

## 第33回 | 打开终端设备文件

Original 闪客 低并发编程 2022-04-13 17:30

收录于合集

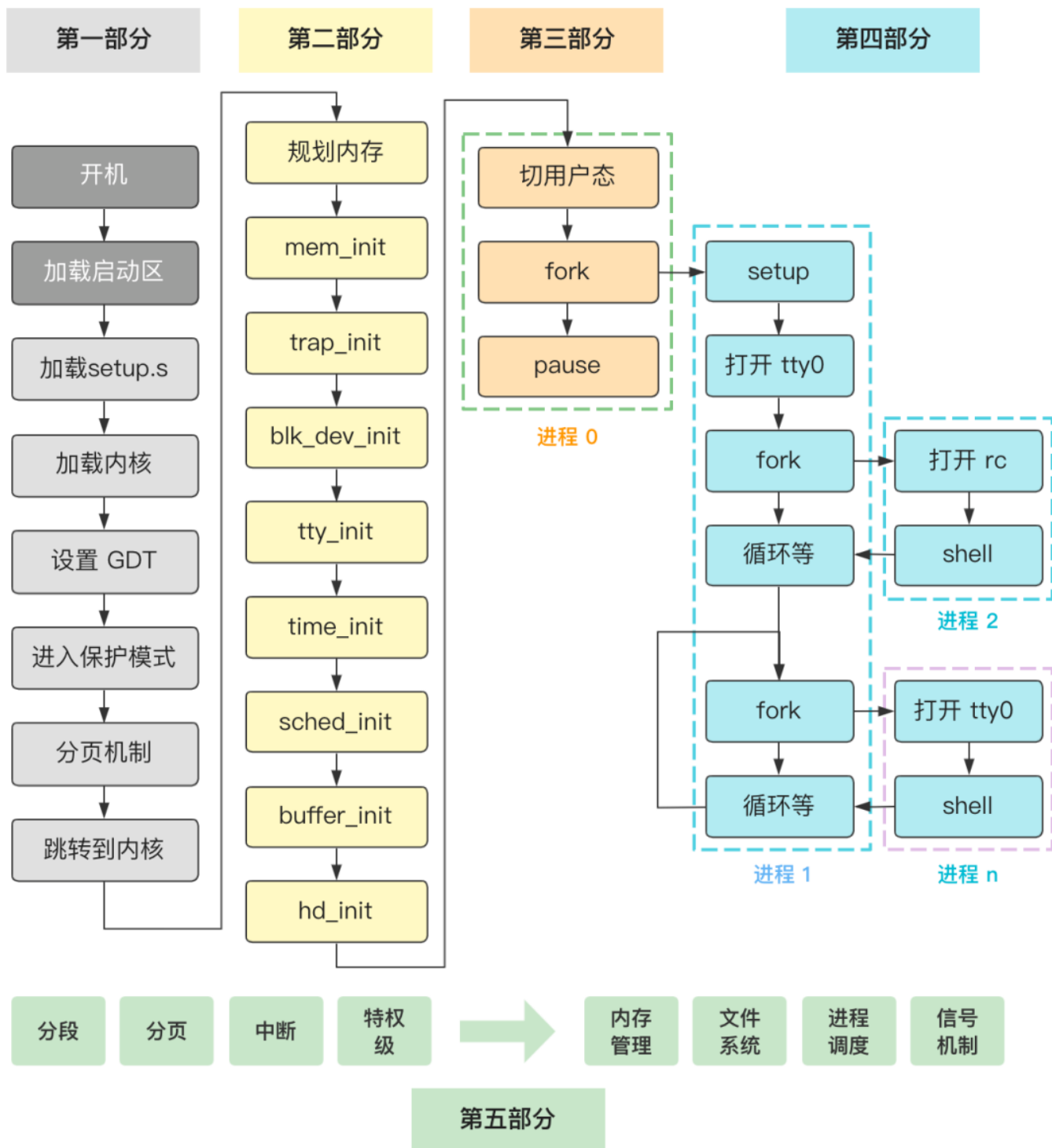
#操作系统源码

43个

新读者看这里，老读者直接跳过。

本系列会以一个读小说的心态，从开机启动后的代码执行顺序，带着大家阅读和赏析 Linux 0.11 全部核心代码，了解操作系统的技术细节和设计思想。

本回的内容属于第四部分。



你会跟着我一起，看着一个操作系统从啥都没有开始，一步一步最终实现它复杂又精巧的设计，读完这个系列后希望你能发出感叹，原来操作系统源码就是这破玩意。

以下是**已发布文章**的列表，详细了解本系列可以先从开篇词看起。

开篇词

**第一部分 进入内核前的苦力活**

第1回 | 最开始的两行代码  
第2回 | 自己给自己挪个地儿  
第3回 | 做好最最基础的准备工作  
第4回 | 把自己在硬盘里的其他部分也放到内存来  
第5回 | 进入保护模式前的最后一次折腾内存  
第6回 | 先解决段寄存器的历史包袱问题  
第7回 | 六行代码就进入了保护模式  
第8回 | 烦死了又要重新设置一遍 idt 和 gdt  
第9回 | Intel 内存管理两板斧：分段与分页  
第10回 | 进入 main 函数前的最后一跃！  
第一部分总结与回顾

## 第二部分 大战前期的初始化工作

第11回 | 整个操作系统就 20 几行代码  
第12回 | 管理内存前先划分出三个边界值  
第13回 | 主内存初始化 mem\_init  
第14回 | 中断初始化 trap\_init  
第15回 | 块设备请求项初始化 blk\_dev\_init  
第16回 | 控制台初始化 tty\_init  
第17回 | 时间初始化 time\_init  
第18回 | 进程调度初始化 sched\_init  
第19回 | 缓冲区初始化 buffer\_init  
第20回 | 硬盘初始化 hd\_init  
第二部分总结与回顾

## 第三部分：一个新进程的诞生

第21回 | 新进程诞生全局概述  
第22回 | 从内核态切换到用户态  
第23回 | 如果让你来设计进程调度  
第24回 | 从一次定时器滴答来看进程调度  
第25回 | 通过 fork 看一次系统调用  
第26回 | fork 中进程基本信息的复制  
第27回 | 透过 fork 来看进程的内存规划  
第三部分总结与回顾

第28回 | 番外篇 - 我居然会认为权威书籍写错了...  
第29回 | 番外篇 - 让我们一起来写本书？  
第30回 | 番外篇 - 写时复制就这么几行代码

## 第四部分：shell 程序的到来

第31回 | 拿到硬盘信息  
第32回 | 加载根文件系统  
第33回 | 打开终端设备文件（本文）

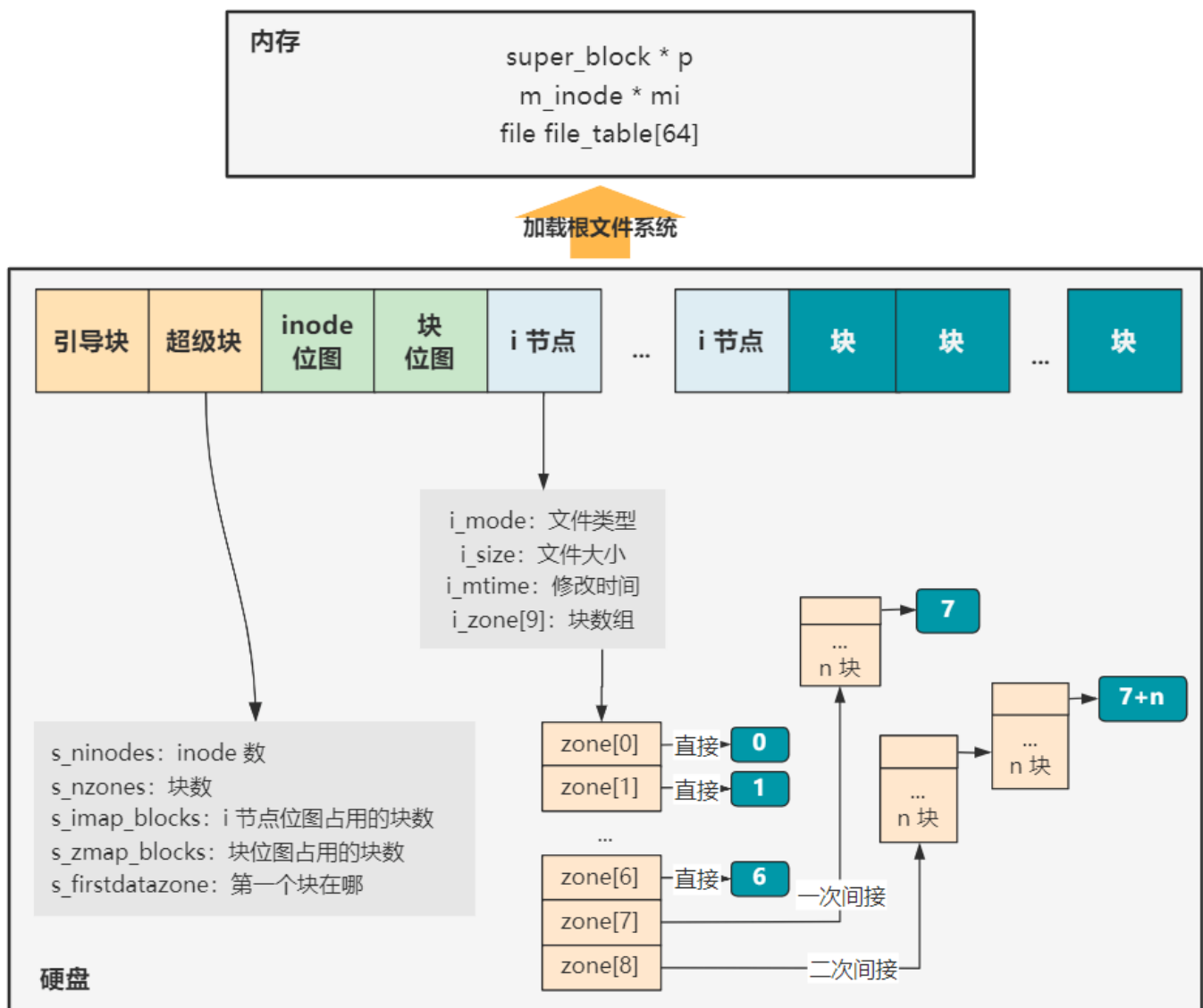
-----

本系列的 GitHub 地址如下，希望给个 star 以示鼓励（文末阅读原文可直接跳转）

<https://github.com/sunym1993/flash-linux0.11-talk>

----- 正文开始 -----

书接上回，上回书咱们说到，`setup` 函数的一番折腾，加载了根文件系统，顺着根 `inode` 可以找到所有文件，为后续工作奠定了基础。



而有了这个功能后，下一行 `open` 函数可以通过文件路径，从硬盘中把一个文件的信息方便地拿到。

```
void init(void) {  
    setup((void *) &drive_info);  
    (void) open("/dev/tty0", O_RDWR, 0);  
    (void) dup(0);  
    (void) dup(0);  
}
```

那我们接下来的焦点就在这个 `open` 函数，以及它要打开的文件 `/dev/tty0`，还有后面的两个 `dup`。

`open` 函数会触发 `0x80` 中断，最终调用到 `sys_open` 这个系统调用函数，相信你已经很熟悉了。

open.c

```
struct file file_table[64] = {0};

int sys_open(const char * filename,int flag,int mode) {
    struct m_inode * inode;
    struct file * f;
    int i,fd;
    mode &= 0777 & ~current->umask;

    for(fd=0 ; fd<20; fd++)
        if (!current->filp[fd])
            break;
    if (fd>=20)
        return -EINVAL;
    current->close_on_exec &= ~(1<<fd);

    f=0+file_table;
    for (i=0 ; i<64 ; i++,f++)
        if (!f->f_count) break;
    if (i>=64)
        return -EINVAL;

    (current->filp[fd]=f)->f_count++;

    i = open_namei(filename,flag,mode,&inode);

    if (S_ISCHR(inode->i_mode))
        if (MAJOR(inode->i_zone[0])==4) {
            if (current->leader && current->tty<0) {
                current->tty = MINOR(inode->i_zone[0]);
                tty_table[current->tty].pgrp = current->pgrp;
            }
        } else if (MAJOR(inode->i_zone[0])==5)
            if (current->tty<0) {
                iput(inode);
                current->filp[fd]=NULL;
                f->f_count=0;
                return -EPERM;
            }
}
```

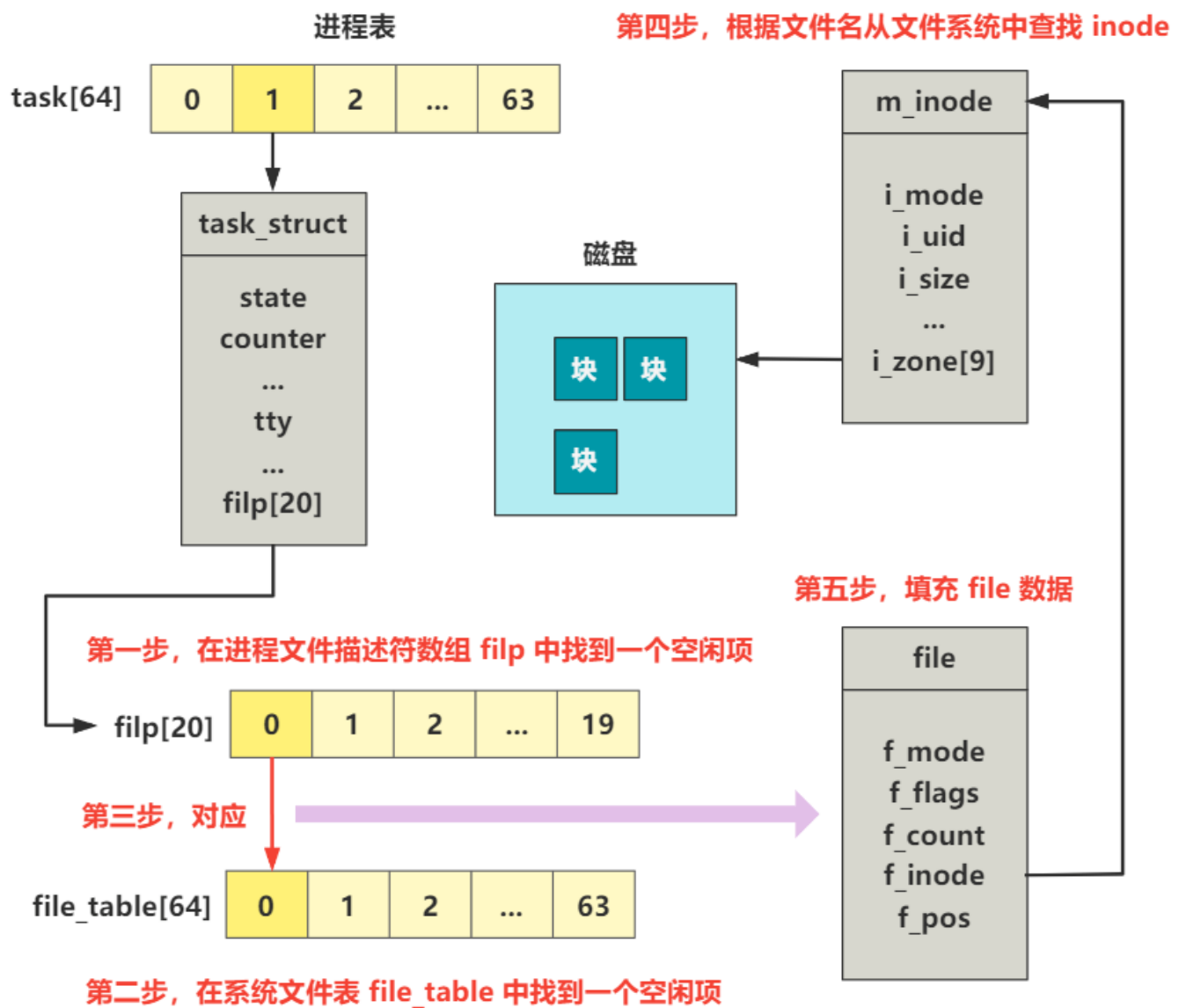
```

if (S_ISBLK(inode->i_mode))
    check_disk_change(inode->i_zone[0]);

f->f_mode = inode->i_mode;
f->f_flags = flag;
f->f_count = 1;
f->f_inode = inode;
f->f_pos = 0;
return (fd);
}

```

这么大一坨别怕，我们慢慢来分析，我先用一张图来描述这一大坨代码的作用。



第一步，在进程文件描述符数组 `filp` 中找到一个空闲项。还记得进程的 `task_struct` 结构

吧，其中有一个 `filp` 数组的字段，就是我们常说的文件描述符数组，这里先找到一个空闲项，将空闲地方的索引值即为 `fd`。

```
int sys_open(const char * filename,int flag,int mode) {  
    ...  
    for(int fd=0 ; fd<20; fd++)  
        if (!current->filp[fd])  
            break;  
    if (fd>=20)  
        return -EINVAL;  
    ...  
}
```

由于此时当前进程，也就是进程 1，还没有打开过任何文件，所以 0 号索引处就是空闲的，`fd` 自然就等于 0。

**第二步，在系统文件表 `file_table` 中找到一个空闲项。**一样的玩法。

```
int sys_open(const char * filename,int flag,int mode) {  
    int i;  
    ...  
    struct file * f=0+file_table;  
    for (i=0 ; i<64; i++,f++)  
        if (!f->f_count) break;  
    if (i>=64)  
        return -EINVAL;  
    ...  
}
```

注意到，进程的 `filp` 数组大小是 20，系统的 `file_table` 大小是 64，可以得出，每个进程最多打开 20 个文件，整个系统最多打开 64 个文件。

**第三步，将进程的文件描述符数组项和系统的文件表项，对应起来。**代码中就是一个赋值操作。



```
int sys_open(const char * filename,int flag,int mode) {  
    ...  
    current->filp[fd] = f;  
    ...  
}
```

**第四步，根据文件名从文件系统中找到这个文件。**其实相当于找到了这个 `tty0` 文件对应的 `inode` 信息。

```
int sys_open(const char * filename,int flag,int mode) {  
    ...  
    // filename = "/dev/tty0"  
    // flag = O_RDWR 读写  
    // 不是创建新文件，所以 mode 没用  
    // inode 是返回参数  
    open_namei(filename,flag,mode,&inode);  
    ...  
}
```

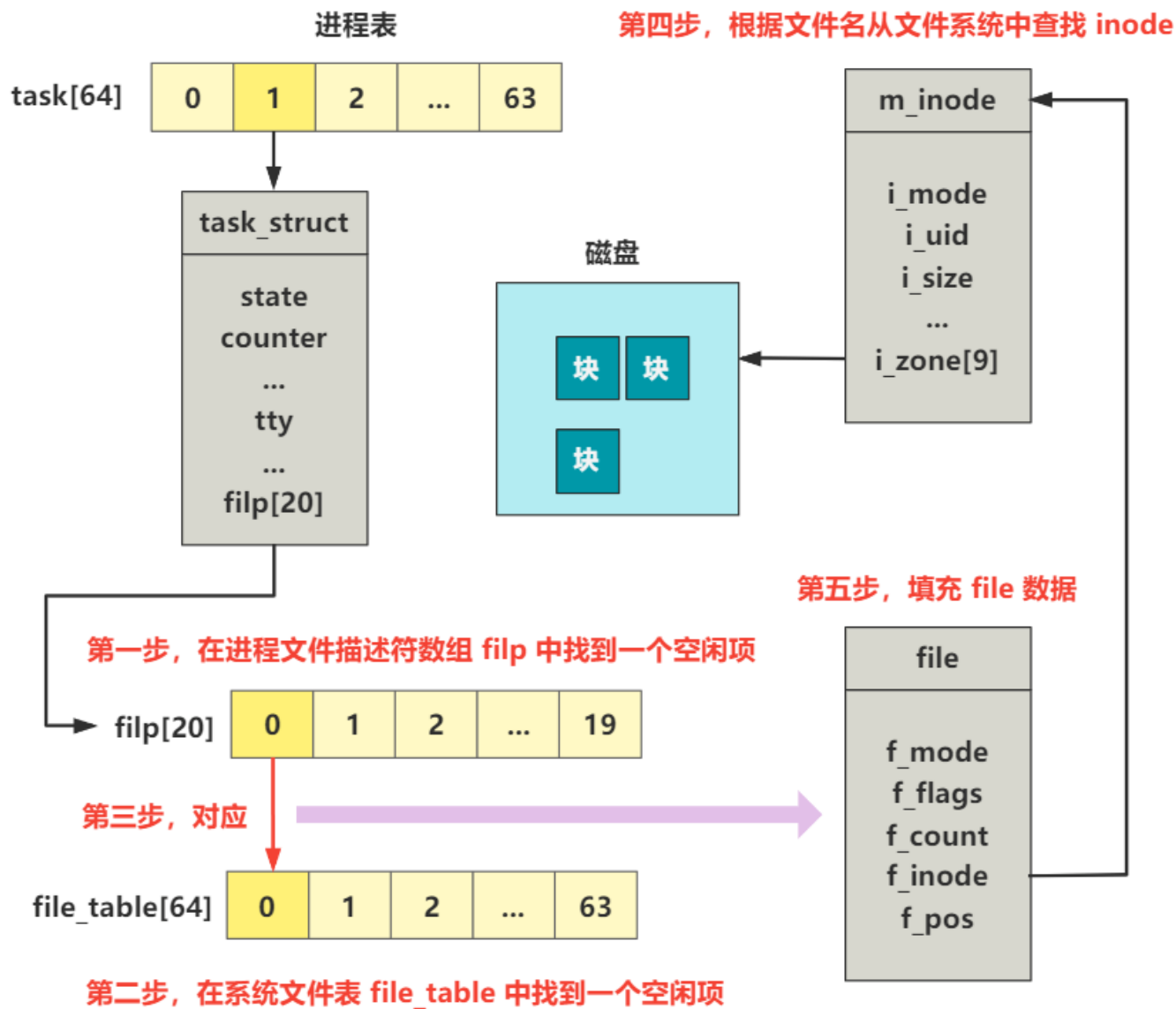
接下来判断 `tty0` 这个 `inode` 是否是字符设备，如果是字符设备文件，那么如果设备号是 4 的话，则设置当前进程的 `tty` 号为该 `inode` 的子设备号。并设置当前进程 `tty` 对应的 `tty` 表项的父进程组号等于进程的父进程组号。

这里我们暂不展开讲。

**最后第五步，填充 `file` 数据。**其实就是初始化这个 `f`，包括刚刚找到的 `inode` 值。最后返回给上层文件描述符 `fd` 的值，也就是零。

```
int sys_open(const char * filename,int flag,int mode) {  
    ...  
    f->f_mode = inode->i_mode;  
    f->f_flags = flag;  
    f->f_count = 1;  
    f->f_inode = inode;  
    f->f_pos = 0;  
    return (fd);  
    ...  
}
```

最后再回过头看这张图，是不是就有感觉了？



其实打开一个文件，即刚刚的 open 函数，就是在上述操作后，返回一个 int 型的数值 fd，称作文件描述符。

之后我们就可以对着这个文件描述符进行读写。

之所以可以这么方便，是由于通过这个文件描述符，最终能够找到其对应文件的 inode 信息，有了这个信息，就能够找到它在磁盘文件中的位置（当然文件还分为常规文件、目录文件、字符设备文件、块设备文件、FIFO 特殊文件等，这个之后再说），进行读写。

比如读函数的系统调用入口。

```
int sys_read(unsigned int fd, char *buf, int count) {  
    ...  
}
```

写函数的系统调用入口。

```
int sys_write(unsigned int fd, char *buf, int count) {  
    ...  
}
```

入参都有个 int 型的文件描述符 fd，就是刚刚 open 时返回的，就这么简单。

好，我们回过头看。

```
void init(void) {  
    setup((void *) &drive_info);  
    (void) open("/dev/tty0", O_RDWR, 0);  
    (void) dup(0);  
    (void) dup(0);  
}
```

上一讲中我们讲了 setup 加载根文件系统的事情。

这一讲中利用之前 setup 加载过的根文件系统，通过 open 函数，根据文件名找到并打开了一个文件。

打开文件，返回给上层的是一个文件描述符，然后操作系统底层进行了一系列精巧的构造，使得一个进程可以通过一个文件描述符 fd，找到对应文件的 inode 信息。

好了，我们接着再往下看两行代码。接下来，两个一模一样的 dup 函数，什么意思呢？

其实，刚刚的 open 函数返回的为 0 号 fd，这个**作为标准输入设备**。

接下来的 dup 为 1 号 fd 赋值，这个作为**标准输出设备**。

再接下来的 dup 为 2 号 fd 赋值，这个作为**标准错误输出设备**。

熟不熟悉？这就是我们 Linux 中常说的 **stdin**、**stdout**、**stderr**。

那这个 `dup` 又是什么原理呢？非常简单，首先仍然是通过系统调用方式，调用到 `sys_dup` 函数。

```
int sys_dup(unsigned int fildes) {
    return dupfd(fildes, 0);
}

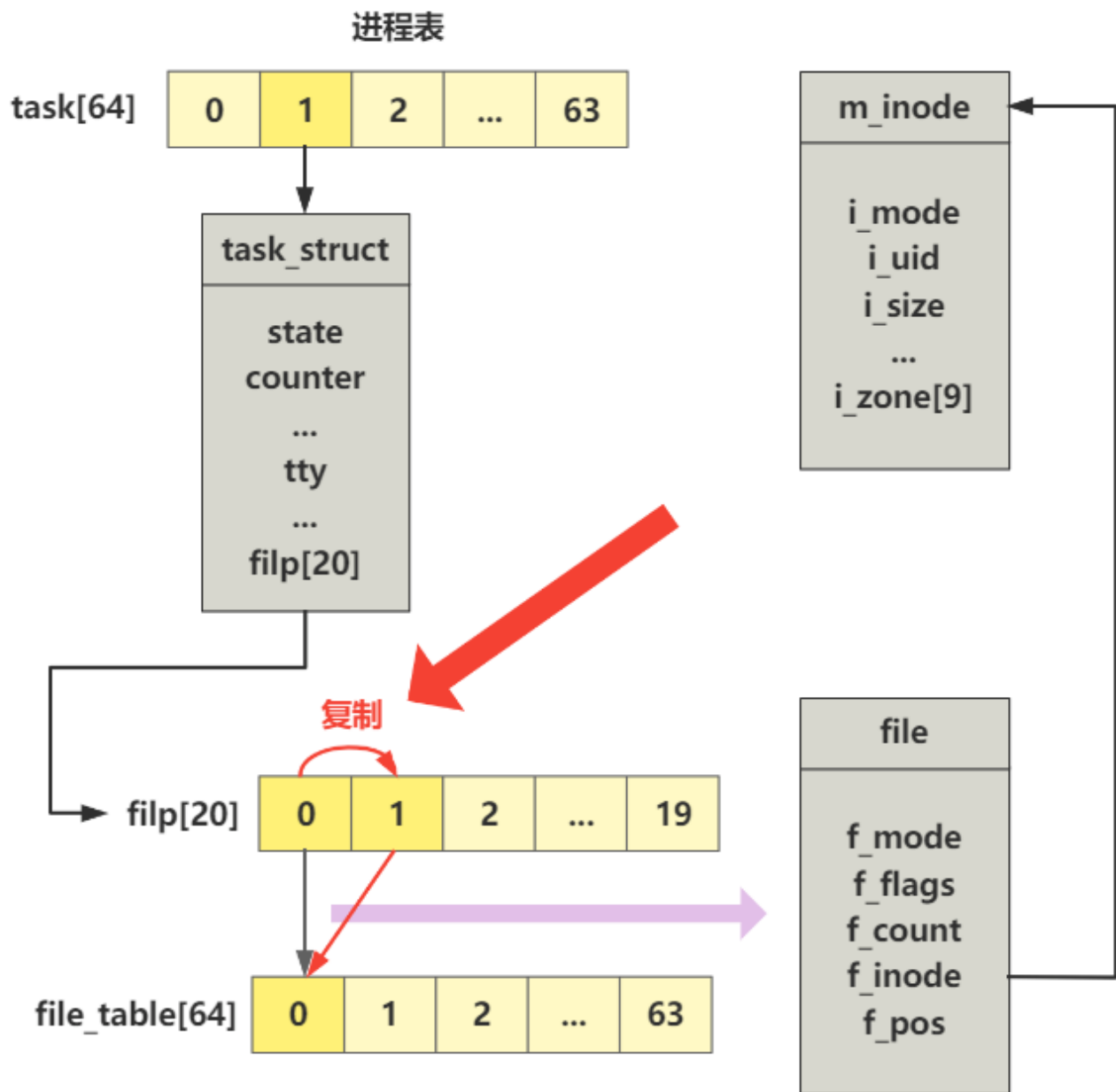
// fd 是要复制的文件描述符
// arg 是指定新文件描述符的最小数值

static int dupfd(unsigned int fd, unsigned int arg) {
    ...
    while (arg < 20)
        if (current->filp[arg])
            arg++;
        else
            break;
    ...
    (current->filp[arg] = current->filp[fd])->f_count++;
    return arg;
}
```

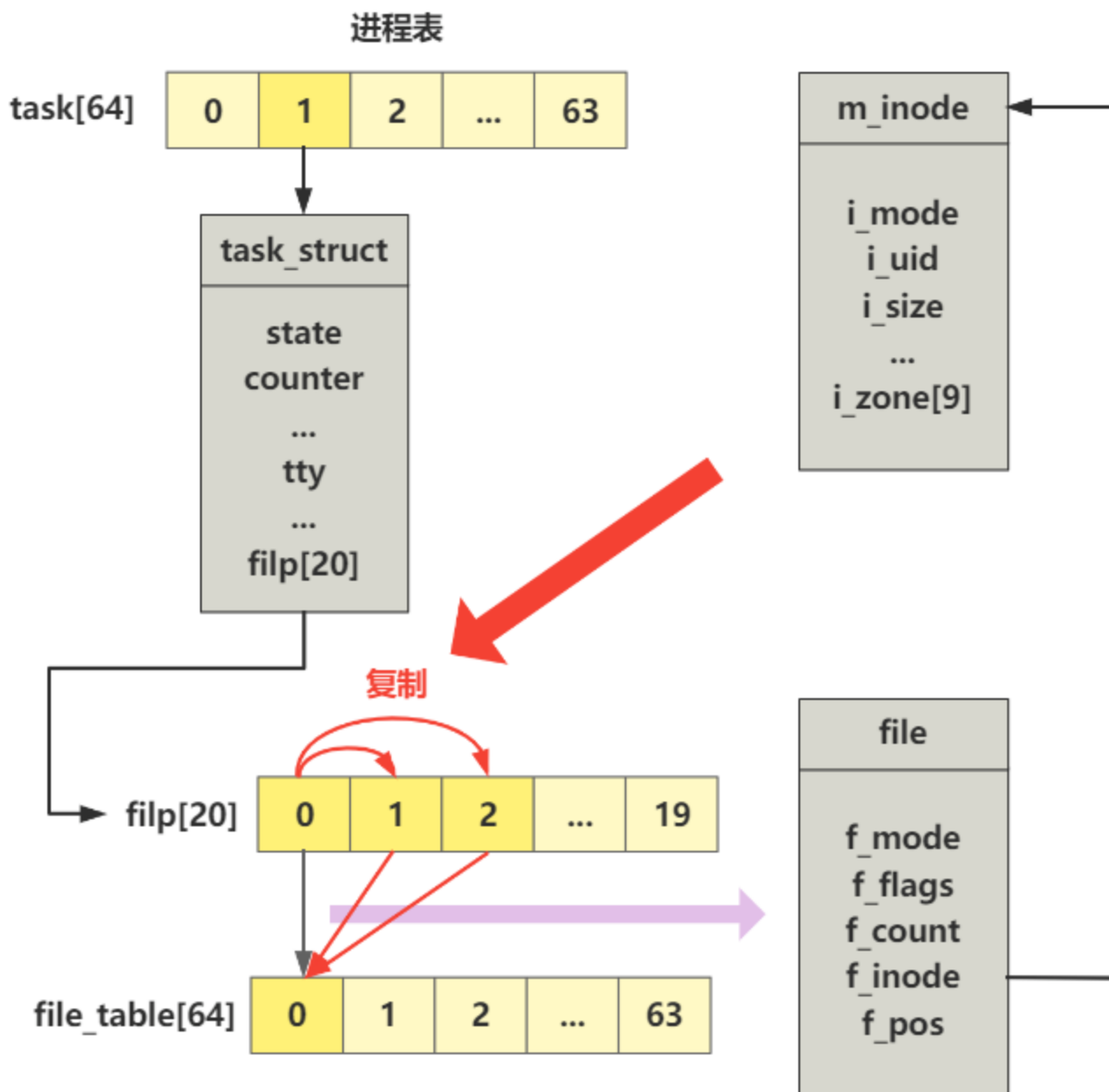
我仍然是把一些错误校验的旁路逻辑去掉了。

那这个函数的逻辑非常单纯，就是从进程的 `filp` 中找到下一个空闲项，然后把要复制的文件描述符 `fd` 的信息，统统复制到这里。

那根据上下文，这一步其实就是把 0 号文件描述符，复制到 1 号文件描述符，那么 0 号和 1 号文件描述符，就统统可以通过一条路子，找到最终 `tty0` 这个设备文件的 `inode` 信息了。



那下一个 `dup` 就自然理解了吧，直接再来一张图。



气不气，消耗了你两次流量，谁让你不懂呢，哈哈哈哈~

ok，进程 1 的 `init` 函数的前四行就讲完了，此时进程 1 已经比进程 0 多了与 **外设交互的能力**，具体说来是 `tty0` 这个外设（也是个文件，因为 Linux 下一切皆文件）交互的能力，这句话怎么理解呢？什么叫多了这个能力？

因为进程 `fork` 出自己子进程的时候，这个 `filp` 数组也会被复制，那么当进程 1 `fork` 出进程 2 时，进程 2 也会拥有这样的映射关系，也可以操作 `tty0` 这个设备，这就是“能力”二字的体现。

而进程 0 是不具备与外设交互的能力的，因为它并没有打开任何的文件，filp 数组也就没有任何作用。

进程 1 刚刚创建的时候，是 fork 的进程 0，所以也不具备这样的能力，而通过 setup 加载根文件系统，open 打开 tty0 设备文件等代码，使得进程 1 具备了与外设交互的能力，同时也使得之后从进程 1 fork 出来的进程 2 也天生拥有和进程 1 同样的与外设交互的能力。

好了，本文就讲到这里，再往后看两行找找感觉，我们就结束。

```
void init(void) {
    setup((void *) &drive_info);
    (void) open("/dev/tty0", O_RDWR, 0);
    (void) dup(0);
    (void) dup(0);
    printf("%d buffers = %d bytes buffer space\n\r", NR_BUFFERS, \
           NR_BUFFERS*BLOCK_SIZE);
    printf("Free mem: %d bytes\n\r", memory_end-main_memory_start);
}
```

接下来的两行是个打印语句，其实就是基于刚刚打开并创建的 0,1,2 三个文件描述符而做出的操作。

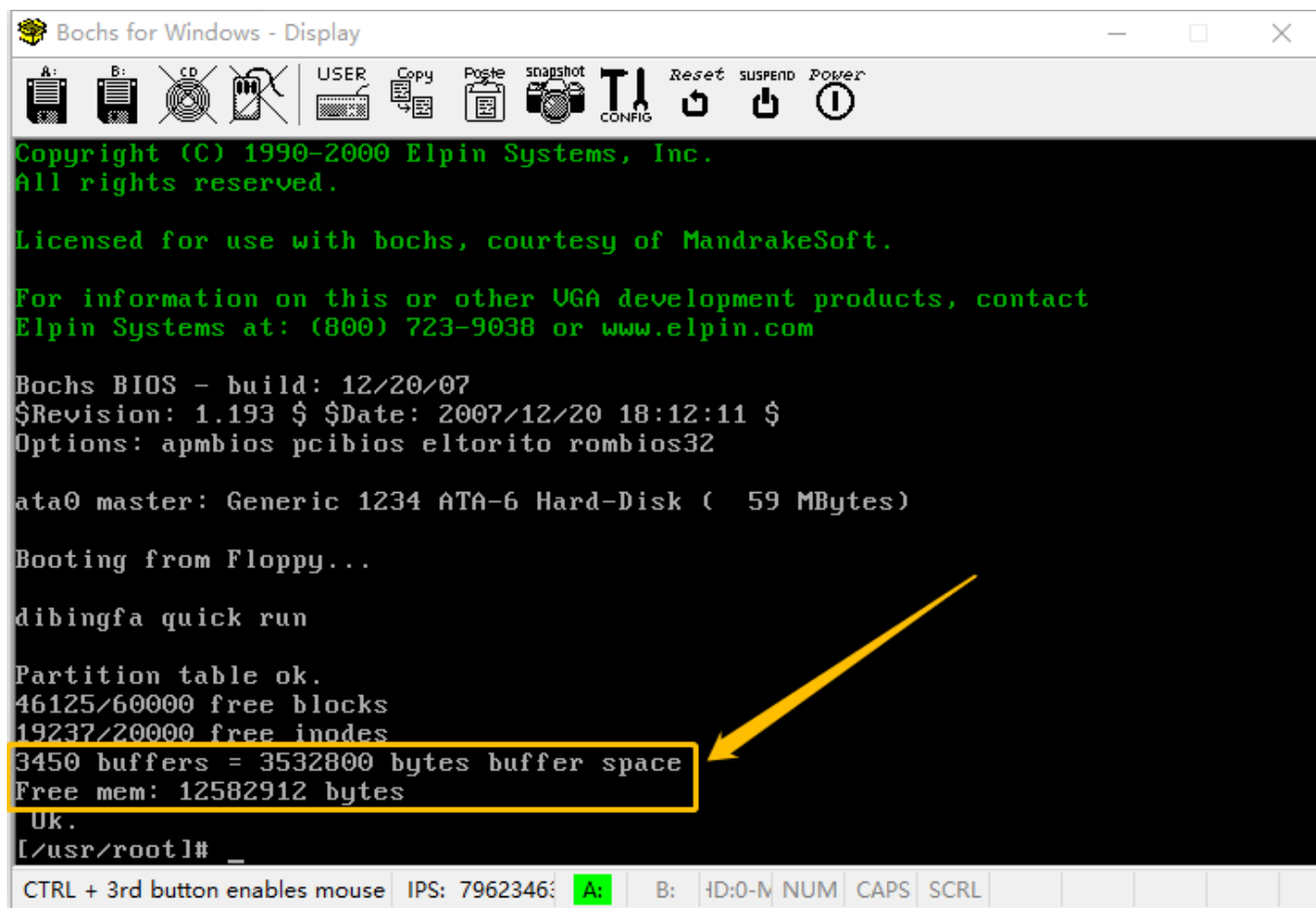
刚刚也说了 1 号文件描述符被当做标准输出，那我们进入 printf 的实现看看有没有用到它。

```
static int printf(const char *fmt, ...) {
    va_list args;
    int i;
    va_start(args, fmt);
    write(1, printbuf, i=vsprintf(printbuf, fmt, args));
    va_end(args);
    return i;
}
```

看，中间有个 write 函数，传入了 1 号文件描述符作为第一个参数。

细节我们先不展开，这里知道它肯定是顺着这个描述符寻找到了相应的 tty0 也就是终端控制台设备，并输出在了屏幕上。我们赶紧看看实际上有没有输出。

仍然是 bochs 启动 Linux 0.11 看效果。



```
Bochs for Windows - Display
A: B: CD Floppy USER Copy Paste Snapshot CONFIG Reset SUSPEND Power
Copyright (C) 1990-2000 Elpin Systems, Inc.
All rights reserved.

Licensed for use with bochs, courtesy of MandrakeSoft.

For information on this or other UGA development products, contact
Elpin Systems at: (800) 723-9038 or www.elpin.com

Bochs BIOS - build: 12/20/07
$Revision: 1.193 $ $Date: 2007/12/20 18:12:11 $
Options: apmbios pcibios eltorito rombios32

ata0 master: Generic 1234 ATA-6 Hard-Disk ( 59 MBytes)

Booting from Floppy...

dibingfa quick run

Partition table ok.
46125/60000 free blocks
19237/20000 free inodes
3450 buffers = 3532800 bytes buffer space
Free mem: 12582912 bytes
Ok.
[usr/root]# _

CTRL + 3rd button enables mouse  IPS: 79623463  A:  B:  HD:0-M NUM CAPS SCRL
```

看到了吧，真的输出了，你偷偷改下这里的源码，再看看这里的输出有没有变化吧！

经过今天的讲解之后，init 函数后面又要 fork 子进程了，也标志着进程 1 的工作基本结束了，准确说是能力建设的工作结束了，接下来就是控制流程和创建新的进程了，可以到开头的全局视角中展望一下。

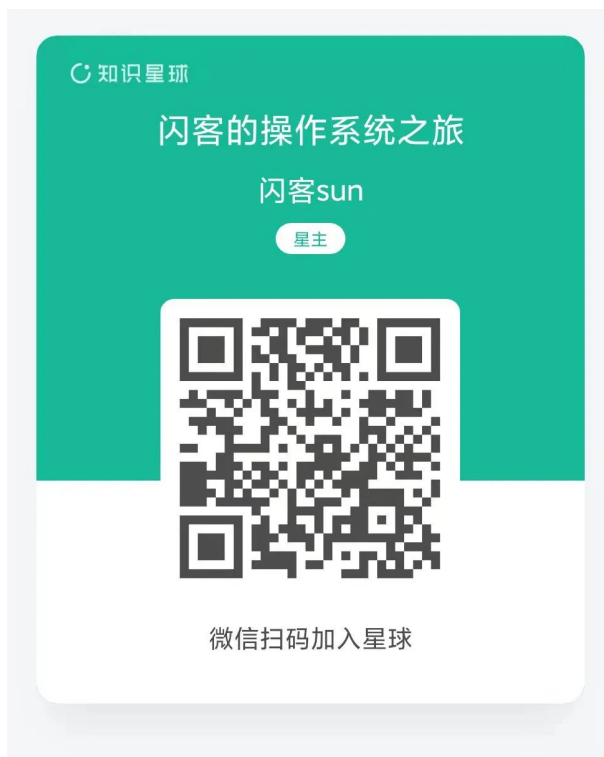
欲知后事如何，且听下回分解。

----- 关于本系列 -----



本系列的开篇词看这，[开篇词](#)

本系列的番外故事看这，[让我们一起来写本书？](#)也可以直接无脑加入星球，共同参与这场旅行。



最后，本系列**完全免费**，希望大家能多多传播给同样喜欢的人，同时给我的 [GitHub](#) 项目点个 star，就在[阅读原文](#)处，这些就足够让我坚持写下去了！我们下回见。



低并发编程

战略上藐视技术，战术上重视技术

175篇原创内容

Official Account

收录于合集 [#操作系统源码](#) 43

[上一篇](#)

[第32回 | 加载根文件系统](#)

[下一篇](#)

[第34回 | 进程2的创建](#)

Modified on 2022-05-17

[Read more](#)

People who liked this content also liked

为什么要旗帜鲜明地反对 orm 和 sql builder

TechPaper



解决前端常见问题：竞态条件

前端巅峰



深入理解 Linux CPU 上下文切换

程序喵大人

