载段加载到内存中。
2.找到对应动态链接器的名称，并进行设置一些标记以指示动态链接器
3.控制权传给动态链接器

动态连接器工作:
4.检查共享库的依赖性，必要时加载该库
5.对于外部引用的函数、参数等进行重定位。
6.执行在ELF文件中标记为.init的节的代码，进行程序运行的初始化。
7.把控制权交给程序，从ELF头文件指定的entry_point开始执行程序。

文献来源：

- https://blog.csdn.net/zjwson/article/details/53337212
- https://wangdh15.github.io/2020/12/22/xv6%E6%BA%90%E7%A0%81%E5%89%96%E6%9E%90%E4%B9%8B%E8%BF%9B%E7%A8%8B%E3%80%81%E7%B3%BB%E7%BB%9F%E8%B0%83%E7%94%A8/#more
- https://blog.csdn.net/weixin_39616565/article/details/111860817
- https://zhuanlan.zhihu.com/p/365217605
- https://www.cnblogs.com/qingergege/p/6601807.html
- https://blog.csdn.net/amoscykl/article/details/80354052
- https://blog.csdn.net/gatieme/article/details/51628257

THANKS