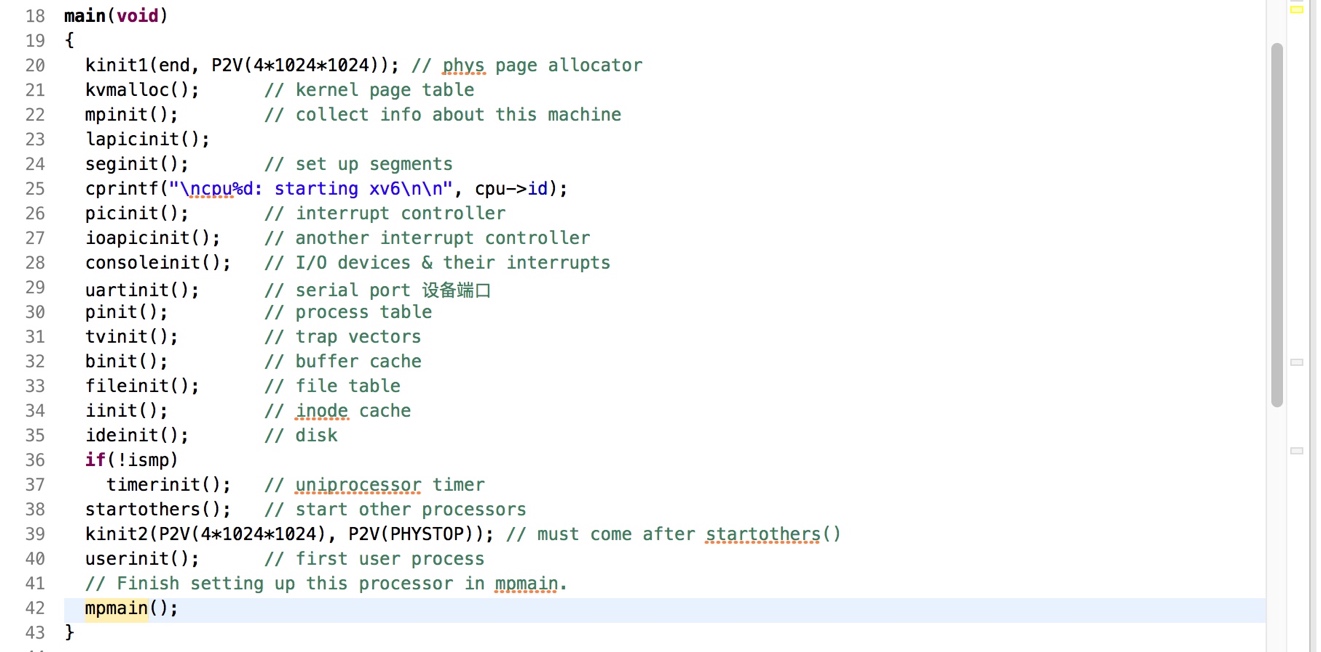
**XV6中断与系统调用部分**

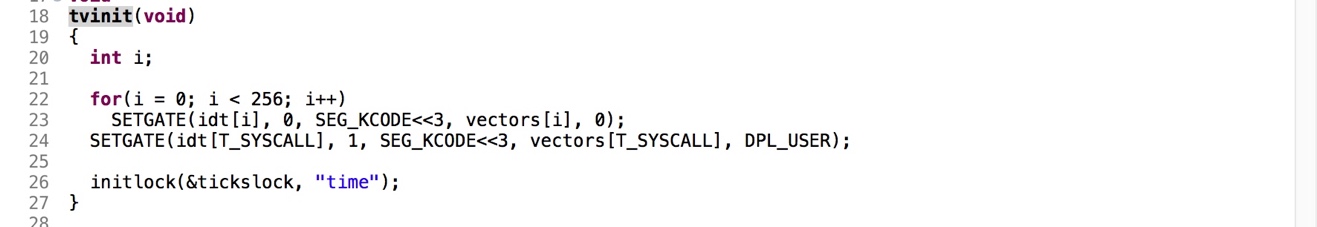
1. **XV6的中断管理是如何初始化的? XV6是如何实现内核态到用户态的转变的? XV6中的硬件中断是如何开关的?**

（1）由于XV6在开始运行阶段可能出现中断源，但是中断的处理程序还没有初始化。当操作系统从BIOS固有程序的接过控制权之后，中断管理初始化之前，需要在bootasm.S文件中执行cli，关中断。于是XV6在bootasm.S中用cli命令禁止中断发生（因为以下就要进入系统的初始化过程）。接下来中断管理初始化在main.c中的main()函数逐个调用。

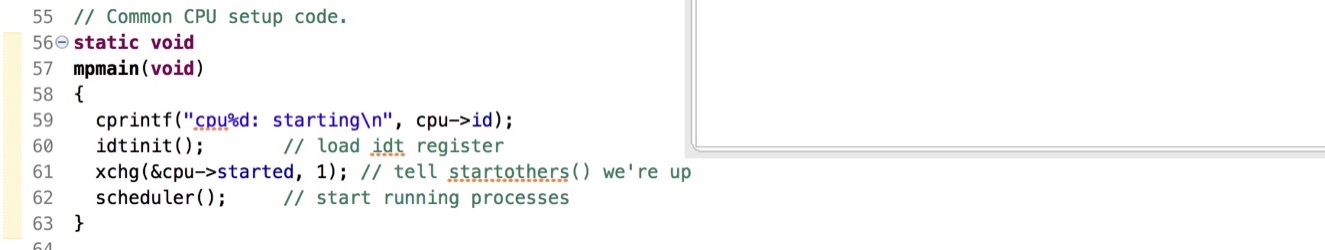


其中：

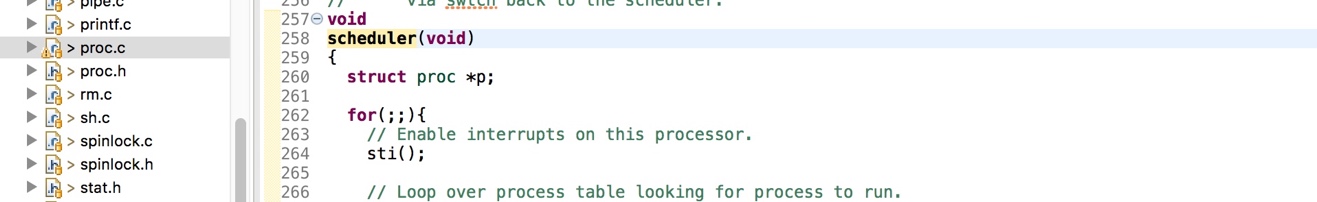
* picinit()和ioapicinit()用于初始化可编程中断控制器；
* consoleinit()初始化I/O设备中断；
* uartinit()设置设备端口的中断；
* tvinit()初始化中断描述符表，关联vectors.S中的256个中断IDT表项；



main()调用mpmain函数，启动CPU：



在调度开始前调用idtinit()，加载idt寄存器。最后，执行调度函数scheduler()，其中中调用sti()开中断，完成中断管理初始化。



（2）从内核态切换到用户态需要通过第一个进程来完成。

第一个用户程序的初始化在pro.c中调用userinit()完成，其中，会为第一个进程构造中断返回帧（trapframe），并将第一个进程的trapframe中的cs,ds,es,ss设置在用户模式下（DPL\_USE）。同时，调用allocproc函数分配空间时，将sp指针设置成trapret，而trapret是一段位于trapasm.S的汇编代码，可将用户态向内核态切换时保存的上下文信息恢复。因此当第一个用户进程开始被调度执行的时候，会从trapret处开始执行。也就是说，执行完后内核态切换到用户态。

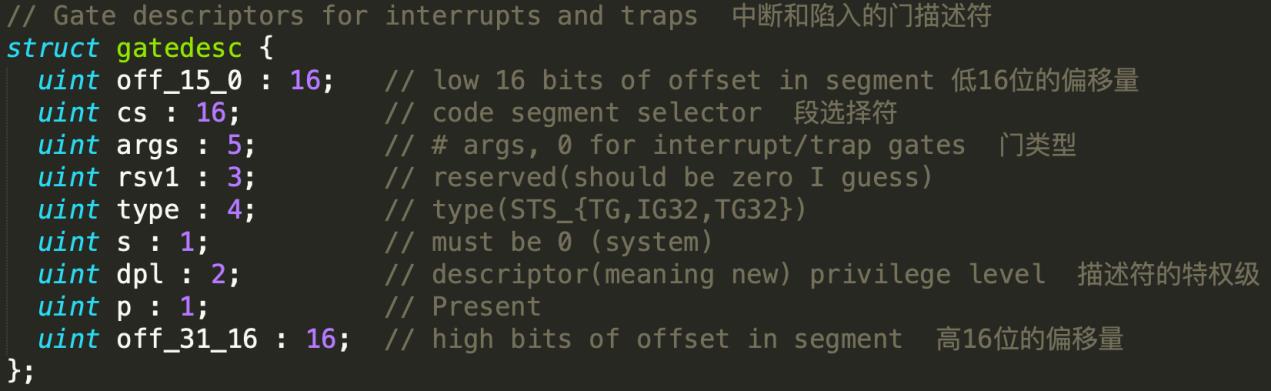
其他用户程序在用fork()创建时，因为父进程调用fork()时必会进入内核态，因此其用户态的上下文信息保存在了内核栈中。而fork()创建子进程的中断返回帧是复制父进程的。同时，子进程第一次运行时是以中断返回的形式的，所以将从内核态恢复到用户态。

（3）XV6的硬件中断由picirq.c，ioapic.c，timer.c中的代码对可编程中断控制器进行设置和管理。例如，通过调用ioapicenable控制IOAPIC中断；通过设置eflags寄存器中的IF位，控制自己是否想要收到中断；通过命令cli（Clear Interrupt）关中断，sti（Set Interrupt）开中断，STI 中断标志置1指令使IF = 1，CLI中断标志置0指令使IF = 0。

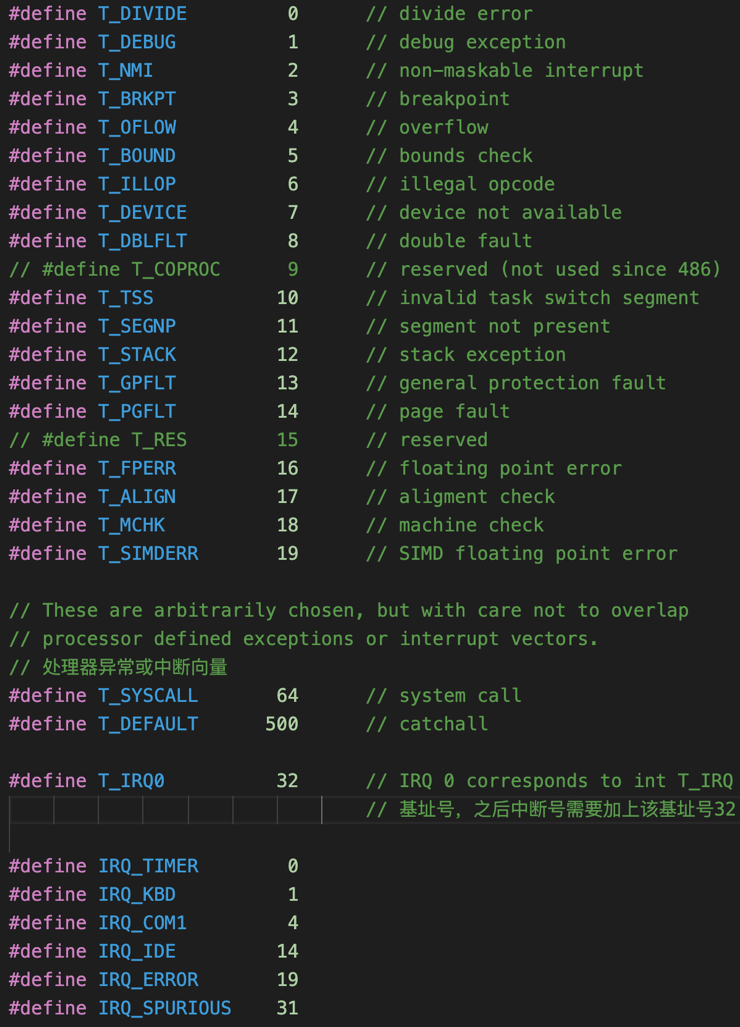
1. **在 XV6 里是用什么数据结构来表示中断描述符，中断描述符表? 请总结一下。**

中断描述符包含了中断相关信息如：偏移量、特权级 DPL、门类型和段选择符等。实模式下直接通过中断向量表得到中断处理程序地址，保护模式下用中断描述符描述中断向量，通过中断描述符表、段描述符表等寻址中断处理程序地址。

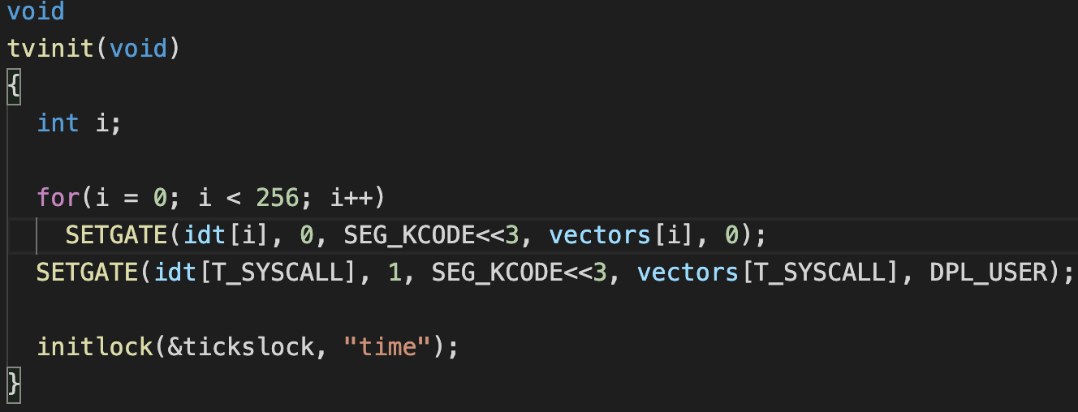
中断描述符表（IDT）负责存储中断描述符，表的地址存储在中断描述符表寄存器 IDTR 中， XV6 中用门描述符（gatedesc）来表示中断描述符，结构定义如下图所示:



XV6中的门描述符类型有中断门和陷阱门，中断和陷入常量被定义在traps.h文件中，其中0-19号为不可屏蔽中断和异常，32号之后为外部中断（IRQ），值得注意的是64号为系统调用，其门描述符类型为陷入门，其余均为中断门。



中断向量表在XV6中由整数数组vectors表示，其存放了对应中断号的中断处理程序的地址偏移，门描述符在trap.c的tvinit()方法被初始化，如下图所示，利用定义在mmu.h文件的SETGATE()方法初始化了中断描述符表的256个表项，门描述符类型全为中断门，之后再初始化64号系统调用的门描述符，更改其门类型为陷入门，DPL特权级为用户级。



1. 请以某一个中断(如除零，页错误等)为例，详细描述 XV6 一次中断的处理过程。包括:  涉及哪些文件的代码?如何跳转?内核态，用户态如何变化?涉及哪些数据结构等等。

以除零错误中断为例，CPU根据当前是用户态保存相应寄存器，然后访问IDT 表，表项指向的中断处理函数入口捕获到中断，将对应的中断号压栈，并调用alltraps:

.globl vector0

vector0:

pushl $0

pushl $0

jmp alltraps

alltraps继续保存处理器的寄存器，设置数据和CPU段，然后压入 %esp，调用trap，到此时已完成用户态到内核态的转变：

...

movw $(SEG\_KDATA<<3), %ax

movw %ax, %ds

...

pushl %esp

call trap

trap会根据%esp指向对应的tf，首先根据trapno判断该中断是否是系统调用，之后判断硬件中断，由于除零不是以上两种，于是判断为代码错误中断，并且是发生在用户空间的。接着处理程序将该进程标记为killed，并退出，继续下一个进程的调度。

switch(tf->trapno){

...

//PAGEBREAK: 13

default:

if(proc == 0 || (tf->cs&3) == 0){

// In kernel, it must be our mistake.

...

}

// In user space, assume process misbehaved.

...

proc->killed = 1;

}

...

// Check if the process has been killed since we yielded

if(proc && proc->killed && (tf->cs&3) == DPL\_USER)

exit();

涉及到的主要数据结构:

extern uint vectors[]; // in vectors.S: array of 256 entry pointers

struct gatedesc idt[256];

// hardware and by trapasm.S, and passed to trap().

struct trapframe {

// registers as pushed by pusha

uint edi;

...

};

另外，如果是执行的是系统调用,则会通过syscall.c中的syscall()调用对应的处理程序，之后控制流返回trapasm.S，还会恢复被压入栈的寄存器，执行iret跳回到用户空间，完成内核态到用户态的转变。

1. 请以系统调用setrlimit (该系统调用的作用是设置资源使用限制)为例，叙述如何在 XV6 中实现一个系统调用。(提示:需要添加系统调用号，系统调用函数，用户接口等等)。

在syscall.h中添加系统调用号 #define SYS\_setrlimit 22；

// syscall.h

……

#define SYS\_mkdir 20

#define SYS\_close 21

#define SYS\_setrlimit 22 // add by yangyu

在syscall.c中添加对应的处理程序的调用接口

// syscall.c

……

static int (\*syscalls[])(void) = {

……

[SYS\_mkdir] sys\_mkdir,

[SYS\_close] sys\_close,

[SYS\_setrlimit] SYS\_setrlimit, // add by yangyu

在sysproc.c中添加系统调用函数int sys\_setrlimit(void)，具体实现对于进程资源使用限制的设置；

// syspro.c

……

int sys\_uptime(void)

{

uint xticks;

acquire(&tickslock);

xticks = ticks;

release(&tickslock);

return xticks;

}

// 在这里面写逻辑,限制进程资源的使用

int sys\_setrlimit(void)

{

// to do

}

在user.h中声明系统调用接口int setrlimit(int resource, const struct rlimit \* rlim)；

// syspro.c

……

// system calls

int fork(void);

int exit(void) \_\_attribute\_\_((noreturn));

…… // 调用该接口陷入内核执行系统调用

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

在usys.S添加SYSCALL(setrlimit)。

// usys.S

……

SYSCALL(sleep)

SYSCALL(uptime)

SYSCALL(setrlimit)

Q：了解与文件系统相关的系统调用及相应数据结构，简述各个系统调用的作用。

A：

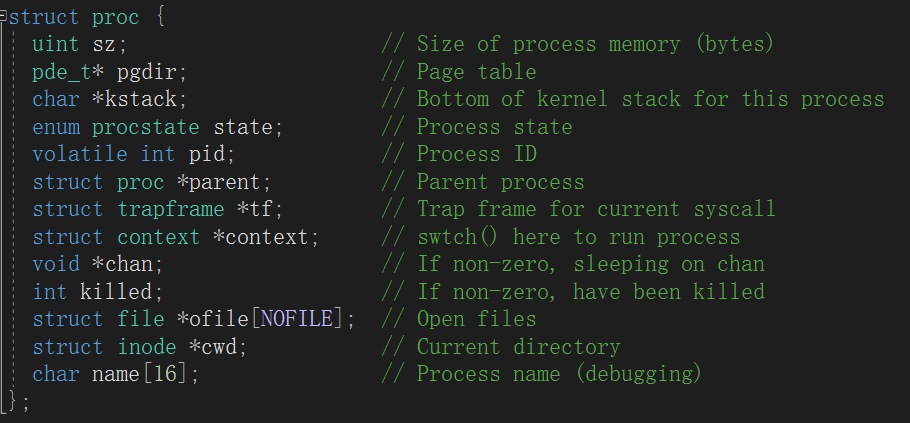
* 系统调用：

|  |  |
| --- | --- |
| 函数 | 功能 |
| static int argfd(int n, int \*pfd, struct file \*\*pf) | 获取第n个字大小的系统调用参数作为文件描述符，然后返回描述符和相应的struct文件。 |
| static int fdalloc(struct file \*f) | 为给定文件分配文件描述符。成功时从调用者那里接管文件引用。 |
| int sys\_dup(void) | 文件描述符引用+1 |
| int sys\_read(void) | 读文件描述符 |
| int sys\_write(void) | 写文件描述符 |
| int sys\_close(void) | 关闭文件 |
| int sys\_fstat(void) | 改变文件统计信息 |
| int sys\_link(void) | 创建路径new作为到旧inode的链接。 |
| static int isdirempty(struct inode \*dp) | 目录dp是否为空（“.”和”..”除外） |
| int sys\_unlink(void) | 移除link |
| static struct inode\* create(char \*path, short type, short major, short minor) | 创建inode |
| int sys\_open(void) | 打开文件 |
| int sys\_mkdir(void) | 创建目录 |
| int sys\_mknod(void) | 创建文件 |
| int sys\_chdir(void) | 更改当前目录 |
| int sys\_exec(void) | 执行用户程序 |
| int sys\_pipe(void) | 使用管道 |

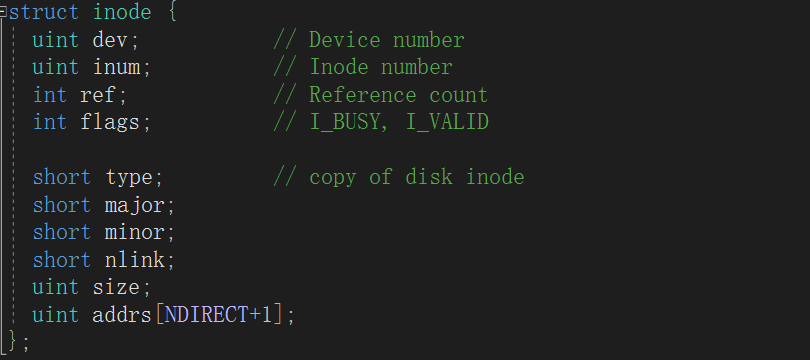
* 对应关系：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| System call numbers | | | | | |
| fork | 1 | **fstat** | 8 | **open** | 15 |
| exit | 2 | **chdir** | 9 | **write** | 16 |
| wait | 3 | **dup** | 10 | **mknod** | 17 |
| pipe | 4 | **getpid** | 11 | **unlink** | 18 |
| read | 5 | **sbrk** | 12 | **link** | 19 |
| kill | 6 | **sleep** | 13 | **mkdir** | 20 |
| exec | 7 | **uptime** | 14 | **close** | 21 |

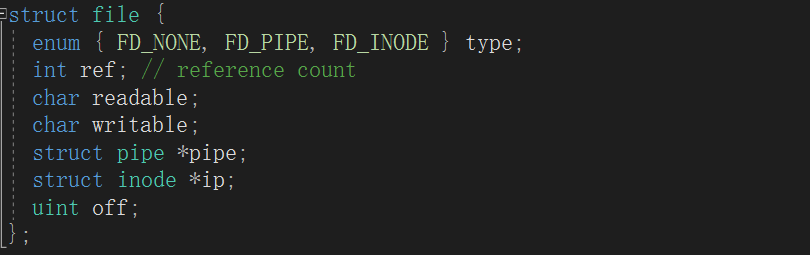
* 数据结构：
  + 进程状态proc



* + 索引节点的内存副本



* + file

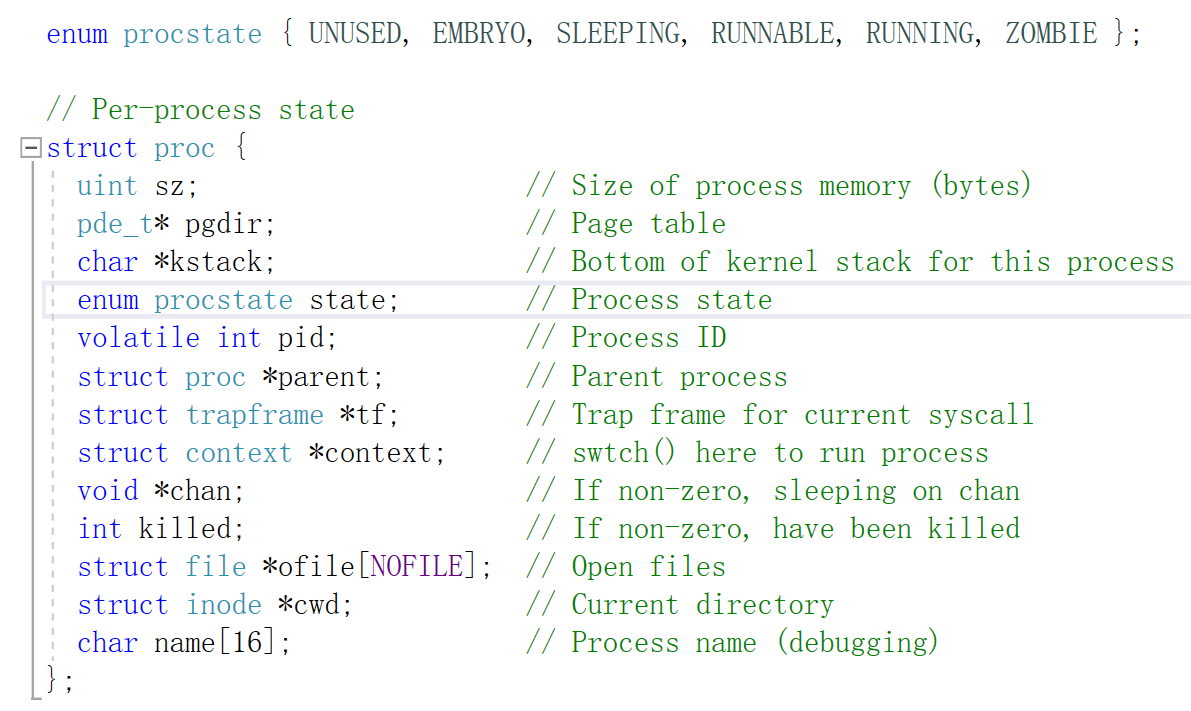


**XV6进程线程部分**

5. **XV6中进程管理的数据结构是什么? 其中包含哪些内容? 是如何进行管理的? 它们是如何初始化的?**

在xv6中进程管理的数据结构即PCB为proc结构体，定义在proc.h中。

在proc结构体中，定义了进程占用的内存空间规模sz；pgdir以x86 硬件要求的格式保存了进程的页表，提供每个线程独有的地址空间；kstack进程内核栈的基地址，即地址空间的最高位；enum procstate state，进程的状态，包括未使用、初始态、睡眠、就绪、运行、僵尸态;pid 进程的标识符；proc patent 父进程;trapframe \*tf 进程中断保存的寄存器;context \*context进程上下文；\*chan睡眠进程列表；killed 若不为0，进程被杀死；\*ofile[NOFILE]进程打开的文件列表；inode\*cwd进程当前的目录；name[16]进程的名字。



Xv6通过数据结构ptable对进程PCB进行管理，代码如下：（定义在proc.c中）

struct {  
struct spinlock lock;  
struct proc proc[NPROC];  
} ptable;

进程的初始化分为第一个进程和后续子进程：

1：第一个进程初始化：自举程序bootmain.c，在主方法main.c中初始化一些设备和子系统后，调用proc.c中的userinit()建立

2：userinit（）首先调用allocproc分配空间。而后userinit调用setupkvm为第一个用户进程分配最初只映射内核区的页表，大小为4096字节即4k。并调用inituvm分配了一个物理页，并将这个物理地址页映射到虚拟地址为0处，将原来的加载到物理地址为0处的initcode.S的代码加载到这个虚拟地址对应的物理页中。另外将进程的trapframe设置为初始的用户模式状态，以便能从内核态返回到用户态)。P->name设置为initcode，p->cwd设置为进程当前工作目录。初始化成功后进程状态变为RUNNABLE，就绪进程。

3：之后的进程初始化：调用fork()函数，在allocproc成功后，利用copyuvm函数以一次一页的方式复制父进程地址空间，并复制父进程的其他内容。Frok函数将子进程的eax设为0，实现子进程返回0，父进程返回子进程pid。

4：无论是userinit()还是fork()都会调用allocproc()函数。

5：在xv6中其他进程的初始化会调用allocproc()函数，allocproc的工作是在页表中分配一个槽（即结构体 struct proc），并初始化进程的状态，为其内核线程的运行做准备。简而言之，即设置好一个特别准备的内核栈和一系列内核寄存器，并将进程状态由UNUSED改写为EMBRYO。

具体的实现为首先定义进程p，然后将进程列表上锁，在进程列表中选出尚未被使用的进程，将未使用进程的状态切换为EMBRYO,如果没有则释放进程列表返回0。然后为线程分配栈空间，栈空间自上到下生长，如果物理空间开辟失败，则重置进程状态为UNUSED,返回0。再分配陷阱帧trapframe，保存系统调用或中断发生时需要保存的信息。再设置进程切换需要保存的上下文，将eip寄存器的值设为forkret的相应地址，用于区分第一个线程及后续线程。

**进程状态的变化就是通过特定的功能函数来实现的：**

allocproc进程分配开辟函数会将进程状态由UNUSED切换为EMBRYO，

如果分配成功，userinit和fork函数就会将进程状态切换为就绪RUNNABLE.

Scheduler函数调用就绪队列会将进程状态转变为RUNNING.

EXIT进程退出函数，退出进程后将进程状态切换为ZOMBIE; 僵尸进程在wait函数中释放；

yeild函数，解除CPU占用，进程状态重新变为就绪RUNNABLE；

sleep函数，进程状态转为SLEEPING；wakeup，唤醒进程，即变为RUNNABLE。Kill杀死进程，如果为SLEEPING则变为RUNNABLE。

**在线程初始化中只利用kalloc开辟了栈空间。**

**6.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?你认为采用这种设计思路有什么好处?**

进程有3个基本状态：运行状态（RUNNING）、就绪状态（READY）和等待状态（WAITING，或称阻塞状态BLOCKING），除此之外还有一些衍生的状态，这些状态根据操作系统设计不同而不同。

XV6中借助枚举类型，枚举了进程的6种状态：UNUSED未使用态、EMBRYO初始态、SLEEPING等待态、RUNNABLE就绪态、RUNNING运行态、ZOMBIE僵尸态。

状态的含义如下：

UNUSED：进程未被创建（即PCB空闲）时的状态，我们认为这就是最开始的状态，而且这里给我们的感觉是，PCB是一开始就存在的，只不过在系统请求创建进程时才会将空闲的PCB分配给这个进程；

EMBRYO：需要分配一个进程控制块且找到一个处于UNUSED状态的进程控制块时，把此进程控制块状态设置为要使用的状态；

SLEEPING：进程由于等待某资源等原因无法执行，进入睡眠状态，即等待态；

RUNNABLE：进程获得了除CPU之外的所有资源，处于可运行状态，即就绪态，这个时候会有一个就绪队列，处于就绪态的进程挨个获得CPU（无抢占的前提下）；

RUNNING：进程获得CPU，正在运行的状态，即执行态；

ZOMBIE：进程结束的状态。

XV6中进程状态转化图如下。



这和Linux中进程状态有点区别。

Linux内核中定义了以下几种状态：

#define TASK\_RUNNING 0

#define TASK\_INTERRUPTIBLE 1

#define TASK\_UNINTERRUPTIBLE 2

#define TASK\_ZOMBIE 4

#define TASK\_STOPPED 8

其中：

TASK\_RUNNING是就绪态，进程当前只等待CPU资源，其他资源已经全部到位。

TASK\_INTERRUPTIBLE和TASK\_UNINTERRUPTIBLE都是阻塞态，进程当前正在等待除CPU外的其他系统资源，两者的区别在于前者可以被信号唤醒，后者不可以。

TASK\_ZOMBIE是僵尸态，进程已经结束运行，但是进程控制块尚未注销，个人感觉属于一个中间的一个临时状态。

TASK\_STOPPED是挂起状态，主要用于调试目的，我们理解的挂起就是把当前程序存入磁盘，进程接收到SIGSTOP信号后会进入该状态，在接收到SIGCONT后又会恢复运行。

我们认为之所以这样设计是为了实现多道程序的交叉运行，和方便设计各种进程调度算法。在时间片轮转调度算法中正处于RUNNING状态的程序在时间片用完的时候，不能直接降级为和什么资源都没有的进程一起等待调度的优先级（因为它还是拥有除CPU外其他资源的，这时只要得到CPU即可马上再次继续运行），所以它就会转成RUNNABLE状态同时让出CPU。而正在RUNNING的程序如果遇到资源申请，就会主动放弃CPU进入SLEEPING状态。但是SLEEPING状态的程序如果得到资源之后并不能马上进入RUNNING而是先进入就绪队列等待CPU调度。我们认为这里的就绪队列就是为了多一层抽象、加一级缓冲。

**7. XV6中的最大进程数是多少? 如何执行进程的切换? 阐述切换过程。**

XV6中最大进程数为NPROC=64个。

切换过程（涉及源码：pro.c switch.s）：

1. 当前进程由running状态变成其它状态后，调用scheduler的入口函数sched

2. sched中，先检查当前进程的状态，CPU关中断的深度等，调用switch函数，把CPU控制权交给scheduler

3.在switch中，XV6会把旧进程的上下文（上下文：寄存器ebp,ebx,esi,edi中的内容）保存在旧进程的内核栈中，然后保存旧栈指针，再切换到scheduler的内核栈的指针，接着把schduler的上下文放入寄存器中。

4. scheduler会从上次的断点继续运行，把页表切换为内核栈中的页表，然后从进程表ptable中找出就绪的进程，加载相应的页表，并且调用switch，再把CPU的控制权交给新进程。若没有找到，则进入释放和申请进程表的锁的循环中，并且每次循环都会尝试遍历进程表找就绪的新进程，找到则停止循环。

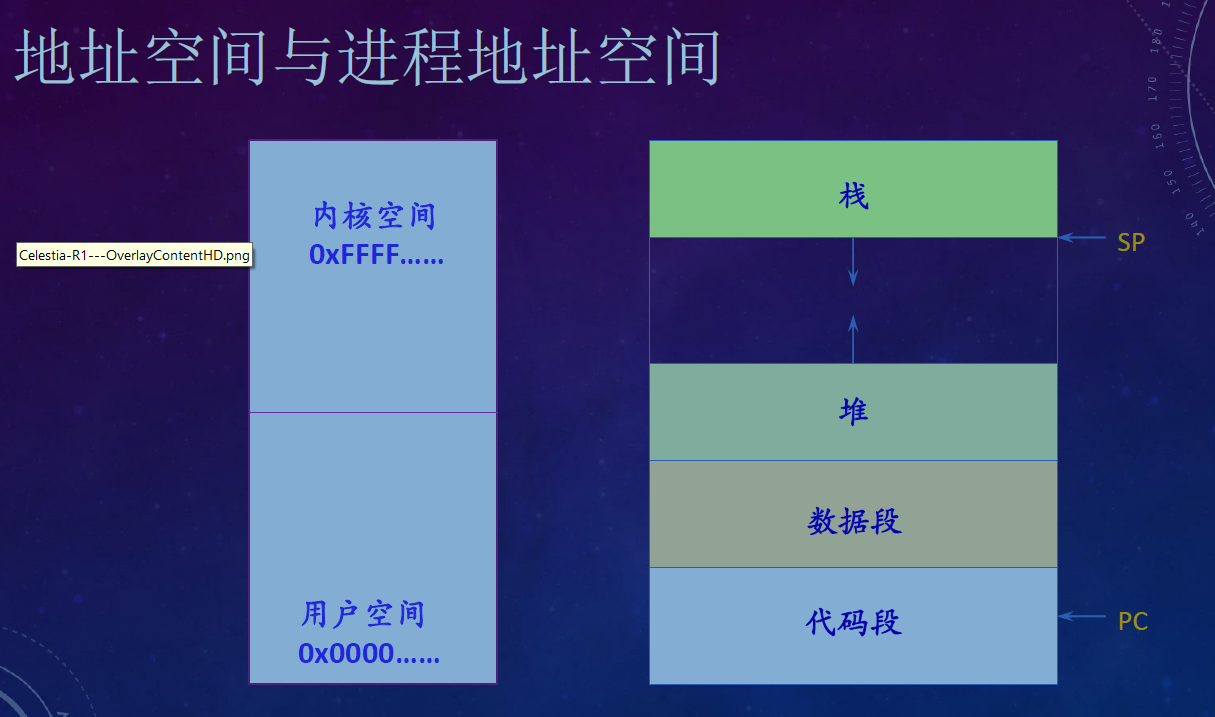
5.换新线程上CPU，同样是通过switch函数，switch会把schduler的上下文保存在内核栈中，然后保存当前栈指针，再将栈指针切换为新进程的栈指针，最后将新进程的上下文存入寄存器内，从而新进程掌握了CPU的控制权。

**8. 阐述XV6中的进程在内存中是如何布局的。**

操作系统给每个进程都分配了一个地址空间，其中内核态进程一般在内核空间，用户态进程一般在用户地址空间。进程在内存中占用的空间包括栈、堆、数据段和代码段。

栈是程序执行前静态分配的地址空间，栈会向下增长。

堆是程序执行时，根据需要动态分配的空间，包括malloc、calloc、realloc函数分配的空间。堆内存由所有线程共享，并会向上增长。



在用户空间中，自底向上依次是代码段，数据段，栈，堆。而在用户空间和内核空间之间有一段0x80000000-0x80100000用于BIOS。

在内核空间中，分别是代码段，数据段，自由存储区

而从0xFE000000开始的Device memory主要用于其他通过内存映射的IO设备的存储

**XV6内存管理部分**

9. XV6的锁是如何实现的，有什么操作? xchg是什么指令，该指令有何特性?

xv6的锁是自旋锁，在请求不到锁时不停自旋。数据结构在spinlock.h中，是一个结构体，包括locked,是否持有锁，锁的名字，占有锁的CPU，调试栈这几个变量。操作在spinlock.c里，包括方法有初始化锁initlock，开中断popcli、关中断pushcli、获得锁acquire、释放锁release、得到调用的栈进行调试getcallerpcs、判断持有锁的是否是当前CPU的holding方法。

xchg是386硬件上的一条特殊指令 ，它是一个原子操作。在这个原子操作中， xchg 交换了内存中的一个字和一个寄存器的值。函数acquire在循环中反复使用xchg，在x86.h中使用内联汇编实现这个原子操作。

while(xchg(&lk->locked, 1) != 0)

每一次都读取 lk->locked ,然后设置1,并返回lk->locked原本的值。

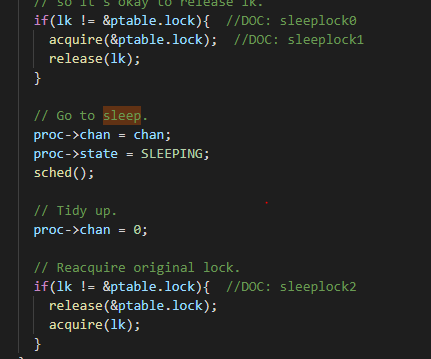
如果锁已经被持有了， lk->locked 就已经为1了，故xchg会返回1然后继续循环。如果xchg 返回0，但是acquire 已经成功获得了锁，即locked已经从0变为了1，这时循环可以停止了。

10. 基于XV6的spinlock，请给出实现信号量的设计方案 (写出相应伪代码)

首先需要看一下spinlock的用法和实现原理。

Spinlock相关的操作有acquire release sleep 与sleep相对应的还有一个wakeup方法，但是这个方法和锁没有关系。

Sleep接受两个参数，一个是chan一个是锁对象，这个chan的作用相当于是个队列。它是这么起作用的，每个进程PCB里都有一个chan成员，代表这个进程当前等在哪个队列上。因此这个chan相当于是个队列标记。Sleep会先获取ptable的锁，也就是含有所有PCB的那个总表，然后释放传进来的锁。



接着就进行调度，换别的进程运行。当自己再次恢复的时候将PCB上chan清除，表示自己不再哪个队列上等待。

Wakeup接受的参数就是chan,他会先获取PTABLE的锁然后到PTABLE里找一个当前在这个chan上等待的进程，如果进程是SLEEP状态就将进程设置成RUNNABLE状态。

struct semaphore {

int value;

struct spinlock lock;

struct proc \*queue[NPROC]; // 进程等待队列,这是⼀个循环队列

int end; // 队尾

int start; // 队头

};

// 初始化信号量

void sem\_init(struct semaphore \*s, int value) {

s->value = value;

initlock(&s->lock, "semaphore\_lock");

end = start = 0;

}

void sem\_wait(struct semaphore \*s) {

acquire(&s->lock); // 竞争锁,如果竞争不到进⼊⾃旋

s->value--;

if (s->value < 0) {

s->queue[s->end] = myproc(); // myproc()获取当前进程, 放⼊队尾

s->end = (s->end + 1) % NPROC; // 循环队列计算新的队尾

// 1. 释放锁(下⼀个sem\_wait 的进程才能进⼊acquire),

// 2. 然后进⼊睡眠等待, 被唤醒时重新竞争锁

sleep(myproc(), &s->lock);

}

release(&s->lock);

}

void sem\_signal(struct semaphore \*s) {

acquire(&s->lock); // 竞争锁

s->value++;

if (s->value <= 0) {

wakeup(s->queue[s->start]); // 唤醒循环队列头的进程

s->queue[s->start] = 0;

s->start = (s->start + 1) % NPROC; // 重新计算队头

}

release(&s->lock);

}

// proc.h

// Per-process state

struct proc {

uint sz; // Size of process memory (bytes)

pde\_t\* pgdir; // Page table

char \*kstack; // Bottom of kernel stack for this process

enum procstate state; // Process state

volatile int pid; // Process ID

struct proc \*parent; // Parent process

struct trapframe \*tf; // Trap frame for current syscall

struct context \*context; // swtch() here to run process

void \*chan; // If non-zero, sleeping on chan

int killed; // If non-zero, have been killed

struct file \*ofile[NOFILE]; // Open files

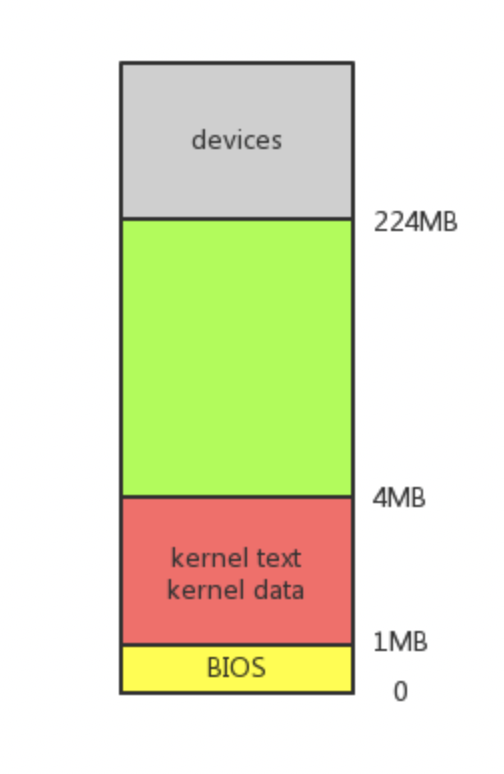
struct inode \*cwd; // Current directory

char name[16]; // Process name (debugging)

};

11. 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)?

如下图所示，xv6默认主存大小为224MB，主存开始0~1MB保留给BIOS存储相关信息，xv6内核代码段和数据段被loader加载到主存开始的1MB~4MB的空间，4MB~224MB这段内存可以自由分配。灰色的devices区域是其它设备引入的存储空间，仅供设备自身使用，并不是主存的一部分。



**初始化过程为：**

内核代码存在于物理地址低地址的0x100000处，页表为main.c文件中的entrypgdir数组，其中虚拟地址低4M映射物理地址低4M，虚拟地址 [KERNBASE, KERNBASE+4MB) 映射到 物理地址[0, 4MB)；

紧接着调用kinit1初始化内核末尾到物理内存4M的物理内存空间为未使用，然后调用kinit2初始化剩余内核空间到PHYSTOP为未使用。kinit1调用前使用的还是最初的页表（也就是是上面的内存布局），所以只能初始化4M，同时由于后期再构建新页表时也要使用页表转换机制来找到实际存放页表的物理内存空间，这就构成了自举问题，xv6通过在main函数最开始处释放内核末尾到4Mb的空间来分配页表，由于在最开始时多核CPU还未启动，所以没有设置锁机制。kinit2在内核构建了新页表后，能够完全访问内核的虚拟地址空间，所以在这里初始化所有物理内存，并开始了锁机制保护空闲内存链表；

然后main函数通过调用void kvmalloc(void)函数来实现内核新页表的初始化；

// kalloc.c

// Initialization happens in two phases.

// 1. main() calls kinit1() while still using entrypgdir to place just

// the pages mapped by entrypgdir on free list.

// 2. main() calls kinit2() with the rest of the physical pages

// after installing a full page table that maps them on all cores.

void

kinit1(void \*vstart, void \*vend)

{

initlock(&kmem.lock, "kmem");

kmem.use\_lock = 0;

freerange(vstart, vend);

}

void

kinit2(void \*vstart, void \*vend)

{

freerange(vstart, vend);

kmem.use\_lock = 1;

}

// kmap.c

……

// This table defines the kernel's mappings, which are present in

// every process's page table.

static struct kmap {

void \*virt;

uint phys\_start;

uint phys\_end;

int perm;

} kmap[] = {

{ (void\*)KERNBASE, 0, EXTMEM, PTE\_W}, // I/O space

{ (void\*)KERNLINK, V2P(KERNLINK), V2P(data), 0}, // kern text+rodata

{ (void\*)data, V2P(data), PHYSTOP, PTE\_W}, // kern data+memory

{ (void\*)DEVSPACE, DEVSPACE, 0, PTE\_W}, // more devices

};

……

12. XV6的动态内存管理是如何完成的?

**动态内存管理管哪块**

有一个 kmem（链表），用于管理可分配的物理内  
存页。（vend=0x00400000，也就是可分配的内存页最大为 4Mb）

**13. XV6的虚拟内存是如何初始化的?画出XV6的虚拟内存布局图，请说出每一部分对应的 内容是什么。**

main函数通过调用void kinit1(void \* vstart, void \* vend), void kinit2(void \* vstart, void \* vend), void kvmalloc(void)函数来实现内核新页表的初始化。虚拟地址与物理地址的转换接口：

// memlayout.h

// Memory layout

#define EXTMEM 0x100000 // Start of extended memory

#define PHYSTOP 0xE000000 // Top physical memory

#define DEVSPACE 0xFE000000 // Other devices are at high addresses

// Key addresses for address space layout (see kmap in vm.c for layout)

#define KERNBASE 0x80000000 // First kernel virtual address

#define KERNLINK (KERNBASE+EXTMEM) // Address where kernel is linked

#ifndef \_\_ASSEMBLER\_\_

static inline uint v2p(void \*a) { return ((uint) (a)) - KERNBASE; }

static inline void \*p2v(uint a) { return (void \*) ((a) + KERNBASE); }

#endif

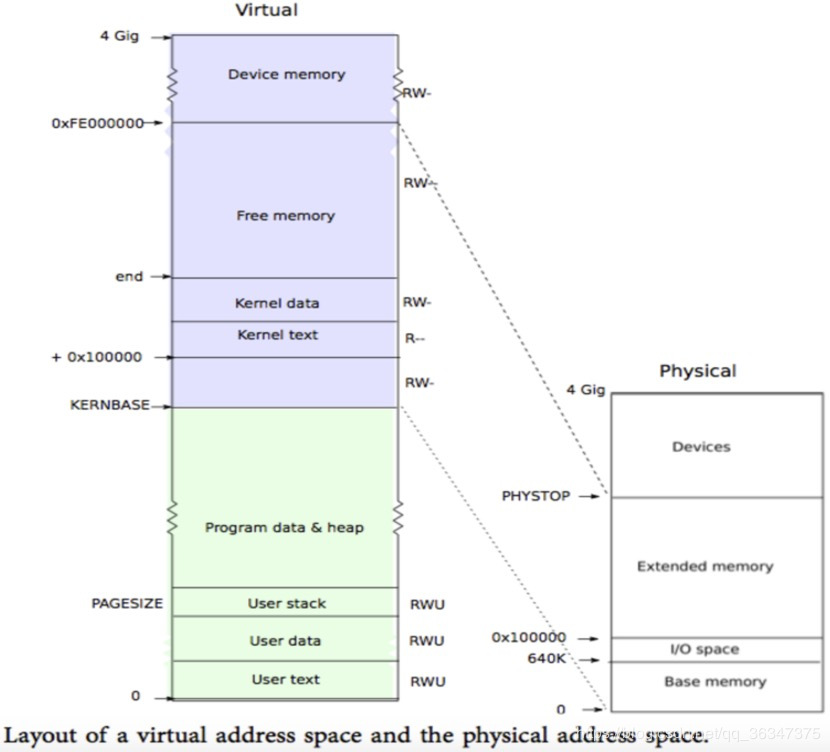
#define V2P(a) (((uint) (a)) - KERNBASE)

#define P2V(a) (((void \*) (a)) + KERNBASE)

#define V2P\_WO(x) ((x) - KERNBASE) // same as V2P, but without casts

#define P2V\_WO(x) ((x) + KERNBASE) // same as V2P, but without casts

内存布局：



**14. 关于XV6的内存页式管理。发生中断时，用哪个页表?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?画出从虚拟地址到物理地址的转换图。**

发生中断的时候，使用的是对应用户进程的页表。因为用户虚拟空间中存放有内核相关的内容，在发生中断陷入内核时无需切换页表。

一个内页的大小是4096B，页目录有1024项，页表有1024项，因而最大支持1024\*1024\*1024B=4GB内存。

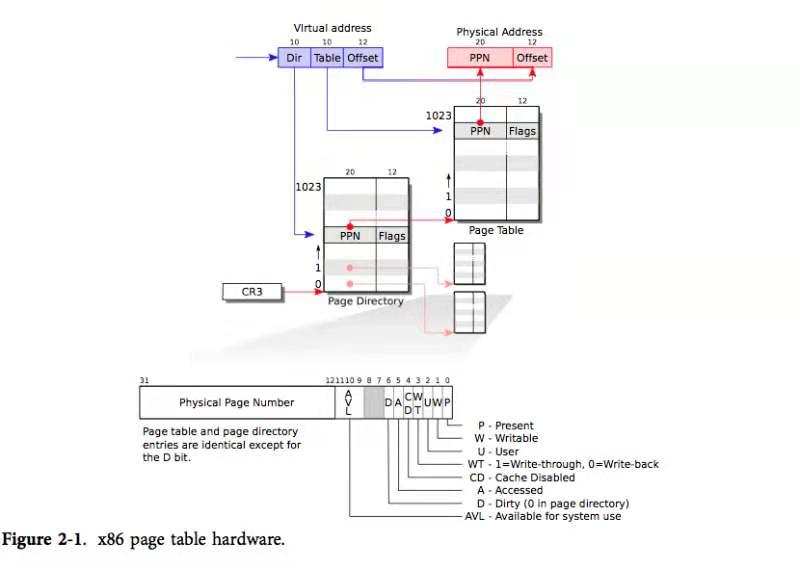
// Page directory and page table constants. 页目录和页表常量

#define NPDENTRIES      1024    // # directory entries per page directory

#define NPTENTRIES      1024    // # PTEs per page table

#define PGSIZE          4096    // bytes mapped by a page

虚拟地址到物理地址的转化图如下：



**补充问题：XV6何时修改cr3寄存器切换页表？**

XV6定义了两个函数用于修改cr3中的页目录指针，分别是：

switchkvm(void)：没有进程在运行时，切换页表为仅有内核部分的页表。

switchuvm(struct proc \*p)：切换页表为当前进程p的页表。

调用这两个函数的地方：

1.系统初始化时，调用switchkvm()创建一个内核页表并启用，从而进行进程调度。

2.非初始CPU启动时，调用switchkvm()启用内核页表。

3.进程调度时，进程运行前调用switchuvm()切换到进程p的页表，运行结束（包括时间片用完等）调用switchkvm()启用内核页表，进行进程调度。

4.调用growproc(int n)增加进程空间时修改了进程页表，修改成功后调用switchuvm()重新载入进程p的页表。

5.exec(char \*path, char \*\*argv)系统调用函数中，调用switchuvm()使用新建进程页表。

15. 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)?

相关函数在vm.c中。main函数调用kvmalloc，创建一个进程的内核地址空间的页表，主要通过setupkvm函数完成，首先分配一页内存作为页目录，然后调用mappages函数按照kmap数据结构映射内核虚拟地址空间到物理地址空间。

虚拟地址与物理地址的映射过程中主要涉及页表项的创建与查询。相关的函数有mappages与walkpgdir。其中mappages负责在页表中建立一段虚拟内存到物理内存的映射，涉及到页表项的创建与填写，在这过程中调用了walkpgdir。Walkpgdir函数负责页表项的创建与查询。

页表项的创建与填写由这两个函数配合完成，Walkpgdir函数有参数alloc，如果alloc为1，则会在页表项不存在时创建。Mappages通过设置alloc参数为1，并传入虚拟地址调用walkpgdir创建页表项。在这过程之中，walkpgdir通过虚拟地址的前10位找到页目录项，如果不存在，说明要找的页表页尚未分配，则会调用kalloc函数分配一页，并填写页目录项。

页表项的查询由walkpgdir完成，首先取虚拟地址的前10位，查询页目录项，取得页表地址，然后在页表页中通过虚拟地址中间10位查询页表，返回对应页表项即可。

**XV6文件系统部分**

第13题 xv6中如何处理page fault

xv6地址翻译由硬件完成。xv6处理中断的c语言代码包含在trap.c的trap()函数,中目前的trap()函数中不包含对page fault的处理，如果需要处理缺页中断，可以在trap()函数中增加对case T\_PGFLT的处理。可以分两种情况处理，即内核态的缺页中断和用户态的缺页中断。

**New 16.文件部分的系统调用**

16. 阅读文件ide.c。这是一个简单的ide硬盘驱动程序，阐述其代码流程。

xv6 的文件系统分6层实现，从底至顶如下：

System calls File descriptors

Pathnames Recursive lookup

Directories Directory inodes

Files Inodes and block allocator

Transactions Logging

Blocks Buffer cache

底层通过块缓冲Buffer cache读写IDE 硬盘，它同步了对磁盘的访问，保证同时只有一个内核进程可以修改磁盘块；

第二层Loggins向上层提供服务，该层实现了文件系统的一致性，使得更高层的接口可以将对磁盘的更新按会话打包，通过会话的方式来保证这些操作是原子操作(要么都被应用，要么都不被应用)；

第三层提供无名文件，每一个这样的文件由一个 i 节点和一连串的数据块组成；

第四层将目录实现为一种特殊的 i 节点，它的内容是一连串的目录项，每一个目录项包含一个文件名和对应的 i 节点；

第五层提供了层次路经名（如/usr/rtm/xv6/fs.c这样的），这一层通过递归的方式来查询路径对应的文件；

最后一层将许多 UNIX 的资源（如管道，设备，文件等）抽象为文件系统的接口，极大地简化了程序员的工作。

**17. 简述XV6中bio.c和buf.h的内容。**

这部分的内容与老师课上所讲的“磁盘高速缓存”的概念对应。

buf.h中的buf结构体

是将一个块文件读取进入内存中管理的数据结构体，包含：

1. 文件系统附加的超参数：包含该block的设备和sector号，在LRU队列上的前一个和后一个指针

2. 实际的存储的块数据（也是硬盘中存储的内容）：data

3. 状态参数，该块在内存中是否忙，是否有效，是否Dirty（指在内存（）被写入但还没有写回磁盘）

而内存中将所有buf的管理结构叫做bcache，它的功能有如下：

（1）同步对磁盘的访问，使得对于每一个块，同一时间只有一份拷贝放在内存中并且只有一个内核线程使用这份拷贝；

（2）缓存常用的块以提升性能。注意，b系列函数一部分定义在bio.c中，一部分在fs.c中（使用日志层的高层级块操作）。

(3) 放在内存中，加入管控信息，进行管控

bcache

定义了将每一个块文件的buf管理起来的内存结构式bcache，其中使用head实际访问具体的buf块。它将所有的buf组织成双向链表的形式，以方便使用LRU进行置换。

对于block的底层操作函数：

1. binit：初始化bcache结构体

2. bget：获取bcache中是否有对应设备和块号的buf块，如果有其他进程也通过bget在使用该buf，则使用锁睡眠等待。直到另一进程唤醒该进程。

如果对应设备块号的块不在内存中，则将一个空闲块或者未使用块（not busy）的信息变为busy，修改元信息，之后再在bread中访问磁盘，将具体内容读出来。

4. bread/bwrite和brelse配合使用，

bread在使用调用iderw进行同步，使用bget将对应块读入缓存后，返回对应的buf结构指针。

而bwrite也使用iderw，将内存中的buf写回磁盘上的对应block，只需将b的dirty位置置为1.

5. brelse直接释放一块buf，并把该buf从双向队列中清除。

**17. 阐述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。**

数据结构bcache维护了一个由struct buf组成的双向链表，同时bcache.lock用户互斥访问；

首先系统调用binit()初始化缓存，随即调用initlock初始化bcache.lock，然后循环遍历buf数组，采用头插法逐个链接到bcache.head后；

上层文件系统读磁盘时，调用bread()，随即调用bget()检查请求的磁盘块是否在缓存中，如果命中，返回缓存命中结果。如果未命中，转到底层的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；

上层文件系统写磁盘时，调用bwrite()直接将缓存中的数据写入磁盘。Buffer Cache层不会尝试执行任何延迟写入的操作，何时调用bwrite()写入磁盘是由上层的文件系统控制的；

上层文件系统可通过调用brelse()释放一块不再使用的缓冲区。

// buf.h

struct buf {

int flags;

uint dev;

uint sector;

struct buf \*prev; // LRU cache list

struct buf \*next;

struct buf \*qnext; // disk queue

uchar data[512];

};

// bio.c

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;

void binit(void)

{

struct buf \*b;

initlock(&bcache.lock, "bcache");

//PAGEBREAK! 头插法,每次都是插入到bcache.head的后面

// Create linked list of buffers

bcache.head.prev = &bcache.head;

bcache.head.next = &bcache.head;

for(b = bcache.buf; b < bcache.buf+NBUF; b++){

b->next = bcache.head.next;

b->prev = &bcache.head;

b->dev = -1;

bcache.head.next->prev = b;

bcache.head.next = b;

}

}

// Return a B\_BUSY buf with the contents of the indicated disk sector.

struct buf\* bread(uint dev, uint sector)

{

struct buf \*b;

// 优先查找缓存

b = bget(dev, sector);

if(!(b->flags & B\_VALID))

iderw(b); // 命中失败时调用下一次接口真真实实读磁盘

return b;

}

// Write b's contents to disk. Must be B\_BUSY.

void bwrite(struct buf \*b)

{

if((b->flags & B\_BUSY) == 0)

panic("bwrite");

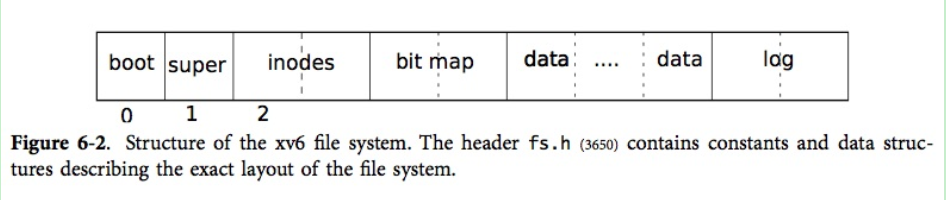
b->flags |= B\_DIRTY;

iderw(b); // 立即写, 未延迟写

}

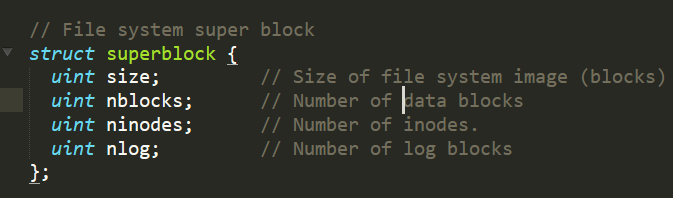
18. 阐述XV6文件系统的硬盘布局。

答：xv6把磁盘划分为几个区块，如下图所示。

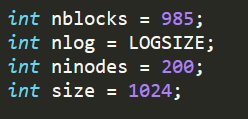


分别是：bootloader区，超级块，i节点区，位图区，数据区，log日志区。

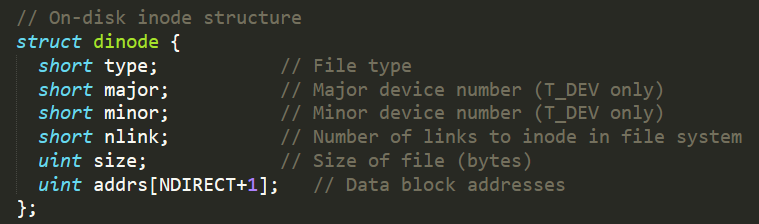
其中第0块存有bootloader，第1块叫做超级块，代码如下：



它包含了文件系统的元信息（如文件系统的总块数，数据块块数，i节点数目已经日志的块数）。具体的信息如下：



即文件系统共有1024个磁盘块，其中数据块有985块，每块大小为512B。Log块有10块，i节点有200个。从第2块开始存放i节点，i节点的定义如下：



每个dinode的大小为2+2+2+2+4+4\*13=64B，所以每个block能放512/64=8个i节点，200个i节点需要200/8=25个磁盘块。i节点后的是位图块，占用了2块（为什么是两块存疑）。剩下的大部分分块是数据块，985块，它们保存了文件和目录的内容。在磁盘的最后是日志块，10块，使用日志保持文件系统的一致性。一共有1+1+2+25+985+10=1024块磁盘块。

19. 阐述XV6的“文件”有哪些，以及文件，i节点，设备相关的数据结构。

A：XV6 中的⽂件有目录⽂件、 普通⽂件、 设备⽂件、 管道⽂件，在 sysfile.c ⽂件中有根据不同类型创建⽂件的⽅法。

调用 sys\_open()⽅法创建⼀个普通⽂件

调用 sys\_pipe()⽅法创建⼀个管道⽂件

调用 sys\_mknod()⽅法创建⼀个设备⽂件

调用 sys\_mkdir()⽅法创建⼀个目录⽂件

2、 file.h 中定义了打开文件 的结构体，其中有⼀项是⽂件的类型

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;

};

xv6 中 inode 对应的数据结构有两种，⼀个是存放在磁盘上的 inode，⼀个是在内存中的 inode，分别地定义在 fs.h 和 file.h 中。

struct dinode {

short type; // File type

short major; // Major device number (T\_DEV only)

short minor; // Minor device number (T\_DEV only)

short nlink; // Number of links to inode in file system

uint size; // Size of file (bytes)

uint addrs[NDIRECT+1]; // Data block addresses

};

struct inode {

uint dev; // Device number

uint inum; // Inode number

int ref; // Reference count

int flags; // I\_BUSY, I\_VALID

short type; // copy of disk inode

short major;

short minor;

short nlink;

uint size;

uint addrs[NDIRECT+1];

};

与设备⽂件相关的数据结构，在 file.c中

struct devsw {

int (\*read)(struct inode\*, char\*, int);

int (\*write)(struct inode\*, char\*, int);

};

全局变量打开文件表，在 file.c中

struct {

struct spinlock lock;

struct file file[NFILE];

} ftable;

每个进程对应的有打开文件表信息和当前所在目录，在proc.h 中

struct proc {

….

struct file \*ofile[NOFILE]; // Open files

struct inode \*cwd; // Current directory

….

}

Pipe.c ⽂件中定义的 pipe 结构体

struct pipe {

struct spinlock lock;

char data[PIPESIZE];

uint nread; // number of bytes read

uint nwrite; // number of bytes written

int readopen; // read fd is still open

int writeopen; // write fd is still open

};