

Interruption and System Calls in Xv6 Operating System

A Focus on Implementation Details

何昊 1600012742

“The devil is in the details”

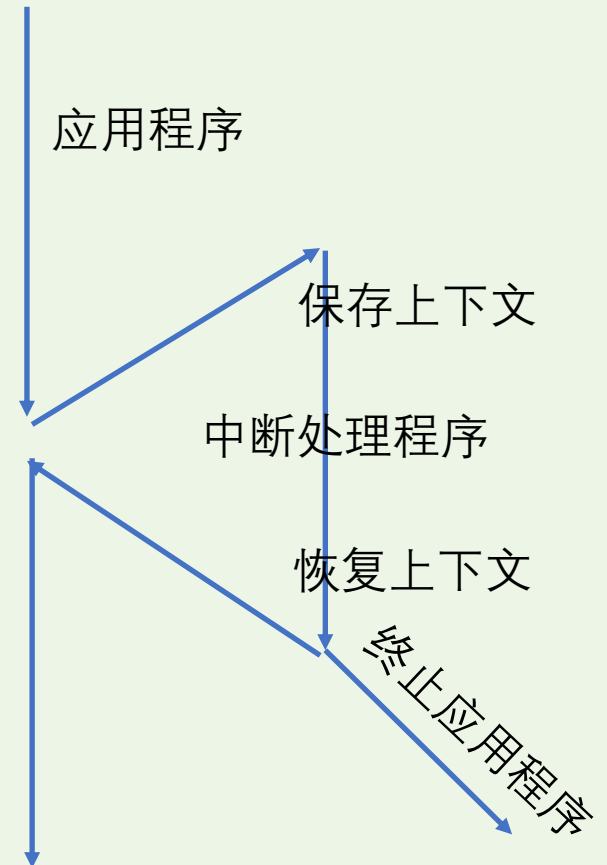
English proverb

Outline

- 知识回顾
 - 基本概念复习
 - X86中的寻址模式
 - 中断描述符表（Interruption Descriptor Table, IDT）
 - 相关X86指令：STI, CLI, INT, LIDT
- Xv6中的中断
 - 相关数据结构
 - 代码的组织 and 执行流
 - 中断调用举例：除零错误
- Xv6中的系统调用
 - 相关数据结构
 - 代码的组织 and 执行流
 - 如何实现一个系统调用

知识回顾 – 基本概念

- 中断与系统调用是操作系统实现**异常控制流**的方式
- 中断是指体系结构**响应内部或外部事件**的机制
 - 系统受到了某种信号，打断了目前执行的应用程序的执行流，进入相应的中断处理程序，在程序中完成对此信号的事件处理，并返回原来的程序执行流
 - 外部中断：由诸如时钟、DMA控制器、鼠标键盘、电源等硬件引发的中断
 - 内部中断：由于中断指令/指令出错等原因引发的中断
 - 需要**软硬件的紧密协同**，涉及大量体系结构细节，代码可能难以理解

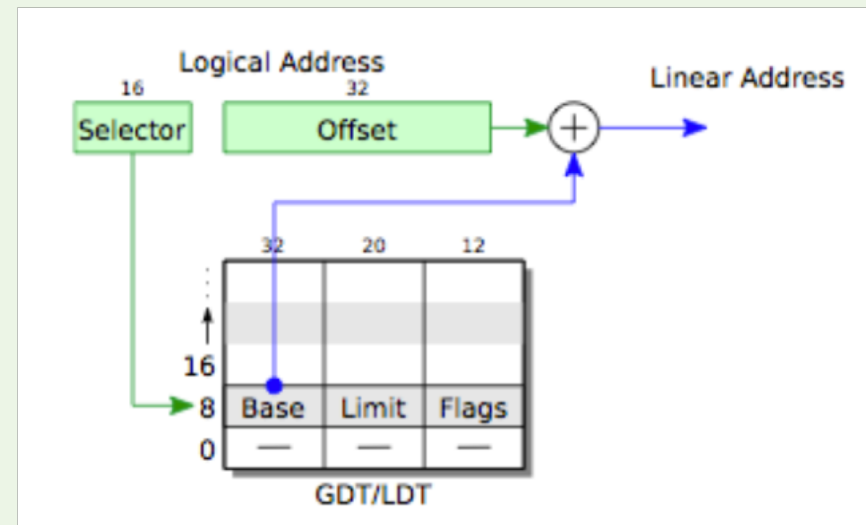


知识回顾 – 基本概念

- 系统调用是指一种应用程序请求操作系统的某种服务的机制
 - 使用中断机制实现，用户程序使用INT指令主动陷入中断，在特定的中断处理程序内实现系统调用的功能
 - 在Xv6系统中，中断与系统调用的代码执行流是一样的，只有参数不一样
 - Xv6系统使用Trap来代指所有由INT指令引发的中断与系统调用
- 在Xv6中允许应用程序请求的系统调用有21种，涵盖以下三类：
 - 进程管理相关：fork、exit、wait、kill、exec、getpid等
 - 输入输出相关：read、write、pipe、dup等
 - 文件管理相关：fstat、open、close、chmod、link、unlink、chdir、mkdir等

知识回顾 – X86中的寻址模式

- X86体系结构中有两种寻址模式：实模式和保护模式
 - 实模式继承自Intel 8086，我们不讨论实模式
 - 保护模式始于Intel 80286，地址空间为32位，采用段+偏移的寻址模式。在逻辑上将地址空间分为代码段、数据段、栈段等等。
 - 关于地址段的信息存储在全局描述符表（GDT, Global Descriptor Table）中，需要由操作系统代码来初始化，段寄存器CS、DS、SS、ES存储GDT表的索引。
 - Xv6系统中设置的GDT项非常简单

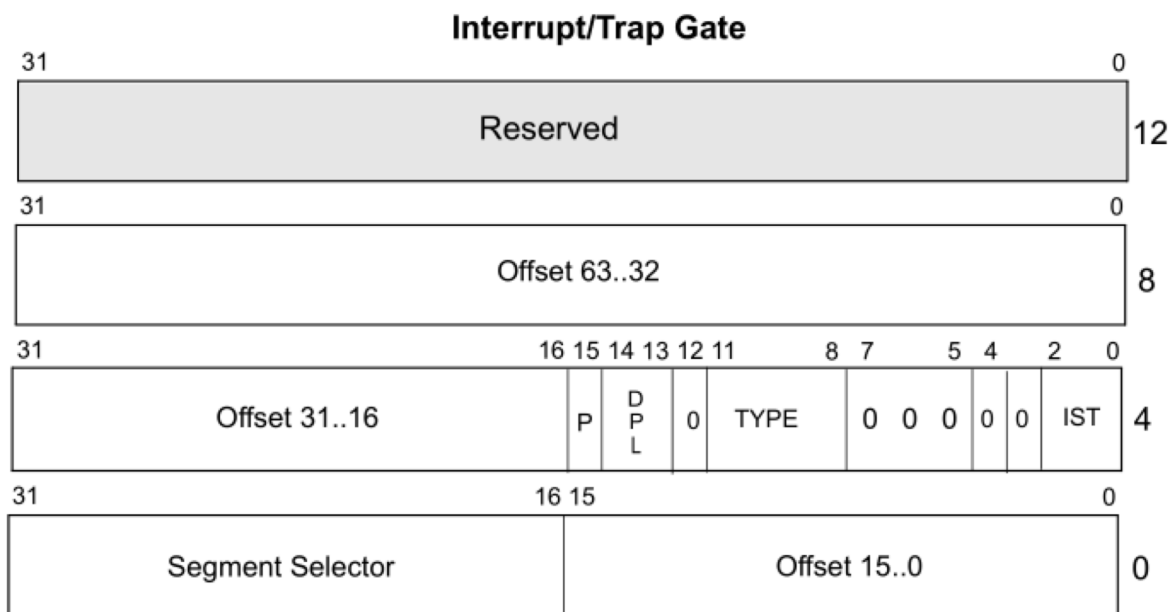


保护模式中段+偏移的寻址过程
Image from Xv6 Documentation

知识回顾 – x86体系结构中的中断

- 中断的触发方式有以下两种
 - 硬件触发
 - 软件显式地调用INT指令
- 每个中断类型有唯一的中断号
- x86体系结构使用中断描述符表来存储中断处理程序的地址
 - 简称IDT, Interrupt Descriptor Table
 - 操作系统必须在某处初始化IDT的值以及指向IDT的寄存器
 - 当中断发生时, 硬件会直接使用中断号作为数组下标取出对应的中断描述符

知识回顾 – 中断描述符表和中断类型举例



DPL Descriptor Privilege Level
 Offset Offset to procedure entry point
 P Segment Present flag
 Selector Segment Selector for destination code segment
 IST Interrupt Stack Table

中断调用号	对应事件
0	Division By Zero
1	Debug
2	Non Maskable Interrupt
3	Breakpoint
12	General Protection Fault
14	Page Fault
64	System Call

Image Courtesy:

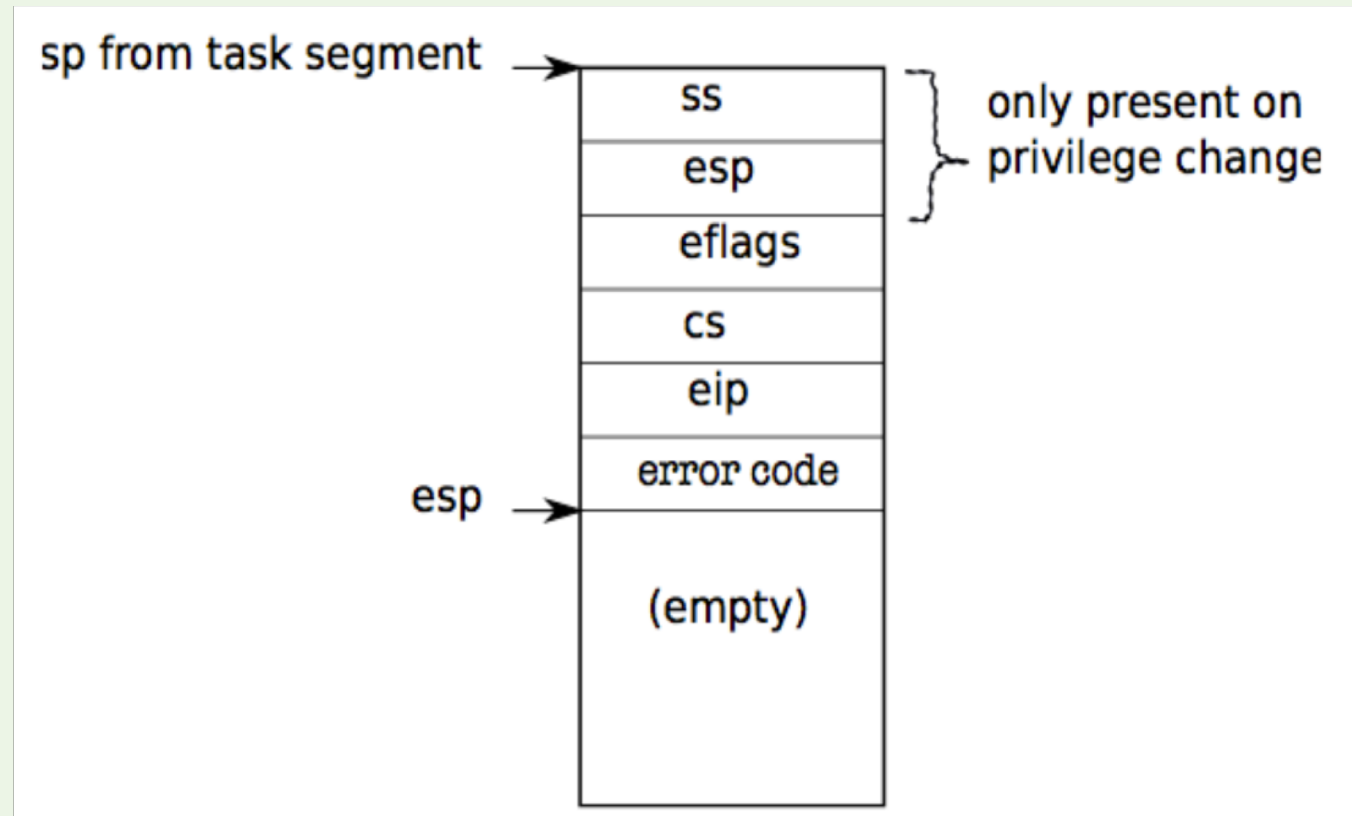
https://codemachine.com/article_interruptdispatching.html

知识回顾 – X86中的中断相关指令

- **INT N**
 - 根据调用号N触发对应的中断
 - Xv6中INT 64对应系统调用
 - 系统调用的参数存在用户进程的栈里
- **CLI**
 - 设置EFLAG寄存器内的IF为0，屏蔽处理器接受中断
- **STI**
 - 设置EFLAG寄存器内的IF为1，允许处理器接受中断
- **LIDT ADDR**
 - 设置IDTR寄存器为中断描述符表的地址

INT N指令执行的具体操作

- 从 IDT 中获得第 n 个描述符, n 就是 `int` 的参数。
- 检查 `%cs` 的域 $CPL \leq DPL$, DPL 是描述符中记录的特权级。
- 如果目标段选择符的 $PL < CPL$, 就在 CPU 内部的寄存器中保存 `%esp` 和 `%ss` 的值。
- 从一个任务段描述符中加载 `%ss` 和 `%esp`。
- 将 `%ss` 压栈。
- 将 `%esp` 压栈。
- 将 `%eflags` 压栈。
- 将 `%cs` 压栈。
- 将 `%eip` 压栈。
- 清除 `%eflags` 的一些位。
- 设置 `%cs` 和 `%eip` 为描述符中的值。



Kernel Stack after an INT instruction

X86中的中断硬件APIC

- APIC全称Advanced Programmable Interrupt Controller
- 目前大多数x86 CPU使用的中断控制器标准
- 在APIC体系结构中，每个CPU核对应一个Local APIC，另有一个或多个I/O APIC
- 中断发送给I/O APIC后，被转发给对应的Local APIC
- 由操作系统设置APIC的行为

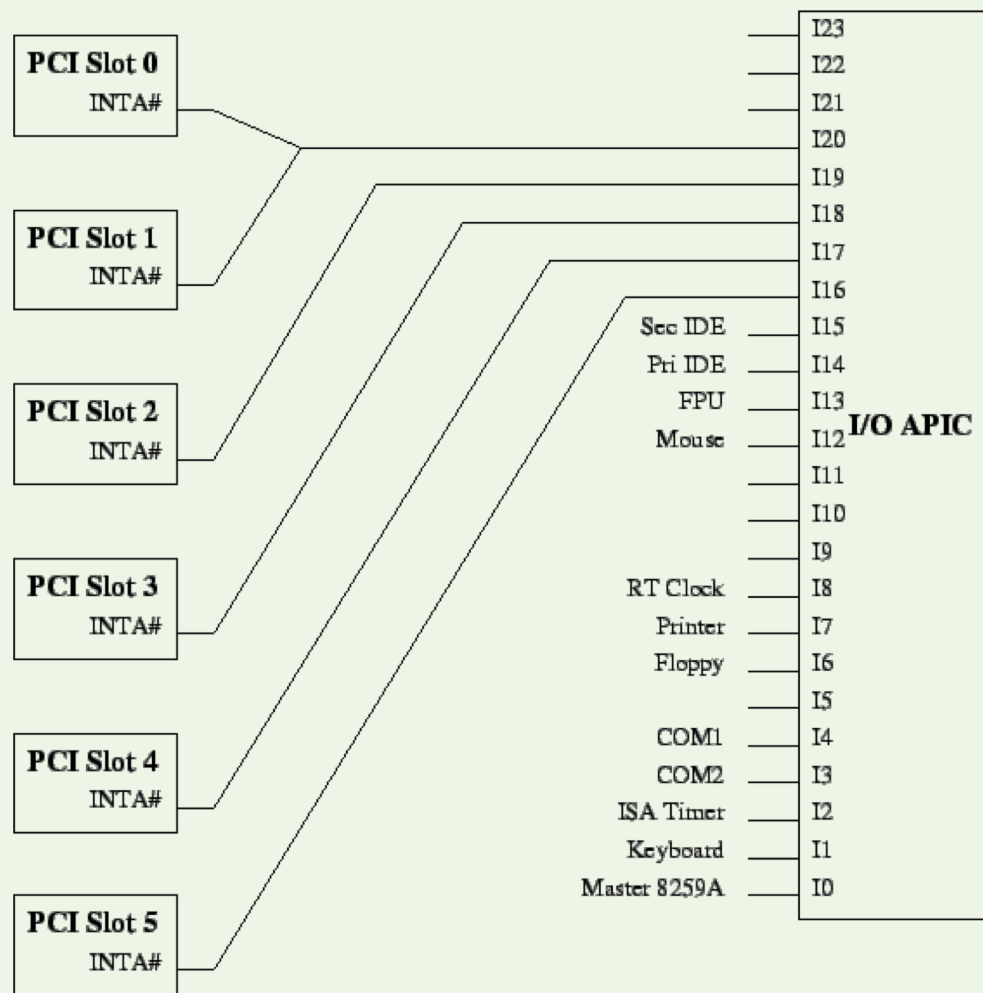


Image courtesy: <https://people.freebsd.org/~jhb/papers/bsdcan/2007/article/node4.html>

Outline

- 先置知识回顾
 - 基本概念复习
 - X86中的寻址模式
 - 中断描述符表 (Interruption Descriptor Table, IDT)
 - 相关X86指令: STI, CLI, INT, LIDT
- Xv6中的中断
 - 相关数据结构
 - 代码的组织 and 执行流
 - 中断调用举例: 除零错误
- Xv6中的系统调用
 - 相关数据结构
 - 代码的组织 and 执行流
 - 如何实现一个系统调用

Xv6中断相关代码的组织结构

文件名	内容
traps.h	中断相关的宏定义
vectors.S	<ol style="list-style-type: none">1. 必须通过vector.pl脚本生成2. 中断的入口，包含全部的256个中断处理函数3. 跳转到trapasm.S
trapasm.S	<ol style="list-style-type: none">1. 两个函数：中断的入口(alltraps)和返回(trapret)2. 负责上下文的保存与恢复3. 中断入口跳转到trap.c内的trap()函数
trap.c	<ol style="list-style-type: none">1. 中断相关的数据结构2. 初始化中断的函数tvinit()和idtinit()3. 真正的中断与系统调用的处理函数trap()

中断相关数据结构

trap.h

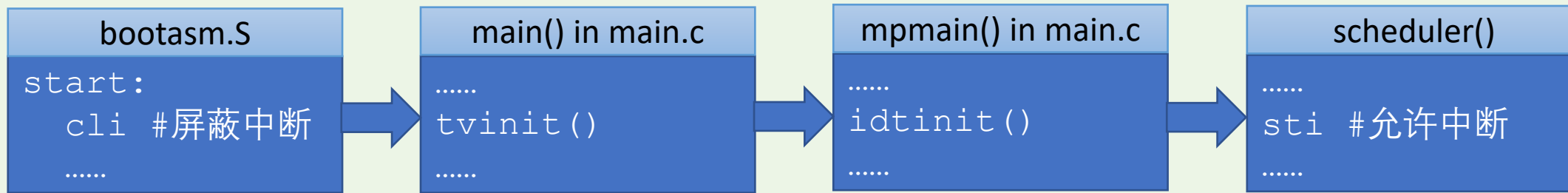
比较重要的域是地址（段寄存器+偏移）和权限级别

```
// Gate descriptors for interrupts and traps
struct gatedesc {
    uint off_15_0 : 16; // low 16 bits of offset in segment
    uint cs : 16;       // code segment selector
    uint args : 5;     // # args, 0 for interrupt/trap gates
    uint rsv1 : 3;     // reserved(should be zero I guess)
    uint type : 4;     // type(STS_{IG32,TG32})
    uint s : 1;       // must be 0 (system)
    uint dpl : 2;     // descriptor(meaning new) privilege level
    uint p : 1;       // Present
    uint off_31_16 : 16; // high bits of offset in segment
};
```

trap.c

```
// Interrupt descriptor table (shared by all CPUs).
struct gatedesc idt[256];
extern uint vectors[]; // in vectors.S: array of 256 entry pointers
```

中断的初始化



```
trap.c
```

```
void tvinit(void) {  
    int i;  
    for(i = 0; i < 256; i++)  
        SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);  
    SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);  
    initlock(&tickslock, "time");  
}  
void idtinit(void) {  
    lidt(idt, sizeof(idt));  
}
```

注意在这里，任何用户程序使用INT 64以外的中断指令都是非法的，因为只有系统调用的描述符表的权限是DPL_USER

这里开始才真正让硬件得知中断描述表的存在

中断举例 – 除零错误

应用程序执行流

DIV 0

另一个应用程序

CPU 发现除零后，执行一系列压栈操作，然后跳转到IDT[0]处的程序

调度算法选择了另一个进程执行

```
vector.S
vector0:
    pushl $0
    pushl $0
    jmp alltraps
```

```
trapasm.S
alltraps:
    把寄存器压栈
    设置内核数据段
    pushl %esp
    call trap
```

```
trap.c
void trap(struct trapframe *tf)
{
    先处理系统调用
    再检查是不是外部硬件中断
    如果都不是：
        如果是内核态下触发除零错误：
            系统停机
        如果是用户态下触发除零错误：
            myproc()->killed = 1;
}
```

在适当的时候调用Exit()

```
trap.c
void exit(void) {
    进行一些进程管理的操作
    curproc->state = ZOMBIE;
    sched();
}
```


Outline

- 先置知识回顾
 - 基本概念复习
 - X86中的寻址模式
 - 中断描述符表 (Interruption Descriptor Table, IDT)
 - 相关X86指令: STI, CLI, INT, LIDT
- Xv6中的中断
 - 相关数据结构
 - 代码的组织 and 执行流
 - 中断调用举例: 除零错误
- Xv6中的系统调用
 - 相关数据结构
 - 代码的组织 and 执行流
 - 如何实现一个系统调用

系统调用相关文件

文件名	内容
syscall.h	21个系统调用的宏定义
trap.c	1. 在中断处理函数内trap()处理系统调用 2. 如果是系统调用， trap()会跳转到syscall.c内的syscall()函数
syscall.c	1. 系统调用参数提取函数 2. syscall()函数用于跳转到对应的系统调用的处理函数
sysproc.c sysfile.c	包含系统调用的实现， 按类别分为两个文件 会大量调用其他内核函数
user.h	声明了供用户进程使用的系统调用函数原型
usys.S	用户使用的系统调用进程的函数实现

系统调用相关数据结构和函数

syscall.c

```
static int (*syscalls[])(void) = {
[SYS_fork] sys_fork,
[SYS_exit] sys_exit,
[SYS_wait] sys_wait,
[SYS_pipe] sys_pipe,
[SYS_read] sys_read,
[SYS_kill] sys_kill,
[SYS_exec] sys_exec,
[SYS_fstat] sys_fstat,
[SYS_chdir] sys_chdir,
[SYS_dup] sys_dup,
[SYS_getpid] sys_getpid,
[SYS_sbrk] sys_sbrk,
.....
};
```

syscall.c

```
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;
    }
}
```

系统调用号在原进程的eax寄存器中

用户使用的系统调用接口

user.h

```
// system calls
int fork(void);
int exit(void) __attribute__((noreturn));
int wait(void);
int pipe(int*);
int write(int, const void*, int);
int read(int, void*, int);
int close(int);
int kill(int);
int exec(char*, char**);
int open(const char*, int);
int mknod(const char*, short, short);
.....
```

usys.S

```
#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)
.....
```

如何实现一个系统调用 – 以setrlimit为例

- `setrlimit`是Linux的一个系统调用，用于设置进程资源使用限制
- 实现系统调用需要完成的事情
 - 在`syscall.h`内添加新的系统调用定义`sys_setrlimit()`
 - 在`syscall.c`的指针数组内添加新的系统调用函数指针`sys_setrlimit()`
 - 在`sysproc.c`中声明并实现这个函数
 - 在`user.h`内声明`setrlimit`函数的用户调用接口`setrlimit()`
 - 在`usys.S`内实现这个接口`setrlimit()`

1. 添加新的系统调用定义sys_setrlimit()

syscall.h

```
// System call numbers
#define SYS_fork 1
#define SYS_exit 2
#define SYS_wait 3
#define SYS_pipe 4
#define SYS_read 5
.....
#define SYS_link 19
#define SYS_mkdir 20
#define SYS_close 21
#define SYS_setrlimit 22
```

2. 在数组内添加新的系统调用函数指针

syscall.c

```
static int (*syscalls[])(void) = {
[SYS_fork] sys_fork,
[SYS_exit] sys_exit,
[SYS_wait] sys_wait,
[SYS_pipe] sys_pipe,
[SYS_read] sys_read,
[SYS_kill] sys_kill,
[SYS_exec] sys_exec,
[SYS_fstat] sys_fstat,
.....
[SYS_link] sys_link,
[SYS_mkdir] sys_mkdir,
[SYS_close] sys_close,
[SYS_setrlimit] sys_setrlimit,
};
```

3. 实现新的系统调用函数

sysproc.c

```
// return how many clock tick interrupts have occurred  
// since start.  
int sys_uptime(void) {  
    uint xticks;  
    acquire(&tickslock);  
    xticks = ticks;  
    release(&tickslock);  
    return xticks;  
}  
  
int sys_setrlimit(void) {  
// Extract arguments from trap frame  
// Set the maximum memory for a process, etc  
}
```


4. 添加系统调用函数的用户接口

user.h

```
// system calls
int fork(void);
int exit(void) __attribute__((noreturn));
.....
int uptime(void);
int setrlimit(int resource, const struct rlimit *rlim);
```

usys.S

```
SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
.....
SYSCALL(uptime)
SYSCALL(setrlimit)
```

Reference

- 《深入理解计算机系统》 第三版
- 《操作系统概念》 第七版
- X86指令在线手册, <https://x86.puri.sm>
- Xv6中文文档, <https://th0ar.gitbooks.io/xv6-chinese/content/>
- Various Articles from Wikipedia: <https://en.wikipedia.org/>
- Various Articles from OS Dev Wiki: <https://wiki.osdev.org/>
- Linux Man Page, <https://linux.die.net/man/2/setrlimit>

Thanks!

Download: <https://hehao98.github.io/files/Xv6中断与系统调用.pdf>