



# 计算机科学与技术

## 软件设计与开发实践实验报告

班 级：英才班  
姓 名：张宁  
学 号：1150310607  
指导老师：曲明成

## 摘 要

操作系统是一种复杂的系统软件。xv6 是一个运行在基于 x86 架构的计算机系统上的类似 UNIX 的教学用操作系统。xv6 起源于 MIT。在目前的 MIT 本科生课程“6.828: Operating Systems Engineering”中，xv6 目前主要用于讲课。

我是在 win 系统上使用 Oracle VM VirtualBox 安装 32 位乌班图。

虚拟机来作为实验环境，在虚拟机上我可以使用 QEMU 来实现一个 x86 计算机，让 xv6 操作系统运行在一个用软件仿真出来的 x86 计算机上

# 目录

1 xv6 环境配置 .....	4
2 Lab 2: Memory Management .....	7
2.1 实验目的 .....	7
2.2 实验要求 .....	7
2.3 实验内容 .....	7
2.4 实验分析 .....	7
3 xv6 实验系统分析 .....	8
3.1 xv6 的系统结构（结合示意图） .....	8
3.2 xv6 的引导 .....	10
3.2.1 代码分析 .....	12
3.3 xv6 的进程与调度分析 .....	18
3.3.1 进程的概念 .....	18
3.3.2 进程控制块 .....	19
3.4 xv6 的内存管理 .....	35
3.5 xv6 的文件系统 .....	43
3.5.1 磁盘结构 .....	44
3.5.2 Inode 结构 .....	45
3.5.3 目录与普通文件 .....	48
3.5.4 块缓存 .....	48
3.5.5 磁盘 ide 驱动文件 .....	51
3.5.6 文件操作 .....	52
3.5.7 文件系统服务器 .....	52
3.5.8 用户使用的 fd 文件描述符 .....	54
3.5.9 日志系统 .....	54
3.6 xv6 的 I/O .....	55
3.6.1 使用系统调用的文件描述符 .....	55
3.6.2 命名服务 .....	56
3.6.3 I/O 子系统 .....	56

3.6.4 文件层的系统调用 .....	57
3.6.5 无文件开启时 .....	60
3.6.6 执行 open(" /carmi/f0" ) .....	61
3.6.7 执行 dup(0) .....	62
3.6.8 执行 close(0) .....	63
3.6.9 执行 dup(1) .....	64
3.6.10 执行 open(" /console" ) .....	65
3.6.11 执行 fork() .....	66
3.6.12 执行 P0 close(1) .....	67
3.6.13 执行 P1 close(2) .....	68
3.6.14 执行 P0 pipe() .....	69
3.6.15 文件层调度 .....	69
3.6.16 I/O 缓存 Buf 结构 .....	72
3.6.17 APIC .....	72
4 实验感想 .....	74
A linux 学习资料 .....	75
B 用户态和内核态是什么, 有什么区别 .....	76
C 为什么需要打开 A20 地址线 .....	77
D 操作系统相关的基本概念 .....	78
D.1 操作系统内核 (Operating System Kernel) .....	78
D.2 不可抢占型内核 (Non-Preemptive Kernel) .....	78
D.3 可抢占型内核 (Preemptive Kernel) .....	79
D.4 进程 (process) .....	80
D.5 进程状态 .....	80
D.6 多进程 .....	81
D.7 调度 (Scheduling) .....	81
D.8 进程调度算法 .....	81
D.9 可重入性 (Reentrancy) .....	82
D.10 上下文切换 (Context Switch or Task Switch) .....	83
D.11 进程优先级 .....	84
D.11.1 静态进程优先级 .....	84
D.11.2 动态进程优先级 .....	84

D.12 资源 .....	84
D.13 共享资源 .....	84
D.14 竞争状态 (race condition) .....	84
D.15 程序临界区 .....	85
D.16 互斥条件 .....	85
D.16.1 屏蔽中断和使能中断 .....	86
D.16.2 测试并置位指令 .....	86
D.16.3 禁止进程切换, 然后允许进程切换 .....	86
D.16.4 信号量 (Semaphores) .....	87
D.17 死锁 (或抱死) (Deadlock (or Deadly Embrace)) .....	89
D.18 同步 .....	89
D.19 进程间的通信 (Intertask Communication) .....	90
D.20 中断 .....	90
D.21 时钟节拍 (Clock Tick) .....	91
D.22 对存储器的需求 .....	91

## xv6 环境配置

对于 linux 并不太熟悉的同学, 我建议你们可以参考一些资料来快速入手一款 linux 操作系统, Ubuntu 由于有很多用户基础, 所以如果是新手的话, Ubuntu 是非常不错的选择, 后面的附录部分, 给出了一些我觉得比较好的学习资料. 在安装完成乌班图虚拟机之后, 我可以进行下一步的实验环境配置. 准备安装

```
1 # 安装git    apt-get install -y git
2 # 安装g++apt-get install -y g++
3 # 安装gdbapt-get install -y gdb
4 # 安装编译工具链apt-get install -y build-essential
5 # 下载xv6的代码并且cd 到对应路径
6 git clone git://github.com/mit-pdos/xv6-public.git
```

```
lablinux [正在运行] - Oracle VM VirtualBox
管理 控制 视图 热键 设备 帮助
root@oslab-VirtualBox:~$ sudo -a
sudo: 选项需要一个参数 -- a
usage: sudo -h | -K | -k | -V
usage: sudo -v [-AknS] [-g group] [-h host] [-p prompt] [-u user]
usage: sudo -l [-AknS] [-g group] [-h host] [-p prompt] [-U user] [-u user]
[command]
usage: sudo [-AbEHknPS] [-r role] [-t type] [-C num] [-g group] [-h host] [-p
prompt] [-u user] [VAR=value] [-i|-s] [<command>]
usage: sudo -e [-AknS] [-r role] [-t type] [-C num] [-g group] [-h host] [-p
prompt] [-u user] file ...
oslab@oslab-VirtualBox:~$ clear
oslab@oslab-VirtualBox:~$ sudo -s
[sudo] oslab 的密码:
root@oslab-VirtualBox:~# apt-get install -y g++
正在读取软件包列表... 完成
正在分析软件包的依赖关系树
正在读取状态信息... 完成
g++ 已经是最新版 (4:5.3.1-1ubuntu1)。
升级了 0 个软件包，新安装了 0 个软件包，要卸载 0 个软件包，有 159 个软件包未被升级。
root@oslab-VirtualBox:~# apt-get install -y gdb
正在读取软件包列表... 完成
正在分析软件包的依赖关系树
正在读取状态信息... 完成
gdb 已经是最新版 (7.11.1-0ubuntu1~16.5)。
升级了 0 个软件包，新安装了 0 个软件包，要卸载 0 个软件包，有 159 个软件包未被升级。
root@oslab-VirtualBox:~# apt-get install -y git
正在读取软件包列表... 完成
正在分析软件包的依赖关系树
正在读取状态信息... 完成
git 已经是最新版 (1:2.7.4-0ubuntu1.3)。
升级了 0 个软件包，新安装了 0 个软件包，要卸载 0 个软件包，有 159 个软件包未被升级。
root@oslab-VirtualBox:~#
```

```
升级了 0 个软件包，新安装了 0 个软件包，要卸载 0 个软件包，有 159 个软件包未被升级。
root@oslab-VirtualBox:~# apt-get install -y git
正在读取软件包列表... 完成
正在分析软件包的依赖关系树
正在读取状态信息... 完成
git 已经是最新版 (1:2.7.4-0ubuntu1.3)。
升级了 0 个软件包，新安装了 0 个软件包，要卸载 0 个软件包，有 159 个软件包未被升级。
root@oslab-VirtualBox:~# apt-get install -y build-essential
正在读取软件包列表... 完成
正在分析软件包的依赖关系树
正在读取状态信息... 完成
build-essential 已经是最新版 (12.1ubuntu2)。
升级了 0 个软件包，新安装了 0 个软件包，要卸载 0 个软件包，有 159 个软件包未被升级。
root@oslab-VirtualBox:~#
```

安装 qemu # 安装 qemu apt-get install qemu

```

root@oslab-VirtualBox:~# apt-get install -y qemu
正在读取软件包列表... 完成
正在分析软件包的依赖关系树
正在读取状态信息... 完成
将会同时安装下列软件：
binfmt-support cpu-checker ipxe-qemu libaiol libboost-random1.58.0 libboost-thread1.58.0 libcaca0 libfdt1 libiscsi2 librados2
librbd1 libSDL1.2debian libspice-server1 libusbredirparser1 libxen-4.6 libxenstore3.0 msr-tools qemu-block-extra qemu-slof
qemu-system qemu-system-arm qemu-system-common qemu-system-mips qemu-system-misc qemu-system-ppc qemu-system-sparc
qemu-system-x86 qemu-user qemu-user-binfmt qemu-utils seabios sharutils
建议安装：
qemu-user-static samba vde2 openbios-ppc openhwware sgabios ovmf debootstrap bsd-mailx | mailx
下列【新】软件包将被安装：
binfmt-support cpu-checker ipxe-qemu libaiol libboost-random1.58.0 libboost-thread1.58.0 libcaca0 libfdt1 libiscsi2 librados2
librbd1 libSDL1.2debian libspice-server1 libusbredirparser1 libxen-4.6 libxenstore3.0 msr-tools qemu qemu-block-extra qemu-slof
qemu-system qemu-system-arm qemu-system-common qemu-system-mips qemu-system-misc qemu-system-ppc qemu-system-sparc
qemu-system-x86 qemu-user qemu-user-binfmt qemu-utils seabios sharutils
升级了 0 个软件包，新安装了 33 个软件包，要卸载 0 个软件包，有 159 个软件包未被升级。
需要下载 39.9 MB 的归档。
解压缩后会消耗 249 MB 的额外空间。
获取:1 http://cn.archive.ubuntu.com/ubuntu xenial/main amd64 libiscsi2 amd64 1.12.0-2 [51.5 kB]

```

编译 xv6 cd /到对应位置/xv6-public make qemu 生成 qemu 环境下运行的 xv6 成功

```

root@oslab-VirtualBox:~/xv6-public
[sudo] oslab 的密码:
root@oslab-VirtualBox:~/xv6-public#
root@oslab-VirtualBox:~/xv6-public# make qemu
dd if=/dev/zero of=xv6.img count=10000
记录了 10000+0 的读入
记录了 10000+0 的写出
5120000 bytes (5.1 MB, 4.9 MiB) copied, 0.0293134 s, 175 MB/s
dd if=bootblock of=xv6.img conv=notrunc
记录了 1+0 的读入
记录了 1+0 的写出
512 bytes copied, 0.000196716 s, 2.6 MB/s
dd if=kernel of=xv6.img seek=1 conv=notrunc
记录了 333+1 的读入
记录了 333+1 的写出
170868 bytes (171 kB, 167 KiB) copied, 0.0013816 s, 124 MB/s
qemu-system-i386 -serial mon:stdio -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512
xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ 
```

## Lab 2: Memory Management

2.1 实验目的

2.2 实验要求

2.3 实验内容

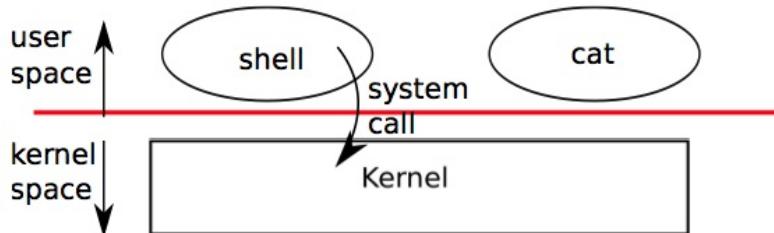
2.4 实验分析

## Lab 2: Memory Management

# xv6 实验系统分析

## 4.1 xv6 的系统结构（结合示意图）

SMP 是 Symmetric Multi Processing 的简称，意为对称多处理计算机系统，内有许多紧耦合多处理器，这种系统的最大特点就是共享所有资源。在 SMP 架构中，每个处理器在功能上都是对称、等价的。每个处理器有独自的中断处理控制器（硬件）。内存所有处理器之间共享。因此处理器可以通过共享的内存区域进行通讯。Xv6 就是一个 SMP 操作系统，可以有多个 CPU 共同工作。xv6 基于典型的 UNIX 操作系统设计思路。简单地说，xv6 是一种能区分内核态和用户态，基于扁平内存管理的层次型单体内核，应用程序和操作系统是处于不同的特权状态和地址空间。代表应用程序的用户态进程运行在 CPU 的用户态（又称非特权模式，用户模式），无法直接访问系统硬件和操作系统中的系统数据，而操作系统运行在 CPU 的核心态（又称特权模式，内核模式），可以访问系统硬件和核心数据。下面给出一个 xv6 系统结构示意图



**Figure 0-1. A kernel and two user processes.**

下面分别从系统调用接口、进程/线程管理、内存管理、文件系统、I/O 管理等几个方面进行总体分析。进程通过系统调用使用内核服务。系统调用是应用程序访问操作系统的接口。在系统调用接口上，通用操作系统与基于此操作系统的应用程序处于两个不同的 CPU 特权态，操作系统处于核心态，而应用程序处于用户态。在核心态可以执行 CPU 特权指令，而用户态无法执行特权指令，且只能通过特定的指令或中断来访问操作系统提供的各种功能。这在一定程度上保证了系统整体的安全，避免应用程序对操作系统可能的破坏。内核使用了 CPU 的硬件保护机制来保证用户进程只能访问自己的内存空间。内核拥有实现保护机制所需的硬件权限 (hardware privileges)，而用户程序没有这些权限。当一个用户程序进行一次系统调用时，硬件会提升特权级并且开始执行一些内核中预定义的功能。内核提供的一系列系统调用就是用户程序可见的操作系统接口，xv6 内核提供了 Unix 传统系统调用的一部分，它们是：

## xv6 system calls

System call	Description
fork()	Create process
exit()	Terminate current process
wait()	Wait for a child process to exit
kill(pid)	Terminate process pid
getpid()	Return current process's id
sleep(n)	Sleep for n seconds
exec(filename, *argv)	Load a file and execute it
sbrk(n)	Grow process's memory by n bytes
open(filename, flags)	Open a file; flags indicate read-/write

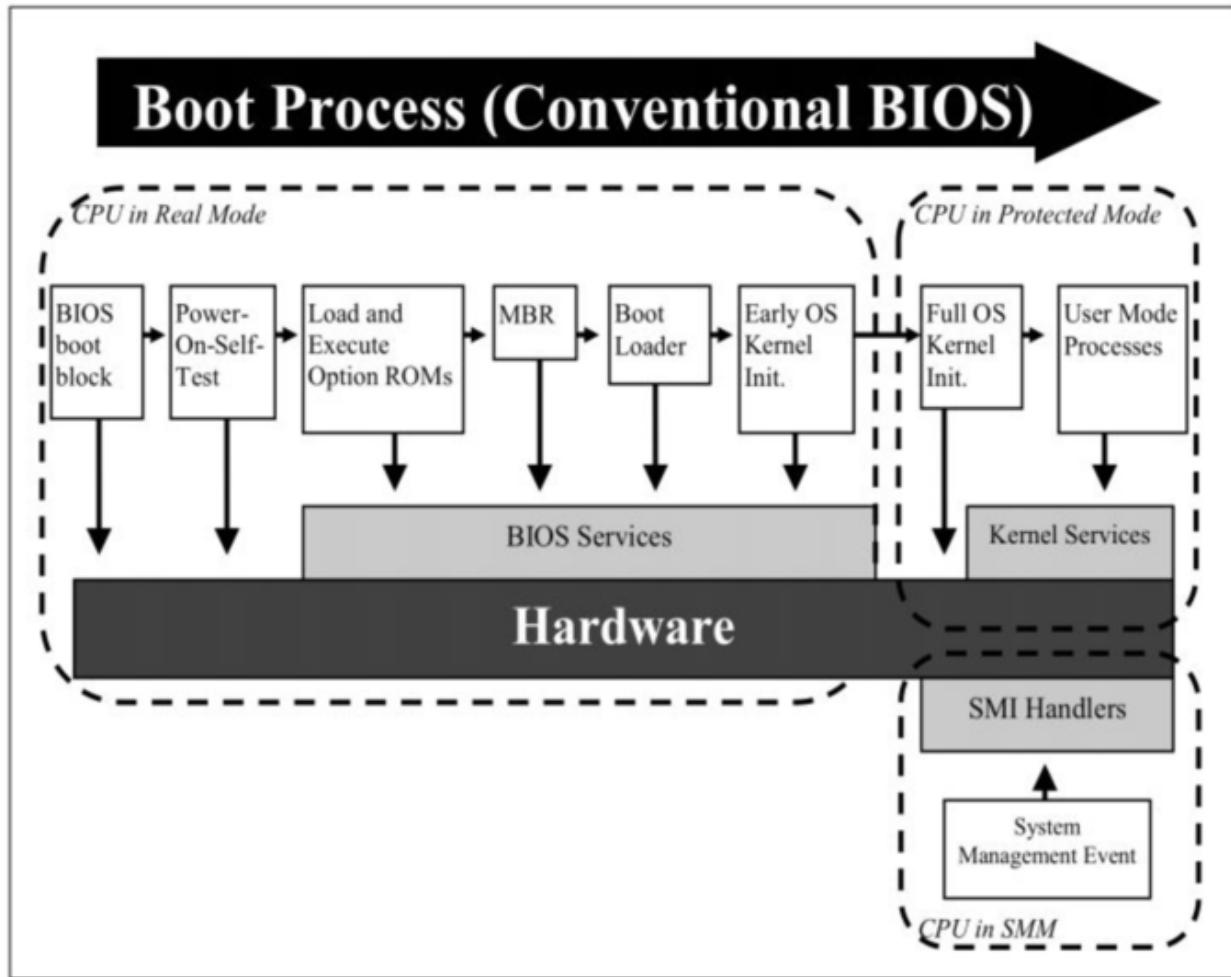
## xv6 system calls (2)

System call	Description
read(fd, buf, n)	Read n bytes from an open file into buf
write(fd, buf, n)	Write n bytes to an open file
close(fd)	Release open file fd
dup(fd)	Duplicate fd
pipe(p)	Create a pipe and return fd's in p
chdir(dirname)	Change the current directory
mkdir(dirname)	Create a new directory
mknod(name, major, minor)	Create a device file
fstat(fd)	Return info about an open file
link(f1, f2)	Create another name (f2) for the file f1
unlink(filename)	Remove a file

在内存管理方面，通用操作系统采用了虚拟内存管理方式，这样可以让内存需求超过实际物理内存的进程/线程能够执行，其主要思想是把重要和常用的数据和执行代码放在物理内存中，把不常用的数据和执行代码放到二级存储（这里主要指的是硬盘等可在掉电后保存数据的存储介质），随时根据系统执行情况替换放在内存中的数据和代码。而且通过虚存管理可以实现对不同内存区域的保护，不同进程之间，或者应用程序和操作系统之间的地址空间相对隔离。这样一般情况下不同进程的地址空间不能直接访问，且应用程序不能直接访问内核地址空间。所以一个与错误的应用程序不会导致系统的崩溃，从而增加了系统的可靠性。xv6 操作系统没有采用虚拟内存管理，而是采用了简单的基于 X86 段模式的单一地址空间管理方式。在内存分配和释放的管理上，xv6 相对实现得比较简单，采用基于可变分区分配的首次适配算法，容易产生内存碎片。在进程/线程管理方面，当前通用操作系统结合虚存管理，采用进程和线程结合的管理方式。进程代表了一个程序执行的过程以及其所占用的计算机资源（包括 CPU、内存、文件等），进程的执行流可用线程来表示。操作系统的调度单位可以是进程或线程。一个进程可以包含多个线程，属于同一进程的多个线程共享进程管理的资源，比如属于同一进程的多个线程共享进程所管理的内存，这样这些线程可以直接访问属于进程的全局地址空间。xv6 操作系统实现了一个基于进程（没有实现线程）的简单进程管理机制。在文件系统管理方面，当前通用操作系统结合虚存管理，实现了多种复杂、高效且可靠的文件系统，且建立了一个统一的虚拟文件系统层，屏蔽不同文件系统的差异，对上层提供统一的接口。且与用户管理和进程管理结合，可实现安全管理，保证对文件的安全访问。xv6 操作系统实现了一个相对简单的基于 inode 索引方式的文件系统。在 I/O 管理方面，xv6 操作系统与通用操作系统（特别是类 UNIX 操作系统）差别不是特别大，都把设备“看成”是一种特殊的设备文件，有设备号，用文件的访问接口来进行打开、关闭、读、写和控制等操作。在灵活性方面，xv6 驱动程序不能象通用操作系统那样根据硬件情况动态加载，而是在编译时候就静态确定的。

## 4.2 xv6 的引导

当计算机加电后，一般不直接执行操作系统，而是执行引导加载程序。简单地说，引导加载程序就是在操作系统内核运行之前运行的一段小程序。通过这段小程序，我可以初始化硬件设备、建立系统的内存空间映射图，从而将系统的软硬件环境带到一个合适的状态，以便为最终调用操作系统内核准备好正确的环境。最终引导加载程序把操作系统内核映像加载到 RAM 中，并将系统控制权传递给它。下面是一个通用操作系统的启动过程



对于绝大多数计算机系统而言，操作系统和应用软件是存放在磁盘（硬盘/软盘）、光盘、EPROM、ROM、Flash 等可在掉电后继续保存数据的存储介质上。计算机启动后，CPU 一开始会到一个特定的地址开始执行指令，这个特定的地址存放了系统软件（不仅是操作系统，还可能是引导加载程序等）。引导加载程序（bootloader）是系统加电后运行的第一段软件代码。对于 PC386 的体系结构而言，PC 机中的引导加载程序由 BIOS（Basic Input Output System，即基本输入/输出系统，其本质是一个固化在主板 Flash/CMOS 上的软件）和位于软盘/硬盘引导扇区中的 OS Boot Loader 一起组成。BIOS 实际上是被固化在计算机 ROM（只读存储器）芯片上的一个特殊的软件，为上层软件提供最底层的、最直接的硬件控制与支持。更形象地说，BIOS 就是 PC 计算机硬件与上层软件程序之间的一个“桥梁”，负责访问和控制硬件。以 PC386 为例，计算机加电后，CPU 从物理地址 0x000FFFFF0（由初始化的 CS: EIP 确定，此时 CS 和 IP 的值分别是 0xF000 和 0xFFFF0）开始执行。在 0x000FFFFF0 这里只是存放了一条跳转指令，通过跳转指令跳到 BIOS 例程起始点。BIOS 做完计算机硬件自检和初始化后，会选择一个启动设备（例如软盘、硬盘、光盘等），并且读取该设备的第一扇区（即启动扇区）到内存一个特定的地址 0x7c00 处，然后 CPU 控制权会转移到那个地址继续执行。至此 BIOS 的初始化工作做完了，进一步的工作交给了 xv6。整个 xv6 系统的启动流程

大致是这样的：首先 BIOS 将把 OS 的 Boot Loader 从磁盘上（一般是位于第一个扇区）拷贝到内存当中。当 BIOS 将基本的初始化程序完成后，将跳转到 Boot Loader 所在内存的位置继续执行。Boot Loader 将把 OS 的内核从磁盘上拷贝到然后运行。这样就完成了启动。在 xv6 的源码中，整个启动过程主要牵涉到 bootasm.S 和 bootmain.c 两个文件

#### 4.2.1 代码分析

bootloader 代码分析 bootloader 的组成在 makefile 中 96 行 102 行有如下语句

```
96 |bootblock: bootasm.S bootmain.c
97 | $(CC) $(CFLAGS) -fno-pic -O -nostdinc -I. -c bootmain.c
98 | $(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c bootasm.S
99 | $(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o bootblock.o bootasm.o bootmain.o
100 | $(OBJDUMP) -S bootblock.o > bootblock.asm
101 | $(OBJCOPY) -S -O binary -j .text bootblock.o bootblock
102 | ./sign.pl bootblock
```

从中可以看出 bootloader 包含两个文件，bootasm.S 和 bootmain.c。生成的 bootloader 会写到一个主引导扇区上面。作为主引导扇区，其位置在软盘或硬盘的第一个扇区，其大小为 512 个字节，在此扇区的最后两个字节是一个主引导扇区特征码为”55AA”。Makefile 的 97 行和 98 行是通过 gcc 把 bootmain.c 和 bootasm.S 编译成目标文件 bootmain.o 和 bootasm.o。Makefile 的 99 行是通过 ld 程序把目标文件 bootmain.o 和 bootasm.o 链接成目标文件 bootblock.o，且定义了起始执行的点（也称入口点）为 start 函数，具体的代码段起始地址为 0x7C00。Makefile 的 100 行是通过 objdump 程序把 bootblock.o 反汇编成 bootlock.asm。Makefile 的 101 行是通过 objcopy 程序把 bootblock.o 变成二进制码 bootlock。Makefile 的 102 行是通过 sign.pl 程序把 bootlock 扩展到 512 个字节，并把最后两个字节写成”55AA”。

bootloader 的启动主要涉及到 bootasm.S、bootmain.c。其中 bootasm.S 的主要作用是从实模式转化到保护模式。bootmain 的作用是把内核从磁盘拷贝到内存中。bootasm.S 在进入实模式向保护模式切换之前，首先需要把中断关闭（”cli” at line 13），保证转换过程不被硬件中断打断。

```

1 #include "asm.h"
2 #include "memlayout.h"
3 #include "mmu.h"
4
5 # Start the first CPU: switch to 32-bit protected mode, jump into C.
6 # The BIOS loads this code from the first sector of the hard disk into
7 # memory at physical address 0x7c00 and starts executing in real mode
8 # with %cs=0 %ip=7c00.
9
10 .code16                      # Assemble for 16-bit mode
11 .globl start
12 start:
13     cli                      # BIOS enabled interrupts; disable
14
15     # Zero data segment registers DS, ES, and SS.
16     xorw    %ax,%ax          # Set %ax to zero
17     movw    %ax,%ds          # -> Data Segment
18     movw    %ax,%es          # -> Extra Segment
19     movw    %ax,%ss          # -> Stack Segment

```

但是实模式刚运行完，很多遗留下的一些寄存器的值需要我整理下，所以我需要“在 16 ~ 19 行中将实模式的东西处理完，我的重点是要转换到保护模式中去开始真正的工作。为此我需要打开 A20 开关，将 32 位地址总线打开，使得地址位从 20 位切换到 32 位。关于为什么需要打开 A20 开关，可以参考文档最后的附录部分。下面的代码实现了打开 A20 开关。

```

23 seta20.1:
24     inb    $0x64,%al          # Wait for not busy
25     testb $0x2,%al
26     jnz   seta20.1
27
28     movb  $0xd1,%al          # 0xd1 -> port 0x64
29     outb  %al,$0x64
30
31 seta20.2:
32     inb    $0x64,%al          # Wait for not busy
33     testb $0x2,%al
34     jnz   seta20.2
35
36     movb  $0xdf,%al          # 0xdf -> port 0x60
37     outb  %al,$0x60
38
39     # Switch from real to protected mode. Use a bootstrap GDT that makes
40     # virtual addresses map directly to physical addresses so that the
41     # effective memory map doesn't change during the transition.
42     lgdt   gdtdesc
43     movl   %cr0,%eax
44     orl   $CR0_PE,%eax
45     movl   %eax,%cr0

```

接下来加载全局符号表 GDT 第 42 行代码通过引导加载器执行 lgdt 指令来把指向 gdt 的指针 gdtdesc 加载到全局描述符表(GDT)寄存器中。第 44 行正式开启保护模式(CR0\_PE=1)接下来第 51 行处理器还需要将 16 位模式(因为之前是实模式,所以目前还是处于 16 位的模式)切换到 32 位。由于我没法直接修改%cs,所以使用了一个 ljmp 指令 ljmp \$(SEG\_KCODE«3), \$start32。跳转指令会接着在下一行执行,但这样做实际上将%cs 指向了 gdt 中的一个代码描述符表项。该描述符描述了一个 32 位代码段,这样处理器就切换到了 32 位模式下。

```

39 # Switch from real to protected mode. Use a bootstrap GDT that makes
40 # virtual addresses map directly to physical addresses so that the
41 # effective memory map doesn't change during the transition.
42 lgdt gdtdesc
43 movl %cr0, %eax
44 orl $CR0_PE, %eax #我们通过这一条语句实现了把CR0_PE赋值成1,正式开启保护模式
45 movl %eax, %cr0
46
47 //PAGEBREAK!
48 # Complete the transition to 32-bit protected mode by using a long jmp
49 # to reload %cs and %ip. The segment descriptors are set up with no
50 # translation, so that the mapping is still the identity mapping.
51 ljmp $(SEG_KCODE«3), $start32
52
53 .code32 # Tell assembler to generate 32-bit code now.

```

接下来完成保护模式之下的初始化数据,然后开始执行 bootmain.c 代码

```

54 start32:
55     # Set up the protected-mode data segment registers
56     movw $(SEG_KDATA«3), %ax      # Our data segment selector
57     movw %ax, %ds                # -> DS: Data Segment
58     movw %ax, %es                # -> ES: Extra Segment
59     movw %ax, %ss                # -> SS: Stack Segment
60     movw $0, %ax                # Zero segments not ready for use
61     movw %ax, %fs                # -> FS
62     movw %ax, %gs                # -> GS
63
64     # Set up the stack pointer and call into C.
65     movl $start, %esp
66     call bootmain

```

bootmain.c 在这个文件中主要有四个函数:bootmain、waitdisk、readsect 和 readseg。其中 bootmain 是加载内核,其余三个都是对磁盘进行访问的程序。首先来看一下 waitdisk、readsect 和 readseg。readseg 函数的作用是从磁盘的 offset 处开始读取 count 个字符到 pa 处。在读取数据时是通过调用 readsect 以扇区为单位进行的。因此在 86 行保证 pa 是从一个扇区起始位置开始,因此要对 pa 进行对齐。readsect 是对磁盘进行读取,在读取之前每次调用 waitdisk 等待磁盘的准备过程,一旦磁盘准备好后就可以进行读取了。

```
76 // Read 'count' bytes at 'offset' from kernel into physical address 'pa'.
77 // Might copy more than asked.
78 void
79 readseg(uchar* pa, uint count, uint offset)
80 {
81     uchar* epa;
82
83     epa = pa + count;
84
85     // Round down to sector boundary.
86     pa -= offset % SECTSIZE;
87
88     // Translate from bytes to sectors; kernel starts at sector 1.
89     offset = (offset / SECTSIZE) + 1;
90
91     // If this is too slow, we could read lots of sectors at a time.
92     // We'd write more to memory than asked, but it doesn't matter --
93     // we load in increasing order.
94     for(; pa < epa; pa += SECTSIZE, offset++)
95         readsect(pa, offset);
96 }
97
```

```
58 // Read a single sector at offset into dst.
59 void
60 readsect(void *dst, uint offset)
61 {
62     // Issue command.
63     waitdisk();
64     outb(0x1F2, 1);    // count = 1
65     outb(0x1F3, offset);
66     outb(0x1F4, offset >> 8);
67     outb(0x1F5, offset >> 16);
68     outb(0x1F6, (offset >> 24) | 0xE0);
69     outb(0x1F7, 0x20); // cmd 0x20 - read sectors
70
71     // Read data.
72     waitdisk();
73     insl(0x1F0, dst, SECTSIZE/4);
74 }
```

```

15 void readseg(uchar*, uint, uint);
16
17 void
18 bootmain(void)
19 {
20     struct elfhdr *elf;
21     struct proghdr *ph, *eph;
22     void (*entry)(void);
23     uchar* pa;
24
25     elf = (struct elfhdr*)0x10000; // scratch space
26
27     // Read 1st page off disk
28     readseg((uchar*)elf, 4096, 0);
29
30     // Is this an ELF executable?
31     if(elf->magic != ELF_MAGIC)
32         return; // let bootasm.S handle error
33
34     // Load each program segment (ignores ph flags).
35     ph = (struct proghdr*)((uchar*)elf + elf->phoff);
36     eph = ph + elf->phnum;
37     for(; ph < eph; ph++){
38         pa = (uchar*)ph->paddr;
39         readseg(pa, ph->filesz, ph->off);
40         if(ph->memsz > ph->filesz)
41             stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
42     }
43

```

然后看一下 bootmain 过程。bootmain 的目的是从磁盘中加载内核到内存中，其中内核是 ELF 格式的二进制文件。为了读取 ELF 头，在第 28 行 bootmain 载入 ELF 文件的前 4096 字节，并将其拷贝到内存中 0x10000 处。其中包含了 ELF 执行文件格式的头。从中可以知道读取镜像的大小以及存放的位置第 31 行通过 ELF 头检查这是否的确是一个 ELF 文件。在第 39 行 bootmain 调用 readseg 将数据从磁盘中载入，第 41 行并调用 stosb 将段的剩余部分置零。stosb 使用 x86 指令 rep stosb 来初始化内存块中的每个字节。当完成拷贝后，在第 46 行 bootmain 获取内核入口程序的地址，然后进入该入口 bootloader 紧接着就是 entry 开始运行，这个时候 x86 的分页硬件在此时还没有开始工作；所以这时的虚拟地址是直接映射到物理地址上的。boot loader 把 xv6 内核装载到物理地址 0x100000 处。之所以没有装载

到内核指令和内核数据应该出现的 0x80100000，是因为小型机器上很可能没有这么大的物理内存。而之所以在 0x100000 而不是 0x0 则是因为地址 0xa0000 到 0x100000 是属于 I/O 设备的。为了让内核的剩余部分能够运行，entry 的代码设置了页表，将 0x80000000 开始的虚拟地址映射到物理地址 0x0 处。它将 entrypgdir 的物理地址载入到控制寄存器%cr3 中。分页硬件必须知道 entrypgdir 的物理地址，因为此时它还不知道如何翻译虚拟地址；它也还没有页表。entrypgdir 这个符号指向内存的高地址处，但只要用宏 V2P\_WO 减去 KERNBASE 便可以找到其物理地址。为了让分页硬件运行起来，xv6 会设置控制寄存器%cr0 中的标志位 CR0\_PG。

```
37 # By convention, the _start symbol specifies the ELF entry point.
38 # Since we haven't set up virtual memory yet, our entry point is
39 # the physical address of 'entry'.
40 .globl _start
41 _start = V2P_WO(entry)
42
43 # Entering xv6 on boot processor, with paging off.
44 .globl entry
45 entry:
46     # Turn on page size extension for 4Mbyte pages
47     movl    %cr4, %eax
48     orl    $(CR4_PSE), %eax
49     movl    %eax, %cr4
50     # Set page directory
51     movl    $(V2P_WO(entrypgdir)), %eax
52     movl    %eax, %cr3
53     # Turn on paging.
54     movl    %cr0, %eax
55     orl    $(CR0_PG|CR0_WP), %eax
56     movl    %eax, %cr0
57
58     # Set up the stack pointer.
59     movl $(stack + KSTACKSIZE), %esp
60
61     # Jump to main(), and switch to executing at
62     # high addresses. The indirect call is needed because
63     # the assembler produces a PC-relative instruction
64     # for a direct jump.
65     mov $main, %eax
66     jmp *%eax
67
68 .comm stack, KSTACKSIZE
```

Entry 在完成设置页表和栈指针的操作之后，跳转到内核的 C 代码，并在内存的高地址中执行它了。首先它将栈指针%esp 指向被用作栈的一段内存。所有的符号包括 stack 都在高

地址，所以当低地址的映射被移除时，栈仍然是可用的。entry 跳转到高地址的 main 代码中。我必须使用间接跳转，否则汇编器会生成 PC 相关的直接跳转（PC-relative direct jump），而该跳转会运行在内存低地址处的 main。main 不会返回，因为栈上并没有返回 PC 值。好了，现在内核已经运行在高地址处的函数 main 中了。

到现在为止,xv6 的引导工作全部完成，接下来开始执行内核代码.

## 4.3 xv6 的进程与调度分析

### 4.3.1 进程的概念

程序与进程的概念是不可分的。当用户在计算机上运行一个程序时，此程序对应的进程就诞生了，并实际完成各种程序提供的功能，而用户关闭一个程序时，进程也随之终止。程序是为了完成某项任务编排的语句序列，它要告诉计算机如何执行，因此程序是需要通过 CPU 来运行的，且在程序的运行过程中需要占有计算机的各种资源（比如内存等）才能运行下去。如果计算机系统在任一时刻限制只有一个程序在运行，则程序在整个运行过程中独占计算机中的全部资源，这样整个程序运行和管理就简单了。就象在一个家中只住了一个 A，他想看书就到书房去看书，想睡觉就到睡房的床上去睡觉，想看电视就到电视厅看电视，想吃饭就去餐厅吃饭，没人和他抢占资源。但为了提高计算机系统的资源利用率，需要支持多个程序并发执行。这就会带来许多新的问题，如资源的共享与竞争，同步与互斥等。比如此人与 B 成家并有了小孩 C，那就是三口之家同时住一套房，当 A 想去看足球比赛直播电视节目的时候，如果发现电视厅已经有 B 在坐着看连续剧电视节目了，A 就得等待或干别的事情。除非 A 在买一个电视，并在另外一个房间看他的球比赛直播。由于程序是静态的，我看到的程序是存储在存储介质（如硬盘、光盘等）上的，它无法反映出程序执行过程中的动态特性，而且程序在执行过程中会不断申请资源或释放资源，这样让程序作为共享资源的基本单位是不合适的，所以需要引入一个概念，它能描述程序的执行过程而且可以作为共享资源的基本单位，这个概念就是进程。简单地说，一个进程是一个具有一定独立功能的程序在一个数据集合上的一次动态执行过程。每个进程都是整个应用的某一部分。操作系统在逻辑上维护了进程的运行状态信息，即与进程运行直接相关的 CPU 寄存器和栈空间（只有这样才能实现进程切换）。操作系统根据当前进程的情况设置进程的状态，并根据进程的状态（比如优先级）进行选择一个进程占用 CPU 并运行，这个过程成为调度。进程的状态从诞生到死亡要经历若干个阶段，也会有生老病死。简单地说，进程有三种状态：就绪、执行、等待。多种原因可以导致创建一个进程，比如，当操作系统把一个程序从硬盘调入内存后，在开始执行前，操作系统就要为此程序创建一个对应的进程。又比如，一个进程可以自己创建一个子进程，子进程被创建后就是在内存中，处于就绪态，所谓就绪态就是万事具备，只欠 CPU 这个东风了；一旦进程占有了 CPU，就可以执行实际的工作了，其状态就变成了执行态；进程在执行中如果需要等待某个资源（如等硬盘输入数据），则进程会放弃 CPU，且其状态就变为等待态，这时

操作系统又会从处于就绪态的另一个进程中挑选一个进程占有 CPU，则这另一个进程的状态就变成了执行态。当前一个进程所等到数据到来后，处于等待态的前一个进程又被唤醒，并把状态变成为就绪态。在实际的操作系统中，进程的状态往往多于三种，比如找 xv6 中，进程具有多种状态，包括：EMBRYO, SLEEPING, RUNNABLE, RUNNING 和 ZOMBIE。定义在 proc.h 文件中：

```
35 enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
```

```
36
```

状态的含义如下：

- UNUSED：进程未被创建（即进程控制块空闲）时的状态；
- 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；
- 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

### 4.3.2 进程控制块

程序的运行是通过进程体现的，操作系统对进程进行管理和控制，那么操作系统怎么了解到进程的状态并掌握进程占有的资源分配呢，而且进程做状态转换时 CPU 的现场保存在那呢？这实际是通过进程控制快（Process Control Block, 简称 PCB）。PCB 是进程的唯一标志，在其中记录了进程的全部信息，相当于进程的档案。操作系统通过 PCB 感知进程的存在，通过 PCB 了解进程和控制进程的运行。在 xv6 中，所有的 CPU 共享一个进程控制块池，即源代码中为 proc.c 中的 proc（即进程控制块）数组。在这个进程数组保存的进程控制块结构分成两类：一类是未使用的进程控制块结构，另一类是正在使用的进程控制块结构。每次要创建一个进程时，只需要从进程控制块数组中取得一个未使用进程控制块结构进行相应的处理即可。

xv6 中结构体 struct proc 包含进程大小、页表、内核栈、进程状态、进程 id、父进程、目录等等，代码如下：

```
37 // Per-process state
38 struct proc {
39     uint sz;           |          // Size of process memory (bytes)
40     pde_t* pgdir;    |          // Page table
41     char *kstack;    |          // Bottom of kernel stack for this process
42     enum procstate state; |      // Process state
43     int pid;          |          // Process ID
44     struct proc *parent; |       // Parent process
45     struct trapframe *tf;   |       // Trap frame for current syscall
46     struct context *context; |    // swtch() here to run process
47     void *chan;        |          // If non-zero, sleeping on chan
48     int killed;        |          // If non-zero, have been killed
49     struct file *ofile[NFILE]; | // Open files
50     struct inode *cwd;   |          // Current directory
51     char name[16];     |          // Process name (debugging)
52 };
53
54 // Process memory is laid out contiguously, low addresses first:
55 //   text
56 //   original data and bss
57 //   fixed-size stack
58 //   expandable heap
```

- sz 是记录进程所占有的内存空间大小;
- pgdir 记录了进程页表的线性地址
- kstack 是进程在内核态的栈;
- 枚举类型变量 state 记录了进程的状态;
- pid 是进程的 ID;
- parent 记录父进程;
- tf 是当前系统调用的中断帧;
- context 是切换进程需要维护的硬件寄存器内容, 是进程运行的入口;
- chan 不为 NULL 时, 是进程睡眠时所挂的睡眠队列;
- killed 不为 0 时, 表示进程被杀死了;
- ofile 数组是进程打开的文件数组;

- cwd 是进程运行时所处的当前目录；
- name 保存了进程的名字（用于调试）。

struct proc 在 proc.h 中，并且在该文件之中定义了几个关键的数据结构。其中 context 是在内核进行上下文切换需要保存的寄存器：edi,esi,ebx,ebp,eip.

```
27     struct context {  
28         uint edi;  
29         uint esi;  
30         uint ebx;  
31         uint ebp;  
32         uint eip;  
33     };
```

proc[NPROC] 数组（在 proc.c 文件的 12 行）定义了 xv6 所能够支持的进程所需的相关数据，NPROC 表示了 xv6 可支持进程个数。

```
9  
10  struct {  
11      struct spinlock lock;  
12      struct proc proc[NPROC];  
13  } ptable;  
14
```

cpu 结构是记录计算机中所有 CPU 的相关信息:

```
1 // Per-CPU state
2 struct cpu {
3     uchar apicid;          // Local APIC ID
4     struct context *scheduler; // swtch() here to enter scheduler
5     struct taskstate ts;    // Used by x86 to find stack for interrupt
6     struct segdesc gdt[NSEGDS]; // x86 global descriptor table
7     volatile uint started;   // Has the CPU started?
8     int ncli;               // Depth of pushcli nesting.
9     int intena;             // Were interrupts enabled before pushcli?
10    struct proc *proc;      // The process running on this cpu or null
11};
```

- apicid 代表此 CPU 的 id 编号;
- scheduler 表示 scheduler 的地址
- ts 是 Task state segment, 用于在中断时找到栈
- gdt 是此 CPU 的全局描述符表 (GDT);
- started 表示是否此 CPU 已经启动了;
- ncli 表示执行 pushcli 函数的次数;
- intena 表示是否能被打断;
- curproc 是当前正在此 CPU 上运行的进程控制块;

spinlock 的作用在于当进程请求得到一个正在被占用的锁时, 将进程处于循环检查, 等待锁被释放的状态。

```
1 // Mutual exclusion lock.
2 struct spinlock {
3     uint locked;           // 锁是否处于锁住状态
4
5     // For debugging:
6     char *name;            // 锁名称
7     struct cpu *cpu;       // 占有该锁的CPU信息
8     uint pcs[10];          // 占有该锁的指令栈
9};
```

进程在虚拟内存中的布局如下：

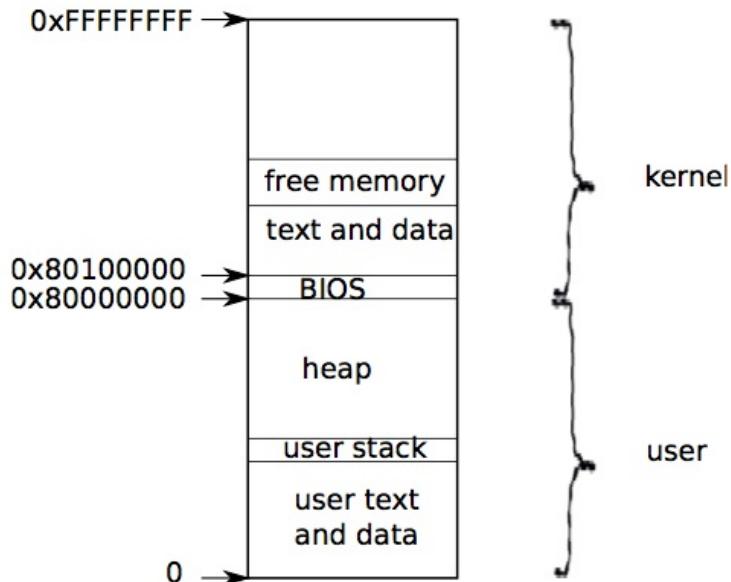


Figure 1-1. Layout of a virtual address space

Xv6 使用了一些全局变量来记录进程的状态。进程数组 proc[NPROC] 数组就是进程池。进程池访问锁 proc\_table\_lock 是一个 spinlock 是用来保护对进程池的临界区访问。当前运行进程 curproc 则是记录每个 cpu 当前运行的进程。第一个进程 initproc 是记录第一个创立的进程。这个进程十分特殊，它将托管所有没有父进程的进程（也就是父进程先于子进程结束）。下一进程号 nextid 是用来产生进程号的，在系统启动后会一直保持递增。Xv6 对于进程的处理过程与进程相关的处理函数集中在 proc.c 中，下面将依次介绍相关处理过程。进程管理初始化 pinit 过程仅仅是初始化 proc\_table\_lock 锁。

```

23 void
24 pinit(void)
25 {
26     initlock(&ptable.lock, "ptable");
27 }
```

初始化用户进程 init userinit 是初始化第一个用户进程 init。其处理过程如下所示：userinit 首先调用 allocproc。allocproc（第 126 行）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。allocproc 会在 proc 的表中找到一个标记为 UNUSED 的槽位。当它找到这样一个未被使用的槽位后，allocproc 将其状态设

置为 EMBRYO，使其被标记为被使用的并给这个进程一个独有的 pid。接下来，它尝试为进程的内核线程分配内核栈。如果分配失败了，allocproc 会把这个槽位的状态恢复为 UNUSED 并返回 0 以标记失败。

```

118 //PAGEBREAK: 32
119 // Set up first user process.
120 void
121 userinit(void)
122 {
123     struct proc *p; //第一个程序保存在这里，储存二进制代码的位置和大小。
124     extern char _binary_initcode_start[], _binary_initcode_size[];
125
126     p = allocproc();
127
128     initproc = p; //调用setupkvm()用来创建一个仅仅内核使用的页表。
129     if((p->pgdir = setupkvm()) == 0)
130         panic("userinit: out of memory?");
131     // 调用inituvvm()分配一页物理内存，将虚拟地址0映射到该内存，将二进制代码拷贝到该页中。
132     inituvvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
133     p->sz = PGSIZE;
134     memset(p->tf, 0, sizeof(*p->tf));
135     //将trap frame赋值为原始的用户模式状态%cs包含一个SEG_UCODE段选择器，优先级是DPL_USER，其他的类似。
136     p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
137     p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
138     p->tf->es = p->tf->ds;
139     p->tf->ss = p->tf->ds;
140     p->tf->eflags = FL_IF;
141     p->tf->esp = PGSIZE;
142     p->tf->eip = 0; // beginning of initcode.S
143
144     safestrcpy(p->name, "initcode", sizeof(p->name));
145     p->cwd = namei("/");
146
147     // this assignment to p->state lets other cores
148     // run this process. the acquire forces the above
149     // writes to be visible, and the lock is also needed
150     // because the assignment might not be atomic.
151     acquire(&ptable.lock);
152 //将进程的状态赋值为RUNNABLE。
153     p->state = RUNNABLE;
154
155     release(&ptable.lock);
156 }
```

创建子进程在 xv6 中，可以通过 fork 来复制父进程内容并创建一个新的子进程。其处理过程如下所示：

一个进程调用 fork() 函数后，系统先给新的进程分配资源，例如存储数据和代码的空间。然后把原来的进程的所有值都复制到新的进程中，只有少数值与原来的进程的值不同。相当于克隆了一个自己。fork() 函数通过系统调用创建一个与原来进程几乎完全相同的进程，也就是两个进程可以做完全相同的事（即复制了 fork() 函数后的代码），但如果初始参数或者传入的变量不同，两个进程也可以做不同的事。需要注意的是，两个进程拥有不同的内存空间

和寄存器 (fork() 函数之前的变量值是一样的, 但存储在不同地方), 改变一个进程中的变量不会影响另一个进程。fork() 调用的一个奇妙之处就是它仅仅被调用一次, 却能够返回两次, 它可能有三种不同的返回值:

- 在父进程中, fork() 返回新创建子进程的进程 pid;
- 在子进程中, fork() 返回 0;
- 如果出现错误, fork() 返回一个负值;

不同的进程有一个唯一的不同进程 pid, 通过 getpid() 函数可以知道当前进程 pid, 可以说我就是通过这个 pid 知道当前所在进程。创建新进程成功后, 系统中出现两个基本完全相同的进程, 这两个进程执行没有固定的先后顺序, 哪个进程先执行要看系统的进程调度策略。

调度进程任何操作系统都可能碰到进程数多于处理器数的情况, 这样就需要考虑如何分享处理器资源。理想的做法是让分享机制对进程透明。通常我对进程造成一个自己独占处理器的假象, 然后让操作系统的多路复用机制 (multiplex) 将单独的一个物理处理器模拟为多个虚拟处理器。

记下来介绍 xv6 是如何调度进程. Xv6 系统允许有多个 cpu, 并且每一个 cpu 获取进程调度的权利是相同的。CPU 在 scheduler 里面进行轮询操作, 每次从线程池中选择一个 RUNNABLE 的进程进行运行。直到运行完毕, 或一单位时间片结束, 或者进程主动 yield 或 sleep。进程调度的时机: 处于运行状态下的进程如果发生等待事件或者时间耗尽就会被 xv6 强制终止, 变成就绪状态. 这样的话,cpu 就可以执行其他的进程, 提高了 cpu 的使用率.

Xv6 要完成进程切换的功能需要实现从运行中的一个进程切换到另一个进程并且让上下文切换透明化, 还有就是 xv6 支持多个 cpu, 所以可能出现多个 CPU 同时切换进程的情况, 那么我必须使用一个带锁的方案来避免竞争, 最后还有进程退出时必须释放其占用内存与资源.

进程切换: 当 CPU 启动之后, 执行 scheduler 函数, 无限循环。在每个周期里, 从进程表中找到一个 RUNNABLE 的进程, 切换为进程的上下文, 此时开始执行函数。当函数运行结束时, 调用 return 函数, 此时切换为 CPU 的上下文, 开始下一循环。

进程唤醒与睡眠: 如果一个程序需要等待 IO, 则 CPU 会将其设置为睡眠状态, 此时不能被执行。当 IO 信号到达时, 执行的进程会将 IO 信号对应的进程设置为 RUNNABLE, 即唤醒。下一个 scheduler 周期的时候, 该进程就可能会被执行, 处理 IO 信号。

进程表锁: 对于多处理器架构而言, 需要用到进程表的时候都需要事先获得表的锁, 当结束之后再释放, 这样保证了对进程表操作的原子化, 可以避免多处理器的竞争问题。

下面通过代码来具体介绍 xv6 如何实现进程调度部分. scheduler 对每个 CPU 进行进程调度。CPU 到没有运行用户进程时, 就会进入这个过程中。这个过程不停的从进程中选出一个 RUNNABLE 的进程, 然后运行这个进程 (通过 swtch.S 中的 swtch 函数进行)。

```
322 void
323 scheduler(void)
324 {
325     struct proc *p;
326     struct cpu *c = mycpu();
327     c->proc = 0;
328
329     for(;;){
330         // 在每次执行一个进程之前，需要调用sti()函数开启CPU的中断
331         sti();
332
333         // 遍历进程表找到一个进程执行
334         acquire(&ptable.lock); // 获取进程表的锁，避免其他CPU更改进程表
335         for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
336             if(p->state != RUNNABLE)
337                 continue; // 如果进程的状态为不可运行，则略过
338
339             // 切换到选择的进程，释放进程表锁，当进程结束时，再重新获取
340             c->proc = p;
341             switchuvm(p);
342             p->state = RUNNING;
343
344             swtch(&(c->scheduler), p->context);
345             switchkvm();
346
347             // Process is done running for now.
348             // It should have changed its p->state before coming back.
349             c->proc = 0;
350         }
351         release(&ptable.lock);
352
353     }
354 }
```

sched 是放弃当前的用户进程，进入 scheduler()。只需用通过 swtch 进行上下文切换即可。

```
354 // Enter scheduler. Must hold only ptable.lock
355 // and have changed proc->state. Saves and restores
356 // intena because intena is a property of this
357 // kernel thread, not this CPU. It should
358 // be proc->intena and proc->ncli, but that would
359 // break in the few places where a lock is held but
360 // there's no process.
361 void
362 sched(void)
363 {
364     int intena;
365     struct proc *p = myproc();
366     // 是否获取到了进程表锁
367     if(!holding(&ptable.lock))
368         panic("sched ptable.lock");
369     // 是否执行过pushcli
370     if(mycpu()->ncli != 1)
371         panic("sched locks");
372     // 执行的程序应该处于结束或者睡眠状态
373     if(p->state == RUNNING)
374         panic("sched running");
375     // 判断中断是否可以关闭
376     if(readeflags()&FL_IF)
377         panic("sched interruptible");
378     intena = mycpu()->intena;
379     // 上下文切换至scheduler
380     swtch(&p->context, mycpu()->scheduler);
381     mycpu()->intena = intena;
382 }
```

放弃 CPU yield 是用户进程用来主动放弃当前 CPU, 进入 scheduler()。因为进入 scheduler 需要加锁所以把 proc\_table\_lock 加上。

```

385 //yield()函数将CPU主动让出一个调度周期(scheduling round), 实际应用在于当一个进程正在使用CPU, 同时中断处于打开状态, 需要查看nlock。
386 // Give up the CPU for one scheduling round.
387 void
388 yield(void)
389 {
390     // 获取进程表锁
391     acquire(&ptable.lock); //DOC: yieldlock
392     // 将进程状态设为可运行, 以便下次遍历时可以被唤醒
393     myproc()->state = RUNNABLE;
394     // 执行sched函数, 准备将CPU切换到scheduler context
395     sched();
396     // 释放进程表锁
397     release(&ptable.lock);
398 }

```

设置用户进程返回 forkret 是用于新 fork 的进程进行返回地址用的。由于调用 sys\_fork 创建的新进程是通过 shceduler() 调用的，因此运行之前需要把 proc\_table\_lock 锁释放。

```

400 // A fork child's very first scheduling by scheduler()
401 // will swtch here. "Return" to user space.
402 void
403 forkret(void)
404 {
405     static int first = 1;
406     // Still holding ptable.lock from scheduler.
407     release(&ptable.lock);
408
409     if (first) {
410         // Some initialization functions must be run in the context
411         // of a regular process (e.g., they call sleep), and thus cannot
412         // be run from main().
413         first = 0;
414         iinit(ROOTDEV);
415         initlog(ROOTDEV);
416     }
417
418     // Return to "caller", actually trapret (see allocproc).
419 }

```

睡眠进程 sleep 功能是让进程休眠。首先获得 proc\_table\_lock，然后将记录 chan，也就是说只有相关的 chan 才能把其唤起。然后调用 sched()。当期被唤醒时，将继续运行，此时将 chan 等记录清空，并且将 proc\_table\_lock 锁释放，而获得 lk 锁。

```
421 // Atomically release lock and sleep on chan.  
422 // Reacquires lock when awakened.  
423  
424 void  
425 sleep(void *chan, struct spinlock *lk)  
426 {  
427     struct proc *p = myproc();  
428  
429     if(p == 0)  
430         panic("sleep");  
431  
432     if(lk == 0)  
433         panic("sleep without lk");  
434     // 释放锁lk  
435     // Must acquire ptable.lock in order to  
436     // change p->state and then call sched.  
437     // Once we hold ptable.lock, we can be  
438     // guaranteed that we won't miss any wakeup  
439     // (wakeup runs with ptable.lock locked),  
440     // so it's okay to release lk.  
441     if(lk != &ptable.lock){ //DOC: sleeplock0  
442         acquire(&ptable.lock); //DOC: sleeplock1  
443         release(lk);  
444     }  
445     // 更改状态为SLEEPING，并切换至CPU context  
446     p->chan = chan;  
447     p->state = SLEEPING;  
448  
449     sched();  
450  
451     // Tidy up.  
452     p->chan = 0;  
453  
454     // 重新获得刚刚释放的lk锁  
455     if(lk != &ptable.lock){ //DOC: sleeplock2  
456         release(&ptable.lock);  
457         acquire(lk);  
458     }  
459 }
```

唤醒进程值得注意的是，使进程进入睡眠需要两个锁，lk 和 ptable.lock，由于之前已经得到了 ptable.lock，所以 wakeup 在此期间不会执行，直至进程完全进入睡眠状态，所以 lk 这个锁可以释放。wakeup 函数的主体部分位于 wakeup1 函数中。wakeup 功能是唤醒进程。先获取 proc\_table\_lock，然后调用函数 wakeup1，然后释放 proc\_table\_lock。wakeup1 是将由于 chan 休眠的进程都唤醒，即把进程的状态由 SLEEPING 改成 RUNNABLE。

```
461 //PAGEBREAK!
462 // Wake up all processes sleeping on chan.
463 // The ptable lock must be held.
464 //wakeup1之所以与wakeup作为两个独立的函数，是因为除了被wakeup调用之外，还在exit中调用，后面会详细讲到。
465 static void
466 wakeup1(void *chan)
467 {
468     struct proc *p;
469     // 遍历进程表，当发现有符合运行条件的程序时，将其标记为RUNNABLE|
470     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
471     if(p->state == SLEEPING && p->chan == chan)
472         p->state = RUNNABLE;
473 }
474
475 // Wake up all processes sleeping on chan.
476 void
477 wakeup(void *chan)
478 {
479     // 先获取ptable.lock，确保sleep不会执行，避免出现missed wakeup
480     acquire(&ptable.lock);
481     wakeup1(chan);
482     // 唤醒结束，释放ptable.lock
483     release(&ptable.lock);
484 }
```

杀死进程 kill 的作用是杀死某个进程。只需要从进程池中找到拥有相应进程 ID 的进程，将其 killed 状态置 1 即可。如果目标进程出于 sleeping 状态则将其设置成 RUNNABLE 使其尽早被杀掉。详细的代码如下所示。

```
490 // Kill the process with the given pid.  
491 // Process won't exit until it returns  
492 // to user space (see trap in trap.c).  
493 int  
494 kill(int pid)  
495 {  
496     struct proc *p;  
497  
498     acquire(&ptable.lock);  
499     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){  
500         if(p->pid == pid){  
501             p->killed = 1;  
502             // Wake process from sleep if necessary.  
503             if(p->state == SLEEPING)  
504                 p->state = RUNNABLE;  
505             release(&ptable.lock);  
506             return 0;  
507         }  
508     }  
509     release(&ptable.lock);  
510     return -1;  
511 }
```

退出进程 exit 完成了进程结束时的资源释放以及子进程处理等工作。在退出之前，用户进程需要关闭打开的文件 (238-243 行)，将打开的文件夹关闭 (358-359)。同时唤醒父进程，因为其可能出于 wait 状态。并将其子进程托管于 init。然后改变其状态为 ZOMBIE (等待父进程进行回收)。最后调用 sched 放弃 CPU。

```
224 // Exit the current process. Does not return.  
225 // An exited process remains in the zombie state  
226 // until its parent calls wait() to find out it exited.  
227 void  
228 exit(void)  
229 {  
230     struct proc *curproc = myproc();  
231     struct proc *p;  
232     int fd;  
233  
234     if(curproc == initproc)  
235         panic("init exiting");  
236  
237     // 关闭之前打开的文件  
238     for(fd = 0; fd < NOFILE; fd++){  
239         if(curproc->ofile[fd]){  
240             fileclose(curproc->ofile[fd]);  
241             curproc->ofile[fd] = 0;  
242         }  
243     }  
244     begin_op();  
245     iput(curproc->cwd);  
246     end_op();  
247     curproc->cwd = 0;  
248     acquire(&ptable.lock);  
249     // 唤醒父进程，因为父进程可能在队列之中处于休眠状态。  
250     wakeup1(curproc->parent);  
251     // 将子进程移交给initproc  
252     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){  
253         if(p->parent == curproc){  
254             p->parent = initproc;  
255             // 如果子进程处于zombie状态，则唤醒其新父亲initproc来料理后事  
256             if(p->state == ZOMBIE)  
257                 wakeup1(initproc);  
258         }  
259     }  
260     // 移交给scheduler，等待父进程处理  
261     curproc->state = ZOMBIE;  
262     sched();  
263     panic("zombie exit");  
264 }
```

等待进程结束 wait 是用于父进程等待子进程结束只用的。做法是不断的循环知道找到一个子进程变成 ZOMBIE 状态，则释放 ZOMBIE 子进程（回收其资源），然后结束循环返回其子进程的进程 ID。在循环的时候需要判断是否被外部进程 Kill，或有无活跃子进程，如果被

杀死或无活跃子进程，则推出循环。

```

266 //wait函数用于父进程等待子进程结束，如果没有子进程，则返回-1，否则返回已经结束的子进程的pid。
267 int
268 wait(void)
269 {
270     struct proc *p;
271     int havekids, pid;
272     // 获取进程表锁
273     struct proc *curproc = myproc();
274
275     acquire(&ptable.lock);
276     for(;;){
277         // 遍历查找是否有处于zombie状态的子进程
278         havekids = 0;
279         for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
280             if(p->parent != curproc)
281                 continue;
282             // 如果发现有子进程
283             havekids = 1;
284             // 如果进程状态为zombie，则将其释放并返回该子进程的pid
285             if(p->state == ZOMBIE){
286                 // Found one.
287                 pid = p->pid;
288                 kfree(p->kstack);
289                 p->kstack = 0;
290                 freevm(p->pgdir);
291                 p->pid = 0;
292                 p->parent = 0;
293                 p->name[0] = 0;
294                 p->killed = 0;
295                 p->state = UNUSED;
296                 release(&ptable.lock);
297                 return pid;
298             }
299         }
}

```

进程上下文切换在操作系统中通常也把上下文切换称为进程切换 (process switch)。当操作系统决定运行另外的进程时，它需要保存当前正在运行进程的当前上下文 (Context，也可称“运行状态信息”), 即 CPU 寄存器中的全部内容。这些内容保存在进程的堆栈空间或特定的上下文保存区。完成保存工作后，操作系统就可以把下一个将要运行进程的当前上下文从该进程的栈或上下文保存区中恢复到 CPU 的寄存器中，这样就可以继续下一个进程的运行。这个过程叫做进程切换。一般有两种情况的上下文切换：

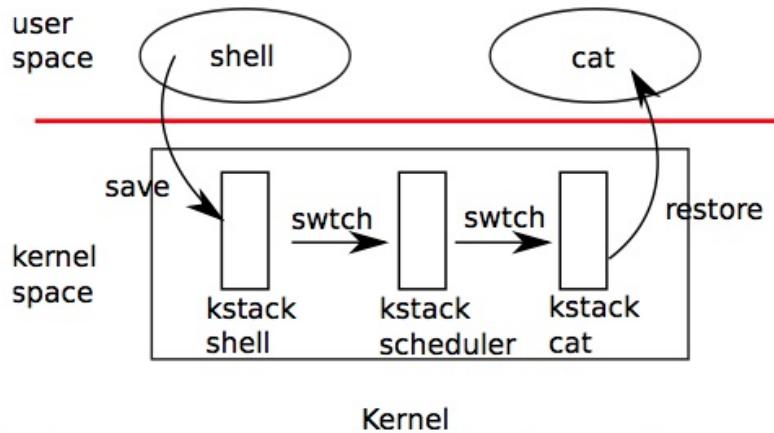
- 高优先级的进程因为需要某种临界资源，主动请求阻塞，让出处理器，此时将调度就绪状态的低优先级进程获得执行。这种切换称为进程级的上下文切换。
- 进程因为时钟中断或其它中断到来而被打断，在中断服务例程处理完毕后，内核发现有

更高优先级进程处于就绪态，则在中断处理结束后直接切换到高优先级进程执行。这种调度也称为中断级的上下文切换。

xv6 在低层次中实现了两种上下文切换：从进程的内核线程切换到当前 CPU 的调度器线程，从调度器线程到进程的内核线程。xv6 永远不会直接从用户态进程切换到另一个用户态进程；这种切换是通过用户态-内核态切换（系统调用或中断）、切换到调度器、切换到新进程的内核线程、最后这个陷入返回实现的。

```
300     // 如果没有子进程则直接返回
301     // No point waiting if we don't have any children.
302     if(!havekids || curproc->killed){
303         release(&ptable.lock);
304         return -1;
305     }
306     // 如果有子进程处于睡眠状态，则将父进程置于睡眠状态
307     // Wait for children to exit. (See wakeup1 call in proc_exit.)
308     sleep(curproc, &ptable.lock); //DOC: wait-sleep
309 }
310 }
```

swtch.S 该文件主要执行上下文切换的功能。它并不了解线程，它只是简单地保存和恢复寄存器集合，即上下文信息。当进程让出 CPU 时，进程的内核线程调用 swtch 来保存自己的上下文然后返回到调度器的上下文中。每个上下文都是以结构体 struct context\* 表示的，这实际上是一个保存在内核栈中的指针。swtch 有两个参数：struct context \*\*old、struct context \*new。它将当前 CPU 的寄存器压入栈中并将栈指针保存在 \*old 中。然后 swtch 将 new 拷贝到%esp 中，弹出之前保存的寄存器，然后返回。下面是 swtch.S 的代码解释。Swtch 在第 12 和第 13 行从栈中弹出参数，放入寄存器%eax 和%edx 中；swtch 必须在改变栈指针以及无法获得%esp 前完成这些事情。然后 swtch 在第 18-21 行压入寄存器，在当前栈上建立一个新的上下文结构。仅有被调用者保存的寄存器此时需要被保存；保存了旧寄存器后，swtch 就准备要恢复新的寄存器了。它将指向新上下文的指针放入栈指针中。新的栈结构和旧的栈相同，因为新的上下文其实是之前某次的切换中的旧上下文。所以 swtch 就能颠倒一下保存旧上下文的顺序来恢复新上下文。



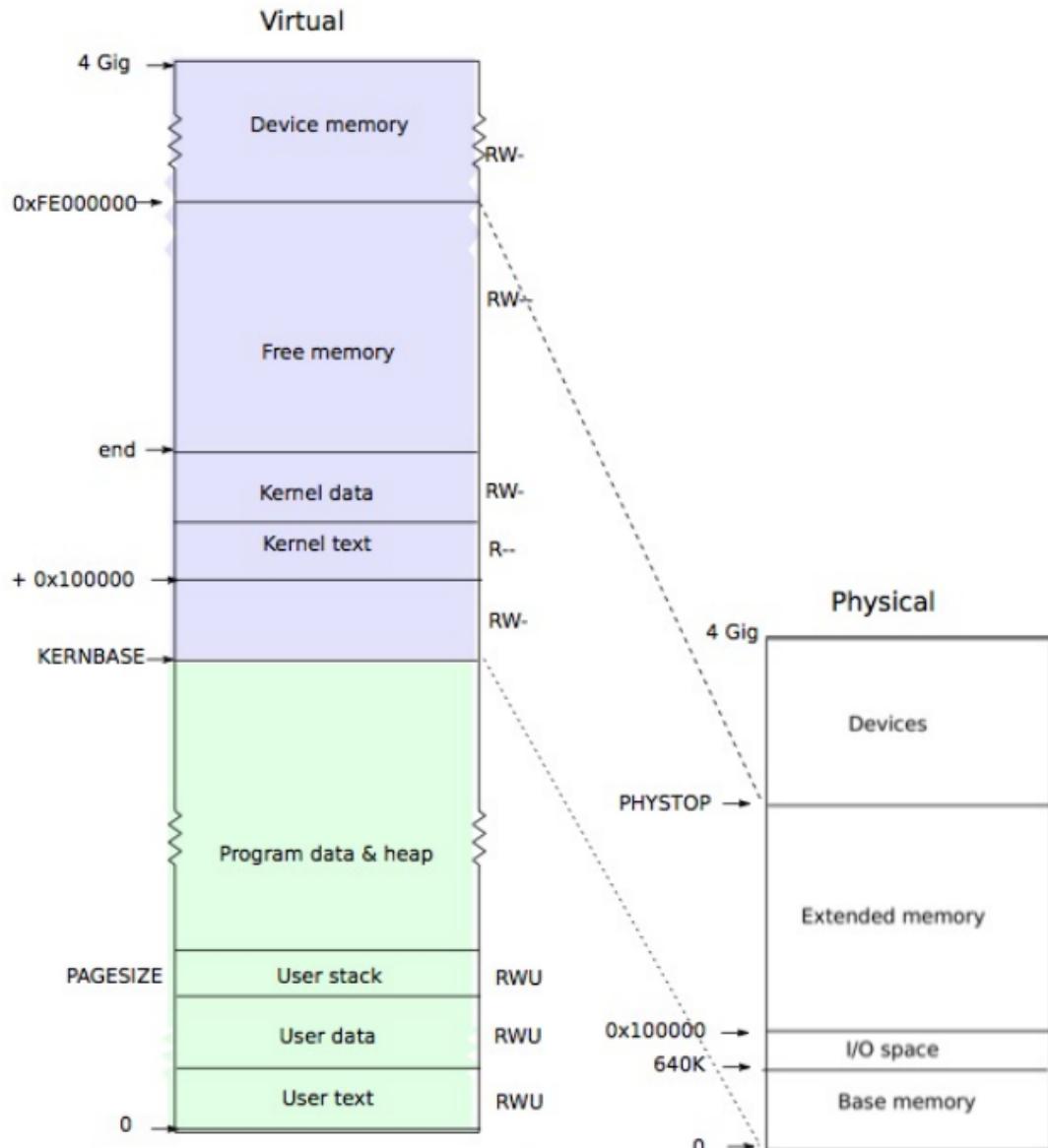
**Figure 5-1.** Switching from one user process to another. In this example, xv6 runs with one CPU (and thus one scheduler thread).

## 4.4 xv6 的内存管理

内存管理机制是实时操作系统的重要组成部分。操作系统通过页表机制实现了对内存空间的控制。页表使得 xv6 能够让不同进程各自的地址空间映射到相同的物理内存上，还能够为不同进程的内存提供保护。除此之外，我们还能够通过使用页表来间接地实现一些特殊功能。xv6 主要利用页表来区分多个地址空间，保护内存。另外，它也使用了一些简单的技巧，即把不同地址空间的多段内存映射到同一段物理内存（内核部分），在同一地址空间中多次映射同一段物理内存（用户部分的每一页都会映射到内核部分），以及通过一个没有映射的页保护用户栈。下面我们就来详细介绍一下 xv6 的内存管理。xv6 初始化之后的物理内存图



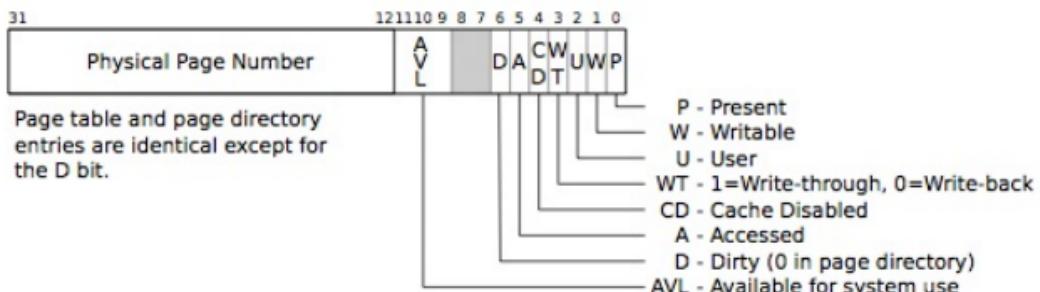
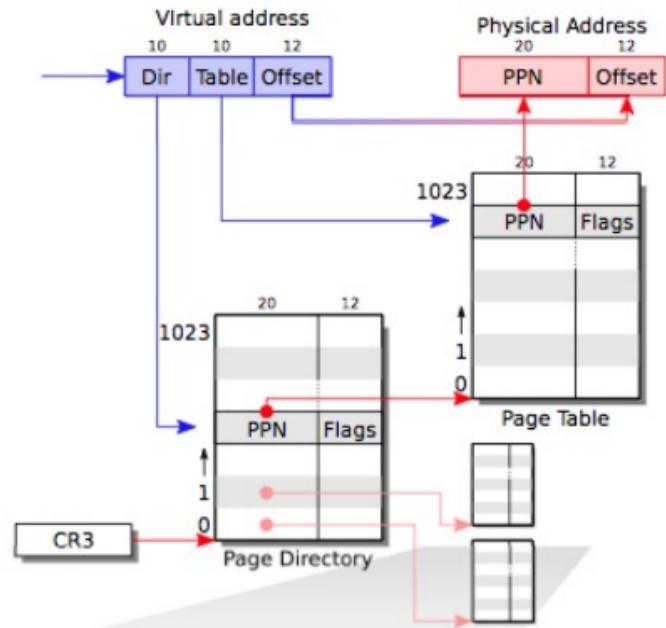
下面这个图展示的是 XV6 中虚拟地址到物理地址的映射关系.



Layout of a virtual address space and the physical address space.

xv6 使用页表（由硬件实现）来为每个进程提供其独有的地址空间。页表将虚拟地址（x86 指令所使用的地址）翻译（或者说“映射”）为物理地址（处理器芯片向主存发送的地址）。Xv6 实现了两级页表，第一级页表有 1024 项（占有 4096 字节），第二级页表也有 1024 项（占有 4096 字节），xv6 为每一个进程分配了这样的一个二级页表来实现将进程的虚拟地址转化为物理地址的过程。分页硬件使用虚拟地址的高 10 位来决定对应页目录条目。如果想要的条目已经放在了页目录中，分页硬件就会继续使用接下来的 10 位来从页表页中选择出对应的 PTE。否则，分页硬件就会抛出错误。通常情况下，大部分虚拟地址不会进行映射，而这样的二级结构就使得页目录可以忽略那些没有任何映射的页表页。在分页机制之中，虚拟地址的高 20 位来找到该虚拟地址在页表中的索引，然后把其高 20 位替换为对应 PTE 的 PPN。而低 12 位

是会被分页硬件原样复制的。因此在虚拟地址-物理地址的翻译机制下，页表可以为操作系统提供对一块块大小为 4096 ( $2^{12}$ ) 字节的内存片，这样的一个内存片就是一页。



### x86 page table hardware.

从图中可以看到，每个 PTE 都包含一些标志位，说明分页硬件对应的虚拟地址的使用权限。PTE\_P 表示 PTE 是否陈列在页表中：如果不是，那么一个对该页的引用会引发错误（也就是：不允许被使用）。PTE\_W 控制着能否对页执行写操作；如果不能，则只允许对其进行读操作和取指令。PTE\_U 控制着用户程序能否使用该页；如果不能，则只有内核能够使用该页。图 2-1 对此进行了说明。这些的标志位和页表硬件相关的结构体都在 mmu.h 定义。

```

134 // Page table/directory entry flags.
135 #define PTE_P          0x001    // Present
136 #define PTE_W          0x002    // Writeable
137 #define PTE_U          0x004    // User
138 #define PTE_PWT        0x008    // Write-Through
139 #define PTE_PCD        0x010    // Cache-Disable
140 #define PTE_A          0x020    // Accessed
141 #define PTE_D          0x040    // Dirty
142 #define PTE_PS         0x080    // Page Size
143 #define PTE_MBZ        0x180    // Bits must be zero
144

```

每个进程都有自己的页表，xv6 会在进程切换时通知分页硬件切换页表。如图所示，

```

17 int
18 main(void)
19 {
20     kinit1(end, P2V(4*1024*1024)); // phys page allocator
21     kvmalloc(); // kernel page table
22     mpinit(); // detect other processors
23     lapicinit(); // interrupt controller
24     seginit(); // segment descriptors
25     picinit(); // disable pic
26     ioapicinit(); // another interrupt controller
27     consoleinit(); // console hardware
28     uartinit(); // serial port
29     pinit(); // process table
30     tvinit(); // trap vectors
31     binit(); // buffer cache
32     fileinit(); // file table
33     ideinit(); // disk
34     startothers(); // start other processors
35     kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
36     userinit(); // first user process
37     mpmain(); // finish this processor's setup
38 }

```

进程的用户内存从 0 开始，最多能够增长到 KERNBASE，这使得一个进程最多只能使用 2GB 的内存。当进程向 xv6 要求更多的内存时，xv6 首先要找到空闲的物理页，然后把这些页对应的 PTE 加入该进程的页表中，并让 PTE 指向对应的物理页。xv6 设置了 PTE 中的 PTE\_U、PTE\_W、PTE\_P 标志位。大多数进程是用不完整个内存空间的；xv6 会把没有被使用的 PTE 的 PTE\_P 标志位设为 0。不同进程的页表将其用户内存映射到不同的物理内存中，因此每个进程就拥有了私有的用户内存。xv6 在每个进程的页表中都包含了内核运行所需要的所有映射，而这些映射都出现在 KERNBASE 之上。它将虚拟地址 KERNBASE:KERNBASE+PHYSTOP 映射到 0:PHYSTOP。这样映射的原因之一是内核可

以使用自己的指令和数据；原因之一是内核有时需要对物理页进行写操作，譬如在创建页表的时候，而使得每一个物理页都在对应的虚拟地址上被映射就让这些操作变得很方便。这样的安排有一个缺点，即 xv6 无法使用超过 2GB 的物理内存。有一些使用内存映射的 I/O 设备的物理内存 0xFE000000 之上，对于这些设备 xv6 页表采用了直接映射。KERNBASE 之上的页对应的 PTE 中，PTE\_U 位均被置 0，因而只有内核能够使用这些页。每个进程的页表同时包括用户内存和内核内存的映射，这样当用户通过中断或者系统调用转入内核时就不需要进行页表的转换了。大多数情况下，内核都没有自己的页表，所以内核几乎都是在借用用户进程的页表。Xv6 如何建立一个地址空间代码分析在第 21 行 main 调用 kvmalloc 来创建并切换到一个拥有内核运行所需的 KERNBASE 以上映射的页表。

```

138 // Allocate one page table for the machine for the kernel address
139 // space for scheduler processes.
140 void
141 |kvmalloc(void)
142 {
143     kpgdir = setupkvm();
144     switchkvm();
145 }
```

这里的大多数工作都是由 setupkvm 完成的。

```

117 // Set up kernel part of a page table.
118 pde_t*
119 |setupkvm(void)
120 {
121     pde_t *pgdir;
122     struct kmap *k;
123
124     if((pgdir = (pde_t*)kalloc()) == 0)
125         return 0;
126     memset(pgdir, 0, PGSIZE);
127     if (P2V(PHYSTOP) > (void*)DEVSPACE)
128         panic("PHYSTOP too high");
129     for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
130         if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
131                     (uint)k->phys_start, k->perm) < 0) {
132             freevm(pgdir);
133             return 0;
134         }
135     return pgdir;
136 }
```

首先，它会分配一页内存来放置页目录，然后在第 130 行调用 mappages 来建立内核需要

的映射，这些映射可以在 kmap 数组中找到。这里的映射包括内核的指令和数据，PHYSTOP 以下的物理内存，以及 I/O 设备所占的内存。setupkvm 不会建立任何用户内存的映射，这些映射稍后会建立。

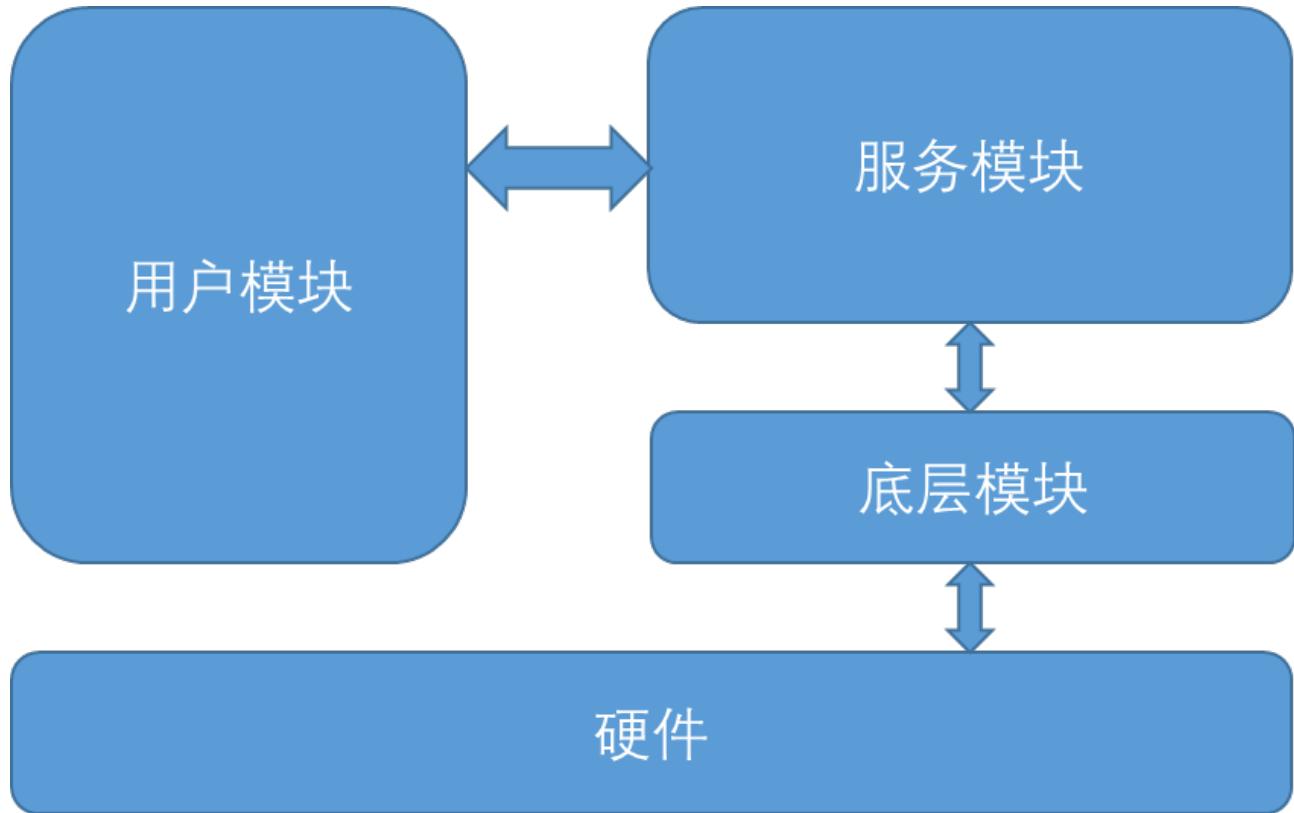
```
57 // Create PTEs for virtual addresses starting at va that refer to
58 // physical addresses starting at pa. va and size might not
59 // be page-aligned.
60 static int
61 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
62 {
63     char *a, *last;
64     pte_t *pte;
65
66     a = (char*)PGROUNDDOWN((uint)va);
67     last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
68     for(;;){
69         if((pte = walkpgdir(pgdir, a, 1)) == 0)
70             return -1;
71         if(*pte & PTE_P)
72             panic("remap");
73         *pte = pa | perm | PTE_P;
74         if(a == last)
75             break;
76         a += PGSIZE;
77         pa += PGSIZE;
78     }
79     return 0;
80 }
```

mappages 做的工作是在页表中建立一段虚拟内存到一段物理内存的映射。它是在页的级别，即一页一页地建立映射的。对于每一个待映射虚拟地址，mappages 调用 walkpgdir 来找到该地址对应的 PTE 地址。然后初始化该 PTE 以保存对应物理页号、许可级别（PTE\_W 和/或 PTE\_U）以及 PTE\_P 位来标记该 PTE 是否有效。

```
32 // Return the address of the PTE in page table pgdir
33 // that corresponds to virtual address va. If alloc!=0,
34 // create any required page table pages.
35 static pte_t *
36 walkpgdir(pde_t *pgdir, const void *va, int alloc)
37 {
38     pde_t *pde;
39     pte_t *pgtab;
40
41     pde = &pgdir[PDX(va)];
42     if(*pde & PTE_P){
43         pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
44     } else {
45         if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
46             return 0;
47         // Make sure all those PTE_P bits are zero.
48         memset(ptab, 0, PGSIZE);
49         // The permissions here are overly generous, but they can
50         // be further restricted by the permissions in the page table
51         // entries, if necessary.
52         *pde = V2P(ptab) | PTE_P | PTE_W | PTE_U;
53     }
54     return &ptab[PTX(va)];
55 }
```

walkpgdir 模仿 x86 的分页硬件为一个虚拟地址寻找 PTE 的过程。walkpgdir 通过虚拟地址的前 10 位来找到在页目录中的对应条目，如果该条目不存在，说明要找的页表页尚未分配；如果 alloc 参数被设置了，walkpgdir 会分配页表页并将其物理地址放到页目录中。最后用虚拟地址的接下来 10 位来找到其在页表中的 PTE 地址。物理内存的分配 Xv6 在运行时，内核需要为页表、进程的用户内存、内核栈及管道缓冲区分配空闲的物理内存。xv6 使用从内核结尾到 PHYSTOP 之间的物理内存为运行时分配提供内存资源。每次分配，它会将整块 4096 字节大小的一页分配出去。xv6 还会通过维护一个物理页组成的链表来寻找空闲页。所以，分配内存需要将页移出该链表，而释放内存需要将页加入该链表。

## 4.5 xv6 的文件系统

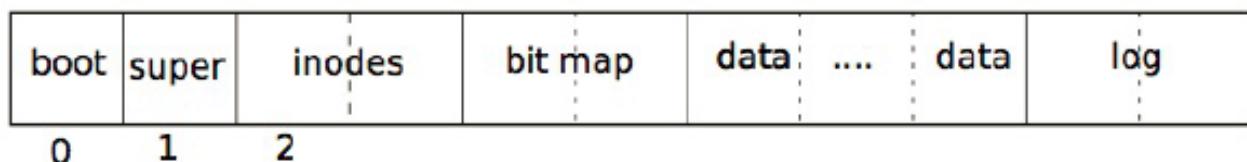


文件系统架构总体上分为 3 大模块，分别是用户模块，服务模块，底层模块。服务器模块负责打包底层模块文件读写函数，为外界提供统一的文件读写接口。底层模块负责详细的页面划分，加载，缓存以及控制磁盘驱动。用户模块与文件服务系统进程通信来进行文件读写活动。各层功能层次如下图

**System calls****File descriptors****Pathnames****Recursive lookup****Directories****Directory inodes****Files****Inodes and block allocator****Transactions****Logging****Blocks****Buffer cache**

xv6 的文件系统分 6 层实现, 如上图所示。最下面一层通过块缓冲读写 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。

#### 4.5.1 磁盘结构



磁盘扇区大小 512 字节, 而 xv6 在此之上组织的块大小为 4kB。Xv6 是类 UNIX 操作系统, 使用 inode 作为管理文件的基本单位, 类似于 jos 实验中的文件元数据。磁盘上文件结构是

- Block0 引导扇区，分区表
- Block1 超级块
- Block2-k inodes 块，存放磁盘文件的 inode 数据。
- Block k+1-t inodes 块，Bitmap 块，存储磁盘块占用信息
- Block t+1-l 数据块
- Block l+1-N-1 log 块，存放日志信息。

#### 4.5.2 Inode 结构

```
// 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];
};
```

## struct File (256 bytes)

Name: "foo"

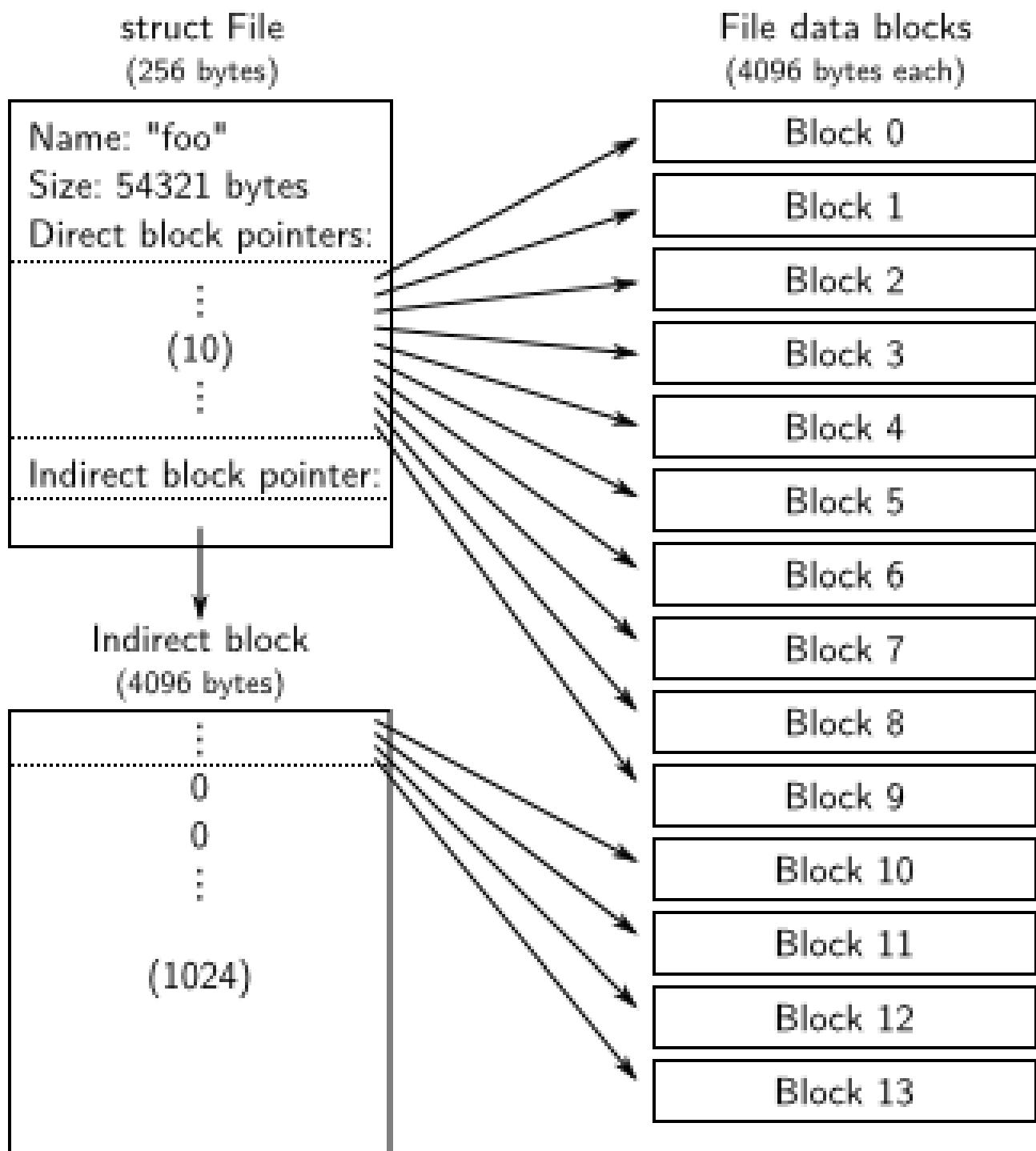
Size: 54321 bytes

Direct block pointers:

.....  
:  
(10)  
:  
.....

Indirect block pointer:

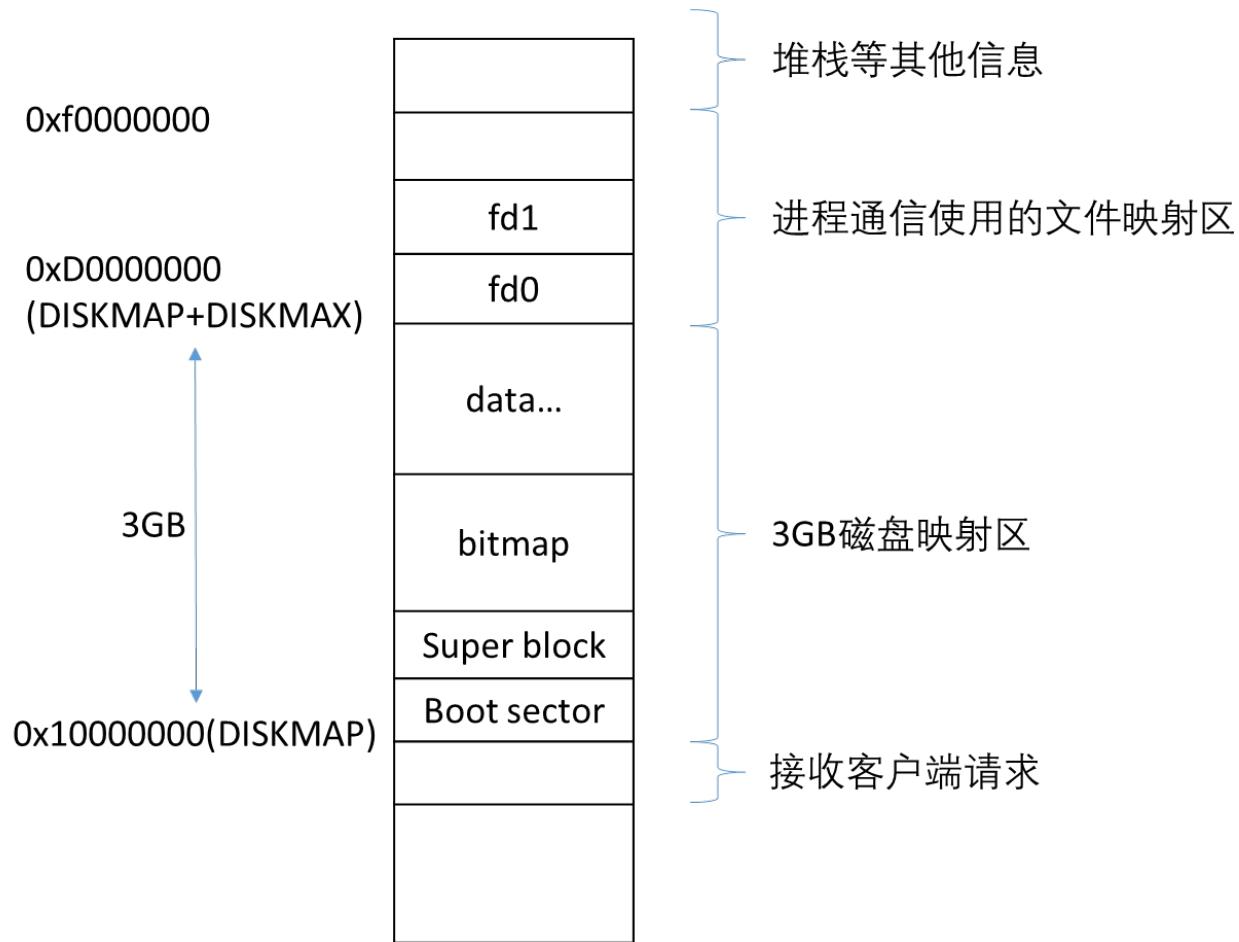
文件元数据使用 inode 结构体存储，包含文件名，文件大小，12 个直接块，一个间接块。文件的前 12 个 block 号会存在直接块里，其他的 block 会存在间接块指向的 block 里，可以额外存储  $4096/4=1024$  个文件块。此文件系统不支持双重间接块和三重间接块，所以文件不能大于 1036 个文件块。



### 4.5.3 目录与普通文件

文件系统以相同的方式管理目录文件与普通文件，以 `f_type` 区分，当文件块描述目录时，关联数据块内容为目录下的其他文件 `File` 数据块。

### 4.5.4 块缓存



Xv6 中我们通过缓存技术来实现访问磁盘块。该机制可以支持最大 3GB 的磁盘。我们将文件系统服务进程的虚拟地址空间 (`0x10000000 (DISKMAP)`) 到 `0xD0000000 (DISKMAP+DISKMAX)`) 对应到磁盘的地址空间 (3GB)。由于现代磁盘大于 3GB，在 32 位机器上的真正的文件系统实现会很麻烦。这种缓冲区高速缓存管理方法在 64 位地址空间的机器上仍然是合理的。映射方法：我们假装整个磁盘都已经被缓存到内存中，当我们想访问虚拟空间中的一个页时，由于虚拟空间还没有被映射，会发生页错误。在页错误处理程序中则会实际进行磁盘块到虚拟地址的映射并将该块磁盘内容缓存到内存中。此时就可以恢复文件系统进程进行正常的文件访问。`Bc_pgfault` 函数负责处理页面错误，处理的同时进行页面映射并从磁盘中缓存对应的块到内存中。处理的步骤：

- 根据地址计算出对应的 blockno (块编号)
- 检查地址范围是否超出 (0x10000000 (DISKMAP) 到 0xD0000000 (DISKMAP+DISKMAX)) 的映射边界
- 检查块编号是否超出块数
- 将地址对齐到页起始位置
- 页面映射
- 调用 ide\_read 函数读取缓存块对应的数个磁盘页到内存中
- 更新脏位

```
static void
bc_pgfault(struct UTrapframe *utf)
{
    void *addr = (void *) utf->utf_fault_va;
    uint32_t blockno = ((uint32_t)addr - DISKMAP) / BLKSIZE;
    int r;

    // Check that the fault was within the block cache region
    if (addr < (void*)DISKMAP || addr >= (void*)(DISKMAP + DISKSIZE))
        panic("page fault in FS: eip %08x, va %08x, err %04x",
              utf->utf_eip, addr, utf->utf_err);

    // Sanity check the block number.
    if (super && blockno >= super->s_nblocks)
        panic("reading non-existent block %08x\n", blockno);
```

```
// Sanity check the block number.  
if (super && blockno >= super->s_nblocks)  
    panic("reading non-existent block %08x\n", blockno);  
  
// Allocate a page in the disk map region, read the contents  
// of the block from the disk into that page.  
// Hint: first round addr to page boundary. fs/ide.c has code to read  
// the disk.  
//  
// LAB 5: you code here:  
addr = ROUNDDOWN(addr, PGSIZE);  
if (sys_page_alloc(0, addr, PTE_P | PTE_U | PTE_W) != 0) {  
    panic("bc_pgfault alloc page error\n");  
}  
if (ide_read(blockno * BLKSECTS, addr, BLKSECTS) != 0) {  
    panic("bc_pgfault ide_read failed\n");  
}  
  
// Clear the dirty bit for the disk block page since we just read the  
// block from disk  
if ((r = sys_page_map(0, addr, 0, addr, uvpt[PGNUM(addr)] & PTE_SYSCALL)) < 0)  
    panic("in bc_pgfault, sys_page_map: %e", r);  
  
// Check that the block we read was allocated. (exercise for  
// the reader: why do we do this *after* reading the block  
// in?)  
if (bitmap && block_is_free(blockno))  
    panic("reading free block %08x\n", blockno);
```

Flush\_block 函数负责将缓存中的块写回到磁盘中。处理步骤：

- 根据地址计算出对应的 blockno (块编号)
- 检查地址范围是否超出 (0x10000000 (DISKMAP) 到 0xD0000000 (DISKMAP+DISKMAX)) 的映射边界
- 检查如果对应块没有映射就不写回
- 检查如果对应块不是脏块 (没有发生改动) 就不写回
- 最后将缓存写回到磁盘，检查是否成功并更新脏位

```
.. void
flush_block(void *addr)
{
    uint32_t blockno = ((uint32_t)addr - DISKMAP) / BLKSIZE;
    if (addr < (void*)DISKMAP || addr >= (void*)(DISKMAP + DISKSIZE))
        panic("flush_block of bad va %08x", addr);
    // LAB 5: Your code here.
    addr = ROUNDDOWN(addr, PGSIZE);
    if (!va_is_mapped(addr)) {
        return;
    }
    if (!va_is_dirty(addr)) {
        return;
    }
    if (ide_write(blockno * BLKSECTS, addr, BLKSECTS) != 0) {
        panic("flush_block: ide_write failed");
    }
    if (sys_page_map(0, addr, 0, addr, uvpt[PGNUM(addr)] & PTE_SYSCALL) != 0) {
        panic("flush_block: sys_page_map failed");
    }
}
```

#### 4.5.5 磁盘 ide 驱动文件

Fs/ide.c 中实现了 ide\_read 和 ide\_write 方法

```

ide_read(uint32_t secno, void *dst, size_t nsecs)
{
    int r;

    assert(nsecs <= 256);

    ide_wait_ready(0);

    outb(0x1F2, nsecs);
    outb(0x1F3, secno & 0xFF);
    outb(0x1F4, (secno >> 8) & 0xFF);
    outb(0x1F5, (secno >> 16) & 0xFF);
    outb(0x1F6, 0xE0 | ((diskno&1)<<4) | ((secno>>24)&0xF));
    outb(0x1F7, 0x20);      // CMD 0x20 means read sector

    for (; nsecs > 0; nsecs--, dst += SECTSIZE) {
        if ((r = ide_wait_ready(1)) < 0)
            return r;
        insl(0x1F0, dst, SECTSIZE/4);
    }

    return 0;
}

```

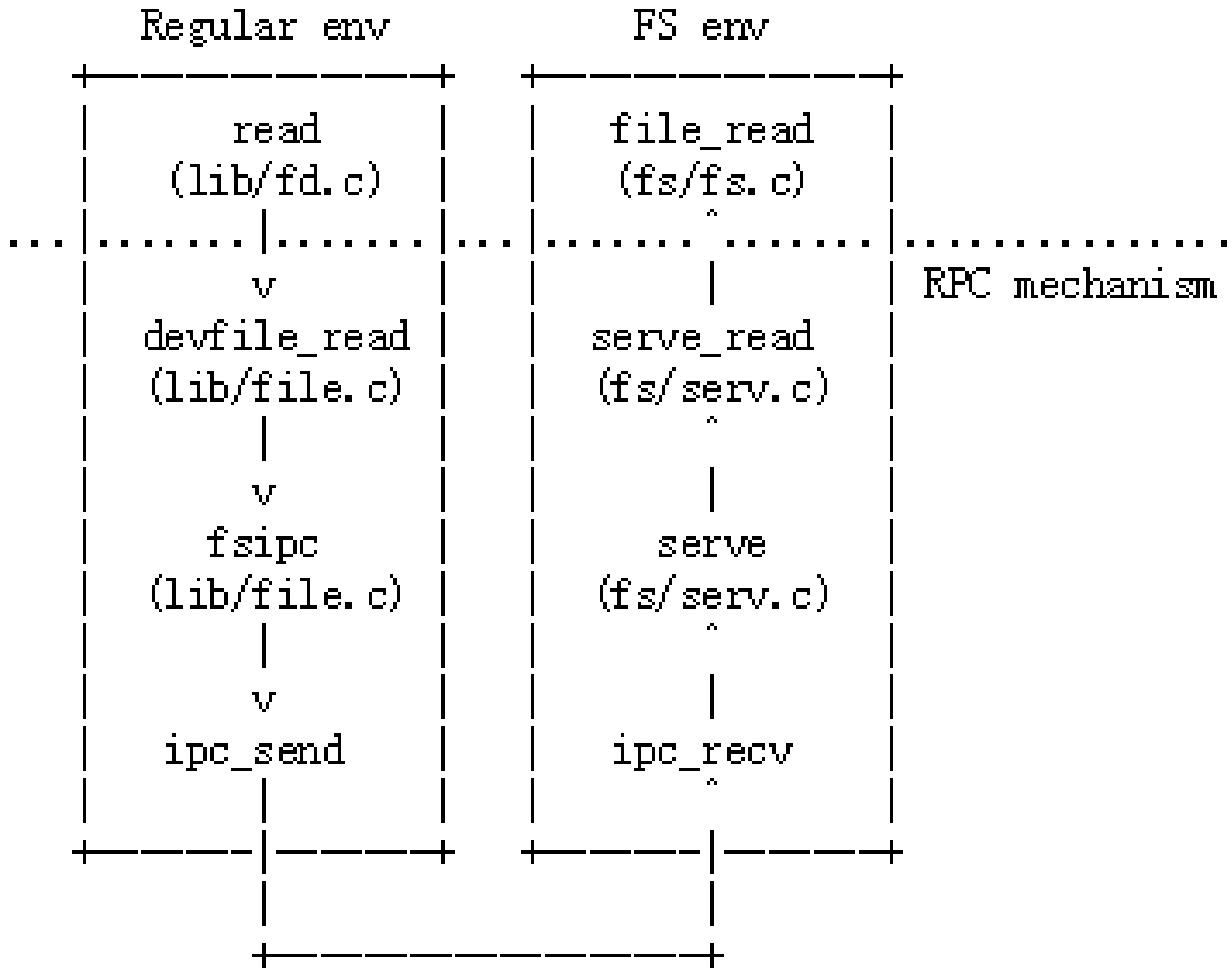
#### 4.5.6 文件操作

fs / fs.c 中提供了各种函数来实现您将需要解释和管理 File 结构，扫描和管理目录文件的条目，从根目录文件系统来解析绝对路径名。有以下几个主要函数：

- Walk\_path 完成目录解析
- File\_block\_walk 找到一个文件的某一块
- file\_get\_block 获取文件某一块
- file\_read 读取某一个文件
- file\_write 写入某一个文件

#### 4.5.7 文件系统服务器

与启动引导时把文件读写作为系统调用不同，而是作为一个用户进程于是我们在创建文件读写进程时需要把 I/O 读写权限打开文件服务器进程采用轮询的方式接收用户进程发送的文件读写请求，向用户进程返回信息



虚线下面的所有内容都是从常规环境到文件系统环境的读取请求的机制。从头开始, `read` (我们提供的) 在任何文件描述符上工作, 并简单地调度到适当的设备读取功能, 在这种情况下 `devfile_read` (我们可以有更多的设备类型, 如管道)。`devfile_read` 实现 `read` 专门针对磁盘上的文件。这个和 `lib / file.c` 中的其他 `devfile_*` 函数实现了 FS 操作的客户端, 并且都以大致相同的方式工作, 将请求结构中的参数捆绑在一起, 调用发送 IPC 请求, 解包并返回结果。该 `fsipc` 函数只是处理向服务器发送请求和接收答复的常见细节。文件系统服务器代码可以在 `fs / serv.c` 中找到。它在 `serve` 函数中循环, 无休止地通过 IPC 接收请求, 将该请求分派给相应的处理函数, 并通过 IPC 发回结果。在读取的例子中, `serve` 将调度到 `serve_read`, 将处理特定的 IPC 细节读取请求, 如解开请求结构, 最后调用 `file_read` 实际执行文件读取。

#### 4.5.8 用户使用的 fd 文件描述符

```
struct FdFile {
    int id;
};

struct Fd {
    int fd_dev_id;
    off_t fd_offset;
    int fd_onode;
    union {
        // File server files
        struct FdFile fd_file;
    };
};

struct Stat {
    char st_name[MAXNAMELEN];
    off_t st_size;
    int st_isdir;
    struct Dev *st_dev;
};
```

#### 4.5.9 日志系统

日志系统用来记录文件操作的记录，主要针对系统崩溃时可以使用日志来进行鉴别和恢复。日志存在于磁盘末端已知的固定区域。它包含了一个起始块，紧接着一连串的数据块。起始块包含了一个扇区号的数组，每一个对应于日志中的数据块，起始块还包含了日志数据块的计数。xv6 在提交后修改日志的起始块，而不是之前，并且在将日志中的数据块都拷贝到文件系统之后将数据块计数清 0。提交之后，清 0 之前的崩溃就会导致一个非 0 的计数值。每一个系统调用都可能包含一个必须从头到尾原子完成的写操作序列，我们称这样的一个序列为一个会话，虽然他比数据库中的会话要简单得多。任何时候只能有一个进程在一个会话之

中，其他进程必须等待当前会话中的进程结束。因此同一时刻日志最多只记录一次会话。日志的操作：

```

1 begin_trans();
2 bp = bread( ... );
3 bp->data[ ... ] = ... ;
4 log_write(bp);
5 commit_trans();
6 begin_trans 像锁一样，获取日志的控制权
7 bread 操作缓冲块
8 log_write 像是 bwrite
9 的一个代理，它把块中新的内容记录到日志中，并且把块的扇区号记录在内存。
10 commit_trans 将日志的起始块写到磁盘上，类似于 git 的 commit，
11 并产生一个检查点，可用于恢复。

```

## 4.6 xv6 的 I/O

处理器必须像和主存交互一样同设备交互。x86 处理提供了特殊的 in, out 指令来在设备地址（称为‘I/O 端口’）上读写。这两个指令的硬件实现本质上和读写内存是相同的。早期的 x86 处理器有一条附加的地址线：0 表示从 I/O 端口读写，1 则表示从主存读写。每个硬件设备会处理它所在 I/O 端口所接收到的读写操作。设备的端口使得软件可以配置设备，检查状态，使用设备；例如，软件可以通过对 I/O 端口的读写，使磁盘接口硬件对磁盘扇区进行读写。很多计算机体系结构都没有单独的设备访问指令，取而代之的是让设备拥有固定的内存地址，然后通过内存读写实现设备读写。实际上现代 x86 体系结构就在大多数高速设备上（如网络、磁盘、显卡控制器）使用了该技术，叫做内存映射 I/O。但由于向前兼容的原因，in, out 指令仍能使用，而比较老的设备如 xv6 中使用的 IDE 磁盘控制器仍使用这两个指令。Xv6 系统的 I/O 主要面向 3 个设备：磁盘，键盘，屏幕。磁盘 I/O 控制方式为程序直接控制（查询）方式。调用磁盘 ide 驱动来进行硬件读写，向上提供统一的 open, read, write 等接口。键盘输入方式为中断控制方式。将键盘中断捕获，交给控制台输入。屏幕显示方式为程序直接控制（查询）方式。调用显示驱动向 cga 接口输出字符信息。首先，我们从用户模式介绍 xv6 I/O 系统调用部分

### 4.6.1 使用系统调用的文件描述符

```

1 read (int fd, char* buf, int len);
2 write (int fd, char* buf, int len);

```

```

3 stat ( int fd , struct stat * );
4 dup ( int fd );
5 close ( int fd );

```

## 4.6.2 命名服务

```

1 link ( char * oldpath , char * newpath );
2 fd = open ( char * path , int flags );
3 unlink ( char * path );
4 mkdir ( char * path );
5 rmdir ( char * path );

```

还有管道函数用作文件缓冲 pipe (int pipefd[2]); 然后，我们开始进入 xv6 的核心 xv6 核心是一个分层系统，其中文件层由 pipe 子系统和 inode 子系统组成，inode 子系统包括 name 层、inode 层、buffer 层和 driver 层，系统调用调用文件层，name 层和 inode 层。简单的介绍后，我们来探讨一下 xv6 如何在不同的硬件，结构和函数方法下，向用户模式提供统一的界面。xv6 文件的抽象格式如下：

```

1 struct file{
2 enum {FD_NONE, FD_PIPE , FD_INODE} type;
3 int ref; // reference count
4 char readable;
5 char writeable;
6 struct pipe* pipe;
7 struct inode * ip;
8 uint off;
9 };

```

其中各个变量功能下文均有解释，并且均将细化到代码模块

## 4.6.3 I/O 子系统

由以下几个模块组成 FD\_PIPE. FD\_INODE: T\_FILE, T\_DIR, T\_DEV. 每个 I/O 子系统都是由一个结构体和一个操作集合来定义的，子系统中的各个模块将在下文予以解释 FD\_PIPE 模块

```

1 struct pipe{
2 struct spinlock lock;

```

```

3 char data[PIPE_SIZE];
4 uint nread;//读取的字符数目
5 uint nwrite;//写入的字符数目
6 int readopen;//读取的文件仍然开启
7 int writeopen;/写入的文件仍然开启
8 };

```

还有 pipe 的一些相关基本操作 pipalloc, pipeclose, piperead, pipewrite. FD\_INODE 模块

```

1 struct inode{
2     uint dev;//设备数字
3     uint inum;//Inode数字
4     int ref;//Reference count
5     int flags;//IBUSY,INVALID
6     short type;// inode的磁盘镜像
7     short major;
8     short minor;
9     short nlink;
10    uint size;
11    uint addrs[NDIRECT+1];
12 };

```

F\_INODE 相关方法 Listing1: 读入 T\_FILE or T\_DIR 调用如下方法 namei, create, ilock, readi, writei, stati, iunlock, iput Listing2: 读入 T\_DEV consoleread, consolewrite

#### 4.6.4 文件层的系统调用

sys dup, sys read, sys write, sys fstat, sys close.

在 fd 系统调用中使用的函数

```

1 struct file * filedup(struct file *f)
2 fileclose(struct file *f);
3 fileread(struct file *f, char *buf, int len);
4 filewrite(struct file *f, char* buf, int len);
5 filestat(struct file*f, struct stat*s);

```

fdalloc: 用于返回最低限度的被数字标记的空余文件槽.

```

1 static int fdalloc(struct file *f){
2     int fd;

```

```

1   for (fd = 0; fd < NOFILE; fd ++){
2       if (proc->ofile [ fd]==0){
3           proc->ofile [ fd ] = f ;
4           return fd ;
5       }
6   }
7   return -1;
8 }
```

sys\_dup 部分

```

1 sys_dup( void ){
2 struct file *f;
3 int fd;
4 if (argfd(0,0,&f)<0) return -1;
5 if ((fd = fdalloc(f))<0) return -1;
6 filedup(f);
7 return fd;
8 }
```

sys\_read 部分

```

1 int sys_read( void ){
2 struct file *f;
3 int n;
4 char *p;
5 if (argfd(0,0,&f)<0||argint(2,&n)<0||argptr(1,&p,n)<0) return -1;
6 return fileread(f,p,n);
7 }
```

sys\_write 部分

```

1 int sys_write( void ){
2 struct file *f;
3 int n;
4 char *p;
5 if (argfd(0,0,&f)<0||argint(2,&n)<0||argptr(1,&p,n)<0) return -1;
6 return filewrite(f,p,n);
7 }
```

sys\_fstat 部分

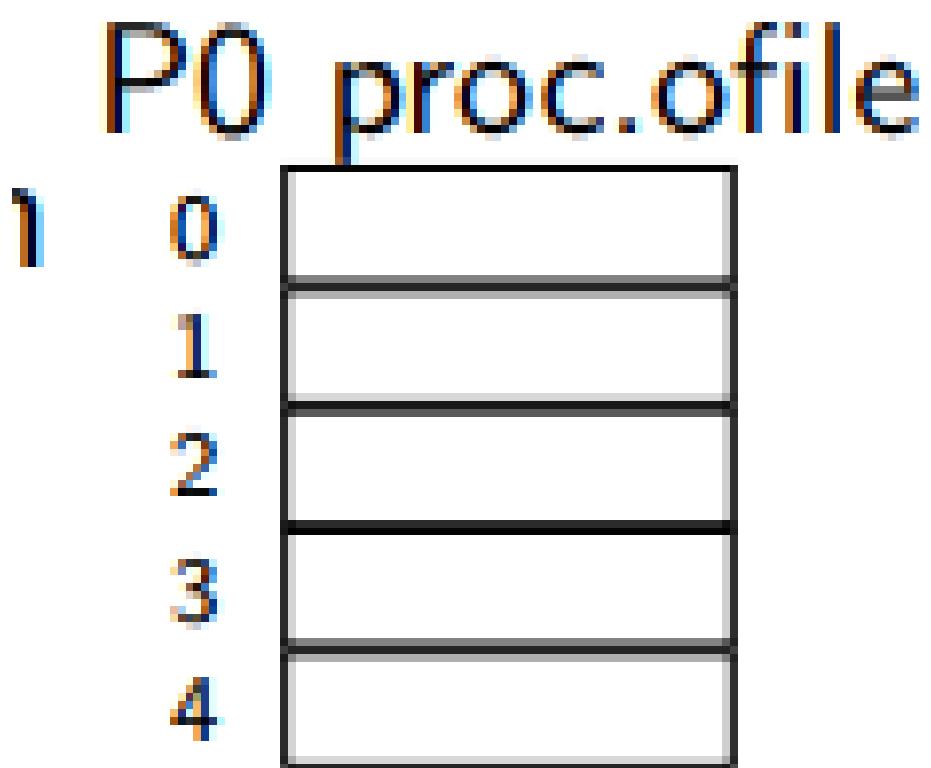
```
1 int sys_fstat(void){  
2     struct file *f;  
3     struct stat *st;  
4     if(argfd(0,0,&f)<0||argptr(1,(void*)&st,sizeof(*st))<0) return -1;  
5     return filestat(f,st);  
6 }
```

sys\_close 部分

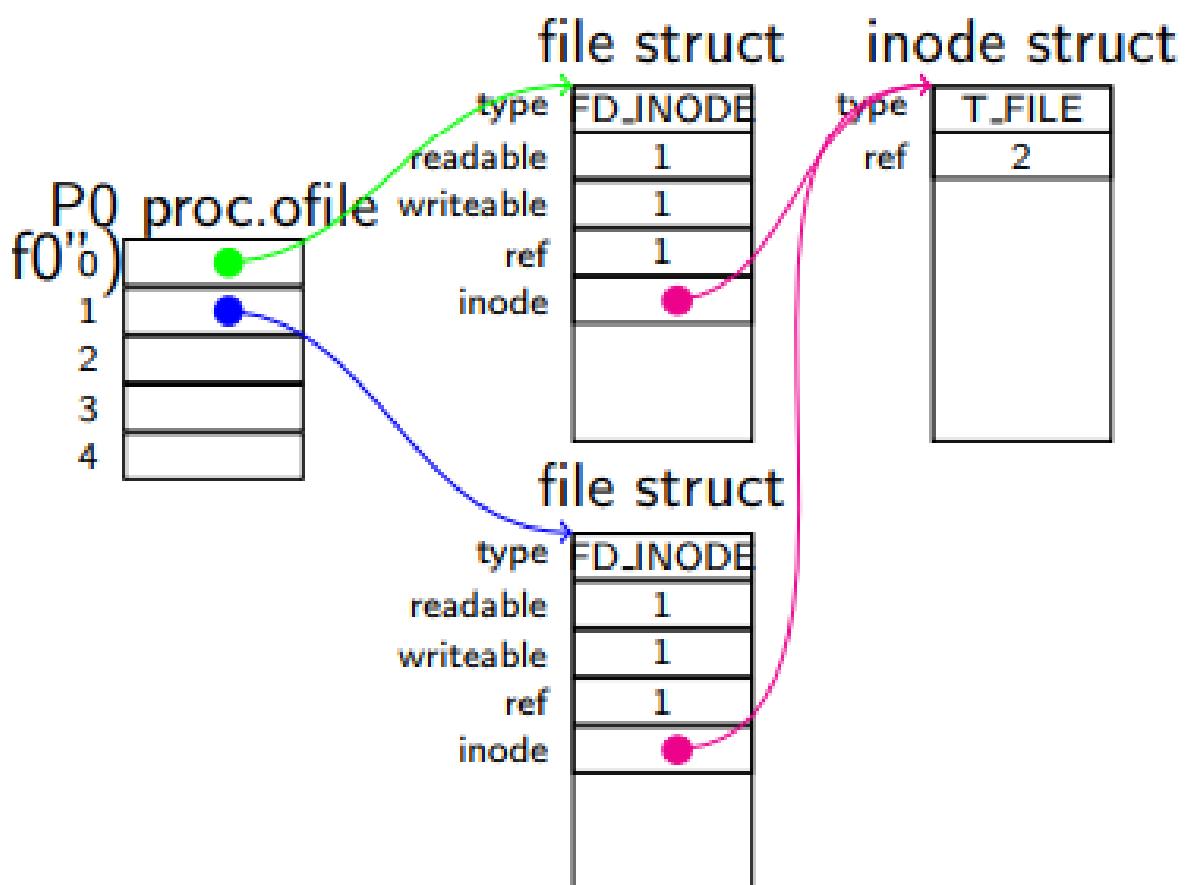
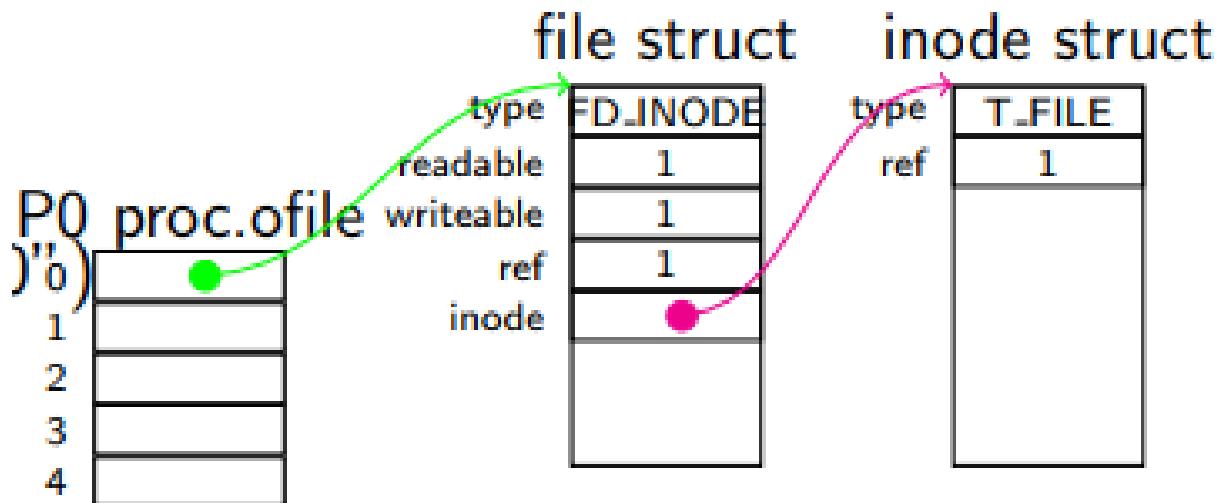
```
1 int sys_close(void){  
2     int fd;  
3     struct file*f;  
4     if(argfd(0,&fd,&f)<0) return -1;  
5     proc->ofile[fd]=0;  
6     fileclose(f);  
7     return 0;  
8 }
```

文件层的实现通过以下五个函数: filedup, fileread, filewrite, filestat, fileclose. 我们先来看一下 ofile/file/inode/pipe 的执行流程

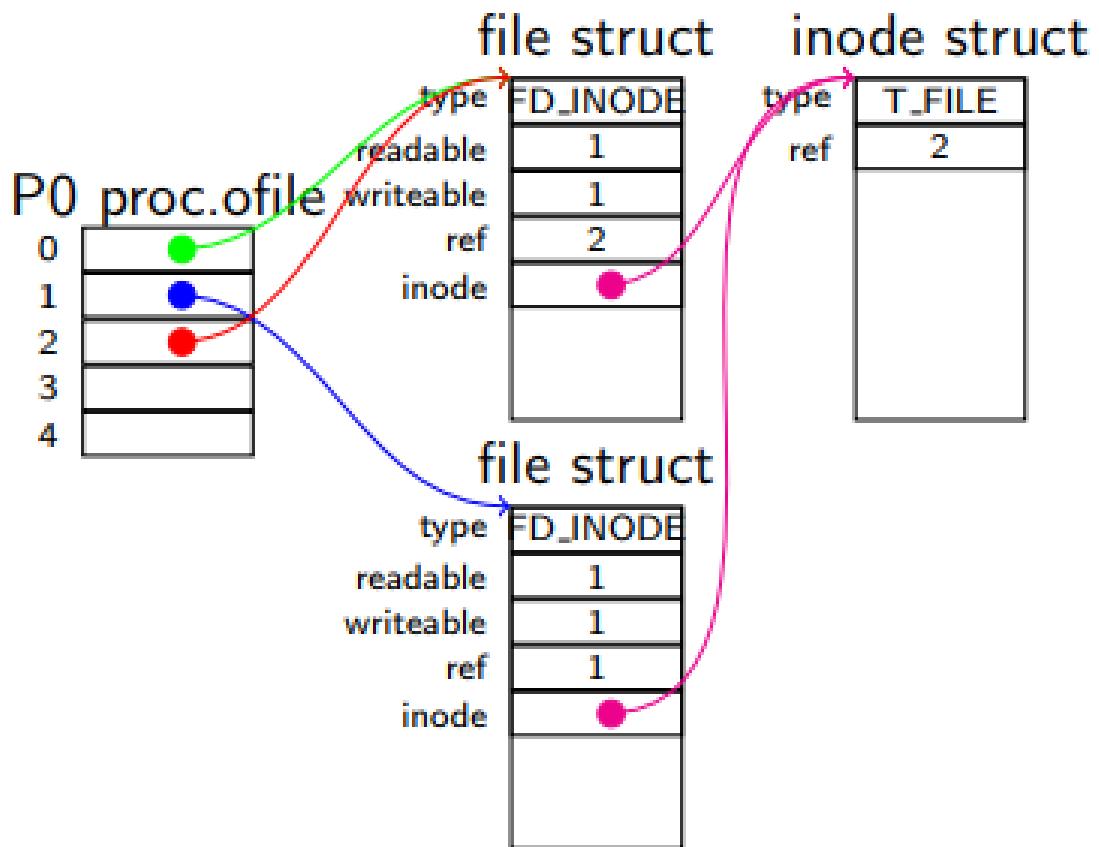
#### 4.6.5 无文件开启时



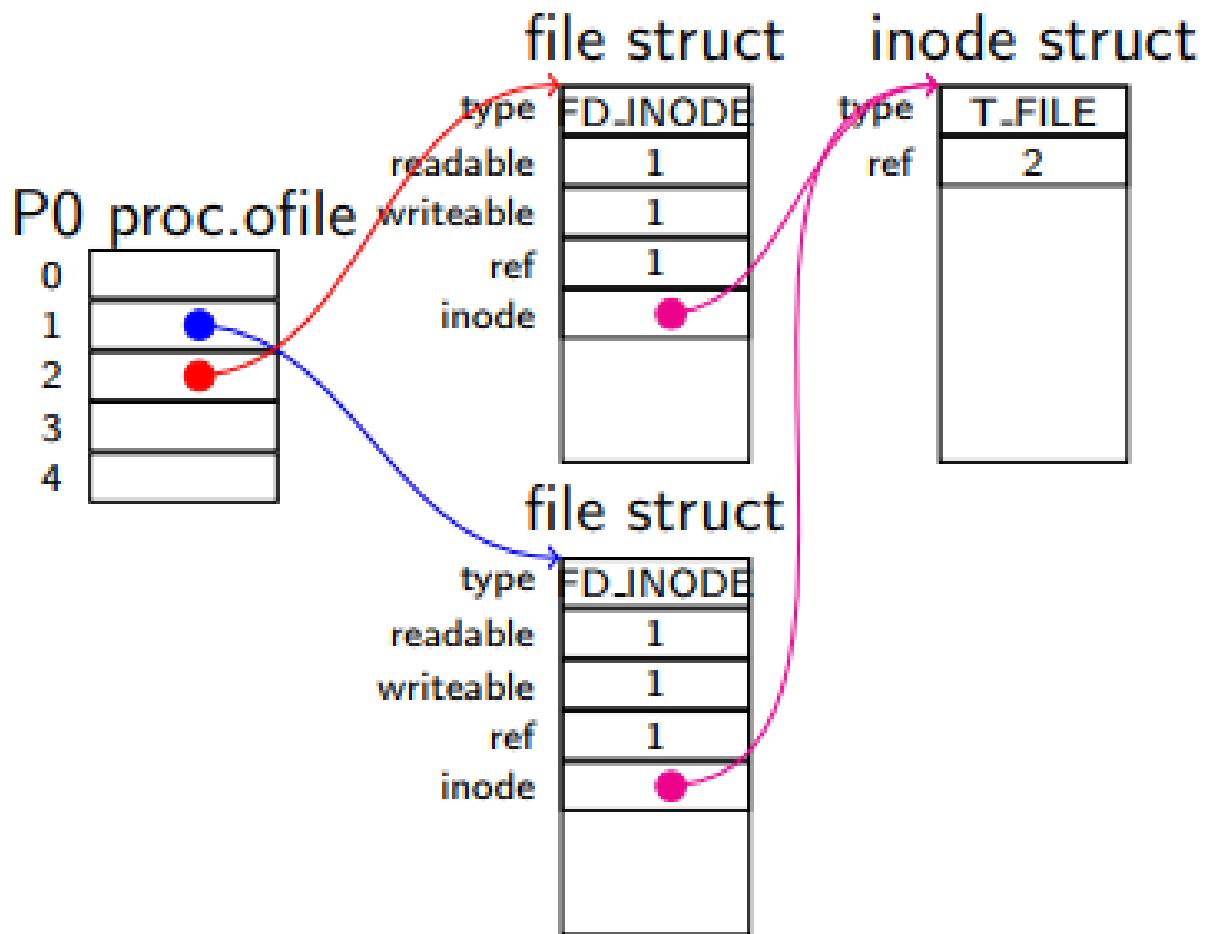
#### 4.6.6 执行 open(" /carmi/f0" )



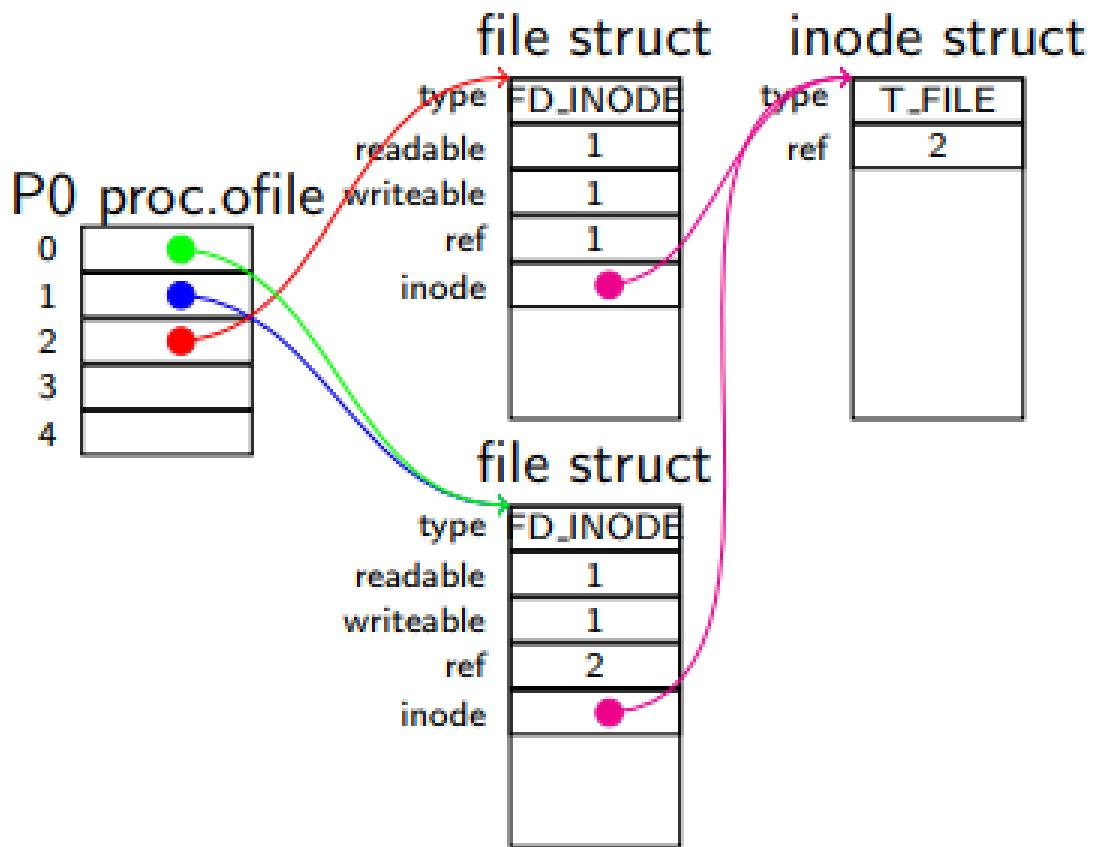
## 4.6.7 执行 dup(0)



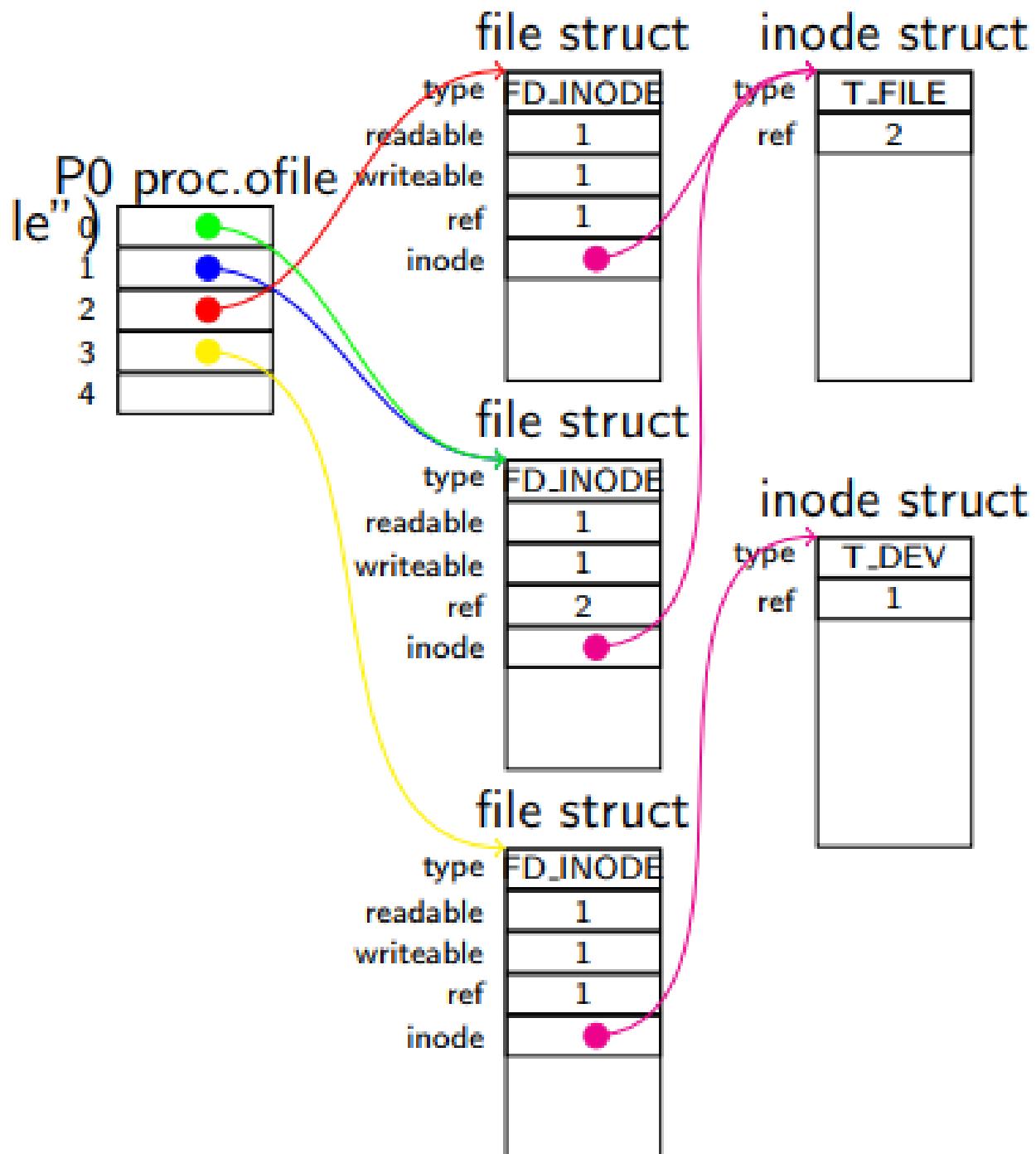
## 4.6.8 执行 close(0)



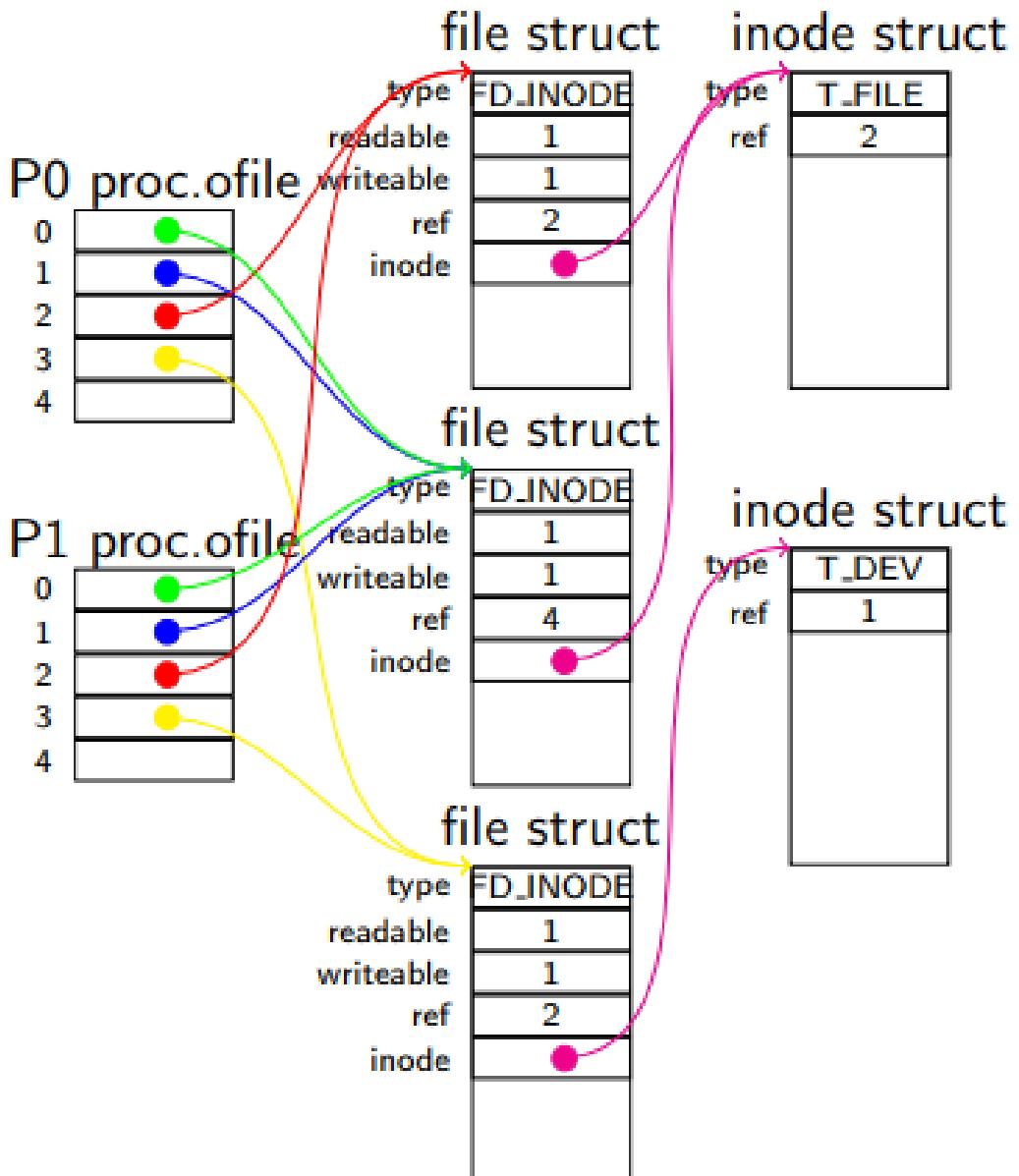
## 4.6.9 执行 dup(1)



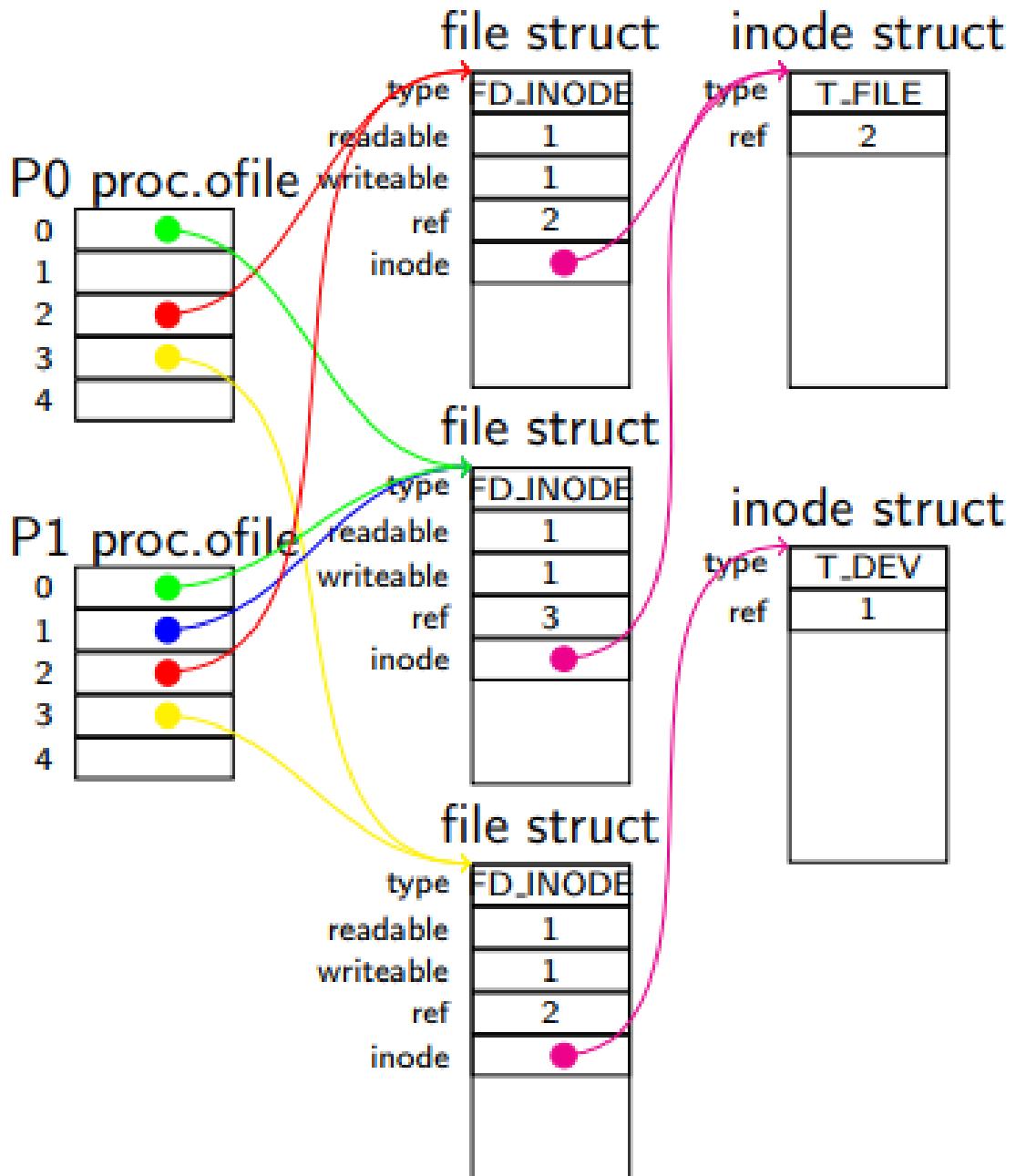
## 4.6.10 执行 open(" /console" )



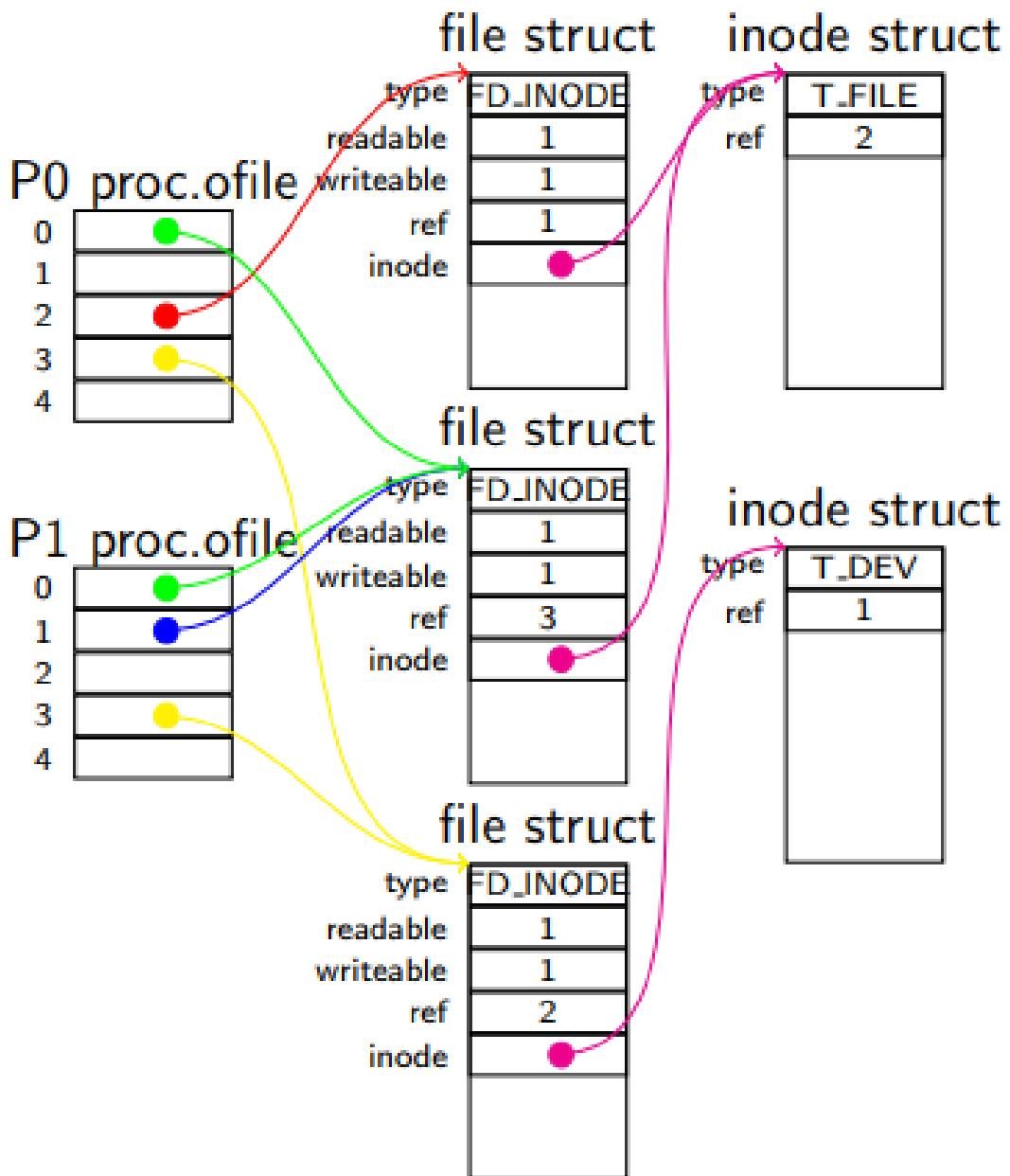
## 4.6.11 执行 fork()



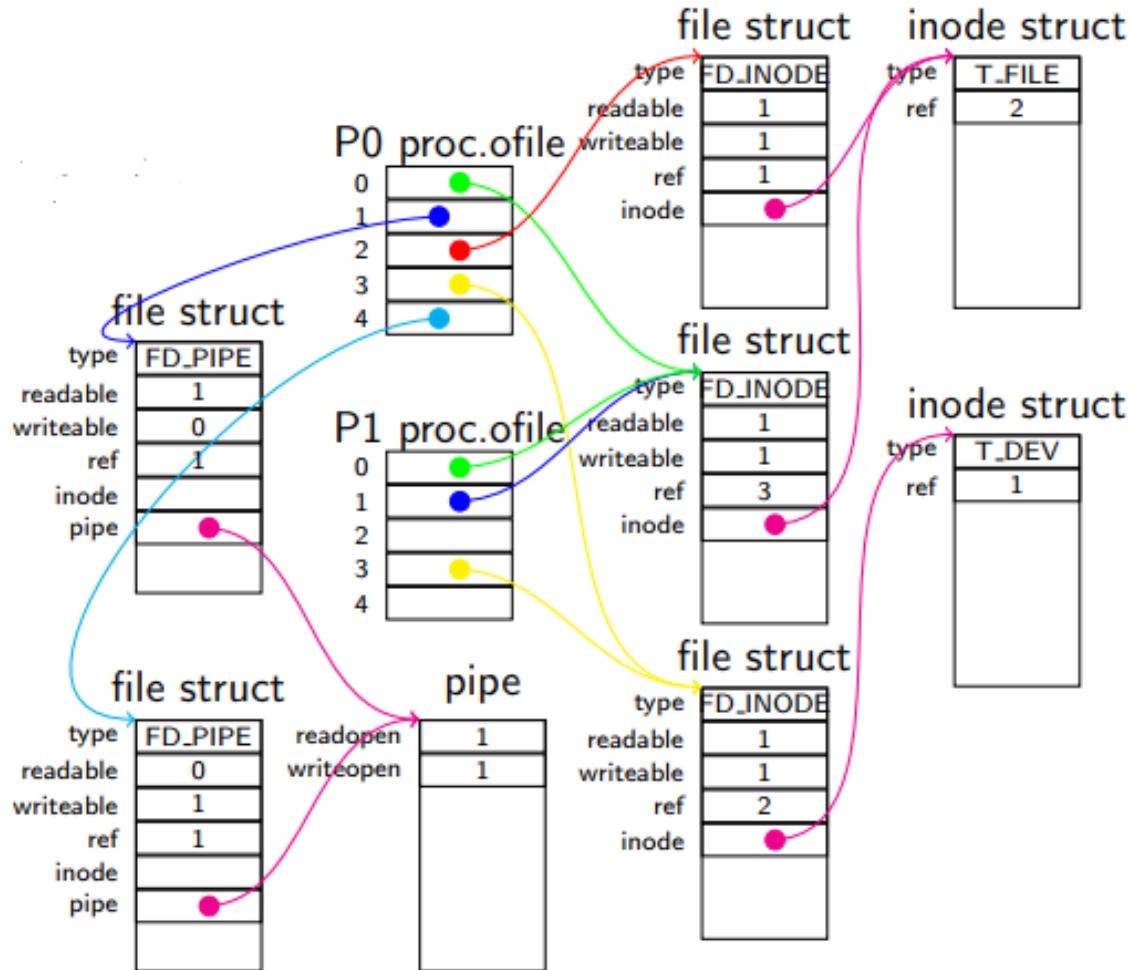
## 4.6.12 执行 P0 close(1)



## 4.6.13 执行 P1 close(2)



#### 4.6.14 执行 P0 pipe()



#### 4.6.15 文件层调度

文件层将调度一下子系统之一:

- pipe.
- inode.

pipe 子系统和 inode 层使用的函数 Pipe 子系统:

- `piperead(pipe *p, char *addr, int len);`
- `pipewrite(pipe *p ,char *addr, int len);`
- `pipeclose(pipe *p ,int writeside);`

inode 子系统:

- ilock(inode \*ip);
- iunlock(inode \*ip);
- readi(inode \*ip, char \*addr, int len);
- writei(inode \*ip, char\* adr, int len);
- iput(inode \*ip);
- stati(inode \*ip, stat\* stat);
- begintrans();
- committrans();

模块详细说明 file\_dup 模块

```

1 struct file* file_dup( struct file*f){
2 acquire(&ftable.lock );
3 if (f->ref<1) panic( " filedup " );
4 f->ref++;
5 release(&ftable.lock );
6 return f;
7 }
```

根据类型, file\_read 将读取委托给以下其中一个: 管道,readi。对于 FD\_INODE 类型, 处理文件位置。file\_read 模块

```

1 int file_read( structfile *f , char *addr ,int n){
2 int r ;
3 if ( f->readable==0) return -1;
4 if ( f->type==FDPIPE) return pipe_read( f->pipe ,addr ,n );
5 if ( f->type==FDINODE){
6 ilock ( f->ip );
7 if (( r=readi( f->ip ,addr ,f->off ,n ))>0) f->off+=r ;
8 iunlock ( f->ip );
9 return r ;
10 }
11 panic( " fileread " );
12 }
```

根据类型, filewrite 将写入委托给以下其中一个: pipe 写入, writei。对于 FD\_INODE 类型, 处理文件位置。file\_write 模块

```

1 int file_write(struct file *f, char* addr, int n){
2 if (f->writable==0) return -1;
3 if (f->type==FDPIPE) return pipe_write(f->pipe,addr,n);
4 if (f->type==FDINODE){
5 int max=((LOGSIZE-1-1-2)/2)*512;
6 for (int i=0;i<n;){
7 int n1=n-i;
8 if (n1>max) n1=max;
9 begintrans();
10 ilock(f->ip);
11 if ((r=writei(f->ip,addr+i,f->off,n1))>0) f->off+=r;
12 iunlock(f->ip);
13 committrans();
14 if (r<0) break;
15 if (r!=n1) panic("shortfilewrite");
16 i+=r;
17 }
18 return i==n ? n:-1;
19 }
20 panic("filewrite");
21 }
```

reference count 已更新。如果 reference count 下降到零, 我们委托给其中一个: pipe 关闭, iput。由于 iput 我们必须释放需要注意, 因此需要复制文件结构。file\_stat 模块

```

1 int file_stat(struct file *f, struct stat *st){
2 if (f->type==FDINODE){
3 ilock(f->ip);
4 stati(f->ip,st);
5 iunlock(f->ip);
6 return 0;
7 }
8 return -1;
9 }
```

file\_close 模块

```

1 void file_close(struct file *f){
```

```

2 acquire(&ftable.lock);
3 if (f->ref<1) panic("fclose");
4 if (--f->ref>0){
5 release(&ftable.lock);
6 return;
7 }
8 struct ff=*f;
9 f->ref=0;
10 f->type=FDNONE;
11 release(&ftable.lock);
12 if (ff.type==FDPIPE) pipeclose(ff.pipe, ff.writable);
13 elseif (ff.type==FDINODE){
14 begintrans(); iput(ff.ip); committrans();
15 }
16 }
```

#### 4.6.16 I/O 缓存 Buf 结构

xv6 用 buf 对磁盘上的 block 块进行缓存。buf 是一个 LRU 链表。也就说最近访问的 buf 总是放在链表的首部。bio.c binit 是将整个 buf 链表初始化，并将其链成一个表。bget 则寻找一个扇区的 buf。首先在 buf 链表中寻找是否有此扇区的 buf，如果没有的话，则加入开辟一个新的 buf 存储扇区。bread 是读取某一个扇区。首先通过 bget 找到相应的 buf。如果发现状态不是 B\_VALID，则通过 ide\_rw 进行磁盘同步。bwrite 是将 buf 中的内容写入扇区。调用 bwrite 之前需要在外部对 buf，从而保证一致性。brelse 是用来释放 buf。这个函数是当对 buf 访问结束时调用的。此时基于 LRU 算法，将把 buf 放在 buf 链表的头部。ide.c 此文件是对 ide 磁盘进行读写操作。I/O 读写是异步的。其中主要的流程如下：首先看 ide\_rw。对于每个读写操作，都有一个 ide\_queue 进行排队。在 ide\_rw 函数中，会把当前的进程加入到 ide\_queue 中。如果之前 disk 没有开始读写，则唤起 disk 读写。(141-142 行)。然后在 145-146 进行轮询等待读写进程的完成。轮询中并不是盲等，而是会 sleep 当前进程，等待进程完成是被唤起。

ide\_start\_request 过程是将 buf b 进程请求发送到磁盘。当进程完成时，磁盘将发出完成的硬件中断。ide\_intr 则是相应此中断的过程。ide\_intr 将在 106-108 行清除状态，并唤醒等待此 buf 的进程。最后如果此进程在 ide\_queue 有后继进程，则启动此进程。

#### 4.6.17 APIC

在对称多处理实现中，CPU 需要处理各个设备发来的中断，我们将这种中断纳入 I/O 的管理。APIC (Advanced Programmable Interrupt Controller) 是一个与 8259A 兼容的高级中断处理器。它不但实现了中断处理的功能，还实现了以下功能：

- 提供与中断相关的设备通讯
- 提供多处理器（或多 CPU）之间的中断共享与中断通讯

事实上，xv6 与外部设备的很大一部分通讯处理，都是通过 APIC 来实现的。APIC 分为两层：Local APIC 和 I/O APIC。

### I/O APIC

I/O APIC 是用来与外部设备通讯的，它完成了 APIC 最主要的功能：中断处理。I/O APIC 提供了两个模式，普通模式和 8259A 兼容模式。xv6 在单核环境下，会选择使用 8259A 兼容模式，而在多处理器环境下，则会使用普通模式。

### Local APIC

Local APIC 是 APIC 的顶层，每个核都有一个对应的 Local APIC。它负责进行多处理器之间的中断传输，屏蔽中断，还提供了一个可编程的 Timer。由于 I/O APIC 已经提供了中断处理的功能，Local APIC 只是起辅助作用，可以屏蔽不用。xv6 在单核环境中，Local APIC 被屏蔽不用；而只有在多处理器环境中，Local APIC 才被打开，完成初始化每个核、开关中断、Timer 等功能。

### Timer

如上文所提到的，Local APIC 提供了一个可编程的 Timer。所以 xv6 在多处理器环境下，每个核使用其对应的 Local APIC 提供的 Timer。而在单核环境下，Local APIC 并没有被打开。xv6 使用了 8253PIT(Programmable Interval Timer) 来实现时钟中断。

# 实验感想

通过学习 xv6 实验, 对于操作系统的知识有了更加深入的理解, 之前虽然知道调度算法的伪代码, 但是在自己写出来了一个调度算法之后, 对于调度算法的理解会深刻很多.

在做这些实验的过程之中, 会不断遇到老师上课的时候讲过的一些知识, 在做实验的过程之中, 复习巩固了这些知识点.

我觉得咱们操作系统实验课程需要更加注重实验, 如果只是通过上课的时候听讲来学习始终是纸上谈兵, 对操作系统的知识点理解的并不透彻, 如果通用自己动手写出来一个操作系统, 不仅锻炼了大家的代码能力学会了使用 linux, 而且对于学生深入理解操作系统有很大的帮助.

## 附录 A linux 学习资料

鸟叔的 linux 私房菜 (去天猫买吧)

Ubuntu Linux 新手入门指引 <http://wiki.ubuntu.org.cn/新手入门指引>

Ubuntu Linux 入门指南 <http://wiki.ubuntu.org.cn/Ubuntu 桌面入门指南>

Ubuntu Linux 编程指南 <http://wiki.ubuntu.org.cn/编程语言>

Linux 就该这么学 <http://www.linuxprobe.com/club>

关于操作系统的资料的总结 <https://pdos.csail.mit.edu/6.828/2017/labguide.html>

Mit 实验指导 <https://pdos.csail.mit.edu/6.828/2017/xv6.html>

MIT 操作系统工程的教学操作系统 Xv6 的源码剖析中文翻译项目，使用 ANSI 标准 C 重新在 X86 架构上实现 Unix v6 <https://github.com/deyuhua/xv6-book-chinese>

乐学网上的操作系统资料.(操作系统的课件和一些演示代码, 还有一些辅助的资料和文件) <https://cms.hit.edu.cn/course/view.php?id=44>

## 附录 B 用户态和内核态是什么，有什么区别

多数系统将处理器工作状态划分为内核态和用户态。前者一般指操作系统管理程序运行的状态，具有较高的特权级别，又称为特权态、系统态或管态；后者一般指用户程序运行时的状态，具有较低的特权级别，又称为普通态、目态。如果操作系统允许用户对计算机做任何操作都可以的话，我的计算机一天死 N 次机都不足为奇。区分了用户态和内核态就是限定用户什么操作可以做，什么操作不能让用户直接做。如果遇到不能让用户直接做的操作，用户就必须请求操作系统做系统调用，这样操作系统就会进入内核态进行系统操作。内核态的进程就是系统进入内核态之后进行系统操作所产生的进程。而用户态进程是用户通过请求操作而产生的进程。它们的主要区别有：

- 运行在不同的系统状态，用户态进程执行在用户态，内核态进程执行在内核态。
- 进入的方式不同用户态进程直接进入而内核态必须通过运行系统调用命令。
- 返回方式不同，用户态进程直接返回，内核态进程有重新调度过程。
- 内核态进程优先级要高于用户态进程。并且内核态进程特权级别最高，它可以执行系统级别的代码。

## 附录 C 为什么需要打开 A20 地址线

在 8086 年代，8086 提供了 20 跟地址线，那么提供的可寻址空间范围即  $0-2^{20}(00000H-FFFFFH)$  的 1M 空间，而由于 8086 的数据处理位宽位 16 位，所以 8086 提供了段地址加偏移地址的地址转换机制，就是我常见的“段地址：偏移地址（或有效地址）”，实际的计算方法为：“段地址 \*10H + 偏移地址”，作为段地址的数据是放在段寄存器中的（16 位），而座位偏移地址的数据则是通过 8086 提供的寻址方式来计算而来的（16 位）。而“段值：偏移”这种表示法能够表示的最大内存为  $10FFEEh(FFFF0 + FFFF)$ ，所以当寻址到超过 1MB 的内存时，会发生“回卷”（不会发生异常）。但是到了 80286 提供了 24 根地址线，cpu 的寻址范围变为  $2^{24}=16M$ ，同时也提供了保护模式，真的可以访问到 1MB 以上的内存了，此时如果遇到“寻址超过 1MB”的情况，系统不会再“回卷”了，这就造成了向上不兼容。为了保持完全的兼容性，IBM 决定在 PC AT 系统上加个逻辑，来模仿以上的回绕特征。他们的方法就是把 A20 和键盘控制器的一个输出进行 AND，这样来控制 A20 的打开和关闭。一开始时 A20 是被屏蔽的（总为 0），直到系统软件去打开它。注意 A20 而非 A20-A31 被控制，所以在 A20 关闭时会发生一些有趣的副作用。就是在访问奇数 M 地址空间的时候，实际的地址会减少 1M。例如访问 1M 2M-d1 时实际访问的是 0-1M-1；访问 3M-4M-1 时为 2M-3M-1，等等。

当 A20 Gate 禁止时，则程序就像在 8086 中运行， $100000h-100FFEFh$  的地址是不可访问的。在保护模式下 A20 Gate 是要打开的。为了使能所有地址位的寻址能力，必须向键盘控制器 8042 发送一个命令。键盘控制器 8042 将会将它的某个输出引脚的输出置高电平，作为 A20 门的输入。一旦设置成功之后，内存将不会再被绕回（memory wrapping），这样我就可以寻址整个 286 的 16M 内存，或者是寻址 80386 级别机器的所有 4G 内存了。8042 键盘控制器的 IO 端口是 0x60-0x6f，实际上 IBM PC/AT 使用的只有 0x60 和 0x64 两个端口（0x61、0x62 和 0x63 用于与 XT 兼容目的）。8042 通过这些端口给键盘控制器或键盘发送命令或读取状态。输出端 P2 用于特定目的。位 0（P20 引脚）用于实现 CPU 复位操作，位 1（P21 引脚）用户控制 A20 信号线的开启与否。系统向输入缓冲（端口 0x64）写入一个字节，即发送一个键盘控制器命令。可以带一个参数。参数是通过 0x60 端口发送的。命令的返回值也从端口 0x60 去读。

知乎上还有一些关于为什么要打开 A20 的答案 <https://www.zhihu.com/question/29375534>

## 附录 D 操作系统相关的基本概念

本节描述的与操作系统相关的基本概念对理解后续的分析 xv6 操作系统设计与实现很有帮助。

### D.1 操作系统内核 (Operating System Kernel)

操作系统是计算机系统中的系统软件，它是这样一些程序模块的集合：它们能有效地组织和管理计算机系统中的软硬件资源，合理地组织计算机工作流程，控制程序的执行，并向用户提供各种服务功能，使得用户能够灵活、方便、有效地使用计算机，使整个计算机系统能高效地运行。

操作系统内核是操作系统中的核心部分，在多进程系统中，操作系统内核提供的最基本服务是进程调度与切换、中断服务。操作系统内核为每个进程分配 CPU 时间，并且负责进程之间的通信。

进程调度与切换是实现多进程并发运行的重要机制，而中断服务是实现可抢占的进程调度与切换的高效手段（其中最主要一种中断服务的是时间中断管理），同时也是实现与外设交互的必要措施。扩展的服务包括同步互斥、内存管理、文件系统管理、网络管理等，在具体的操作系统中不一定必须提供。

操作系统内核一般提供同步互斥等服务，如信号量、消息队列等。操作系统内核提供的文件管理为保存和读写需要长期保存的数据提供了一种有效的方法。操作系统内核提供的网络管理主要是包括基于 TCP/IP 或其它通信协议的协议栈实现。

### D.2 不可抢占型内核 (Non-Preemptive Kernel)

不可抢占型内核要求每个进程自我放弃 CPU 的所有权，所以不可抢占型调度也称作合作型多进程调度，其调度的完成需要基于各个进程对 CPU 主动“弃权”，只要进程自己不放弃 CPU，它就可以一直运行下去。虽然对于外设的异步事件还是由中断服务来处理，即中断服务可以使一个高优先级的进程由阻塞状态变为就绪状态，但中断服务以后控制权还是回到原来被中断了的那个进程。只有直到该进程主动放弃 CPU 的使用权时，那个高优先级的进程

才能获得 CPU 的使用权。

在不可抢占型内核中，由于每个进程要运行到自身结束时才释放 CPU 的控制权，所以可以在应用进程中使用不可重入函数（函数的可重入性在本节的靠后部分有介绍），而不必担心其它进程可能也会正在使用该函数，从而造成对共享数据的破坏。当然该不可重入型函数本身不能放弃 CPU 控制权。不可抢占型内核的另一个优点是，几乎不需要使用某种互斥机制（如信号量等）保护共享数据。这时由于运行着的进程始终占有 CPU，而不必担心被别的进程抢占。但这也不是绝对的，比如在处理 I/O 设备时，可能存在中断服务例程与应用进程共享某些资源，这就仍需要使用某种互斥机制来“安全”地访问这些共享资源。不可抢占型内核的主要缺点是过于缓慢的响应时间。其原因在于虽然高优先级的进程已经进入就绪态，但还不能运行，要等到当前运行着的进程释放 CPU 后才能运行。这导致不可抢占型内核的进程级响应时间是不确定的，不知道什么时候最高优先级的进程才能获得 CPU 的控制权。

### D.3 可抢占型内核 (Preemptive Kernel)

绝大多数的操作系统内核都是可抢占型内核。在可抢占型内核中，处于最高优先级的进程一旦就绪（即就等 CPU 这个资源了），内核就会马上调度此进程，让这个最高优先级的进程总能得到 CPU 的控制权。我也可以换一种说法，当一个运行着的进程使一个比它优先级高的进程进入了就绪态，当前进程的 CPU 使用权就被抢占了，或者说被阻塞了，那个高优先级的进程立刻得到了 CPU 的控制权。如果是中断服务例程使一个高优先级的进程进入就绪态，中断完成时，中断了的进程被阻塞，优先级最高的那个进程（不一定是那个被中断了的进程）开始运行。使用可抢占型内核，最高优先级的进程何时可以得到 CPU 的控制权并运行是可确定的。这样使用可抢占型内核使得进程级响应时间得以最优化。

使用可抢占型内核时，应用程序不应直接使用不可重入型函数。如果调用不可重入型函数时，低优先级的进程被高优先级进程抢占，那么不可重入型函数中的数据有可能被破坏。如果确实需要调用不可重入型函数，那么在调用不可重入型函数时，必须满足互斥条件，这一点可以用信号量等互斥机制来实现。

## D.4 进程 (process)

在前面，已经多次提到进程了，但进程到底是什么呢？从操作系统原理上看，一个进程是一个具有一定独立功能的程序在一个数据集合上的一次动态执行过程。进程的组成包括程序、数据和进程控制块（即进程状态信息）。在操作系统中，进程一般也称作线程（thread），因为在一般的操作系统中的所有进程共享操作系统提供的各种资源（内存、CPU、同步互斥资源等），且与操作系统一起位于同一地址空间，都处于特权态。每个进程都是整个应用的某一部分，每个进程被赋予一定的优先级，且每个进程在逻辑上维护了进程的运行状态信息，即与进程运行直接相关的 CPU 寄存器和栈空间（只有这样才能实现进程切换）。一般应用实际上是由多个进程组成，由操作系统根据当前进程的情况设置进程的状态，并根据进程的优先级进行调度。

## D.5 进程状态

在进程的运行过程中，每个进程都处在以下五种状态之一：

- 创建 (new) 态：一个进程正在被创建，还没被转到就绪状态之前的状态。
- 就绪 (ready) 态：该进程获得了除 CPU 之外的一切所需资源，已经准备好运行了，但由于还没有占用 CPU，所有还暂时不能运行，一旦得到处理机即可运行。
- 运行 (running) 态：该进程获得了 CPU 使用权，正在运行中。
- 阻塞 (blocked) 态：该进程在等待某一事件（如某个资源可用的事件等）发生，而暂停运行。
- 退出 (exit) 态：一个进程正在从系统中消失时的状态，这是因为进程结束或由于其他原因所导致。

可能的状态变化如下所示：

- NULL→New：一个新进程被产生出来执行一个程序。
- New→Ready：当进程被创建完成并初始化后，一切就绪准备运行时，变为就绪状态。

- Ready→Running: 处于就绪状态的进程被进程调度程序选中后，就分配到 CPU 上来运行
- Running→Exit: 当进程表示它已经完成或者出错，当前运行进程会由操作系统作结束处理。
- Running→Ready: 处于运行状态的进程在其运行过程中，由于分配给它的处理机时间片用完或被中断服务例程、高优先级进程抢占而让出 CPU。
- Running→Blocked: 当进程请求某资源且必须等待。
- Blocked→Ready: 当进程要等待某事件到来时，它从阻塞状态变到就绪状态。

## D.6 多进程

多进程运行的实现实际上是靠操作系统利用中断机制和调度机制，让不同的进程按照优先级等因素分时占用 CPU。多进程运行使 CPU 得到充分利用，并使应用模块化。

## D.7 调度 (Scheduling)

调度 (Scheduling) 就是要决定该轮到哪个进程运行了。这是操作系统内核的主要职责之一。多数操作系统内核是基于优先级调度算法，即每个进程根据其重要程度的不同被赋予一定的优先级，由操作系统内核选择某一个优先级最高且可以马上执行的进程占用 CPU 来运行。根据何时让高优先级进程掌握 CPU 的使用权，可以把操作系统内核分为两种类型，即不可抢占型操作系统内核和可抢占型操作系统内核。

## D.8 进程调度算法

当进程间的优先级不同时，选择优先级高的进程获得 CPU 控制权，这种调度方法称为优先级调度法。当两个或两个以上进程有同样优先级，由两种方法可以采用。一个是 FIFO（先来先出）调度法，即按照进程的到达顺序来选择进程获得 CPU 控制权的顺序，只有当一个进

程执行完毕或阻塞后，下一个同优先级的进程才能获得 CPU 控制权。内核允许一个进程运行事先确定的一段时间，叫做时间片（quantum，也称时间额度），然后切换给另一个进程。这种调度方法称为时间片轮转调度法，也称时间片调度。内核在满足以下条件时，把 CPU 控制权交给下一个进程就绪态的进程：

- 当前进程的时间片用完了
- 当前进程在时间片还没结束时已经完成了

## D.9 可重入性（Reentrancy）

可重入型函数可以被一个以上的进程调用，而不必担心数据的破坏。可重入型函数任何时候都可以被中断，一段时间以后又可以继续运行，而相应数据不会丢失。可重入型函数一般只使用局部变量，即变量保存在 CPU 寄存器中或堆栈中。如果使用全局变量，则要对全局变量予以互斥保护。下面的程序片断是一个可重入型函数的例子。

```
1 void strcpy(char *dest, char *src)
2 {
3     while (*dest++ = *src++) {};
4     *dest = NUL;
5 }
```

函数 strcpy 做字符串复制。因为参数是存在堆栈中的，故函数 strcpy 可以被多个进程调用，而不必担心各进程调用函数期间会互相破坏对方的指针。不可重入型函数的例子如下所示。strcpy2 是一个简单函数，它也是做字符串复制。为便于讨论，假定使用的是可抢占型内核，中断是开着的，temp 是一个全局变量。

```
1 char temp;
2
3 void strcpy2( char *dest, char *src)
4 {
5     while (*src!=NULL) {
6         temp=*src;
```

```
7     src++;
8     *dest=temp;
9     dest++;
10    }
11    *dest = NUL;
12 }
```

假设 strcpy2 函数可以为任何进程所调用，如果一个低优先级的进程 A 正在执行 strcpy2 函数，把 temp 赋值为'a' 后，而此时中断发生了，打断了进程 A 的执行，中断服务例程执行，当中断服务例程执行完毕后，操作系统内核使最高优先级的就绪进程 B 运行，进程 B 也调用 strcpy2 函数，且把 temp 赋值为'b'。这对该进程 B 本身来说，实现两个变量的交换是没有问题的。然后当进程 B 运行完毕后，释放了 CPU 的使用权，低优先级进程 A 得以继续运行。而此时 temp 的值仍为'b'。这样导致进程 A 执行的结果出错。使用以下技术之一即可使 strcpy2 函数具有可重入性：把 temp 定义为局部变量调用 strcpy2 函数之前屏蔽中断，调动后再使能中断用信号量禁止该函数在使用过程中被再次调用

## D.10 上下文切换 (Context Switch or Task Switch)

在操作系统中通常也把上下文切换称为进程切换 (task switch)。当操作系统决定运行另外的进程时，它要保存当前正在运行进程的当前上下文 (Context，也可称“运行状态信息”)，即 CPU 寄存器中的全部内容。这些内容保存在进程的堆栈空间或特定的上下文保存区。完成保存工作后，操作系统就可以把下一个将要运行的进程的当前上下文从该进程的栈或上下文保存区中恢复到 CPU 的寄存器中，这样就可以继续下一个进程的运行。这个过程叫做进程切换。一般有两种情况的上下文切换：

- 高优先级的进程因为需要某种临界资源，主动请求阻塞，让出处理器，此时将调度就绪状态的低优先级进程获得执行。这种切换称为进程级的上下文切换。
- 进程因为时钟中断或其它中断到来而被打断，在中断服务例程处理完毕后，内核发现有更高优先级进程处于就绪态，则在中断处理结束后直接切换到高优先级进程执行。这种调度也称为中断级的上下文切换。

## D.11 进程优先级

每个进程都有其优先级。进程越重要，赋予的优先级应越高。

### D.11.1 静态进程优先级

应用在执行过程中，各个进程优先级不变，则称进程的优先级为静态进程优先级。在这样的操作系统中，各个进程以及它们的时间约束在应用程序编译时是已知的。

### D.11.2 动态进程优先级

应用执行过程中，各个进程的优先级会随着时间的流逝而改变，则称进程的优先级为动态进程优先级。

## D.12 资源

任何为进程所占用的实体都可称为资源。资源可以是 CPU、内存，也可用是 I/O 设备，还可以是一个变量，一个结构或一个数组等。

## D.13 共享资源

可以被一个以上进程使用的资源叫做共享资源。为了防止数据被随意访问（特别是执行写操作），每个进程在与共享资源打交道时，必须独占该资源。这叫做互斥（mutual exclusion）。需要互斥访问的共享资源称为临界资源。

## D.14 竞争状态（race condition）

竞争状态是指两个或多个进程对同一共享资源同时进行读写操作，而最后的结果是不可预测的，而是取决于各个进程具体运行情况。

## D.15 程序临界区

对共享资源的访问，可能会导致竞争状态的出现。我把完成这类事情的那段程序片断称为程序临界区。程序临界区在处理时不可被打断，要保证其操作的原子性。为确保临界区程序执行不被打断，在进入临界区之前要屏蔽中断，而临界区代码执行完以后要立即使能中断，以减少对中断处理延迟的影响。

## D.16 互斥条件

当所有的进程都在一个单一地址空间下，实现进程间通信最简便的办法是使用基于全局变量的共享数据结构（指针、缓冲区、链表、循环缓冲区）。虽然共享数据区法简化了进程间的信息交换，但是必须保证每个进程在处理共享数据时的排它性和互斥性，以避免竞争和数据的破坏。解决临界区问题应满足的要求包括：

- 互斥：如果进程在其临界区执行，那么其它进程不能在其临界区内执行。
- 有空让进（前进要求）：如果没有进程在其临界区且有进程希望进入临界区，那么只有那些不在剩余区内的进程能参加决策，且选择不能无限等待；
- 有限等待：在一个进程  $T_i$  做出进入临界区的请求到该请求被允许期间，其它进程  $T_j$  被允许进入其临界区的等待时间（ $P_i$  进入临界区的次数）存在一个上限（进程不能“死等”）。

与共享资源打交道时，使之满足互斥条件以避免临界区问题的最一般的方法有：

- 屏蔽中断（也称关中断）
- 使用测试并置位指令
- 禁止进程切换
- 信号量

### D.16.1 屏蔽中断和使能中断

处理共享数据时保证互斥，最简便快捷的办法是屏蔽中断（关中断）和使能中断（开中断）。可是，必须十分小心，屏蔽中断的时间不能太长。因为它影响整个系统的中断响应时间，即中断延迟时间。当改变或复制某几个变量的值时，应想到用这种方法来做。这也是在中断服务例程中处理共享变量或共享数据结构的唯一方法。在任何情况下，屏蔽中断的时间都要尽量短。

### D.16.2 测试并置位指令

当两个进程共享一个资源时，一定要约定好，先测试某一全程变量，如果该变量是 0，允许该进程与共享资源打交道。为防止另一进程也要使用该资源，前者只要简单地将全程变量置为 1，这通常称作测试并置位 (Test-And-Set) 操作，或称作 TAS 操作。TAS 操作一般是 CPU 提供的一条不会被中断的指令。通过这个指令可以实现对共享资源的互斥访问。TAS 指令读出标志后设置为 TRUE，下面是 TAS 指令的逻辑流程：

```
1  boolean TAS(boolean *lock) {
2      boolean old;
3      old = *lock; *lock = TRUE;
4      return old;
5 }
```

lock 表示资源的两种状态：TRUE 表示正被占用，FALSE 表示空闲。

### D.16.3 禁止进程切换，然后允许进程切换

如果进程不与中断服务例程共享变量或数据结构，可以使用禁止进程切换，然后允许进程切换的方法。这样两个或两个以上的进程可以共享数据而不发生冲突。注意，此时虽然进程切换是禁止了，但中断还是开着的。如果这时中断来了，中断服务例程会在这一临界区内立即执行。中断服务例程结束时，尽管有优先级高的进程已经进入就绪态，由于设置了禁止进程切换，所以内核还是返回到原来被中断了的进程。直到执行完允许进程切换函数后，内核再看有没有优先级更高的就绪态进程，如果有，则做进程切换。虽然这种方法是可行的，但

应该尽量避免禁止进程切换之类操作，因为内核最主要的功能就是做进程的调度与协调。禁止进程切换显然与内核的初衷相违。

#### D.16.4 信号量 (Semaphores)

信号量和对应的 P/V 原语是二十世纪六十年代中期荷兰的计算机科学家 Edsger Dijkstra 提出的。所以 P、V 分别是荷兰语的 test(proberen) 和 increment(verhogen) 的意思。信号量实际上是一种同步约定机制，在操作系统内核中普遍使用。信号量用于：

- 控制共享资源的使用权 (满足互斥条件)
- 标志某事件的发生
- 使两个进程的行为同步

信号像是一把钥匙，进程要运行下去，得先拿到这把钥匙。如果信号已被别的进程占用，该进程只得被阻塞，直到信号被当前使用者释放。换句话说，申请信号的进程是在说：“把钥匙给我，如果谁正在用着，我只好等！”。在实现上，每个信号量 s 除一个整数值 s.count（计数值）外，还有一个进程等待队列 s.queue，其中是阻塞在该信号量的各个进程的标识。信号量结构体的一般定义如下：

```
1 typedef struct {
2     int count;           // 计数值变量
3     struct TCB *queue;  // 进程等待队列
4 } semaphore;
```

信号量有两种类型：二值 (binary) 信号量，计数 (counting) 信号量。二值信号量只有两个计数值 0 和 1 的信号量量。计数信号量的计数值可以是大于 1 的，具体的范围取决于信号量规约机制使用的是 8 位、16 位还是 32 位。一般地说，对信号量只能实施三种操作：初始化 (INITIALIZE)，也可称作建立 (CREATE)；等信号 (WAIT) 也可称作阻塞 (SUSPEND)；给信号 (SIGNAL) 或发信号 (POST)。信号量初始化时要给信号量赋初值，初始化指定一个非负整数值，表示空闲资源总数（又称为“资源信号量”）——若为非负值表示当前的空闲资源数，若为负值其绝对值表示当前等待临界区的进程数。等待信号量的进程等待队列应清为空。想要得到信号量的进程执行等待 (WAIT) 操作。如果该信号量有效 (即信号量值大于 0)，则

信号量值减 1，进程得以继续运行。如果信号量的值为 0，等待信号量的进程就被列入等待信号量的进程等待队列中。多数内核允许用户定义等待超时，如果等待时间超过了某一设定值时，该信号量还是无效，则等待信号量的进程进入就绪态准备运行，并返回出错代码（指出了发生了等待超时错误）。等待操作的实现逻辑如下所示：

```
1 wait(semaphore s) {  
2     --s.count;           // 表示申请一个资源；  
3     if (s.count < 0)      // 表示没有空闲资源；  
4     {  
5         把进程放入进程等待队列 s.queue;  
6         阻塞调用进程；  
7     }  
8 }
```

进程以发信号操作 (SIGNAL) 释放信号量。如果没有进程在等待信号量，信号量的值仅仅是简单地加 1。如果有进程在等待该信号量，那么就会有一个进程进入就绪态，信号量的值也就不加 1。于是钥匙给了等待信号量的诸进程中一个进程。至于给了那个进程，要看内核是如何调度的。收到信号量的进程可能是以下两者之一。等待信号量进程中优先级最高的进程（基于优先级排队）最早开始等待信号量的那个进程（基于按先进先出的原则 (First In First Out , FIFO)）发信号操作的实现逻辑如下所示：

```
1 post(semaphore s){  
2     ++s.count;           // 表示释放一个资源；  
3     if (s.count ≤ 0)      // 表示有进程处于阻塞状态；  
4     {  
5         从进程等待队列s.queue中取出一个进程T;  
6         进程T进入进程就绪队列；  
7     }  
8 }
```

处理简单的共享变量也使用信号量则是多余的。因为请求和释放信号量的过程是要花相当的时间。这时只需要屏蔽中断、使能中断来处理简单共享变量，可以有效地提高效率。如果屏蔽中断后的处理时间很长，会影响中断延迟时间，这种情况下就有必要使用信号量了。

## D.17 死锁 (或抱死) (Deadlock (or Deadly Embrace))

在一组进程中，每个进程都占用着若干个资源，同时又在等待得到该组进程中另一进程所占用的资源，因而造成的所有进程都无法进展下去的现象，这种现象称为死锁（也称作抱死），这一组进程就称为死锁进程。在死锁状态下，每个进程都动弹不得，既无法运行，也无法释放所占用的资源，它们互为因果、互相等待。例如，设进程 T1 正独享资源 R1，进程 T2 在独享资源 T2，而此时 T1 又要独享 R2，T2 也要独享 R1，于是哪个进程都没法继续执行了，发生了死锁。只有当以下四个条件同时成立时，才会出现死锁：

- 互斥：在任何时刻，每一个资源最多只能被一个进程所使用；
- 占有并等待：进程在占用若干个资源的同时又可以请求新的资源；
- 非抢占：进程已经占用的资源，不会被强制性拿走，而必须由该进程主动释放；
- 循环等待：存在一条由两个或多个进程所组成的环路链，其中每一个进程都在等待环路链中下一个进程所占用的资源。

最简单的防止发生死锁的方法是让每个进程都：

- 先得到全部需要的资源再做下一步的工作
- 用同样的顺序去申请多个资源
- 释放资源时使用相反的顺序

内核大多允许用户在申请信号量时定义等待超时，以此化解死锁。当等待时间超过了某一确定值，信号量还是无效状态，就会返回某种形式的出现超时错误的代码，这个出错代码告知该进程，不是得到了资源使用权，而是系统错误。死锁一般发生在大型多进程系统中，在嵌入式系统中不易出现。

## D.18 同步

同步是指让多个进程之间能够按照某种时序执行。可以利用信号量使某进程与中断服务同步（或者是与另一个进程同步，这两个进程间没有数据交换）。如果内核支持计数信号量，信号量的值表示尚未得到处理的事件数。请注意，可能会有一个以上的进程在等待同一事件的发生，则这种情况下内核会根据以下原则之一发信号给相应的进程：

- 等待信号量进程中优先级最高的进程（基于优先级排队）
- 最早开始等待信号量的那个进程（基于按先进先出的原则 (First In First Out , FIFO)）

## D.19 进程间的通信 (Intertask Communication)

有时很需要进程间的或中断服务与进程间的通信。这种信息传递称为进程间的通信。进程间信息的传递有两个途径：

- 通过全程变量或发消息给另一个进程
- 使用邮箱或消息队列等通信机制

用全程变量时，必须保证每个进程或中断服务程序独享该变量。中断服务中保证独享的唯一办法是屏蔽中断。如果两个进程共享某变量，各进程实现独享该变量的办法可以是屏蔽中断再使能中断，或使用信号量 (如前面提到的那样)。

需要注意的是，进程只能通过全程变量与中断服务程序通信，而进程并不知道什么时候全程变量被中断服务程序修改了，除非中断程序以信号量方式向进程发信号或者是该进程以查询方式不断周期性地查询变量的值。要避免这种情况，用户可以考虑使用邮箱或消息队列等通信机制。

## D.20 中断

中断是一种硬件机制，用于通知 CPU 有个异步事件发生了。中断一旦被识别，CPU 保存部分 (或全部) 现场 (Context) 即部分或全部寄存器的值，跳转到操作系统提供的专门的函数进行处理，称为中断服务例程 (ISR)。中断服务例程做事件处理，处理完成后，程序回到：

- 对不可抢占型内核而言，程序回到被中断了的进程
- 对可抢占型内核而言，让进入就绪态的优先级最高的进程开始运行

中断使得 CPU 可以在事件发生时才予以处理，而不必让微处理器连续不断地查询 (Polling) 是否有事件发生。通过两条特殊指令：屏蔽中断 (Disable interrupt) 和使能中断 (Enable interrupt) 可以让微处理器不响应或响应中断。屏蔽中断的时间应尽量的短。屏蔽中断影响中断

延迟时间。屏蔽中断时间太长可能会引起中断丢失。微处理器一般允许中断嵌套，也就是说在中断服务期间，微处理器可以识别另一个更重要的中断，并服务于那个更重要的中断。

## D.21 时钟节拍 (Clock Tick)

时钟节拍是特定的周期性中断。这个中断可以看作是系统心脏的脉动。中断之间的时间间隔取决于不同的应用，一般在 1ms 到 100ms 之间。时钟的节拍式中断使得内核可以将进程延时若干个整数时钟节拍，以及当进程等待事件发生时，提供等待超时的依据。时钟节拍率越快，系统的额外开销就越大。

## D.22 对存储器的需求

使用多进程操作系统内核和应用所需空间大小取决于多种因素。代码空间总需求量为：  
$$\text{总代码量} = \text{应用程序代码} + \text{内核代码}$$
由于每个进程都是独立运行的，所以至少需要给每个进程提供单独的栈空间。应用程序设计人员决定分配给每个进程多少栈空间时，应该尽可能使之接近实际需求量（有时，这是相当困难的一件事）。栈空间的大小不仅仅要计算进程本身的需求（局部变量、函数调用等等），还需要计算最多中断嵌套层数（保存寄存器、中断服务程序中的局部变量等）。根据不同的目标微处理器和内核的类型，进程栈和系统栈可以是分开的。系统栈专门用于处理中断级代码。这样做有许多好处，每个进程需要的栈空间可以大大减少。所有内核都需要额外的栈空间以保证内部变量、数据结构、队列等。如果内核支持进程栈和系统栈（用于中断），且二者分离，总 RAM 需求量为：  
$$\text{RAM 总需求} = \text{应用程序数据区的 RAM 需求} + \text{内核数据区的 RAM 需求} + \text{各进程栈之总和的 RAM 需求} + \text{最多中断嵌套栈的 RAM 需求}$$