XV6 源码阅读报告

姓名： 翁嘉进
学号： 1801220016
日期： 2019/06/01

目 录

概述

xv6 是是 MIT 6.828 课程中使用的教学操作系统，是在现代硬件上对 Unix V6 系统的重写。xv6 在一定程度上遵守 v6 的结构和风格，但它是用 ANSI C 实现的，并且是基于 x86 多核处理器的。

通过学习 xv6 操作系统来强化理解操作系统的概念，XV6 操作系统提供 Unix 操作系统中的基本接口，同时模仿 Unix 的内部设计。Unix 里机制结合良好的窄接口提供了令人吃惊的通用性。这样的接口设计非常成功，使得包括 BSD，Linux，Mac OS X，Solaris 都有类似 Unix 的接口。理解 xv6 是理解这些操作系统的一个良好开始。

本文为阅读 XV6 源码的学习汇报。学习过程中，利用了 XV6 学习手册、MIT 6.828 课程录像以及一些网络分享的博文。

MIT 6.828 的课程网站: https://pdos.csail.mit.edu/6.828/。

XV6 操作系统有官方文档(中文版) : https://th0ar.gitbooks.io/xv6-chinese/content/。

第 1 章　进程线程

一、源码阅读

阅读要求：

1. 基本头文件：types.h param.h memlayout.h defs.h x86.h asm.h mmu.h elf.h
2. 进程线程部分：vm.c proc.h proc.c swtch.S kalloc.c 以及相关其他文件代码

基本头文件阅读：

这里主要介绍一下基本头文件的功能：

type.h 文件，通过 typedef 定义了数据类型的别名；

param.h 文件，通过#define 定义了特殊参数的具体值，如系统最大进程数为 64、系统最大打开文件数为 100、每个进程的打开文件上限为 16 个等；

memlayout.h 文件，通过#define 定义了内存的布局；

defs.h 文件，不仅声明了其他文件所需的数据结构，如 file（文件）、inode（i 节点）、context（上下文场景），还声明了一些基础的系统方法，如 bread 和 bweite 字节读写操作、系统调用、console 调试等。

x86.h 文件，其主要功能为让 C 代码使用特殊的 x86 指令。

asm.h 文件，通过汇编宏创建 x86 扇区，并定义扇区容量为 4Kb。同时，也定义了可执行扇区、可读扇区以及可写入扇区。

mmu.h 文件，这个文件包含 x86 内存管理单元(MMU)的定义。

elf.h 文件，定义了 elf header，即 elf 可执行文件头。同时，也定义进程空间的头部结构。

以上，就是这些基础头文件的主要功能，以及涉及的方面。它们是 XV6 操作系统实现的底层基础。

基本头文件阅读：

进程是操作系统资源分配的最小单位，线程是操作系统中调度的最小单位。进程是指一个正在运行的程序的实例，而线程是一个 CPU 指令执行流的最小单位。

XV6 系统中只实现了进程，并没有提供对线程的额外支持，一个用户进程永远只会有一个用户可见的执行流。对于每一个进程都存在对应的唯一的进程控制块（PCB），在 XV6 操作系统中，其定义在 proc.h 文件中。如下图所示，PCB 中包含了进程的相关信息，如进程 ID、

占用内存大小、内核栈底指针、进程状态等

```
struct proc {
  uint sz;                     // 占用内存大小
  pde_t* pgdir;                // 所属页表
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // 进程状态
  int pid;                     // 进程 ID
  struct proc *parent;         // 父进程
  struct trapframe *tf;        // 当前系统调用陷入的存储
  struct context *context;     // 上下文信息
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // 若非0，则说明进程已被kill
  struct file *ofile[NOFILE];  // 打开文件
  struct inode *cwd;           // 当前目录-inode
  char name[16];               // 进程名 (debugging)
};
```

此外，还包括进程所调用的文件对应打开文件和所属页表。同时，当进程进行切换时，上下文信息被存储至 context 中。

XV6 操作系统还定义了 ptable 数据结构，用于存储和管理所有进程。其定义于 proc.c 文件中，除了互斥锁 lock 之外， XV6 系统中允许同时存在的进程数量是有上限的。NPROC 为 64，即 XV6 最多只允许同时存在 64 个进程。该参数定义在 param.h 中，可修改。

同时，在 proc.c 文件中，还定义进程的基本操作：

- Fork 操作，用于创建新的进程；
- Exit 操作，用于退出进程，但该进程并未销毁，直到其父进程通过 wait()检测到该进程已退出；
- Wait 操作，用于等待子进程退出，并返回其进程 ID。若返回-1，则说明该进程无子进程；
- Yield 操作，用于使当前进程放弃 cpu，开始下一轮调度；
- ForkRet 操作，用于当一个新创建的进程第一次被调度时，释放 Ptable 的锁，并初始化用户进程空间；
- Sleep 操作，用于当前进程转化为 SLEEPING 状态，并调用 sched 以释放 CPU；
- Wakeup 操作，用于寻找一个睡眠状态的进程并把它标记为 RUNNABLE；
- Kill 操作，用于杀死进程。

二、讨论总结

什么是进程，什么是线程？操作系统的资源分配单位和调度单位分别是什么？？XV6 中的

进程和线程分别是什么，都实现了吗？

从抽象的意义来说，进程是指一个正在运行的程序的实例，而线程是一个 CPU 指令执行流的最小单位。进程是操作系统资源分配的最小单位，线程是操作系统中调度的最小单位。

从实现的角度上讲，XV6 系统中只实现了进程， 并没有提供对线程的额外支持，一个用户进程永远只会有一个用户可见的执行流。

进程管理的数据结构是什么？在 Windows，Linux，XV6 中分别叫什么名字？其中包含哪些内容？操作系统是如何进行管理进程管理数据结构的？它们是如何初始化的？

进程管理的数据结构被叫做进程控制块(Process Control Block, PCB)。一个进程的 PCB 必须存储以下两类信息：

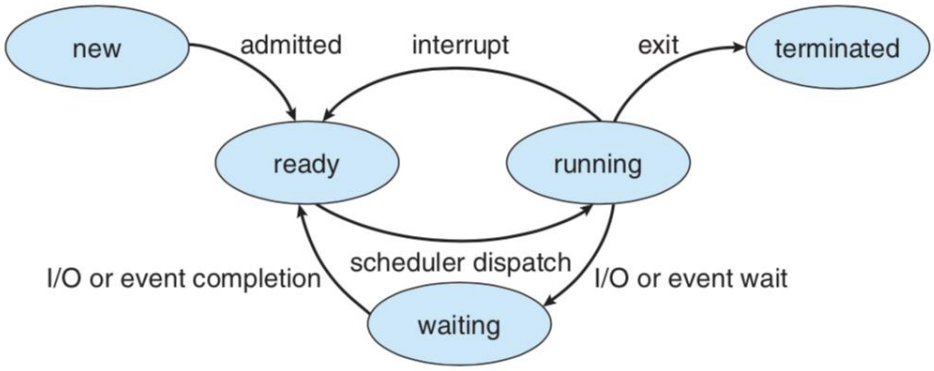
1. 操作系统管理运行的进程所需要信息，比如优先级、进程 ID、进程上下文等
2. 一个应用程序运行所需要的全部环境，比如虚拟内存的信息、打开的文件等。

在 Windows NT 以后的 Windows 系统中，进程用 EPROCESS 对象表示，线程用 ETHREAD 对象表示。Linux 系统的实现中并不刻意区分进程和线程，而是将其一概存储在被称作 task_struct 的数据结构中。在 XV6 操作系统中，与进程有关信息定义在 proc 的数据结构中。

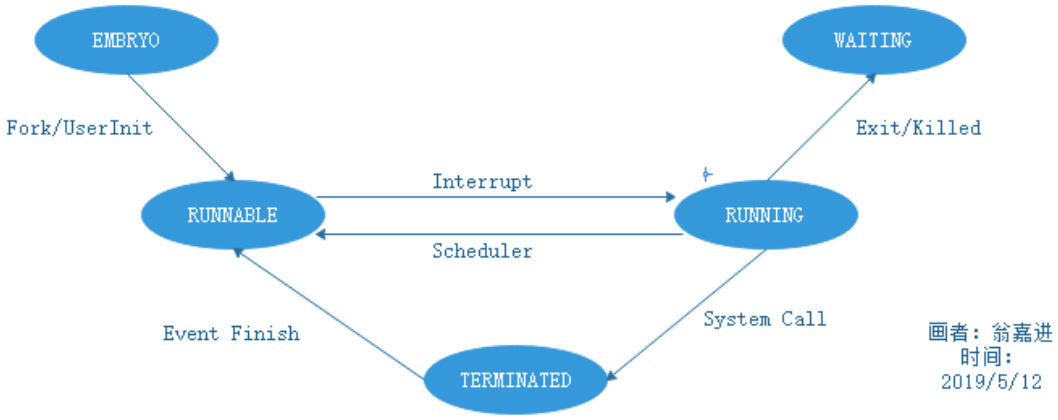
一般操作系统会有统一的 PCB 管理模块，在 XV6 中使用的是 PTable，具体可参考上一节源码阅读部分。

进程有哪些状态？请画出 XV6 的进程状态转化图。在 Linux，XV6 中，进程的状态分别包括哪些

在大多数教科书使用的标准五状态进程模型中，进程分为 New、Ready、Running、Waiting 和 Terminated 五个状态，状态转换图如图所示



XV6 操作系统的进程状态采用的是五状态进程模型。在 XV6 中这五个状态的定义为 EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE。其状态转换图如图所示：



Linux 中的进程转换图与 Xv6 大致相同，但是有如下区别：

1. Linux 中的 Waiting 有两个状态，分别是 Interruptible Waiting 和 Uninterruptible Waiting。
2. Linux 中额外有两个调试用状态 TASK_TRACED 和 TASK_STOPPED。

如何启动多进程（创建子进程）？如何调度多进程？调度算法有哪些？

通过 Fork 操作即可创建子进程。在 XV6 操作系统中，有专门负责进程调度工作的调度器，其定义于 proc.c 文件中。

经典的批处理系统调度算法包括：（1）FCFS-先来先服务；（2）SJF-最短作业优先；（3）SRT-最短剩余时间优先；（4）HRRN-最高相应比优先。这其中，FCFS 相对效率较低，而 SJF 效率最高，但可能出现进程“饥饿”的现象；相比之下，HRRN 则是折衷考虑了等待时间与响应时间。

现今被广发使用的操作系统有 Windows、Linux、UNIX、UNIX(5.3 BSD)，他们的调度算法分别为基于优先级的抢占式多进程调度、抢占式调度、动态优先数法、多级反馈队列法。

操作系统为何要限制一个 CPU 最大支持的进程数？XV6 中的最大进程数是多少？如何执行进程的切换？ 什么是进程上下文？多进程和多 CPU 有什么关系？

因为计算机中 CPU 支持的进程是要受内存大小限制的，每一个进程都拥有独自的进程空间，因而进程的数量一旦超过一定数目，会导致内存溢出等现象。

XV6 中最大进程数为 64 个，它被宏定义与 param.h 中，可修改。

switch.s 文件中定义了线程交换的汇编语言实现，多进程操作系统需要通过时间片分配算法来循环执行进程，当前进程执行一个时间片后会切换到下一个进程。但是，在切换前会保存上一个进程的状态，以便下次切换回这个进程时，可以再次加载这个进程的状态，进程的状态（如寄存器状态，代码执行的位置）即进程上下文。

单 CPU 计算机中的多进程只能是并发，多 CPU 计算机中的多进程可以并行。无论是并发还是并行，使用者来看，看到的都是多进程。

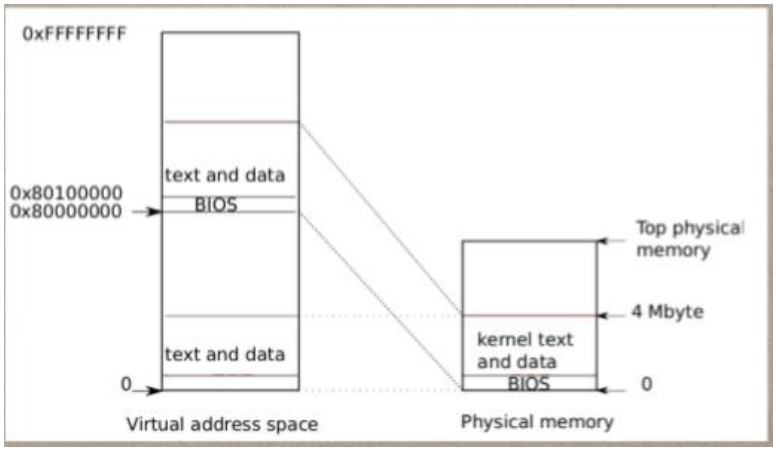
内核态进程是什么？用户态进程是什么？它们有什么区别？

内核态进程，顾名思义，是在操作系统内核态下执行的进程。在内核态下运行的进程一般用于完成操作系统最底层，最为核心，无法在用户态下完成的功能。比如，调度器进程是 Xv6 中的一个内核态进程，因为在用户态下是无法进行进程调度的。

相比较而言，用户态进程用于完成的功能可以多种多样，并且其功能只依赖于操作系统提供的系统调用，不需要深入操作内核的数据结构。比如 init 进程和 shell 进程就是 xv6 中的用户态进程。

进程在内存中是如何布局的，进程的堆和栈有什么区别？

Xv6 进程在虚拟内存中的布局如下。进程的栈用于存放运行时数据和运行时轨迹，包含了函数调用的嵌套，函数调用的参数和临时变量的存储等。栈通常较小，不会在运行时增长，不适合存储大量数据。相比较而言，堆提供了一个存放全局变量和动态增长的数据的机制。堆的大小通常可以动态增长，并且一般用于存储较大的数据和程序执行过程中始终会被访问的全局变量。



第 2 章　虚存管理

一、源码阅读

阅读要求： 内存管理部分： kalloc.c vm.c 以及相关其他文件代码

源码阅读：

虚拟内存是指将内存与外存有机的结合起来使用，从而得到一个容量很大的“内存”。这是一种以时间空间（外）换取空间（内）的技术。虚存管理，是现代操作系统不可缺失的一种管理机制。

在 kalloc.c 文件中，定义了物理内存分配器 kmem。其核心数据为空闲页链表，通过对物理空闲页的分配和回收来管理物理内存，其数据结构如下：

```
struct {
  struct spinlock lock;  // 自旋锁
  int use_lock;          // 是否在使用锁
  struct run *freelist;  // 空闲页链表
} kmem;
```

在该文件中也有分配器的初始化函数，空闲页的分配和回收函数。其中，变量 lock 的意义为：当 lock 为 1 时代表锁已被占用，反之未被占用，初始值为 0。

同时，在调用锁之前，必须对锁进行初始化。其初始化方法分两个阶段进行。首先，main()主函数调用 kinit1()方法，同时仍然使用 entrypgdir 来放置 just。由 entrypgdir 映射到空闲列表的页面。 最后，main()主函数使用其余物理页面调用 kinit2()方法，在安装了一个完整的页面表之后，将它们映射到所有核心上。

kalloc ()方法和 kfree ()方法分别实现对空闲页的分配和回收的功能。

kalloc ()方法：首先，使用 acquire()方法（该方法在同步机制中介绍）申请分配器的 lock；然后，获取空闲页链表的链首，并将空闲页链表指针后移；最后，使用 release()（该方法在同步机制中介绍）释放分配器的 lock。

```
char*
kalloc(void)
{
    struct run *r;

    if(kmem.use_lock)
        acquire(&kmem.lock);
    r = kmem.freelist;
    if(r)
        kmem.freelist = r->next;
    if(kmem.use_lock)
        release(&kmem.lock);
    return (char*)r;
}
```

kfree ()方法：如同分配一样。首先，使用 acquire()方法申请分配器的 lock；然后，获取空闲页链表的链首，并将释放的页面挂在空闲页链表的链首；最后，使用 release()释放分配器的 lock。

每个进程有一个页面表，加上 CPU 不运行任何进程时使用的页面表(kpgdir)。内核在系统调用和中断期间使用当前进程的页表;页面保护位阻止用户代码使用内核的映射。vm.c 文件主要功能就是提供维护这些页表的方法。本文将在下面的讨论区更多的介绍这一块

二、讨论总结

XV6 初始化之后到执行 main.c 时，内存布局是怎样的（其中已有哪些内容）？

当 XV6 初始化之后，执行到 main()方法之前，物理地址的具体内容如下。其中，引导程序是由 BIOS 负责载入内存，而设备区则是硬件规定占用的区域，内核 ELF 文件头和 Xv6 操作系统是由引导程序(bootmain.c)加载进内存的。

```
0x0000-0x7c00       引导程序的栈
0x7c00-0x7d00       引导程序的代码(512字节)
0x10000-0x11000     内核ELF文件头(4096字节)
0xA0000-0x100000    设备区
0x100000-0x400000 Xv6操作系统(未用满)
```

XV6 的动态内存管理是如何完成的？

正如上一节所述，在 kalloc.c 文件中，系统定义了一个物理内存管理模块 kmem。其核心数据为空闲页链表，通过对物理空闲页的分配和回收来管理物理内存，其数据结构如下：

```
struct {
  struct spinlock lock;  // 自旋锁
  int use_lock;          // 是否在使用锁
  struct run *freelist;  // 空闲页链表
} kmem;
```

在该文件中也有分配器的初始化函数，空闲页的分配和回收函数。其中，变量 lock 的意义为：当 lock 为 1 时代表锁已被占用，反之未被占用，初始值为 0。

同时，在调用锁之前，必须对锁进行初始化。其初始化方法分两个阶段进行。首先，main()主函数调用 kinit1()方法，同时仍然使用 entrypgdir 来放置 just。由 entrypgdir 映射到空闲列表的页面。 最后，main()主函数使用其余物理页面调用 kinit2()方法，在安装了一个完整的页面表之后，将它们映射到所有核心上。

XV6 的虚拟内存是如何初始化的？

在 XV6 操作系统中，通过使用 end 指针来标记 Xv6 的 ELF 文件的结尾地址，从而使得[PGROUNDUP(end), 0x400000]范围内的物理页可以被用作内存页来进行分配。正如上一个问题中介绍的，main()主函数调用 kinit1(end, P2V(0x400000))将这部分内存纳入虚拟内存页管理。

kmem 内存分配器必须知道可用的空闲页的内存范围。由于此时虚拟内存已经开启，且页表表项只有两条，因此 Xv6 必须利用已有的虚拟地址空间，在其中创建新的页表。kinit1()函数会调用 freerange()函数，按照前文叙述的方式，建立从 PGROUNDUP(end)地址开始直到 0x400000 为止的全部内存页的链表。kvmalloc()函数获得一个虚拟内存页并将其初始化一级页表。

最后，main()主函数使用其余物理页面调用 kinit2()方法将[0x400000, 0xE00000]范围内的物理地址纳入到内存页管理之中，在安装了一个完整的页面表之后，将它们映射到所有核心上。

XV6 的虚拟内存布局图，请说出每一部分对应的内容是什么。

Xv6 操作系统的虚拟内存布局图及其对应的每一部分内容如下表所示。正如我们所了解的，每个用户进程都有一个页表。其中，有关内核部分(即最后四项)，对于所有用户进程都是一样的。而前面的映射会有所不同，表中的信息根据 init 的进程的 ELF 文件信息和 exec 调用的代码确定。

虚拟地址	映射到物理地址	内容
[0x0, 0x1000]	由分配器提供的地址	用户进程的代码和数据
[0x1000, 0x2000]	由分配器提供的地址	不可访问页，用于检测栈溢出
[0x2000, 0x3000]	由分配器提供的地址	用户进程的栈
[0x80000000, 0x80100000]	[0, 0x100000]	I/O设备
[0x80100000, 0x80000000+data]	[0x100000, data]	内核代码和只读数据
[0x80000000+data, 0x80E00000]	[data, 0xE00000]	内核数据+可用物理内存
[0xFE000000, 0]	[0xFE000000, 0]	其他通过内存映射的I/O设备

发生中断时，用哪个页表？

中断发生时，使用的的页表依然是对应用户进程的页表。由于用户地址空间不仅有用户栈还有内核栈，因此陷入的内核代码依然可以正常执行。只有当中断处理程序决定退出当前进程或者切换到其他进程时，当前页表才会被切换为调度器的页表(全局变量 kpgdir)，并在调度器中切换为新进程的页表。

一个内页是多大？页目录有多少项？页表有多少项？最大支持多大的内存？

XV6 操作系统采用二级页表来管理虚拟内存。在这个二级页表结构中，每个页的大小为 4KB，每个页表的大小也为 4KB，每个页表项的大小为 4 字节，一个页表包含 1024 个页表项。一级页表表项存储的是二级页表的地址，二级页表表项存储的是对应的物理地址。虚拟地址和物理地址的最后 12 位总是相同，因此页表表项中的这 12 位可以被用作标记其他信息。

在 XV6 中，是如何将虚拟地址与物理地址映射的（调用了哪些函数实现了哪些功能）？

userinit()方法的主要功能是来初始化用户进程。userinit()在完成有关进程数据结构管理的工作后，会初始化这个进程自己的页表(struct proc 中的 pgdir)。而将虚拟地址与物理地址映射的正是在这一步进行。

首先，userinit()会使用 setupkvm()生成与前述一模一样的内核页表，然后使用 inituvm()生成第一个用户内存页(映射到虚拟地址 0x0)，并将用户进程初始化代码移动至这个内存页中。

initcode.S 中包含了一个 exec 系统调用，通过这个系统调用来加载进一个真正的用户进程。exec 会从磁盘里加载一个 ELF 文件。ELF 文件中包含了所有代码段和数据段的信息，并且描述了这些段应该被加载到的虚拟地址(这是在编译时就已经确定好的，所以编译器必须遵循某些约定来分配这些虚拟地址)。

最后，exec 会分配两个虚拟内存页，第一个页设置为不可访问，第二个页用作用户栈。

第 3 章　同步机制

一、源码阅读

阅读要求： 锁部分 spinlock.h spinlock.c 以及相关其他文件代码

源码阅读：

锁(Lock)是计算机操作系统中实现进程同步的重要机制。同步(Synchronization)是指让多个进程或线程能够按照程序员期望的顺序来协调执行。当多个进程或线程并发地执行并访问同一资源，并且进程/线程的执行结果依赖于其执行顺序，我们就称这种情况为竞争状态(Race Condition)。

Xv6 中实现了自旋锁(Spinlock)用于内核临界区访问的同步和互斥。自旋锁最大的特征是当进程拿不到锁时会进入无限循环，直到拿到锁退出循环。以下为自旋锁的数据结构，其定义在 spinlock.h 文件中。

```
struct spinlock {
  uint locked;       // Is the lock held?

  // For debugging:
  char *name;        // Name of lock.
  struct cpu *cpu;   // The cpu holding the lock.
  uint pcs[10];      // The call stack (an array of program counters)
                     // that locked the lock.
};
```

其中，核心的变量 locked 的意义为：当 locked 为 1 时代表锁已被占用，反之未被占用，初始值为 0。

同时，在调用锁之前，必须对锁进行初始化。其初始化方法相对简单，即对锁名进行赋值，将 locked 变量与 cpu 变量赋值 0。

acquire()方法和 release()方法分别实现对 locked 变量进行原子操作占用锁和释放锁的功能。

acquire()方法：首先，禁止了中断，并且使用专门的 pushcli()方法，这个方法保证了中断的禁止；然后，采用 xchg 指令（内联汇编）来实现在设置 locked 为 1；同时，获得

其原来的值的操作；最后，使用__sync_synchronize 方法是为了避免编译器对这段代码进行指令顺序调整和 CPU 在这块代码采用乱序执行的优化。

```
void
acquire(struct spinlock *lk)
{
  pushcli();      // 关中断
  if(holding(lk)) // 如果能获得Lock，则返回
    panic("acquire");

  // xchg -> 交换两个变量
  // return &lk->locked
  // and lk->locked = 1
  // 自带锁总线机制
  // 非阻塞
  while(xchg(&lk->locked, 1) != 0)
    ;

  lk->cpu = cpu;    // 记录上锁的CPU
  getcallerpcs(&lk, lk->pcs);
}
```

release()方法：首先，为了保证设置 locked 为 0 的操作的原子性，同样使用了内联汇编；最后，使用 popcli()来允许中断。

```
// Release the lock.
void
release(struct spinlock *lk)
{
  if(!holding(lk)) // 如果 已经释放了
    panic("release");

  lk->pcs[0] = 0; // 释放栈指针
  lk->cpu = 0;

  xchg(&lk->locked, 0); // 同样的xchg 保证了写后读 避免OOD

  popcli();
}
```

同时，为了实现 acquire()方法和 release()方法的开关中断行为，spinlock.c 文件中还实现了 pushcli()方法和 popcli()方法

二、讨论总结

临界区(Critical Section)

临界区是指对共享数据进行访问与操作的代码区域。所谓共享数据，就是可能有多个代

码执行流并发地执行，并在执行中可能会同时访问的数据。

同步(Synchronization)与互斥(Mutual Exclusion)

同步是指让两个或多个进程/线程能够按照程序员期望的方式来协调执行的顺序。比如，让 A 进程必须完成某个操作后，B 进程才能执行。

互斥则是指让多个线程不能够同时访问某些数据，必须要一个进程访问完后，另一个进程才能访问。

竞争状态(Race Condition)

当多个进程/线程并发地执行并且访问一块数据，并且进程/线程的执行结果依赖于它们的执行顺序，我们就称这种情况为竞争状态。

临界区操作时中断是否开启？若中断开启，有何影响？

Xv6 操作系统要求在内核临界区操作时中断必须关闭。如果此时中断开启，那么可能会出现以下死锁情况：A 进程在内核态运行并拿下了 p 锁时，触发中断进入中断处理程序，中断处理程序也在内核态中请求 p 锁，由于锁在 A 进程手里，且只有 A 进程执行时才能释放 p 锁，因此中断处理程序必须返回，p 锁才能被释放。那么此时中断处理程序会永远拿不到锁，陷入无限循环，进入死锁。

XV6 的锁是如何实现的，有什么操作？

Xv6 中实现了自旋锁(Spinlock)用于内核临界区访问的同步和互斥。自旋锁最大的特征是当进程拿不到锁时会进入无限循环，直到拿到锁退出循环。具体操作，可参考上一节中源码阅读部分

xchg 指令是什么，该指令有何特性？

交换指令 XCHG 是两个寄存器，寄存器和内存变量之间内容的交换指令，两个操作数的数据类型要相同，可以是一个字节，也可以是一个字，也可以是双字。该指令的功能和 MOV 指令不同，后者是一个操作数的内容被修改，而前者是两个操作数都会发生改变。寄存器不能是段寄存器，两个操作数也不能同时为内存变量。 XCHG 指令不影响标志位。

三、方案设计

设计要求： 基于 XV6 的 spinlock，请给出实现信号量、读写锁、信号机制的设计方案

这里主要设计了读写锁，对信号量以及信号机制也有讨论和实现。

- **信号量**

基本方案：用 Xv6 提供的接口实现了信号量，格式和命名与 POSIX 标准类似。这个信号量的实现采用等待队列的方式。当一个进程因信号量陷入阻塞时，会将自己放进等待队列并睡眠。当一个进程释放信号量时，会从等待队列中取出一个进程继续执行。

伪码实现：

```
struct semaphore {
  int value;
  struct spinlock lock;     // 自旋锁
  struct proc *queue[NPROC];  // 等待队列
  int end;
  int start;
};

void sem_init(struct semaphore *s, int value) {
  s->value = value;
  initlock(&s->lock, "semaphore_lock");
  end = start = 0;
}

void P(struct semaphore *s) {
  acquire(&s->lock);  // 获取锁

  s->value--;
  if (s->value < 0) {
    s->queue[s->end] = myproc();  // 将进程加入到等待队列
    s->end = (s->end + 1) % NPROC;
    sleep(myproc(), &s->lock) // 睡眠
  }

  release(&s->lock);  // 释放锁
}

void V(struct semaphore *s) {
  acquire(&s->lock);  // 获取锁

  s->value++;
  if (s->value <= 0) {
    wakeup(s->queue[s->start]); // 唤醒等待队列队首进程
    s->queue[s->start] = 0;
    s->start = (s->start + 1) % NPROC;
  }

  release(&s->lock);  // 释放锁
}
```

- **读写锁**

基本方案： 由于 XV6 操作系统支持多处理器执行命令，因而最初方案本小组参考《多核计

算与程序设计》中对读写锁的设计：

1. 将读写操作定义为互斥关系

2. 将不同写操作定义为同步关系，并设置了读者计数器。

其数据结构为：

```
typedef struct RWLOCK_st{
    spinlock ReadLock; // 读锁
    spinlock WriteLock; // 写锁
    UINT uReadcount; // 读者计数器
}RWLOCK;
```

以下方法为对读锁与写锁的获取和释放方法。

```
RWLock_LockRead() {
ReadLock->acquire(); // 上锁锁住计数器变量的读写，即读锁
uReadcount += 1; // 读者计数器加1
if(uReadcount > 0) // 判断是否触发写锁获取条件——读者非零
WriteLock->acquire(); // 获取写锁
ReadLock->release(); // 解锁计数器变量的读写，即读锁
}
RWLock_UnlockRead() {
ReadLock->acquire(); // 上锁锁住计数器变量的读写，即读锁
uReadcount -= 1; // 读者计数器减1
if(uReadcount == 0) // 判断是否触发写锁释放条件——读者为零
WriteLock->release(); // 释放写锁
ReadLock->release(); // 解锁计数器变量的读写，即读锁
}

RWLock_LockWrite() {
WriteLock->acquire(); // 获取写锁
}
RWLock_UnlockWrite() {
WriteLock->release(); // 释放写锁
}
```

优点：保证了多处理器上的读写操作的同步互斥关系 实现简单。

缺点：读操作比较频繁时，计数 uReadCount 可能一直无法归零，会导致写操作饿死现象 ，消耗较大

优化方案： 为读操作设置副本文件，使得读写操作不存在互斥关系。

RWLock_LockRead()：

1. 在获取读锁后，立即判断读者计数器是否为 0。
   1.1. 若为 0，则建立副本文件并链接至读写锁（副本文件唯一）；
2. 读者计数器加 1

3. 释放读锁

RWLock_UnlockRead():

1. 获取读锁
2. 读者计数器减 1
3. 判断读者计数器是否为 0，若为 0，则取消链接并删除副本文件；
4. 释放读锁

优点：解决了写操作可能出现的饿死现象

缺点：副本文件与原文件可能存在滞后的问题。

- **信号机制**

基本方案：条件变量是管程内的等待机制。进入管程的线程因资源被占用而进入等待状态

每一个条件变量均表示一种等待原因，并且对应一个等待队列。其数据结构为：

```
Class Condition{
    int wait_count = 0; // 等待的线程数目
    Wait_Queue wait_queue; // 等待队列
}
```

Wait()方法和 Signal()方法的伪码实现

```
Condition::Wait(lock){
    wait_count++;
    Add this thread t to wait_queue;          // 将当前线程添加到等待队列
    release(lock);                            // 释放锁
    schedule();                               // 需要保证原子性与一致性
    require(lock);                            // 申请锁，实现比等
}

Condition::Signal(){
    if(wait_count>0){                         // 如果等待的线程数目大于0
        Remove a thread t from wait_queue;    // 从等待队列移除一个就绪线程
        wakeup(t);                            // 需要保证原子性与一致性
        wait_count--;
    }
}
```

第 4 章　文件系统

一、源码阅读

阅读要求：

文件系统部分 buf.h fcntl.h stat.h fs.h file.h ide.c bio.c log.c fs.c file.c sysfile.c exec.c

其他文件系统参考：

由于本章存在大量的源码需要配合问题来解释。因而，这里不详述。这里介绍下，Unix 和 Window 的文件系统：

Unix 文件系统是以块(Block)为单位对磁盘进行读写的。一般而言，一个块的大小为 512Byte 或者 4KB。文件系统的所有数据结构都以块为单位存储在硬盘上，一些典型的数据块包括：超级数据块，i 节点块，数据块。

Window 的文件系统 FAT 磁盘物理上分为四部分组成：保留区(含 MBR－Main Boot Record)、保留区(含 DBR－DOS Boot Record)、FAT 区、数据区。保留区含有一个重要的数据结构－系统引导扇区 DBR。FAT12、FAT16 的保留区通常只有一个扇区。而 FAT32 保留的多些，除了 0 号扇区外，还有一些其他的扇区，其中包括了 DBR 的备份扇区。

FAT32 中簇地址是用 4 字节进行编址的，故在 FAT 表中，是以 4 个字节为单位进行划分，每个单元存储一个簇地址。 **0 号地址与 1 号地址被系统保留并存储特殊标志内容。**从 2 号地址开始，第 i 号地址对应数据区中 i 号簇。我们称 FAT 表中的地址为 FAT 表项，FAT 表中记录的值为 FAT 表项值。

当文件系统被创建时，FAT 表会被清空，在 FAT1 和 FAT2 表中的 0 号地址与 1 号地址会被写入特定值。由于创建文件系统的同时会创建根目录，也就是在数据区为根目录分配一个簇的空间(2 号簇，起始簇)在 FAT 表中 2 号地址写入一个结束标记。当 FAT 表中第 i 号地址对应的簇未被使用时，表项值为 0

二、讨论总结

了解 UNIX 文件系统的主要组成部分：超级块（superblock），i 节点（inode），数据块（data block），目录块（directory block），间接块（indirection block）

由于上文介绍了 Unix 文件系统。这里阐述一下各个部分的功能：

1. 超级块：相当于是保存了文件系统的元数据信息，描述文件系统的状态，比如它有多大，何处有空闲空间以及其他一些信息等等。
2. i 节点：i 节点表相当于是 i 节点的一个数组，内核所引用的 i 节点号即是 i 节点在该数组中的下标大小。类似于文件的元数据信息。
3. 数据块：真正存放数据的地方。
4. 目录块：在数据块中保存的数据是一系列的 i 节点 <-> 文件名的 item。
5. 间接块：i 节点中包含了几个数据块的地址。此外，当文件过大，10 个数据块无法存放完毕，系统会动态分配空间用于存储这些超出信息存放的数据块地址，这就是间接块。I 节点后三位存放的是间接块。

阅读文件 ide.c

在 ide.c 文件中，XV6 实现了一个简单的基于 Programmed IO (非 DMA)的 IDE 驱动程序代码。一个 Xv6 中的磁盘读写请求的数据结构如下：

```
struct buf {
  int flags;
  uint dev;
  uint blockno;
  struct sleeplock lock;
  uint refcnt;
  struct buf *prev;  // LRU 缓存队列
  struct buf *next;
  struct buf *qnext; // 磁盘队列
  uchar data[BSIZE];
};
```

在 IDE 中，还实现了下列方法，其具体功能如下：

函数名	功能
idewait()	等待磁盘进入空闲状态
ideinit()	初始化 IDE 磁盘 IO
idestart()	开始一个磁盘读写请求
iderw()	上层文件系统调用的磁盘 IO 接口
ideintr()	当磁盘请求完成后中断处理程序会调用的函数

当系统启动时，main()主函数会通过 ideinit()方法来对 IDE 磁盘进行初始化操作。

而 iderw()方法的主要功能是：提供面向顶层文件系统模块的接口。iderw()方法可用于磁盘的读写操作，需要通过检查 buf->flag 里的 DIRTY 位和 VALID 位来判断是读操作还是写操作。如果请求队列为空，则说明当前磁盘没有处在工作状态，则需要调用 idestart()方法初始化磁盘请求队列，并设置中断。当执行完以上操作，iderw()会将调用者切换到睡眠状态。

当磁盘读取或者写操作完毕时，会触发中断进入 trap.c 中的 trap()方法，trap()方法会调用 ideintr()方法处理磁盘相关的中断。在 ideintr()方法中，如果当前请求是读请求，就读取目前已经在磁盘缓冲区中准备好的数据。最后，ideintr()方法会唤醒正在睡眠等待当前请求的进程，如果队列里还有请求，就调用 idestart()方法来处理新的请求。

了解 XV6 文件系统中 buffer cache 层的内容和实现。描述 buffer 双链表数据结构及其初始化过程。了解 buffer 的状态。了解对 buffer 的各种操作。

在文件系统中，Buffer Cache 担任了一个磁盘与内存文件系统交互的中间层。由于对磁盘的读取是非常缓慢的，因此将最近经常访问的磁盘块缓存在内存里是很有益处的。

Xv6 中 Buffer Cache 的实现在 bio.c 中，Buffer Cache 的数据结构如下：

```
struct {
  struct spinlock lock;
  struct buf buf[NBUF];
  // Linked list of all buffers, through prev/next.
  // head.next is most recently used.
  struct buf head;
} bcache;
```

其中，最重要的数据是在固定长度的数组上维护了一个由 buf 组成的双向链表。此外，lock 保护对 Buffer Cache 链表结构的访问。值得注意的是，对链表结构的访问和对一个 buf 结构的访问需要的是不同的锁。

在系统对缓存进行初始化时，系统调用 binit()方法。binit()方法对缓存内每个元素初始化睡眠锁，并从后往前连接成一个双向链表。

操作系统使用 bread()方法和 bwrite()方法对缓存中的块进行读写操作。关于缓存的全部操作是在 bread()方法与 bwrite()方法中自动完成的，不需要上层文件系统的参与。

bread()方法：首先，调用 bget()方法，而 bget()方法会检查请求的磁盘块是否在缓存中。如果磁盘块在缓存中，则直接返回缓存中对应的磁盘块即可。如果不在缓存中，那么需

要先使用最底层的 iderw()函数先将此磁盘块从磁盘加载进缓存中，再返回此磁盘块。

bwrite()函数的作用是:将缓存中的数据直接写入磁盘。Buffer Cache 层不尝试执行任何延迟写操作，对磁盘的 bwrite()写操作的时间由上层文件系统控制。

上面的文件系统调用 brelse()方法来释放不再使用的脉冲块。brelse()方法主要处理双向链表上的操作，这里不再重复。

了解 XV6 文件系统中的 logging 和 transaction 机制。

在文件系统中添加 Log 层是为了能够使文件系统能够处理电源系统等异常，避免磁盘文件系统上的不一致。Log 层的实现思路是，对于文件系统顶部全部是磁盘操作，它分为一个个 transaction。首先，将所有数据及其对应的每个事务 Log 区域上的磁盘号写入磁盘，并且只有在写入 Log 区域后才完成，然后 数据被写入日志区域实际数据存储区域。通过此设计，如果禁用对日志的写入，则文件系统将假定写入不存在，并且如果禁用对实际区域的写入，则 Log 区域数据可用于恢复文件系统。这样，可以避免文件系统中文件的损坏。

使用 Log 层时，上层文件系统必须首先调用 begin_op（）方法。 begin_op（）方法记录一个新事务。 使用日志层后，上层系统必须调用 end_op（）方法。 仅当没有事务正在执行时，日志才会执行实际磁盘写入。 在 commit（）方法中真正的磁盘写操作，你可以看到只在 end_op()结束时调用 commit()方法，log.outstanding == 0(和启动时间)。 commit（）方法首先调用 write_log（）方法将高速缓存中的磁盘块写入磁盘上的 Log 区域，并将 Log Header 写入磁盘区域。 仅当更新磁盘上存在日志标题的区域数据时，才会完成日志更新。 更新日志区域后，commit（）方法调用 install_trans（）以完成实际磁盘写入步骤，之后调用 write_head（）方法以清除当前 Log 数据。

了解 XV6 文件系统的硬盘布局。

在 Xv6 操作系统的硬盘中，按顺序存储了几个硬盘块：boot block 、 super block 、 log 、 inode blocks 、 free bit map 、 data blocks。 这些硬盘块的索引直接由整数使用。 第一个硬盘块引导块在引导时加载到内存中，磁盘块号为 0.第二个超级块占用硬盘块，编号为 1，Xv6 中的声明如下：

```
struct superblock {
    uint size;         // Size of file system image (blocks)
    uint nblocks;      // Number of data blocks
    uint ninodes;      // Number of inodes.
    uint nlog;         // Number of log blocks
    uint logstart;     // Block number of first log block
    uint inodestart;   // Block number of first inode block
    uint bmapstart;    // Block number of first free map block
};
```

Superblock 中存储有关文件系统的元信息。 操作系统必须首先读取 Superblock 以了解剩余 log 块，inode 块，bitmap 块和 data 块的大小和位置。 在 Superblock 之后顺序存储多个日志块，多个 inode 块和多个位图块。 磁盘的剩余部分存储数据块块。

了解 XV6 的“文件”有哪些，以及文件，i 节点，设备相关的数据结构。了解 XV6 对文件的基本操作有哪些。XV6 最多支持多少个文件？每个进程最多能打开多少个文件？

Xv6 中的文件（包括目录）都由 inode 数据结构表示，所有文件的 inode 都存储在磁盘上。当系统或进程需要使用 inode 时，inode 将被加载到 inode 缓存中。存储在内存中的 inode 将具有比存储在磁盘上的 inode 更多的运行时信息。内存中的 inode 数据结构声明如下。

```
// in-memory copy of an inode
struct inode {
  uint dev;           // Device number
  uint inum;          // Inode number
  int ref;            // Reference count
  struct sleeplock lock; // protects everything below here
  int valid;          // inode has been read from disk?
  short type;         // copy of disk inode
  short major;
  short minor;
  short nlink;
  uint size;
  uint addrs[NDIRECT+1];
};
```

其中，inode.type 指明了这个文件的类型。Xv6 中，这个类型可以是普通文件，目录，或者是特殊文件。

内核在内存中维护一个 i 节点缓存。同时，文件中还实现了有关 i 节点的相关操作。其相关方法及其功能如下：

1. iinit()方法　主要功能是：读取 Superblock，初始化 inode 相关的锁
2. readi()方法　主要功能是：往 inode 读数据
3. ialloc()方法　主要功能是：在磁盘上分配一个 inode

4. ilock()方法　主要功能是：获取指定 inode 的锁
5. iunlock()方法 主要功能是：释放指定 inode 的锁
6. iget()方法　主要功能是：获取指定 inode，更新缓存
7. writei()方法　主要功能是：往 inode 写数据
8. bmap()方法　主要功能是：返回 inode 的第 n 个数据块的磁盘地址
9. iupdate()方法 主要功能是：将内存里的一个 inode 写入磁盘
10. iput()方法　主要功能是：对内存内一个 Inode 引用减 1，引用为 0 则释放 inode

Inode 具有 12 个（NDIRECT）直接映射的磁盘块和 128 个间接映射的磁盘块。 总之，XV6 系统支持的最大文件大小为 140 * 512B = 70KB。

在 XV6 操作系统中，一个文件的数据结构表示如下

```
struct file {
  enum { FD_NONE, FD_PIPE, FD_INODE } type;
  int ref; // reference count
  char readable;
  char writable;
  struct pipe *pipe;
  struct inode *ip;
  uint off;
};
```

由此可见，file 数据结构可以表示 inode 或管道。 多个 file 数据结构可以抽象相同的 inode，但 Offset 可以是不同的。系统所有的打开文件都在全局文件描述符表 ftable 中

从中可以看出 Xv6 最多支持同时打开 100(NFILE)个文件，从 struct proc 中可以看出 Xv6 中每个进程最多同时可以打开 16(NOFILE)个文件。

了解与文件系统相关的系统调用，简述各个系统调用的作用。

利用上一层的实现，大多数系统调用的实现都是比较直接的。Xv6 中支持的文件相关系统调用如下：

1. sys_link()方法　主要功能是：为已有的 inode 创建一个新的名字
2. sys_unlink()方法 主要功能是：为已有的 inode 移除一个名字，可能会移除这个 inode
3. sys_open()方法　主要功能是：打开一个指定的文件描述符
4. sys_mkdir()方法　主要功能是：创建一个新目录
5. sys_mknod()方法　主要功能是：创建一个新文件

6. sys_chdir()方法　主要功能是：改变进程当前目录

7. sys_fstat()方法　主要功能是：改变文件统计信息

8. sys_read()方法　主要功能是：读文件描述符

9. sys_write()方法　主要功能是：写文件描述符

10. sys_dup()方法　主要功能是：增加文件描述符的引用

第 5 章　中断与系统调用

一、阅读要求

1. 启动部分： Bootloader：bootasm.S bootmain.c xv6 初始化模块：main.c bootother.S
2. 中断与系统调用部分：trap.c trapasm.S vectors.S & vectors.pl syscall.c sysproc.c proc.c 以及相关其他文件代码

二、讨论总结

用户态和内核态

在操作系统中，内核态是指运行时操作系统内核的状态。在这种状态下，内核程序可以访问任何现有硬件并执行任何现有指令。用户态是指用户进程执行时系统的状态。在这种状态下，用户进程只能执行部分指令，访问硬件并根据操作系统提供的系统调用与其他进程交互。

将内核态与用户态隔离是为了提高系统的整体安全性和健壮性，并防止恶意进程和错误进程损坏系统。

中断和系统调用

中断是一种允许操作系统响应外部硬件的机制。例如，当用户进程执行并加载了另一个用户进程请求的磁盘文件时，需要设计一个中断信号来通知操作系统暂停当前用户进程。让操作系统处理这个中断事件;系统调用是一种机制，它使用户进程能够进入内核状态并请求某个系统服务，例如使用系统提供的 syscall 指令进入内核，并完成需要内核的输入和输出任务进程的权限。然后返回用户模式，进程继续执行。

当计算机运行时，它通过 CPU 中某些寄存器的权限位知道它是处于内核状态还是用户状态。例如，在 x86 系统中，CPU 通过检查%cs 寄存器中的 CPL 位来检查当前指令的执行权限级别。在 XV6 系统中，CPL0 代表内核模式，CPL3 代表用户模式。如果指令的执行权限与 CPL 位的值不匹配，则生成一般保护错误。

计算机开始运行阶段就有中断吗？XV6 的中断管理是如何初始化的？

计算机开始运行阶段就有 BIOS 支持的中断。由于 xv6 在开始运行阶段没有初始化中断处理程序，于是 xv6 在 bootasm.S 中用 cli 命令禁止中断发生。picinit()方法和 oapicinit()方法初始化可编程中断控制器；consoleinit()方法和 uartinit()方法设置 I / O 和设备端口的中断。 然后，tvinit()调用 trap.c 中的代码来初始化中断描述符表，关联 vectors.S 中的中断 IDT 条目，调用 idtinit()以在调度开始之前设置第 32 个时钟中断，最后在调用中调用它。调度()。 Sti()启动中断并完成中断管理初始化。

XV6 是如何实现内核态到用户态的转变的？XV6 中的硬件中断是如何开关的？

从内核态到用户态的转变主要是靠设置程序状态字，程序状态字存储在%cs 寄存器的低 3 位中。XV6 系统的硬件中断由可编程中断控制器进行设置和管理，比如通过调用 ioapicenable 控制 IOAPIC 中断。处理器可以通过设置 eflags 寄存器中的 IF 位来控制自己是否想要收到中断：通过命令 cli 关中断, 通过命令 sti 开中断。

实际的计算机里，中断有哪几种？

1. 外部设备请求中断。一般的外部设备（例如键盘，打印机，A / D 转换器等）在完成其自己的操作之后，向 CPU 发出中断请求，请求 CPU 为他服务。由计算机硬件异常或故障引起的中断，也称为内部中止;
2. 故障被迫中断。计算机在一些关键部件中具有自动故障检测设备。如操作溢出，内存读取错误，外部设备故障，电源故障等报警信号等，这些设备的报警信号可以中断 CPU 并执行相应的中断处理;
3. 实时时钟请求中断。在控制中遇到定时检测和控制，并且经常使用外部时钟电路（可编程）来控制其时间间隔。当需要定时时，CPU 发出命令以启动时钟电路。一旦达到指定的时间，时钟电路发出中断请求，CPU 就完成检测和控制工作;
4. 数据通道被中断。数据通道中断也称为直接存储器访问（DMA）操作，例如磁盘，磁带驱动器或 CRT，它们是与存储器直接交换数据所必需的。
5. 程序自愿中断。 CPU 执行特殊指令（陷阱指令）或由硬件电路引起的中断是程序自愿中断。它指的是当用户调试程序时程序使用的检查方法，程序主动中断检查中间结果或查找错误，例如断点中断。单步中断等。

什么是中断描述符，中断描述符表？在 XV6 里是用什么数据结构表示的？

中断描述符代表并描述一个特定的中断。中断描述符表是用于在 X86 架构中以受保护模式存储中断服务例程信息的数据结构。 在 XV6 数据结构中，涉及的数据结构如下：

```
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
};
struct gatedesc idt[256];
```

其中，struct gatedesc 的格式与 X86 架构所需的格式完全相同。对于第 i 个中断描述符，CS 寄存器存储内核代码段的段号 SEG_KCODE，偏移部分存储 vector [i]的地址。在 XV6 系统中，所有 vector [i]地址都指向 trapasm.S 中的 alltraps 函数。

请以某一个中断为例，详细描述 XV6 一次中断的处理过程。

以零除错误为例。 在 XV6 指令执行期间遇到除零错误时，CPU 硬件首先发现此错误并触发中断处理机制。 在中断处理机制中，硬件执行以下步骤：

1. 从中断描述符表中获得对应的中断描述符。
2. 检查 CS 的域 CPL <= DPL，即操作是否合法，DPL 是描述符中记录的特权级。
3. 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。
4. 从一个任务段描述符中加载 SS 和 ESP。
5. 将用户信息段 SS/ESP、EFLAGS 寄存器、CS 寄存器、EIP 寄存器依次压栈。
6. 清除 EFLAGS 的一些位。
7. 设置 CS 和 EIP 为描述符中的值。

此时，由于 CS 已设置为描述符中的值（SEG_KCODE），系统已进入内核态，并且 EIP 指向 trapasm.S 中 alltraps 函数的开头。 在 alltrap 函数中，系统按下用户寄存器，构建一个 Trap Frame，并将数据寄存器段设置为内核数据段，然后跳转到 trap.c 中的 trap 函数。在陷阱功能中，首先检查中断呼叫号码，发现这不是系统调用，也不是外部硬件中断，因此请输入以下代码段：

```
if(myproc() == 0 || (tf->cs&3) == 0){
  // In kernel, it must be our mistake.
  cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
          tf->trapno, cpuid(), tf->eip, rcr2());
  panic("trap");
}
// In user space, assume process misbehaved.
cprintf("pid %d %s: trap %d err %d on cpu %d "
        "eip 0x%x addr 0x%x--kill proc\n",
        myproc()->pid, myproc()->name, tf->trapno,
        tf->err, cpuid(), tf->eip, rcr2());
myproc()->killed = 1;
```

根据系统程序或用户进程是否触发中断来执行不同的处理。 如果用户进程错误，系统将终止用户进程；如果内核进程错误，则在输出错误消息后，整个系统进入无限循环。

如果是一个可以修复的错误，比如页错误，那么系统会在处理完后返回 trap()函数进入 trapret()函数，在这个函数中恢复进程的执行上下文，让整个系统返回到触发中断的位置和状态。

请以系统调用 setrlimit 为例，叙述如何在 XV6 中实现一个系统调用。

1. 首先，在 syscall.h 中添加一个新的系统调用号

```
#define SYS_setrlimit  22
```

2. 其次，在 syscall.c 中增加入口函数中添加 setrlimit 的系统调用处理函数的地址

```
static int (*syscalls[])(void) = {
        ...
    [SYS_setrlimit] sys_setrlimit,
};
```

3. 同时，在 sysproc.c 中声明并实现 setrlimit 的系统调用处理函数

```
int sys_setrlimit(int resource, const struct rlimit *rlim) {
    // set max memory for this process, etc
}
```

4. 最后，在 user.h 中声明 setrlimit()这个函数系统调用函数的接口，并在 usys.S 中添加有关的用户系统调用接口。

```
SYSCALL(setrlimit)
int setrlimit(int resource, const struct rlimit *rlim);
```