**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?**

xv6的终端管理初始化各部分通过main.c中的main()函数调用。picinit()和oapicinit()初始化可编程中断控制器consoleinit()和uartinit()设置了I/O、设备端口的中断。接着，tvinit()调用trap.c中的代码初始化中断描述符表，关联vectors.S中的中断IDT表项,在调度开始前调用idtinit()设置32号时钟中断，最后在scheduler()中调用sti()开中断，完成中断管理初始化。

xv6在proc.c中的userinit()函数中，通过设置第一个进程的tf(trap frame)中cs ds es ss处于DPL\_USER(用户模式) 完成第一个用户态进程的设置，然后在scheduler中进行初始化该进程页表、切换上下文等操作，最终第一个进程调用trapret，而此时第一个进程构造的tf中保存的寄存器转移到CPU中,设置了 %cs 的低位，使得进程的用户代码运行在 CPL = 3 的情况下，完成内核态到用户态的转变。

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，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，xv6中通过命令cli关中断,sti开中断。

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。**

中断描述符表的每一项是一个中断描述符，在x86系统中，中断处理程序定义存储在IDT中。XV6的IDT有256个入口点，每个入口点中对应的处理程序不同，在出发trap时，只要找到对应编号的入口，就能得到对应的处理程序；

XV6中的数据结构中中断描述符用struct gatedesc表示：保存了类型、对应处理程序、Offset等信息，中断描述表由struct gatedesc idt[256]表示。

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)

**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中为proc结构体。

在proc结构体中，包含sz进程占用的内存空间规模；pgdir进程页表；kstack进程在内存栈中的基地址；enum procstate state，进程的状态;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对进程进行管理，代码如下：

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

1：第一个进程初始化：自举程序bootmain.c，调用主方法main.c，调用proc.c中的userinit()建立

2：第一个用户进程(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。另外还设置进程的trapframe，以便能从内核态返回到用户态)

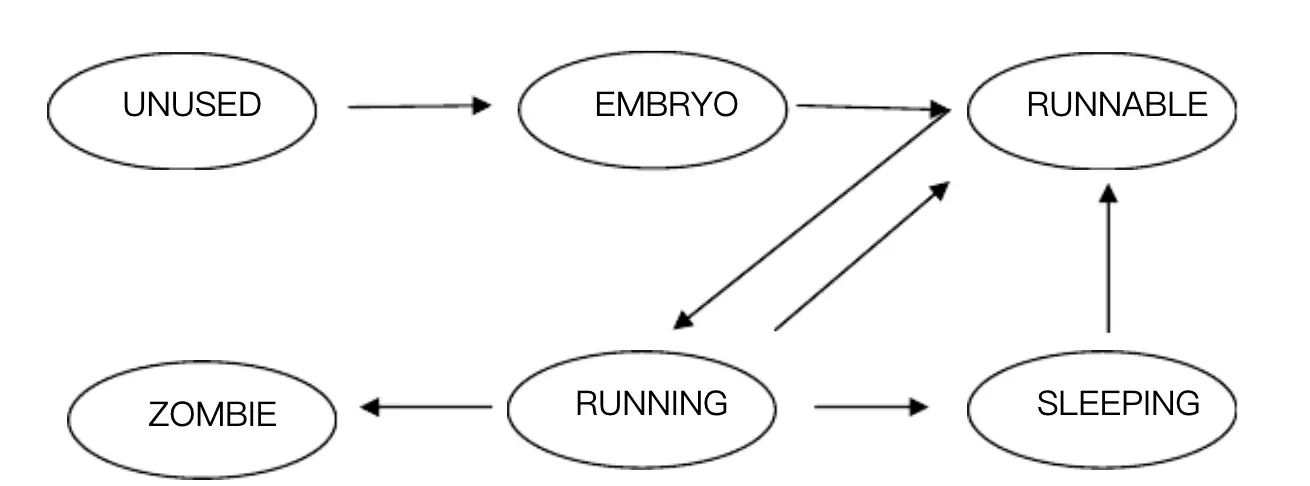
3：之后的进程初始化：调用fork()

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

5：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

在xv6中其他进程的初始化会调用allocproc()函数，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，如果没有则释放进程列表返回0。将未使用进程的状态切换为EMBRYO,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。

**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?**



在Linux中进程状态有就绪、深度睡眠、浅度睡眠、占有CPU执行、暂停和死亡但户口未注销。

在XV6中，进程状态包括unused（未使用）、embryo（初始态）、sleep（等待）、runnable（就绪）、run（运行）、zombie（僵尸），六种状态。

操作系统利用对进程不同状态的控制来实现cpu和系统资源的分配，占用不同的资源对应不同的进程状态，并实现操作系统的并发，即多道程序的交叉运行。

**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。

进程的切换有scheduler函数，从就绪队列中选择进程切换运行，以及中断和陷入导致的进程切换。

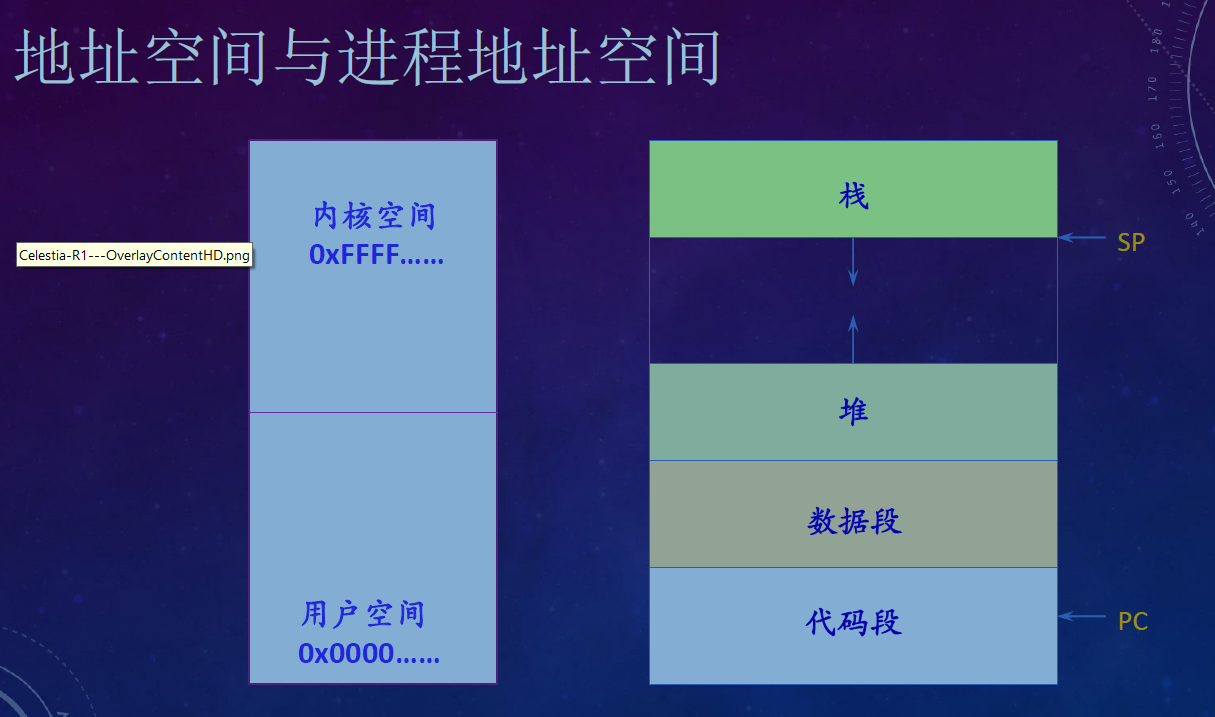
进程上下文在XV6中是当前进程的程序计数器PC和当前运行的CPU中各个寄存器的内容。包括程序状态寄存器、栈指针、通用寄存器和其他控制寄存器的值。将CPU硬件状态从一个进程换到另一个进程的过程称为上下文切换。

**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。



**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的锁利用spinlock结构体及相关函数实现。Xchg是用于交换值的原子操作，并返回原值。

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)

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)?

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；

最后内存布局和地址空间如下：内核末尾物理地址到物理地址PHYSTOP的内存空间未使用，虚拟地址空间KERNBASE以上部分映射到物理内存低地址相应位置。

// 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

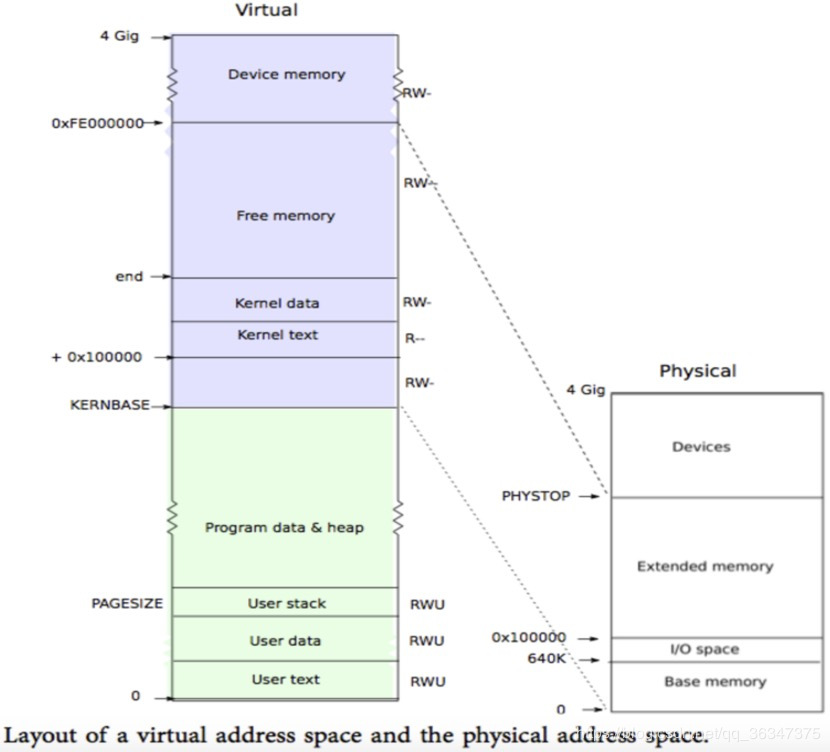
#define V2P(a) (((uint) (a)) - KERNBASE)

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

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

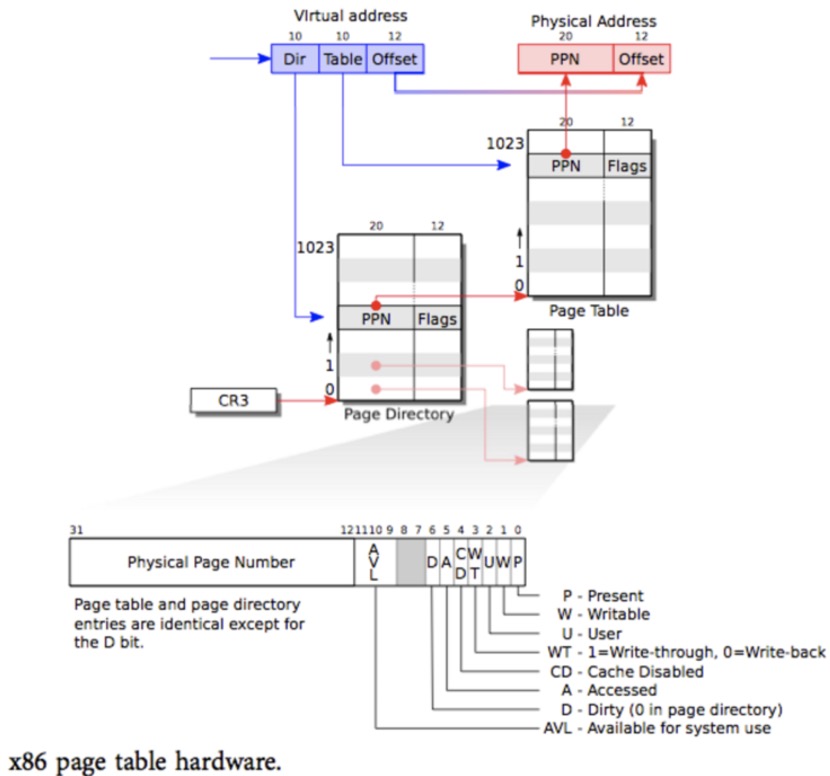
#define P2V\_WO(x) ((x) + KERNBASE) // same as V2P, but without casts

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。**

* 发生中断时，将换入cpu的进程的页表首地址存入cr3寄存器；一个内存页为4k；XV6页表采用的二级目录，一级目录有210210条，二级目录有210∗210210∗210条；页表项为2222Bytes，故页表有212/22=210=1024212/22=210=1024项；最大支持4G内存；



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)?

**XV6文件系统部分**

**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. 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。

// On-disk inode structure

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

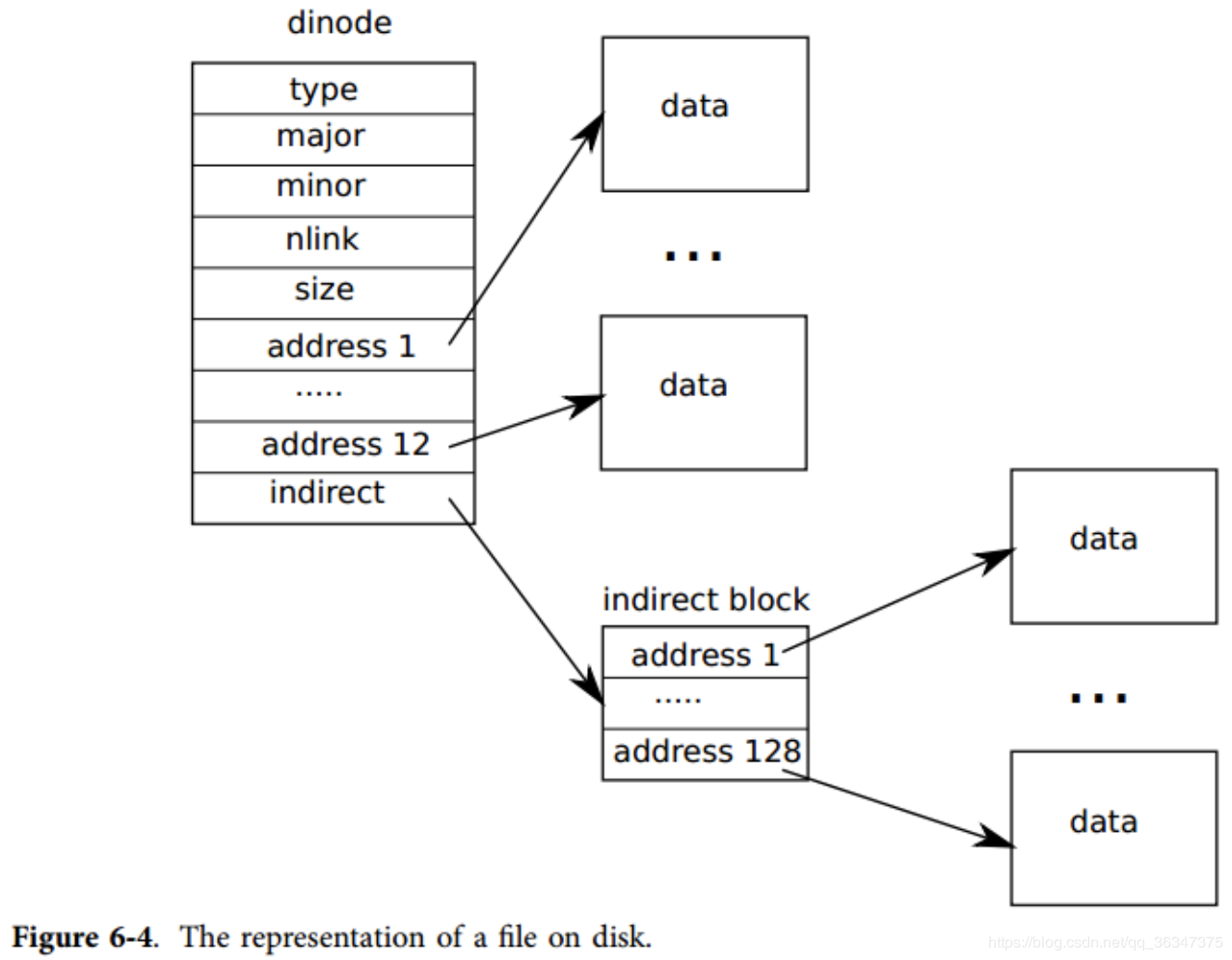
uint size; // Size of file (bytes)

// NDIRECT = 12, 前12个为直接索引,

// 第13个为间接索引, 可容纳128个直接索引

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

};



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。

xv6文件分为管道文件，设备文件和普通文件；

XV6最多支持同时打开100个文件，也就是分配100个文件句柄；

单个进程最多能打开16个文件。

// param.h

#define NOFILE 16 // open files per process

#define NFILE 100 // open files per system