

从 OS 角度漫谈 Hello World

一、前言

`Hello.c` 是一个简单的 C 语言程序，能够实现简单字符串信息的输出

本文从《操作系统》的角度，探讨 `Hello` 从创建进程开始到执行结束过程中，涉及到的进程管理，内存管理，IO，文件系统，缓冲，缺页中断等等，从更为深入细致的角度回顾 `Hello` 的“一生”，主要聚焦于 `020` 的过程，也即将一个可执行文件 `hello` 载入内存并运行的过程中操作系统的运行具体细节。

二、实验内容

本文完成了对 `Linux0.11` 内核启动和初始化过程、`Hello` 从创建进程开始到执行结束过程从理论上进行了较为细致的分析，并从实验角度验证了其中较为核心的部分：

- 修改 `kernel\fork.c` 中的 `find_empty_process` 和 `copy_process` 函数打印进程创建相关信息；
- 修改 `mm\memory.c` 中的 `copy_page_tables` 函数打印页表复制相关信息；
- 修改 `fs\exec.c` 中的 `do_execve` 函数打印读取文件头和内存调整相关信息；
- 修改 `mm\memory.c` 中的 `do_no_page` 和 `put_page` 函数打印缺页中断处理相关信息；
- 在 `mm\memory.c` 中添加 `address_convert` 函数打印线性地址转换为物理地址的转换过程；
- 修改 `kernel\fork.c` 中的 `copy_process`、`kernel\sched.c` 中的 `schedule`、`sys_pause`、`sleep_on`、`interruptible_sleep_on`、`wake_up` 和 `kernel\exit.c` 中的 `do_exit` 函数，打印进程调度相关信息；
- 在 `kernel\chr_drv\tty_io.c` 添加 `press_f12_handle` 函数，修改 `\include\linux\tty.h` 和 `kernel\chr_drv\keyboard.s` 实现对 F12 的识别处理；
- 修改 `kernel\chr_drv\console.c` 中的 `con_write` 函数添加字符 显示处理；
- 编写 `test.c` 实现 `SIGINT` 信号的捕获
- 修改 `kernel\exit.c` 中的 `do_exit` 函数打印进程退出相关内容

具体得到附件如下：

- `fork.log`： 打印进程创建和进程退出任务结构体 `task_struct` 相关信息；
- `fork_page_copy.log`： 页表复制相关信息（页目录项、页表项）；
- `execve.log`： 读取文件头和内存调整后任务结构体 `task_struct` 及 `eip` 等寄存器相关信息；

- `page_fault.log` : 缺页中断处理相关信息和线性地址转换为物理地址的转换过程;
- `process.log` : 进程调度相关信息，包含各进程的执行调度信息，结合 `stat_log.py` 使用;

三、时间之外——内核的初始化

3.1 操作系统的引导启动

当机器上电后，CPU 将自动进入实模式，并从地址 `0xFFFF0` 开始自动执行程序代码，这个地址通常是 ROM-BIOS 中的地址。PC 机的 BIOS 将执行系统的某些硬件检测和诊断功能，并在物理地址 `0` 处开始设置和初始化中断向量。此后，它将磁盘引导扇区（512 字节，以 `0x55AA` 结束）读入内存绝对地址 `0x7C00` 处，并跳转到这个地方开始引导启动机器运行了。

在本节中，操作系统完成了初步的引导启动，实现了实模式到保护模式，再到启用分页机制，最终跳转到 `main` 内核，其整体流程如下：

开机上电 -> 加载启动区 -> 加载 `setup.s` -> 加载内核 -> 分段机制 -> 进入保护模式 -> 中断机制 -> 分页机制 -> 跳转到内核

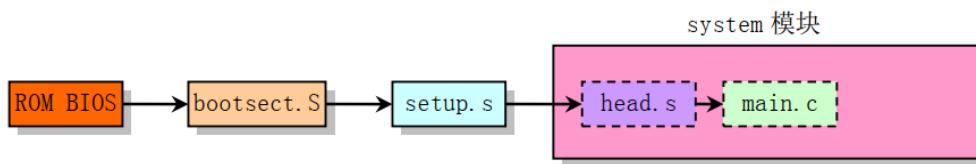


图 1 引导执行顺序

3.1.1 bootsect.s

`bootsect.S` 代码是磁盘引导块程序，由 BIOS 读入内存地址 `0x7C00`，首先移至 `0x90000` 处，设置寄存器 `ds`、`es` 和 `ss`，并将栈顶地址设置为 `0x9FF00`。随后，它利用 ROM BIOS 中断 `INT 0x13` 从磁盘加载 `setup` 模块到 `0x90200`，并在成功加载后显示 "Loading system..."。接着，加载 `setup` 模块后面的 `system` 模块到内存 `0x10000`，最后跳转执行 `setup` 程序。

`bootsect.S` 代码是磁盘引导块程序，由 BIOS 读入到内存绝对地址 `0x7C00` 处，其执行过程大致如下：

- 首先将自己移动到内存绝对地址 `0x90000` 开始处。而后将把 `cs` 寄存器的值分别复制给 `ds`、`es` 和 `ss` 寄存器，然后又把 `0xFF00` 给了 `sp` 寄存器，即
 - 数据段寄存器 `ds` 和代码段寄存器 `cs` 此时都被设置为了 `0x9000`。
 - 栈顶地址被设置为了 `0x9FF00`，具体表现为栈段寄存器 `ss` 为 `0x9000`，栈基址寄存器 `sp` 为 `0xFF00`。栈是向下发展的，不会轻易撞见代码的位置
- 而后利用 ROM BIOS 中断 `INT 0x13` 把从磁盘第 2 个扇区开始的 4 个扇区的 `setup` 模块加载到内存紧接着 `bootsect` 后面位置处（`0x90200`）
 - 在读操作过程中如果读出错，则显示磁盘上出错扇区位置，然后复位驱动器并重试；

- 如果加载成功，则利用 BIOS INT 0x10 在屏幕上显示 "Loading system..." 字符串。

3. 再者把磁盘上 `setup` 模块后面的 `system` 模块（大约 240 个扇区）加载到内存 0x10000 开始的地方。此后还要检查要使用哪个根文件系统设备...

4. 最后长跳转到 `setup` 程序开始处去执行 `setup` 程序。

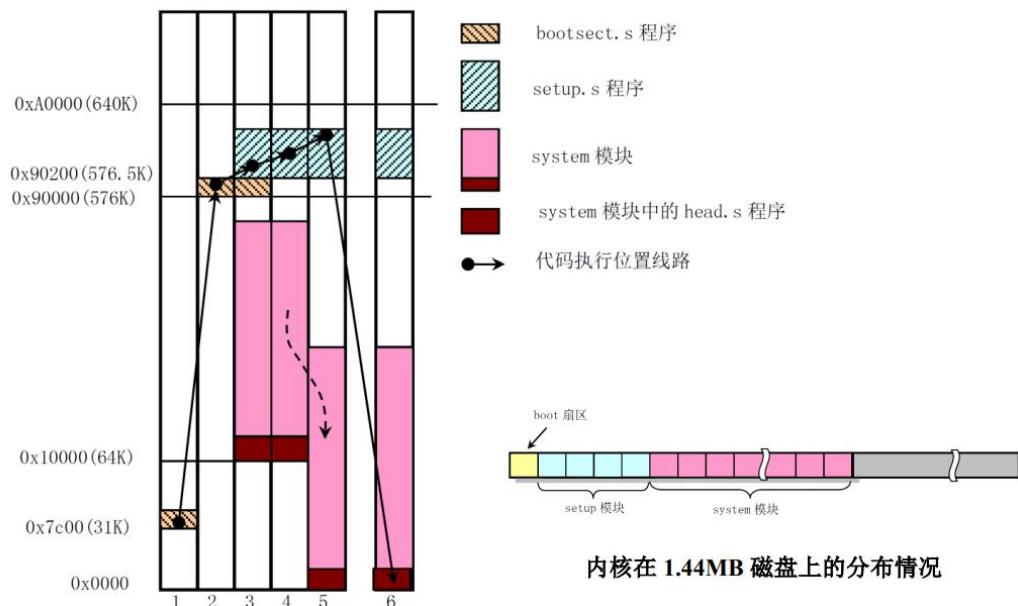


图 2 初始话中各模块在内存中移动过程

3.1.2 setup.s

ROM BIOS 的 INT 0x10 中断读取机器系统数据并保存到 0x90000，覆盖原有的 `bootsect` 程序。接着，关闭中断，将 `system` 模块从 0x10000-0x8ffff 移动到内存地址 0x00000，设置段描述符表。然后，根据临时的 IDT 和 GDT 设置中断描述符表寄存器 (IDTR) 和全局描述符表寄存器 (GDTR)，开启 A20 地址线，重新配置中断控制芯片 8259A 的中断号。最后，设置 CPU 的控制寄存器 CR0 的第 0 位 PE 为 1，进入 32 位保护模式，并跳转到位于 `system` 模块开头的 `head.s` 程序继续运行。

`setup.S` 是一个操作系统加载程序，其主要执行逻辑为：

1. ROM BIOS INT 0x10 中断读取机器系统数据，并将这些数据保存到 0x90000 开始的位置（覆盖掉了 `bootsect` 程序所在的地方）。
2. 关闭中断，将 `system` 模块从 0x10000-0x8ffff 整块向下移动到内存绝对地址 0x00000 处；
3. 根据临时中断描述符表 (IDT) 和全局描述符表 (GDT)，设置中断描述符表寄存器 (IDTR) 和全局描述符表寄存器 (GDTR)，开启 A20 地址线（变成 32 位可用），重新设置两个中断控制芯片 8259A，将硬件中断号重新设置为 0x20 - 0x2f。
4. 最后设置 CPU 的控制寄存器 CR0 的第 0 位 PE 为 1，进入 32 位保护模式运行，并跳转到位于 `system` 模块最前面部分的 `head.s` 程序（内存地址 0）继续运行。

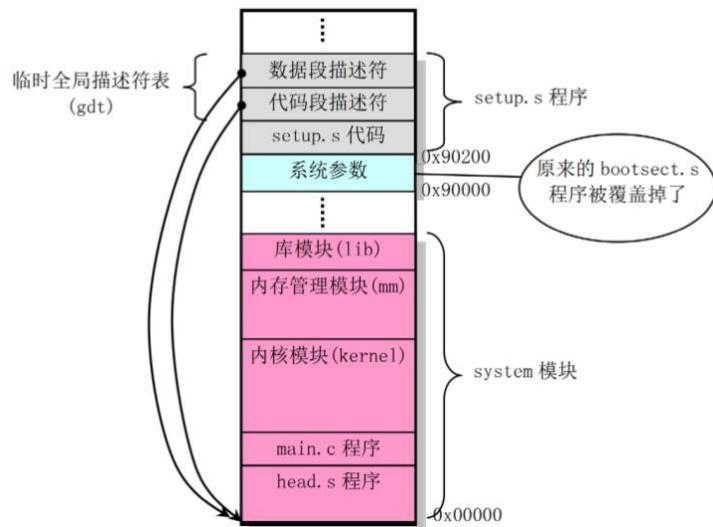


图 3 setup.s 程序结束后内存中程序示意图

3.1.3 head.s

首先，程序加载各个数据段寄存器并重新设置中断描述符表(IDT)，共 256 项，每项指向一个只报告错误的哑中断子程序 `ignore_int`。接着，重新设置全局段描述符表(GDT)，新的 GDT 与之前在 `setup.s` 中设置的 GDT 相似，唯一的区别在于段限长从 8MB 增加到 16MB，主要是因为前面设置的 GDT 在内存 0x902XX 处，计划在内核初始化后用作内存缓存。之后，检测 A20 地址线是否开启，并设置内存的分页处理机制，将控制寄存器 CR0 的第 31 位 PG 设置为 1，将页目录表放在物理地址 0，并在 CR3 寄存器中存储其地址，紧接着放置 4 个页表以寻址 16MB 内存，设置相应的表项。最后，`head.s` 程序通过返回指令弹出预先放置在堆栈中的\init\main.c 程序入口地址，运行 `main` 程序。

`head.s` 程序在被编译生成目标文件后会与内核其他程序的目标文件一起被链接成 `system` 模块，并位于 `system` 模块的最前面开始部分。

- 首先它加载各个数据段寄存器，重新设置中断描述符表 IDT，共 256 项，并使各个表项均指向一个只报错误的哑中断子程序 `ignore_int`。
- 重新设置了全局段描述符表 GDT。实际上新设置的 GDT 表与原来在 `setup.s` 程序中设置的 GDT 表描述符除了在段限长上有些区别以外（原为 8MB，现为 16MB），其他内容基本一致。（这里重新设置 GDT 的主要原因是因为前面设置的 GDT 表处于内存 0x902XX 处。这个地方将在内核初始化后用作内存高速缓冲区的一部分。）
- 接着检测 A20 地址线是否已开启。程序设置管理内存的分页处理机制（设置控制寄存器 CR0 的第 31 位 PG 为 1），将页目录表放在绝对物理地址 0 开始处（存储于 CR3 寄存器），紧随后面会放置共可寻址 16MB 内存的 4 个页表，并分别设置它们的表项。



图 4 CR0 寄存器

4. 最后，`head.s` 程序利用返回指令将预先放置在堆栈中的 `\init\main.c` 程序的入口地址弹出，去运行 `main` 程序。

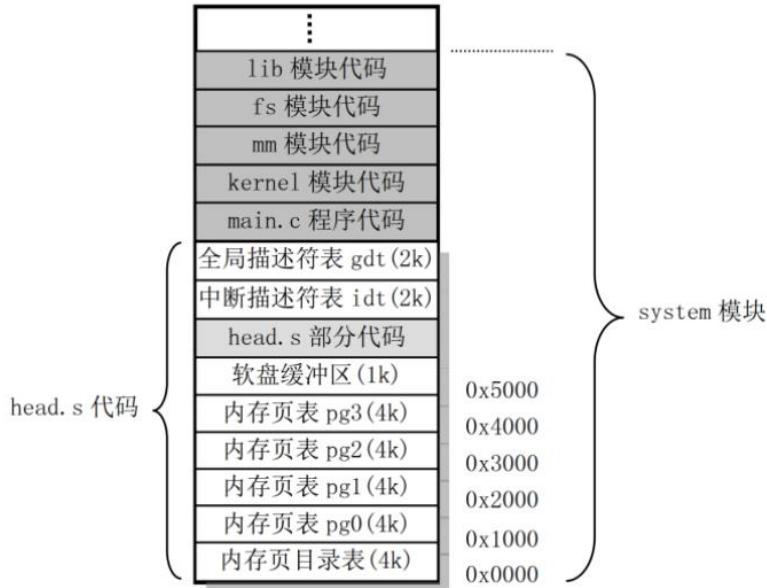


图 5 system 模块在内存中的映像示意图

3.1.4 Linux0.11 的分页机制

Linux0.11 采用两级页表（页目录表和页表），对于一个线性地址，首先将其拆分成高 10 位：中间 10 位：后 12 位。高 10 位负责在页目录表中找到一个页目录项，这个页目录项的值加上中间 10 位拼接后的地址去页表中去寻找一个页表项，这个页表项的值，再加上后 12 位偏移地址，就是最终的物理地址。

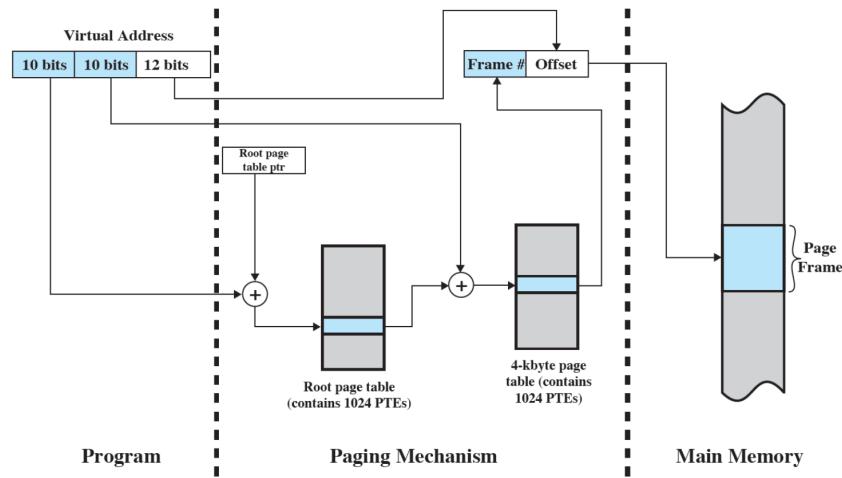


图 6 二级页表

3.2 内核初始化

操作系统内核启动和初始化的过程包括利用 `setup.s` 程序获取的机器参数设置根文件设备号和一些内存全局变量，这些变量指明主内存区的起始地址、系统的内存容量和高速

缓冲区的末端地址。内核随后进行硬件初始化，包括陷阱门、块设备、字符设备和 `tty`，人工设置第一个任务（`task 0`）。完成所有初始化后，程序设置中断允许标志以开启中断并切换到任务 `0` 运行，将执行控制权切换到了用户模式（CPU 从 `0` 特权级切换到了第 `3` 特权级），内核通过任务 `0` 创建一些初始任务，运行 `shell` 程序并显示命令行提示符，使 `Linux` 系统进入正常运行阶段。此外，相关图表展示了内存布局，包括内核程序、高速缓冲、虚拟盘和主内存区。

3.2.1 边界划分

此时中断仍被禁止着，在 `main.c` 中，首先设置根文件系统设备号 `ROOT_DEV` 等变量，接着根据机器物理内存容量设置高速缓冲区和主内存区的位置和范围，即设置变量：

1. 主内存开始地址 -> `main_memory_start`
2. 机器内存容量 -> `memory_end`
3. 高速缓存末端地址 -> `buffer_memory_end`

其实就只是针对不同的内存大小，设置不同的边界值，并且 `main_memory_start` 与 `buffer_memory_end` 实质上取值相等。

但如果在 `Makefile` 文件中定义了内存虚拟盘符号 `RAMDISK`，则初始化虚拟盘。此时主内存将减少。

```

void main(void) /* This really IS void, no error here. */
108 {
109     /* The startup routine assumes (well, ...) this */
110     /*
111      * Interrupts are still disabled. Do necessary setups, then
112      * enable them
113      */
114     ROOT_DEV = ORIG_ROOT_DEV;
115     drive_info = DRIVE_INFO;
116     memory_end = (1 << 20) + (EXT_MEM_K << 10);
117     memory_end &= 0xfffff000;
118     if (memory_end > 16 * 1024 * 1024)
119         memory_end = 16 * 1024 * 1024;
120     if (memory_end > 12 * 1024 * 1024)
121         buffer_memory_end = 4 * 1024 * 1024;
122     else if (memory_end > 6 * 1024 * 1024)
123         buffer_memory_end = 2 * 1024 * 1024;
124     else
125         buffer_memory_end = 1 * 1024 * 1024;
126     main_memory_start = buffer_memory_end;
127 #ifdef RAMDISK
128     main_memory_start += rd_init(main_memory_start, RAMDISK * 1024);
129 #endif

```

图 7 `main.c` 关于边界划分的部分代码

3.2.2 初始化

(1) 内存管理 `mem_init(main_memory_start,memory_end)`

具体主内存区是如何管理和分配，在函数 `mem_init` 内完成（定义在 `mm\memory.c` 中），代码如下：

```

425 void mem_init(long start_mem, long end_mem)
426 {
427     int i;
428
429     HIGH_MEMORY = end_mem;
430     for (i=0 ; i<PAGING_PAGES ; i++)
431         mem_map[i] = USED;
432     i = MAP_NR(start_mem);
433     end_mem -= start_mem;
434     end_mem >>= 12;
435     while (end_mem-->0)
436         mem_map[i++]=0;
437 }
438

```

图 8 `mm\memory.c\mem_init`

该函数把位于 1MB 以上的所有内存划分成一个个页面 (4KB)，并使用一个页面映射字节数组 `mem_map[]` 来管理所有这些页面。

1. 对于 16MB 内存，该数组共有 $(16\text{MB} - 1\text{MB})/4\text{KB} = 3840$ 项，即可管理 3840 个物理页面，而 `start_mem` 通常是 4MB；
2. 缓冲区 (`1MB ~ start_mem`) 直接标记为 USED，产生的效果就是无法再被分配；而剩余的主内存区域，而主内存目前没有任何程序申请，所以初始化时都是零；
3. 每当一个内存页面被占用就把 `mem_map[]` 中对应的字节项增 1；若释放一个页面，就把对应字节值减 1。

- 若字节项为 0，则表示对应页面空闲；
- 若字节值大于或等于 1，则表示页面被占用或被多个进程共享占用。

(2) 陷阱门 (硬件中断向量) `trap_init()`

`trap_init()` 定义在 `kernel\trap.c` 中，用于初始化硬件异常处理中断向量 (陷阱门)，并设置允许 8259A 主芯片中断请求信号的到来。

实质上就是使用 `set_trap_gate()` 与 `set_system_gate()` 函数去将中断向量号 0-16 与具体的中断绑定起来 (在 `head.s` 中全部设定的是默认中断)，而 17-47 的陷阱门均先设置为 `reserved`，以后各硬件初始化时会重新设置自己的陷阱门。

```
void trap_init(void)
{
    int i;

    set_trap_gate(0,&divide_error);
    set_trap_gate(1,&debug);
    set_trap_gate(2,&nmi);
    set_system_gate(3,&int3); /* int3-5 can be called from all */
    set_system_gate(4,&overflow);
    set_system_gate(5,&bounds);
    set_trap_gate(6,&invalid_op);
    set_trap_gate(7,&device_not_available);
    set_trap_gate(8,&double_fault);
    set_trap_gate(9,&coprocessor_segment_overrun);
    set_trap_gate(10,&invalid_TSS);
    set_trap_gate(11,&segment_not_present);
    set_trap_gate(12,&stack_segment);
    set_trap_gate(13,&general_protection);
    set_trap_gate(14,&page_fault);
    set_trap_gate(15,&reserved);
    set_trap_gate(16,&coprocessor_error);
    for (i=17;i<48;i++)
        set_trap_gate(i,&reserved);
    set_trap_gate(45,&irq13);
    outb_p(inb_p(0x21)&0xfb,0x21);
    outb(inb_p(0xA1)&0xdf,0xA1);
    set_trap_gate(39,&parallel_interrupt);
}
```

图 9 kernel\trap.c\trap_init

(3) 块设备初始化 `blk_dev_init()`

`blk_dev_init()` 定义在 `kernel\blk_drv\ll_rw_blk.c` 中，实质上就是给 `request` 这个数组的前 32 个元素的两个变量 `dev` 和 `next` 赋值 -1 和 `NULL`，即将所有请求项置为空闲项。

```
void blk_dev_init(void)
{
    int i;

    for (i=0 ; i<NR_REQUEST ; i++) {
        request[i].dev = -1;
        request[i].next = NULL;
    }
}
```

图 10 kernel\blk_drv\ll_rw_blk.c\blk_dev_init

而 `request` 结构代表了一次读盘请求（定义在 `kernel\blk_drv\blk.h` 中），具体如下：

```
23 struct request {  
24     int dev;          /* -1 if no request */  
25     int cmd;          /* READ or WRITE */  
26     int errors;  
27     unsigned long sector;  
28     unsigned long nr_sectors;  
29     char * buffer;  
30     struct task_struct * waiting;  
31     struct buffer_head * bh;  
32     struct request * next;  
33 };
```

图 11 kernel\blk_drv\blk.h\request

`struct request` 中的字段共同构成了操作系统管理块设备请求的核心结构。`dev` 字段指定设备号，帮助设备驱动程序识别硬件设备，`-1` 表示请求结构体为空闲。`cmd` 字段表示操作类型（读取或写入），用于 I/O 调度以决定请求处理方式。`errors` 记录操作中的错误次数，便于错误处理和调试。`sector` 指定读写操作的起始物理扇区，与文件系统紧密结合以管理数据位置。`nr_sectors` 表示涉及的扇区总数，影响内存管理和 I/O 调度策略。`buffer` 指向存放读写数据的缓冲区，减少磁盘访问频率并在用户空间与内核空间之间传递数据。`waiting` 指向等待操作完成的进程，与进程管理和调度相关，以便有效唤醒相关进程。`bh` 指向缓冲区头信息，跟踪缓冲区状态，确保数据管理和优化。最后，`next` 字段指向链表中的下一个请求项，便于 I/O 调度器依次处理请求，优化处理队列。通过这些字段，操作系统能够高效地协调资源分配和任务处理。

(4) tty 终端初始化 `tty_init()`

字符设备初始化 `chr_dev_init()` 在 `Linux0.11` 中为空 (`kernel\chr_drv\tty_io.c`)，为以后扩展做准备。`tty` 终端初始化函数 `tty_init()`（定义在 `kernel\chr_drv\tty_io.c` 中）初始化所有终端缓冲队列，初始化串口终端和控制台终端。

```
105 void tty_init(void)  
106 {  
107     rs_init();  
108     con_init();  
109 }
```

图 12 kernel\chr_drv\tty_io.c\tty_init

1. `rs_init()`: 这个函数是串口中断的开启，以及设置对应的中断处理程序；
2. `con_init()`: 定义在 `kernel\chr_drv\console.c`，根据显示卡类型和显示内存容量（系统启动初始化时获得的系统信息），设置有关屏幕的一些基本参数值（光标位置等...），用于 `con_write()` 函数（屏幕输出）的操作。

(5) 时钟初始化 `time_init()`

为了让操作系统能自动地准确提供当前时间和日期信息，在初始化时，内核通过 `init\main.c` 程序中的 `time_init()` 函数读取一块 CMOS RAM 芯片中保存的当前时间和日期信息，并通过 `kernel\mktime.c` 程序中的 `kernel_mktime()` 函数转换成从 1970 年 1 月 1 日午夜 0 时开始计起到当前的以秒为单位的时间，我们称之为 UNIX 日历时间。

(6) 调度程序初始化 `sched_init()`

```

404     void sched_init(void)
405     {
406         int i;
407         struct desc_struct * p;
408
409         if (sizeof(struct sigaction) != 16)
410             panic("Struct sigaction MUST be 16 bytes");
411         set_tss_desc(gdt+FIRST_TSS_ENTRY,&(init_task.task.tss));
412         set_ldt_desc(gdt+FIRST_LDT_ENTRY,&(init_task.task.ldt));
413         p = gdt+2*FIRST_TSS_ENTRY;
414         for(i=1;i<NR_TASKS;i++) {
415             task[i] = NULL;
416             p->a=p->b=0;
417             p++;
418             p->a=p->b=0;
419             p++;
420         }
421         /* Clear NT, so that we won't have troubles with that later on */
422         __asm__("pushfl ; andl $0xfffffbfff,(%esp) ; popfl");
423         ltr(0);
424         lldt(0);
425         outb_p(0x36,0x43);      /* binary, mode 3, LSB/MSB, ch 0 */
426         outb_p(LATCH & 0xff , 0x40);    /* LSB */
427         outb(LATCH >> 8 , 0x40);    /* MSB */
428         set_intr_gate(0x20,&timer_interrupt);
429         outb(inb_p(0x21)&~0x01,0x21);
430         set_system_gate(0x80,&system_call);
431     }

```

图 13 kernel\sched.c\sched_init

`sched_init()` 定义在 `kernel\sched.c` 中，主要是为进程调度所需要用到的数据结构做准备：

- 首先初始化了一组 TSS 和 LDT，分别对应保存和恢复进程的上下文的任务状态段和局部描述符表。之后将作为进程 0（也就是现在运行这个的 TSS 和 GDT）

```

52     #define _set_tss_ldt_desc(n,addr,type) \
53     __asm__ ("movw $104,%1\n\t" \
54             "movw %%ax,%2\n\t" \
55             "rrol $16,%eax\n\t" \
56             "movb %%al,%3\n\t" \
57             "movb $" type ",%4\n\t" \
58             "movb $0x00,%5\n\t" \
59             "movb %%ah,%6\n\t" \
60             "rrol $16,%eax" \
61             :::"a" (addr), "m" (*(n)), "m" (*(n+2)), "m" (*(n+4)), \
62             "m" (*(n+5)), "m" (*(n+6)), "m" (*(n+7)) \
63             )

```

图 14 kernel\asm\system.h\set_tss_desc

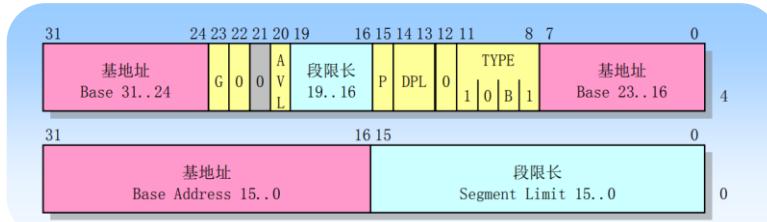


图 15 任务状态段描述符

- 随后清空任务数组 `task`（数据类型为 `task_struct`）和 GDT 描述符表项（均定义在 `include\linux\head.h` 中），`task_struct` 结构就是代表每一个进程的信息（从 1 位置开始，初始任务的描述符仍然保留，以后每创建一个新进程，就会在后面添加一组 TSS 和 LDT 表示这个进程的任务状态段以及局部描述符表信息）

```
struct tss_struct {
    long back_link; /* 16 high bits zero */
55    long esp0;
56    long ss0; /* 16 high bits zero */
57    long esp1;
58    long ss1; /* 16 high bits zero */
59    long esp2;
60    long ss2; /* 16 high bits zero */
61    long cr3;
62    long eip;
63    long eflags;
64    long eax,ecx,edx,ebx;
65    long esp;
66    long ebp;
67    long esi;
68    long edi;
69    long es; /* 16 high bits zero */
70    long cs; /* 16 high bits zero */
71    long ss; /* 16 high bits zero */
72    long ds; /* 16 high bits zero */
73    long fs; /* 16 high bits zero */
74    long gs; /* 16 high bits zero */
75    long ldt; /* 16 high bits zero */
76    long trace_bitmap; /* bits: trace 0, bitmap 16-31 */
77    struct i387_struct i387;
78};

struct task_struct {
    /* these are hardcoded - don't touch */
79    long state; /* -1 unrunnable, 0 runnable, >0 stopped */
80    long counter;
81    long priority;
82    long signal;
83    struct sigaction sigaction[32];
84    long blocked; /* bitmap of masked signals */
85    /* various fields */
86    int exit_code;
87    unsigned long start_code,end_code,end_data,brk,start_stack;
88    long pid,father,pgrp,session,leader;
89    unsigned short uid,euid,suid;
90    unsigned short gid,egid,sgid;
91    long alarm;
92    long utime,stime,cutime,cstime,start_time;
93    unsigned short used_math;
94    /* file system info */
95    int tty; /* -1 if no tty, so it must be signed */
96    unsigned short umask;
97    struct m_inode *pwd;
98    struct m_inode *root;
99    struct m_inode *executable;
100   unsigned long close_on_exec;
101   struct file *filp[NR_OPEN];
102   /* ldt for this task 0 - zero 1 - cs 2 - ds&ss */
103   struct desc_struct ldt[3];
104   /* tss for this task */
105   struct tss_struct tss;
106};

};
```

图 16、17 kernel\linux\sched.h 中 tss_struct 和 task_struct

3. 清除 EFLAGS 标志寄存器中的 NT 位，避免错误的嵌套调用返回。

```
_asm_("pushfl ; andl $0xffffbfff,(%esp) ; popfl");
```

4. 将任务 0 的 TSS 段选择符加载到任务寄存器 tr。将局部描述符表段选择符加载到局部描述符表寄存器 ldtr 中。`ltr(0);lldt(0);`

5. 修改中断控制器屏蔽码，允许时钟中断并添加系统调用，一个作为进程调度的起点，一个作为用户程序调用操作系统功能的桥梁。

```
set_intr_gate(0x20,&timer_interrupt);outb(inb_p(0x21)&~0x01,0x21);set_system_gate(0x80,&system_call);
```

```
#define _set_gate(gate_addr,type,dpl,addr) \
23  __asm__ ("movw %%dx,%ax\n\t" \
24    "movw %0,%dx\n\t" \
25    "movl %%eax,%1\n\t" \
26    "movl %%edx,%2" \
27    : \
28    : "i" ((short) (0x8000+(dpl<<13)+(type<<8))), \
29    "o" (*((char *) (gate_addr))), \
30    "o" (*(_4+(char *) (gate_addr))), \
31    "d" ((char *) (addr)), "a" (0x00000000)) \
32 \
33 #define set_intr_gate(n,addr) \
34  _set_gate(&idt[n],14,0,addr) \
35 \
36 #define set_trap_gate(n,addr) \
37  _set_gate(&idt[n],15,0,addr) \
38 \
39 #define set_system_gate(n,addr) \
40  _set_gate(&idt[n],15,3,addr)
```

图 18 kernel\asm\system.h\set_gate

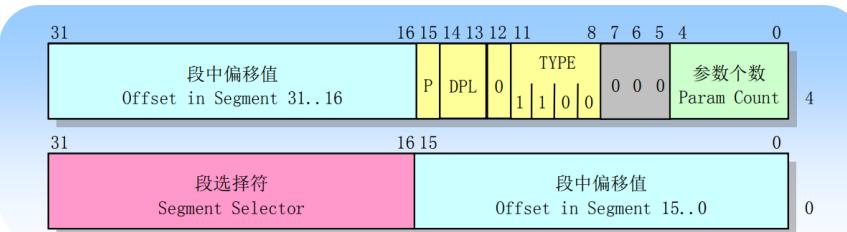


图 19 中断门描述符

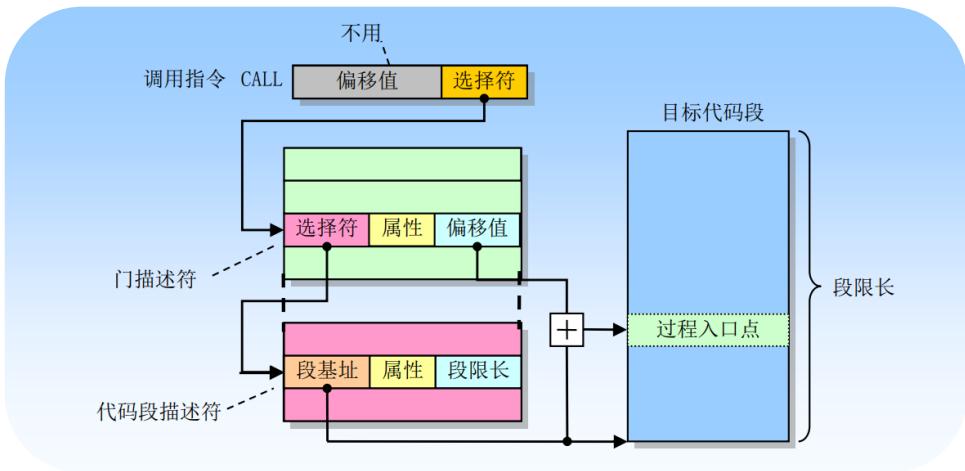


图 20 门调用操作过程

(7) 缓冲管理初始化 buffer_init(buffer_memory_end)

`buffer_init(buffer_memory_end)` 函数（`fs\buffer.c` 中定义）从缓冲区开始位置 `start_buffer` 处和缓冲区末端 `buffer_end` 处分别同时设置（初始化）缓冲块头结构和对应的数据块，直到缓冲区中所有内存被分配完毕。

```

void buffer_init(long buffer_end)
{
    struct buffer_head * h = start_buffer;
    void * b;
    int i;

    if (buffer_end == 1<<20)
        b = (void *) (640*1024);
    else
        b = (void *) buffer_end;
    while ((b -= BLOCK_SIZE) >= ((void *) (h+1))) {
        h->b_dev = 0;
        h->b_dirt = 0;
        h->b_count = 0;
        h->b_lock = 0;
        h->b_uptodate = 0;
        h->b_wait = NULL;
        h->b_next = NULL;
        h->b_prev = NULL;
        h->b_data = (char *) b;
        h->b_prev_free = h-1;
        h->b_next_free = h+1;
        h++;
        NR_BUFFERS++;
        if (b == (void *) 0x100000)
            b = (void *) 0xA0000;
    }
    h--;
    free_list = start_buffer;
    free_list->b_prev_free = h;
    h->b_next_free = free_list;
    for (i=0;i<NR_HASH;i++)
        hash_table[i]=NULL;
}

```

图 21 `fs\buffer.c\buffer_init`

实质上就是创建了 `buffer_head` 结构的 `start_buffer` 和指针 `buffer_end`。缓冲区结尾的 `start_buffer` 每次循环 `-1024`，也就是一页的值，缓冲区结尾的 `start_buffer` 每次循环 `+1`（一个 `buffer_head` 大小的内存），直到碰一块为止。

同时还引入了 `free_list` 指针作为该链表的头指针，指向空闲块链表中第一个“最为空闲的”缓冲块，即近期最少使用的缓冲块。

而 `start_buffer` 维护的显然是一个双向链表，同时初始化了一个哈希数组便于后

续查找读取的块设备中的数据操作。

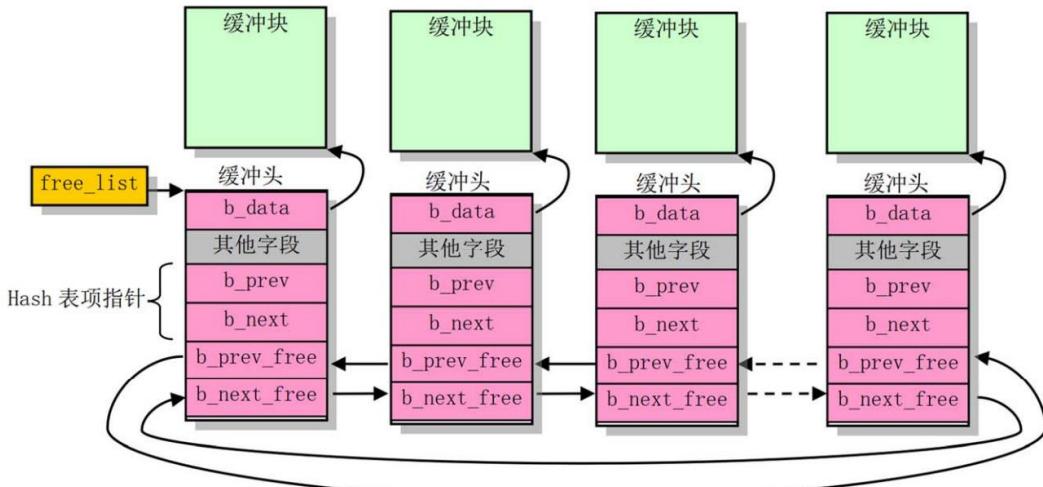


图 22 缓冲块组成的双向循环空闲链表结构

(8) 硬盘和软驱初始化

进入到初始化的最后一步，初始化硬盘和软驱

1. 软驱初始化 `floppy_init()`: 设置软盘块设备请求项的处理函数 `do_fd_request()`, 并设置软盘中断门 (`int 0x26`, 对应硬件中断请求信号 `IRQ6`)。然后取消对该中断信号的屏蔽, 以允许软盘控制器 `FDC` 发送中断请求信号。
2. 硬盘初始化 `hd_init()`: 设置硬盘设备的请求项处理函数指针为 `do_hd_request()`, 并设置硬盘控制器中断描述符 (`int 0x2E`, 对应 `8259A` 芯片的中断请求信号 `IRQ14`), 并复位硬盘控制器中断屏蔽码, 以允许硬盘控制器发送中断请求信号。

实际上就是在相应端口设置并开启一个 `IO` 中断, 用于软\硬盘发送读写请求时, 软\硬盘会发出中断信号给 `CPU`。

3.2.3 分道扬镳

在完成上述繁杂但必要的初始化后, 系统现在终于可以打开中断了 (从 `setup.s` 一直关到现在), 然后从内核态切换到用户态 (这样可以保证之后的进程将一直处于用户态的模式, 除非中断), 然后通过 `fork` 创建出一个新的进程 `1`, 再之后进程 `0` 进入死循环, 直至关机。而只有进程 `1` 会执行 `init()` 函数, 完成如加载根文件系统的任务, 同时这个方法将又会创建出一个新的进程 `2`, 在进程 `2` 里又会加载与用户交互的 `shell` 程序。

```

131     mem_init(main_memory_start, memory_end);
132     trap_init();
133     blk_dev_init();
134     chr_dev_init();
135     tty_init();
136     time_init();
137     sched_init();
138     buffer_init(buffer_memory_end);
139     hd_init();
140     floppy_init();
141     sti();
142     move_to_user_mode();
143     if (!fork())
144     { /* we count on this going ok */
145         init();
146     }

```

图 23 main.c 关于的初始化代码

(1) 特权级切换 move_to_user_mode()

由于处于特权级 0 的代码不能直接把控制权转移到特权级 3 的代码中执行，但中断返回操作是可以的，因此利用中断返回指令 IRET 来启动运行 Linux0.00 的第 1 个任务。

具体的做法是，模拟中断返回时栈的内容，即在堆栈中构筑中断返回指令需要的内容，把返回地址的段选择符设置成任务 0 代码段选择符，其特权级为 3。在返回后从栈中取出对应的段选择符。

```
1 #define move_to_user_mode() \
2     __asm__ ("movl %%esp,%%eax\n\t" \
3             "pushl $0x17\n\t" \
4             "pushl %%eax\n\t" \
5             "pushfl\n\t" \
6             "pushl $0x0f\n\t" \
7             "pushl $1f\n\t" \
8             "iret\n\t" \
9             "1:\tmovl $0x17,%%eax\n\t" \
10            "movw %%ax,%%ds\n\t" \
11            "movw %%ax,%%es\n\t" \
12            "movw %%ax,%%fs\n\t" \
13            "movw %%ax,%%gs" \
14            :::"ax")
```

图 24 kernel\asm\system.h\move_to_user_mode

在 Linux0.11 中也采用的是类似的做法，在代码中模仿 CPU 进行了五次压栈操作，这样在执行 iret 指令时，硬件会按顺序将刚刚压入栈中的数据，分别赋值给 SS、ESP、EFLAGS、CS、EIP 这几个寄存器。而返回的位置恰好是 iret 的下一行，就这样实现了从内核态切换到用户态，而其他基本没有改变。

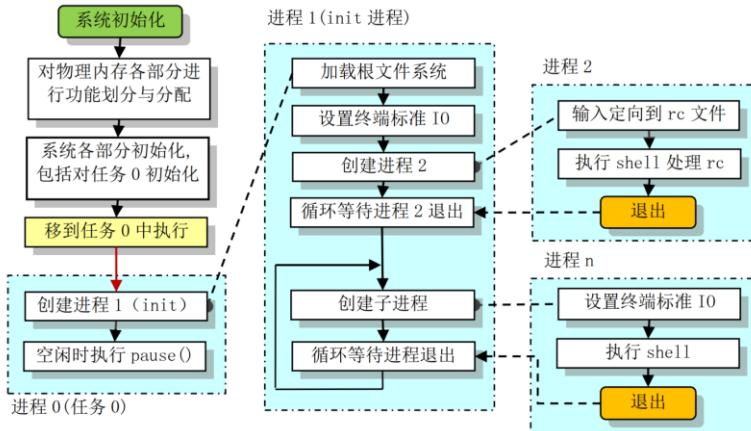


图 25 正式执行

(2) 死循环

在切换至用户模式过后，全系统第一次调用 fork() 建立进程 1。进程 1 调用 init()。

而对于父进程 0，会进入持续执行 pause() 的死循环，pause() 对应的系统调用会把任务 0 转换成可中断等待状态，再执行调度函数。但是调度函数只要发现系统中没有其他任务可运行时就会切换回进程 0，而不依赖于进程 0 的状态。（对于任何其他的进程，pause() 将意味着必须等待收到一个信号才会返回就绪态，但进程 0 是唯一例外情况。因为进程 0 在任何空闲时间里都会被激活）

3.3 shell 程序开始运行

`init()` 函数运行在进程 0 第 1 次创建的子进程 1 中。它首先对第一个将要执行的程序 Shell 的环境进行初始化，然后以登录 Shell 方式加载该程序并执行。

3.3.1 进程 1 的铺垫

在第二次调用 `fork()` 之前，进程 1 首先完成了一些准备工作如下：

```
174 void init(void)
175 {
176     int pid, i;
177
178     setup((void *)&drive_info);
179     (void)open("/dev/tty0", O_RDWR, 0);
180     (void)dup(0);
181     (void)dup(0);
182     (void)open("/var/process.log", O_CREAT | O_TRUNC | O_WRONLY, 0666);
183     (void)open("/var/fork.log", O_CREAT | O_TRUNC | O_WRONLY, 0666);
184     (void)open("/var/fork_page_copy.log", O_CREAT | O_TRUNC | O_WRONLY, 0666);
185     (void)open("/var/execve.log", O_CREAT | O_TRUNC | O_WRONLY, 0666);
186     (void)open("/var/page_fault.log", O_CREAT | O_TRUNC | O_WRONLY, 0666);
187     printf("%d buffers = %d bytes buffer space\n", NR_BUFFERS,
188            NR_BUFFERS * BLOCK_SIZE);
189     printf("Free mem: %d bytes\n", memory_end - main_memory_start);
```

图 26 main.c\init

(1) 硬盘加载 `setup`

`setup` 是个系统调用，会通过中断最终调用到 `sys_setup` 函数（定义在 `kernel\blk_drv\hd.c`）。查看代码其实就是这样：

1. 读取硬盘参数把硬盘的基本信息（`setup.s` 写入）存入了 `hd_info[]`，把硬盘的分区信息存入了 `hd[]`
2. `rd_load()` 加载虚拟盘
3. `mount_root()` 加载根文件系统，从整体上说，它就是要把硬盘中的数据，以文件系统的格式进行解读，加载到内存中设计好的数据结构

那么其实这部分最重要的是加载根文件系统，Linux0.11 采用的是 MINIX 文件系统，其格式如下：

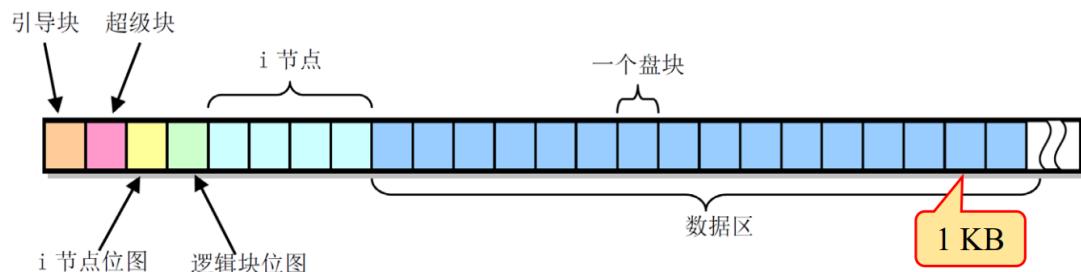


图 27 Minix 文件系统

那么 `mount_root()` 该函数除了用于安装系统的根文件系统以外，还对内核使用文件系统起到初始化的作用。

它对内存中超级块数组 `super_block[]` 进行了初始化，还对文件描述符数组表 `file_table[]` 进行了初始化，并对根文件系统中的空闲盘块数和空闲 `i` 节点数进行了统计并显示出来。具体执行流程如下：

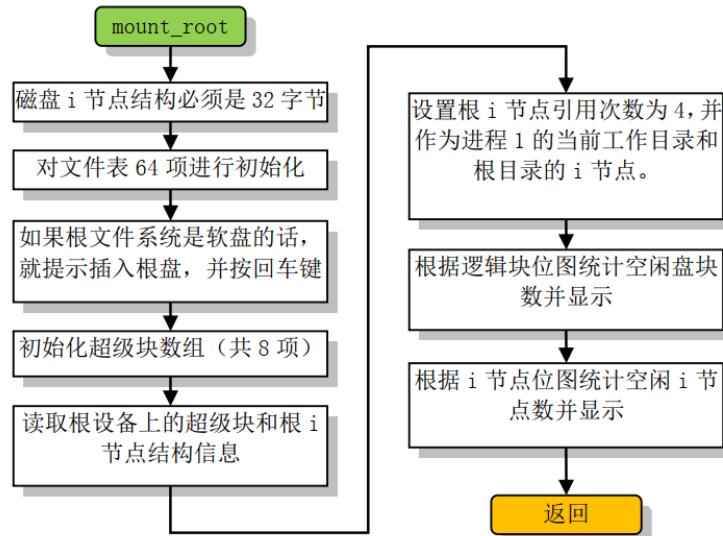


图 28 mount 初始化根文件系统流程

```

24: struct super_block {
125:     unsigned short s_ninodes;
126:     unsigned short s_nzones;
127:     unsigned short s_imap_blocks;
128:     unsigned short s_zmap_blocks;
129:     unsigned short s_firstdatazone;
130:     unsigned short s_log_zone_size;
131:     unsigned long s_max_size;
132:     unsigned short s_magic;
133: /* These are only in memory */
134:     struct buffer_head * s_imap[8];
135:     struct buffer_head * s_zmap[8];
136:     unsigned short s_dev;
137:     struct m_inode * s_isup;
138:     struct m_inode * s_imount;
139:     unsigned long s_time;
140:     struct task_struct * s_wait;
141:     unsigned char s_lock;
142:     unsigned char s_rd_only;
143:     unsigned char s_dirt;
144: } « end super_block » ;

```

图 29 super_block

(2) 文件系统调用 open 和 dup

open 实际上对应 sys_open 这个系统调用函数，用以打开（或创建）文件系统调用。

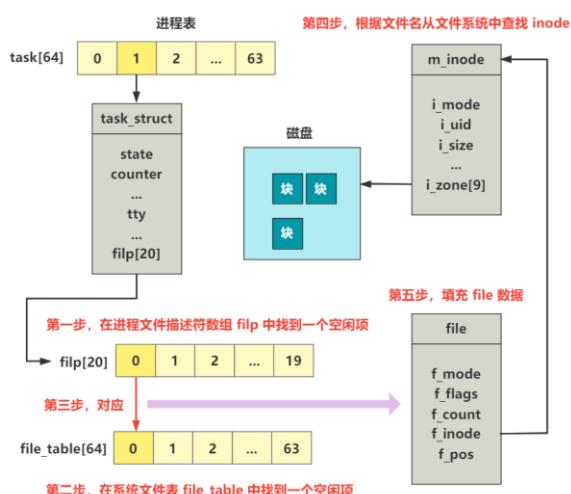


图 30 打开或创建文件的步骤

而 `dup` 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` 的信息复制到这里，在此即文件描述符 `0`、`1` 和 `2` 都对应一个设备文件 `\dev\tty0`。之后就可以通过文件描述符，最终能够找到其对应文件的 `inode` 信息，从而对文件进行读写

3.3.2 Shell 程序走向舞台

在准备好文件描述符后，进程 `1` 可以进行与外设的交互了，接下来就是创建一个新的进程并运行 `Shell` 程序，代码很短

```
190     if (!(pid = fork()))
191     {
192         close(0);
193         if (open("/etc/rc", O_RDONLY, 0))
194             _exit(1);
195         execve("/bin/sh", argv_rc, envp_rc);
196         _exit(2);
197     }
```

图 31 main.c\init 中执行 shell

实际上可以分成这三部分：

1. `fork` 一个新的子进程，此时就是进程 `2` 了。
2. 在进程 `2` 里首先把标准输入 `stdin`（文件描述符 `0`）重定向到 `\etc\rc` 文件。
3. 然后使用 `execve` 替换成 `\bin\sh` 程序并执行。

如果 `execve` 执行成功，则原有进程的代码和数据会被完全替换，而新程序会成为新的进程映像，接着会开始运行。

（注：`fork`、`execve` 以及替换后的缺页中断等在此没有过多纠结细节，但在后续 `Hello` 的运行中也会涉及，在那里结合实际调试再仔细叙述。）

那么进程 `2` 中标准输入为 `\etc\rc` 文件的 `Shell` 进程在读取完 `\etc\rc` 这个文件并执行这个文件里的命令后就会自动退出（可以用来输登录密码什么的）。

而这个 `Shell` 程序的父进程，也就是进程 `1`，它会等待 `Shell` 进程退出，而后再次创建一个子进程，用于运行登录和控制台 `Shell` 程序，只不过这次的文件描述符 `0` 保留为设备文件 `\dev\tty0`（可以与终端 `IO` 交互），如此往复陷入等待-创建 `Shell` 程序的死循环。

于是，`Shell` 程序就成功的运行起来了，在《计算机系统》中我们学过，`Shell` 程序就是个死循环，不断读取（`getcmd`）我们用户输入的命令，创建一个新的进程（`fork`），在新进程里执行（`runcmd`）刚刚读取到的命令，最后等待（`wait`）进程退出，再次进入读取下一条命令的循环中。

四、主角登场——属于 `Hello` 的一生

4.1 键盘输入与读取

让我们再等一下进入 `Hello` 的一生，在 `Shell` 程序过程中，我们需要使用键盘输

入指令才能让操作系统知道用户现在要求执行 Hello 程序了。

那么，用键盘输入一条指令对应操作系统的哪些调用呢？

回忆在内核初始化过程中 `tty_init()` 将键盘中断绑定在了 `keyboard_interrupt` 这个中断处理函数上，这个中断处理程序很复杂，调用也很深，但其流程大致如下：

1. 键盘按下按键后，会引起键盘中断响应（中断请求信号 `IRQ1`，对应中断号 `INT 33`），进入到键盘中断处理程序 `keyboard_interrupt` 里，之后从键盘控制器读入对应的键盘扫描码，根据 `key_table`（定义在 `kernel\chr_drv\keyboard.s` 中）中对应按键的处理函数译成相应字符，最终通过 `put_queue` 函数将字符放入 `read_q` 这个队列。
2. `read_q` 队列里的未处理字符，通过 `do_tty_interrupt` 函数又直接调用行规则函数 `copy_to_cooked` 函数，经过一定的 `termios` 规范处理后，将处理过后的字符 `tty` 辅助队列 `secondary` 中，同时把该字符放入 `tty` 写队列 `write_q` 中

而上层 `shell` 程序经过库函数、文件系统函数等，最终会调用到 `tty_read` 函数，将字符从 `secondary` 队列里取走放入到内存里的某个位置。

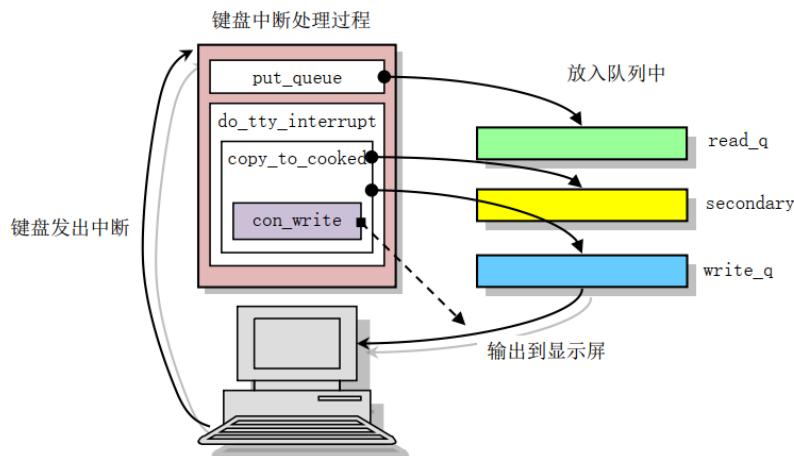


图 32 中断处理过程

在此处因为有两个进程一个读一个写，还涉及到进程调度的问题，简单叙述如下：

- `secondary` 队列为空，即不满足继续读取条件的时候，调用 `sleep_if_empty` 函数让上层读取进程阻塞。
- 再次按下键盘，使得 `secondary` 队列中有字符时，在 `copy_to_cooked` 函数中将字符放入 `secondary` 队列后 `wake_up` 这个队列里的等待进程。

而终端界面的字符显示也是通过 `tty_read` 从 `secondary` 里读字符，然后最终调用写控制台函数 `con_write()` 进行输出（显示）操作。此时如果该终端的回显（`echo`）属性是设置的，则该字符会显示到屏幕上。

4.1.1 Shell 的处理流程

一个简单的 `shell` 程序的处理流程大致可以分为以下几个步骤：

1. 读取用户输入的命令或脚本文件。
2. 将输入字符串切分，分析输入内容，解析命令和参数，将命令行的参数改造为系统调用 `execve()` 内部处理所要求的形式
 - 内置命令，立即执行；
 - 调用 `fork()` 来创建子进程，自身调用 `wait()` 来等待子进程完成，同时在程序执行期间始终接受键盘输入信号，并对输入信号做相应处理。
3. 当子进程运行时，调用 `execve()` 函数，同时根据命令的名字指定的文件到目录中查

找可行性文件，调入内存并执行这个命令。

4. 当子进程完成处理后，向父进程 shell 报告，此时终端进程被唤醒，清除子程序相关在内存中的占用。

4.2 进程创建 fork

在 Linux 系统中，内核通过在 `init\main.c` 中的 `sched_init` 函数中调用 `set_system_gate(0x80, &system_call)`，将 `int 0x80` 中断门绑定到 `system_call` 程序，实现系统调用。用户程序通过嵌入式汇编指令 `syscall0(int, fork)` 触发 `0x80` 中断，从而调用内核中的 `system_call` 函数。内核在 `sys_call_table` 中找到 `sys_fork` 的首地址，并跳转到 `sys_fork` 执行。

在 `sys_fork` 中，首先调用 `find_empty_process` 函数在任务数组 `task[]` 中找到一个空闲位置 `task[nr]`，以存放新进程的任务结构指针 `task_struct`。接着，调用 `copy_process`，该函数负责复制父进程的任务数据结构，使用 `get_free_page` 找到空闲内存页，为新进程申请一页内存。

将父进程的 `task_struct` 信息复制给新进程，使用进程内核栈中压入的参数来设置新进程的相关字段，以保持新进程的状态与父进程在即将进入中断前的状态一致，并将新进程的 `eax` 寄存器设置为 `0`，以确保子进程返回时的返回值为 `0`，表明成功创建了子进程。系统随后在线性地址空间中设置新任务的代码段和数据段描述符的基址和限长，并复制父进程的页目录项和页表项，确保不同进程映射到不同的线性地址空间，但共享相同的物理地址空间。同时，将页表设置为只读，以支持写时复制（`copy-on-write`）。

接下来，增加当前进程打开文件的引用计数，并在 `GDT` 表中设置新任务的 `TSS` 段和 `LDT` 段描述符。最后，将新进程的状态置为 `TASK_RUNNING`，以便将其加入调度队列。

具体代码解释如下：

```
include\unistd.h
...
int fork(void)
{
    #define _syscall0(type,name) \
    type name(void) \
    { \
        long _res; \
        __asm__ volatile ("int $0x80" \
        : "=a" (_res) \
        : "0" (__NR_##name)); \
        if (_res > 0) \
            return (type)_res; \
        errno = -_res; \
        return -1; \
    }
}

init\main.c
...
23: static inline __syscall0(int,fork)
...
124:     main_memory_start += rd_init(main_memory_start, RAMDISK*1024);
125: #endif
126:     mem_init(main_memory_start,memory_end);
127:     trap_init();
128:     blk_dev_init();
129:     chr_dev_init();
130:     tty_init();
131:     time_init();
132:     sched_init();
133:     buffer_init(buffer_memory_end);
134:     hd_init();
135:     floppy_init();
136:     sti();
137:     move_to_user_mode();
138:     if (!fork()) { /* we count on this going ok */
139:         init();
140:     }

```

图 33 用户态使用 `fork`

1. 宏定义：`#define __syscall0(type,name)`：定义一个名为 `_syscall0` 的宏，接收两个参数 `type` 和 `name`。
2. 函数定义：`type name(void)`：根据传入的参数，定义一个函数，其返回类型为 `type`，函数名为 `name`，参数列表为空。

3. 局部变量: `long __res;`: 定义一个长整型变量 `__res`, 用于存储系统调用的返回值。

4. 内联汇编:

- `__asm__ volatile ("int $0x80" ...);`: 使用内联汇编进行系统调用。
- "int \$0x80" 是 Linux 中用于进行系统调用的中断指令。
- : "`=a`" (`__res`): 输出约束, 将 `eax` 寄存器的值 (系统调用的返回值) 存储到 `__res` 中。
- : "`0`" (`__NR_##name`): 输入约束, `__NR_##name` 是系统调用号, 使用 `##` 连接生成的, 该值被存放在 `eax` 寄存器中, 在 `int $0x80` 指令执行时被使用。

5. 返回值处理:

- `if (__res >= 0)`: 检查系统调用是否成功 (返回值大于等于 0)。
- 如果成功, 返回存储在 `__res` 中的值。
- `errno = -__res;`: 如果系统调用失败, 则将 `errno` 设置为负值的返回值, 用于表示错误类型。
- `return -1;`: 函数返回 -1, 表示发生错误。

6. 声明`_syscall0(int, fork)`相当于声明了函数 `int fork()`, 在用户态 C 代码中可以使用户态来实叉 `fork` 来实叉到内核态, 从而实现系统调用。

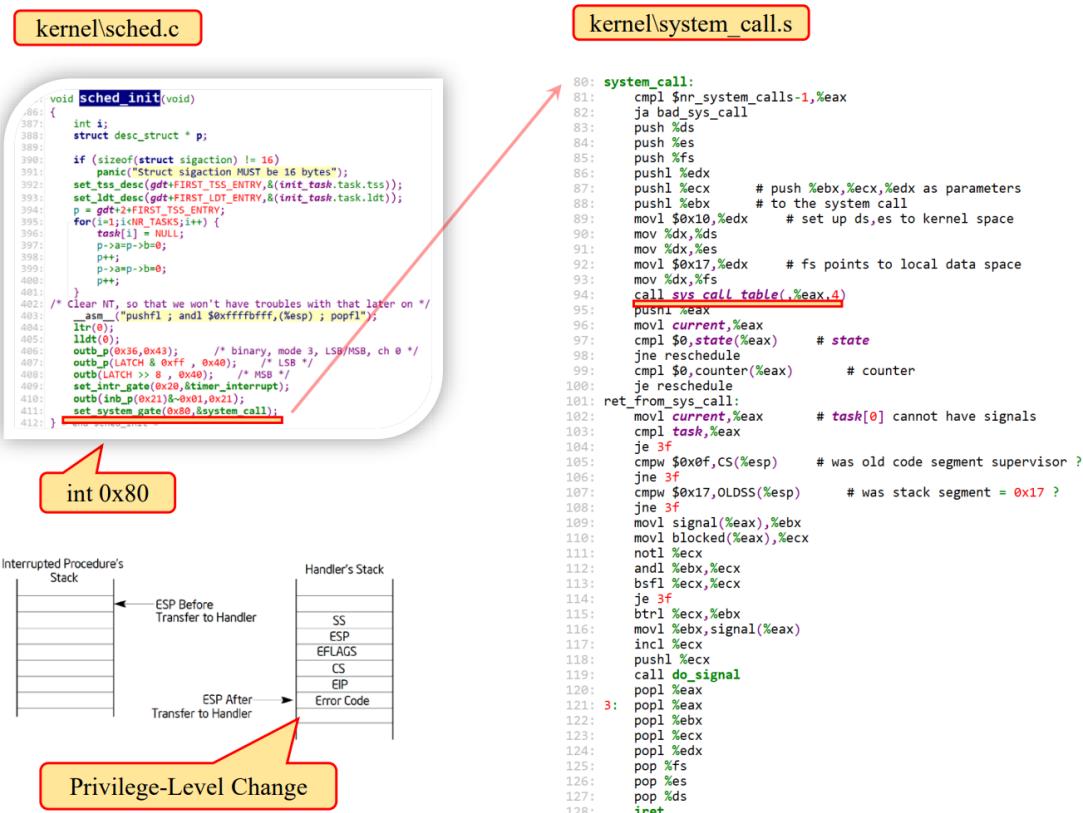


图 34 int80 中断

7. 在 `sched_int` 中创建了 `int 80` 中断门, 中断门选择子指向 `system_call` 代码所在段, 用户使用 `fork` 时将 “`fork`” 值传入 “`eax`” 寄存器中。

8. `system_call` 通过 `sys_call_table` 找到 `sys_fork` 所在段。并跳转。

```

kernel\sys_call.s
int 0x80

80: system_call:
81:    cmpl $nr_system_calls-1,%eax
82:    ja bad_sys_call
83:    push %ds
84:    push %es
85:    push %fs
86:    pushl %edx
87:    pushl %ecx    # push %ebx,%ecx,%edx as parameters
88:    pushl %eax    # to the system call
89:    movl $0x10,%edx    # set up ds,es to kernel space
90:    mov %dx,%ds
91:    mov %dx,%es
92:    movl $0x17,%edx    # fs points to local data space
93:    mov %dx,%fs
94:    call sys_call_table(%eax,4)
95:    pushl %eax
96:    movl current,%eax
97:    cmpl $0,state(%eax)    # state
98:    jne reschedule
99:    cmpl $0,counter(%eax)    # counter
100:   je reschedule
101:  ret_from_sys_call:
102:  movl current,%eax    # task[0] cannot have signals
103:  cmpl task,%eax
104:  je 3f
105:  cmpw $0x0f,CS(%esp)    # was old code segment supervisor ?
106:  jne 3f
107:  cmpw $0x17,OLDSS(%esp)    # was stack segment = 0x17 ?
108:  jne 3f
109:  movl signal(%eax),%ebx
110:  movl blocked(%eax),%ecx
111:  notl %ecx
112:  andl %ebx,%ecx
113:  bsfl %ecx,%ecx
114:  je 3f
115:  btrl %ecx,%ebx
116:  movl %ebx,signal(%eax)
117:  incl %ecx
118:  pushl %ecx
119:  call do_signal
120:  popl %eax
121:  3: popl %eax
122:  popl %ebx
123:  popl %ecx
124:  popl %edx
125:  pop %fs
126:  pop %es
127:  pop %ds
128:  iret

include\linux\sys.h
71: extern int sys_setreuid();
72: extern int sys_setregid();
73:
74: fn_ptr sys_call_table[] = { sys_setup, sys_exit, sys_fork, sys_read,
75:                             sys_write, sys_open, sys_close, sys_waitpid, sys_creat, sys_link,
76:                             sys_unlink, sys_execve, sys_chdir, sys_time, sys_mknod, sys_chmod,
77:                             sys_chown, sys_break, sys_stat, sys_lseek, sys_getpid, sys_mount,
78:                             sys_umount, sys_setuid, sys_getuid, sys_stime, sys_ptrace, sys_alarm,
79:                             sys_fstat, sys_pause, sys_utime, sys_stty, sys_gtty, sys_access,
80:                             sys_nice, sys_fftime, sys_sync, sys_kill, sys_rename, sys_mkdir,
81:                             sys_rmdir, sys_dup, sys_pipe, sys_times, sys_prof, sys_brk, sys_setgid,
82:                             sys_getgid, sys_signal, sys_geteuid, sys_getegid, sys_actc, sys_phys,
83:                             sys_lock, sys_ioctl, sys_fcntl, sys_mpx, sys_setpgid, sys_ulimit,
84:                             sys_uname, sys_umask, sys_chroot, sys_ustat, sys_dup2, sys_getppid,
85:                             sys_getgr, sys_setsid, sys_sigaction, sys_getmask, sys_ssetmask,
86:                             sys_setreuid, sys_setregid };

207: .align 2
208: sys_fork:
209:     call find_empty_process
210:     testl %eax,%eax
211:     js 1f
212:     push %gs
213:     pushl %esi
214:     pushl %edi
215:     pushl %ebp
216:     pushl %eax
217:     call copy_process
218:     addl $20,%esp
219: 1:  ret

36: int find_empty_process(void)
37: {
38:     int i;
39:
40:     repeat:
41:         if ((++last_pid)<0) last_pid=1;
42:         for(i=0 ; i<NR_TASKS ; i++)
43:             if (task[i] && task[i]->pid == last_pid) goto repeat;
44:         if (!task[i])
45:             return i;
46:     return -EAGAIN;
47: }
48: }

Slides-50

```

图 35 系统调用

```

kernel\sys_call.s
int 0x80

69: int copy_process(int nr,long ebp,long edi,long esi,long gs,long none,
70:                   long ebx,long ecx,long edx,
71:                   long fs,long ds,
72:                   long esp,long cs,long esp,long esp,long ss)
73: {
74:     struct task_struct *p;
75:     int i;
76:     struct file *f;
77:
78:     p = (struct task_struct *) get_free_page();
79:     if (!p)
80:         return -EAGAIN;
81:     task[nr] = p;
82:     /*p = current; /* NOTE! this doesn't copy the supervisor stack */
83:     p->state = TASK_UNINTERRUPTIBLE;
84:     p->pid = last_pid;
85:     p->father = current->pid;
86:     p->counter = p->priority;
87:     p->signal = 0;
88:     p->alarm = 0;
89:     p->leader = 0;
90:     p->utime = p->stime = 0;
91:     p->cutime = p->cstime = 0;
92:     p->start_time = jiffies;
93:     p->tss.back_link = 0;
94:     p->tss.esp0 = PAGE_SIZE + (long)p;
95:     p->tss.ss0 = 0x10;
96:     p->tss.esp = esp;
97:     p->tss.flags = eflags;
98:     p->tss.es = es & 0xffff;
99:     p->tss.cs = cs & 0xffff;
100:    p->tss.ds = ds & 0xffff;
101:    p->tss.esp = esp;
102:    p->tss.ebp = ebp;
103:    p->tss.esi = esi;
104:    p->tss.edi = edi;
105:    p->tss.edx = edx;
106:    p->tss.es = es & 0xffff;
107:    p->tss.cs = cs & 0xffff;
108:    p->tss.ds = ds & 0xffff;
109:    p->tss.fs = fs & 0xffff;
110:    p->tss.gs = gs & 0xffff;
111:    p->tss.ldt = LDT(nr);
112:    p->tss.trace_bitmap = 0x80000000;
113:    if (last_task_used_math == current)
114:        __asm__("clts ; fnsave %0::%m" (p->tss.i387));
115:    if (copy_mem(nr,p))
116:    {
117:        task[nr] = NULL;
118:        free_page((long)p);
119:        return -EAGAIN;
120:    }
121:    for (i=0; i<NR_OPEN;i++)
122:        if ((fp->filp[i]))
123:            fp->count++;
124:    if (current->parent)
125:        current->parent->count++;
126:    if (current->root)
127:        current->root->count++;
128:    if (current->executables)
129:        current->executables->count++;
130:    set_tss_desc(gdt+(nr<1)?FIRST_TSS_ENTRY:&(p->tss));
131:    set_ldt_desc(gdt+(nr<1)?FIRST_LDT_ENTRY:&(p->ldt));
132:    p->state = TASK_RUNNING; /* do this last, just in case */
133:    return last_pid;
134: }

include\unistd.h
133: #define _syscall(type,name) \
134: type name(void) \
135: { \
136:     long __res; \
137:     __asm volatile ("int $0x80" \
138:                    : "=a" __res \
139:                     : "0" (_NR_##name)); \
140:     if (__res > 0) \
141:         return (type) __res; \
142:     errno = __res; \
143:     return -1; \
144: }

int 0x80
CS:EIP

kernel\sys_call_table
80: system_call:
81:    cmpl $nr_system_calls-1,%eax
82:    ja bad_sys_call
83:    push %ds
84:    push %es
85:    push %fs
86:    pushl %edx
87:    pushl %ecx    # push %ebx,%ecx,%edx as parameters
88:    pushl %eax    # to the system call
89:    movl $0x10,%edx    # set up ds,es to kernel space
90:    mov %dx,%ds
91:    mov %dx,%es
92:    movl $0x17,%edx    # fs points to local data space
93:    mov %dx,%fs
94:    call sys_call_table(%eax,4)
95:    pushl %eax
96:    movl current,%eax
97:    cmpl $0,state(%eax)    # state
98:    jne reschedule
99:    jne 3f

include\linux\sys.h
207: .align 2
208: sys_fork:
209:     call find_empty_process
210:     testl %eax,%eax
211:     js 1f
212:     push %gs
213:     pushl %esi
214:     pushl %edi
215:     pushl %ebp
216:     pushl %eax
217:     call copy_process
218:     addl $20,%esp
219: 1:  ret

Slides-52

```

图 36 copy_process

9. sys_fork 中最重要的结构是 copy_process：调用 copy_process 复制父进程的任

务数据结构信息

1. 为新进程申请一页内存页，调用 `get_free_page` 函数遍历 `mem_map[]` 数组（在 `mem_init` 中被初始化），找出值为零的项，就表示找到了空闲的一页内存
2. 将父进程的 `task_struct` 的全部值都复制给即将创建的进程 `p`
3. 为新进程修改复制的任务数据结构的某些字段值，如 `TSS` 结构中的各字段值，其中 `esp0` 正好指向该页顶端，让新进程的状态保持父进程即将进入中断过程前的状态。
4. 接着系统在线性地址空间中设置新任务代码段和数据段描述符中的基址和限长（设置 `LDT` 表），并为新进程复制父进程的页目录项和页表项。
 - `LDT` 的复制和改造，使得不同进程分别映射到了不同的线性地址空间。段限长就是取自父进程设置好的段限长；段基址取 `nr \times 64M`
 - 页表的复制，使得不同进程又从不同的线性地址空间被映射到了相同的物理地址空间，同时将新老进程的页表都变成只读状态，为后面写时复制的缺页中断做准备。
5. 把当前进程（父进程）打开文件对应 `i` 节点的引用次数都增 1。
6. 随后在 `GDT` 表中设置新任务 `TSS` 段和 `LDT` 段描述符项。这两个段的限长均被设置成 104 字节。
7. 最后将新进程的状态置为 `TASK_RUNNING`，可以被加入调度队列。

4.2.1 写时复制

在 `fork()` 的执行过程中，内核并不会立刻为新进程分配代码和数据内存页。新进程将与父进程共同使用父进程已有的代码和数据内存页面。只有当以后执行过程中如果其中有一个进程以写方式访问内存时被访问的内存页面才会在写操作前被复制到新申请的内存页面中。

4.2.2 父进程子进程

任务数据结构中有这样几个指针，可以描述父进程与其多个子进程之间的关系，可以发现子进程间并非完全独立的。

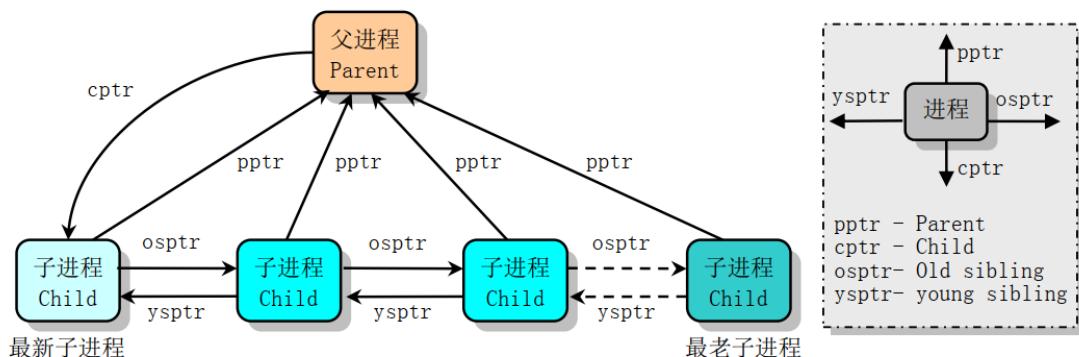


图 37 父进程与子进程

4.3 程序加载 execve

在 Linux 系统中，内核通过在 `kernel\sched.c` 中的 `sched_init` 函数中调用 `set_system_gate(0x80, &system_call)` 将 `int 0x80` 中断门绑定到 `system_call` 程序，实现系统调用。当用户程序通过声明 `_syscall3(int, execve, const char *, file, char *, argv, char *, envp)` 来调用 `int 0x80` 进入内核态时，内核会检查 `eax`

寄存器中的系统调用号`__NR_execve`，并从`sys_call_table`中找到`sys_execve`函数并跳转执行。首先，`sys_execve`会检查调用进程的权限，确保其有权执行指定程序，并通过调用文件系统接口获取文件的`inode`进行存在性和权限验证。在验证权限后，`sys_execve`调用`do_execve`以完成新程序的加载过程。在此过程中，内核会清理旧的页表，分配新的页目录和页表，设置新的内存映射，并更新当前进程的页表指针，确保新程序能够被正确执行。接着，内核会在堆栈中设置参数和环境变量的指针，更新进程的`executable`字段，复位信号处理句柄，关闭不必要的文件描述符，释放旧代码和数据段的内存空间，最后修改堆栈中返回地址为新程序的入口地址，以准备在下次调度时执行新程序。

此过程`execve()`函数的主要功能为：

1. 执行对命令行参数和环境参数空间页面的初始化操作。
 - 设置初始空间起始指针并初始化空间页面指针数组为`(NULL)`；
 - 根据执行文件名取执行对象的`i`节点，检查文件类型和执行权限；
 - 计算参数个数和环境变量个数。
2. 根据执行文件开始部分的头数据结构，对其中信息进行处理。根据被执行文件`i`节点读取文件头部信息，
 - 若是`Shell`脚本程序（第一行以`"#!"`开始），则分析`Shell`程序名及其参数，并以被执行文件作为参数执行该`Shell`程序；
 - 根据文件的幻数以及段长度等信息判断是否可执行；
3. 对当前调用进程进行运行新文件前的初始化操作
 - 指向新执行文件的`i`节点，复位信号处理句柄；
 - 根据头结构信息设置局部描述符基址和段长，设置参数和环境参数页面指针，修改进程各执行字段内容；
4. 替换堆栈上原调用`execve()`程序的返回地址为新执行程序运行地址，运行新加载的程序。

```

10: _syscall3(int,execve,const char *,file,char **,argv,char **,envp)

int execve(const char * file,char ** argv,char ** envp) \
{ \
    long __res; \
    __asm__ volatile ("int $0x80" \
    : "=a" (__res) \
    : "0" (__NR_execve), "b" ((long)(file)), "c" ((long)(argv)), "d" ((long)(envp))); \
    if (__res>=0) \
        return (int) __res; \
    errno=-__res; \
    return -1; \
}
  
```

图 38 execve 声明

具体地，`execve()`对应系统调用`sys_execve()`，经过简单压栈后执行函数`do_execve`，去除一些逻辑校验和权限检验，其实就是三个步骤：

4.3.1 读取加载执行文件

这一过程主要完成的是：根据文件名，找到并读取文件里的内容，解析开头`1KB`的数据为`exec`结构：

- 初始化`128KB`（`32`页）的参数和环境串空间，把所有字节清零；
- 读取可执行程序文件`i`节点，从中取出文件属性信息进行一些权限检测；

- 根据 **i** 节点读取文件头信息，并复制缓冲块数据到 **ex** 结构中；
- 判断是否对 **Shell** 脚本程序，对可执行程序进行合法性检测。

```

1. int do_execve(...) {
2.     for (i=0 ; i<MAX_ARG_PAGES ; i++)           \\ 清空页表
3.         page[i]=0;
4.     if (!(inode=namei(filename)))               \\ 根据文件名获取 inode
5.         return -ENOENT;
6.     \\
7.     if (!(bh = bread(inode->i_dev,inode->i_zone[0]))) { \\ 读取文件第一
8.         retval = -EACCES;
9.         goto exec_error2;
10.    }
11.    struct exec ex = ((struct exec ) bh->b_data);

```

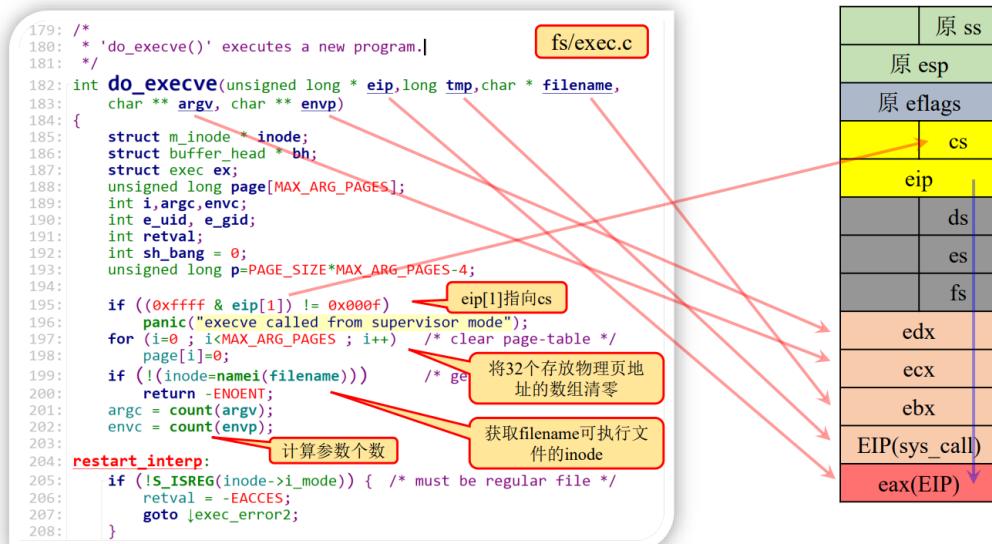


图 39 do_execve 函数参数解析

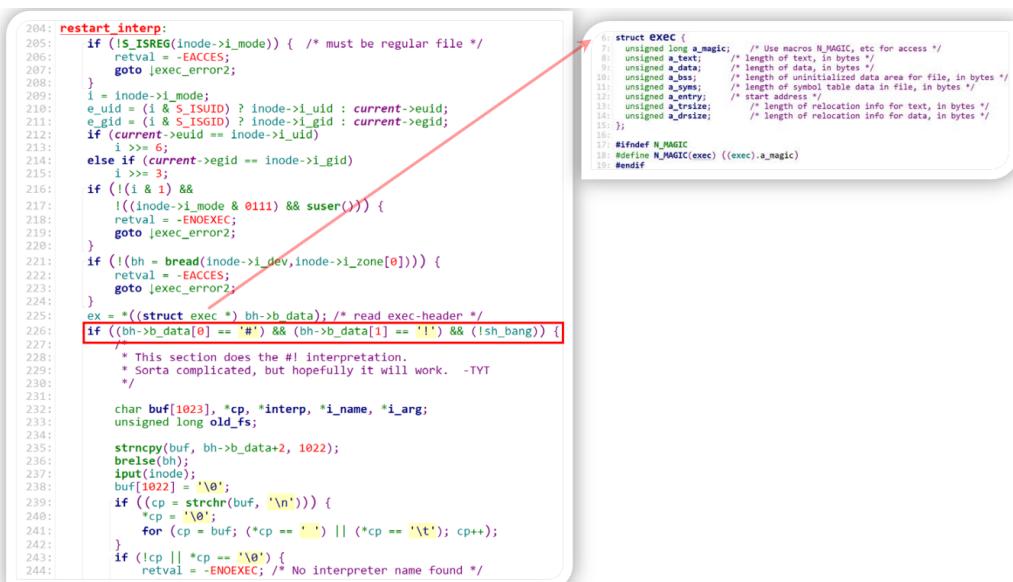


图 40 读取文件 inode

```

298:     brelse(bh);
299:     if (N_MAGIC(ex) != Z_MAGIC || ex.a_trsize || ex.a_drsiz ||
300:         ex.a_text+ex.a_data+ex.a_bss>0x3000000 || 
301:         inode->i_size < ex.a_text+ex.a_data+ex.a_syms+N_TXTOFF(ex)) {
302:         retval = -ENOEXEC;
303:         goto ↓exec_error2;
304:     }
305:     if (N_TXTOFF(ex) != BLOCK_SIZE) {
306:         printk("%s: N_TXTOFF != BLOCK_SIZE. See a.out.h.", filename);
307:         retval = -ENOEXEC;
308:         goto ↓exec_error2;
309:     }
310:     if (!sh_bang) {
311:         p = copy_strings(envc, envp, page, p, 0);
312:         p = copy_strings(argc, argv, page, p, 0);
313:         if (!p) {
314:             retval = -ENOMEM;
315:             goto ↓exec_error2;
316:         }
317:     }

```

各种检测

exec头部必须占有BLOCK_SIZE大小

不是shell

拷贝参数到page当中（里面会分配物理页）

图 41 对文件做各种检查并解析 envp, argv 解析指定解释器的行、设置参数及环境变量

4.3.2 调整内存

```

318: /* OK, This is the point of no return */
319:     if (current->executable)
320:         iput(current->executable);    // 释放当前的可执行文件inode节点
321:     current->executable = inode;   // Line 199: inode=namei(filename)
322:     for (i=0 ; i<32 ; i++)
323:         current->sigaction[i].sa_handler = NULL;
324:     for (i=0 ; i<NR_OPEN ; i++)
325:         if ((current->close_on_exec>>i)&1)
326:             sys_close(i);           // 清空当前进程的页表映射
327:     current->close_on_exec = 0;
328:     free_page_tables(get_base(current->ldt[1]),get_limit(0x0f));
329:     free_page_tables(get_base(current->ldt[2]),get_limit(0x17));
330:     if (last_task_used_math == current)
331:         last_task_used_math = NULL; // 改变ldt
332:     current->used_math = 0;
333:     p += change_ldt(ex.a_text,page)-MAX_ARG_PAGES*PAGE_SIZE; // 制作参数表（类似指针数组）
334:     p = (unsigned long)create_tables((char *)p,argc,envc); // 写入代码结束位置、数据结束位置、bss结束位置
335:     current->brk = ex.a_bss + (current->end_data - ex.a_data +
336:         (current->end_code - ex.a_text)); // 4KB对齐
337:     current->start_stack = p & 0xfffff000;
338:     current->euid = e_uid;
339:     current->egid = e_gid;
340:     i = ex.a_text+ex.a_data;
341:     while (i&0xffff)
342:         put_fs_byte(0,(char *) (i++));
343:     eip[0] = ex.a_entry;           // eip, magic happens :-)
344:     eip[3] = p;                  // stack pointer */
345:     return 0;
346: exec_error2:
347:     iput(inode);               // where
348: exec_error:
349:     for (i=0 ; i<MAX_ARG_PAGES ; i++)
350:         free_page(page[i]);
351:     return(retval);
352: } « end do_execve »

```

```

199: .align 2
200: sys_execve:
201:     lea EIP(%esp),%eax
202:     pushl %eax
203:     call do_execve
204:     addl $4,%esp
205:     ret

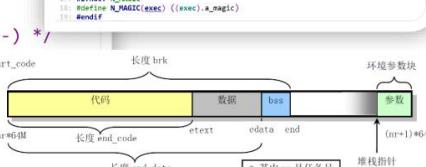
```

将系统调用压栈里的eip，改为应用程序的入口地址。同时将栈顶esp设为0

```

1: struct EXEC {
2:     unsigned long a_magic; /* Use exec N_MAGIC, etc for access */
3:     /* length of text, in bytes */
4:     unsigned a_text; /* length of data, in bytes */
5:     /* length of uninitialized data area for file, in bytes */
6:     unsigned a_bss; /* length of initialized data in file, in bytes */
7:     /* start address */
8:     unsigned a_entry; /* length of relocation info for text, in bytes */
9:     /* length of relocation info for data, in bytes */
10:    unsigned a_trsize;
11:    unsigned a_drsiz;
12: };
13: #ifndef N_MAGIC
14: #define N_MAGIC(exec) ((exec).a_magic)
15: #endif

```



Slides-100

图 42 调整内存

将当前进程可执行文件 i 节点更新，让进程 executable 字段指向新执行文件的 i 节点。

```

12.     if (current->executable)
13.         iput(current->executable);
14.     current->executable = inode;

```

然后复位原进程的所有信号处理句柄，再根据设定的执行时关闭文件句柄（close_on_exec）位图标志，关闭指定的打开文件并复位该标志。

```

15.     for (i=0 ; i<32 ; i++)

```

```

16.     current->sigaction[i].sa_handler = NULL;
17.     for (i=0 ; i<NR_OPEN ; i++)
18.         if ((current->close_on_exec>>i)&1)
19.             sys_close(i);
20.     current->close_on_exec = 0;

```

然后根据当前进程指定的基地址和限长，释放原来程序的代码段和数据段所对应的内存页表指定的物理内存页面及页表本身。

```

21.     free_page_tables(get_base(current->ldt[1]),get_limit(0x0f));
22.     free_page_tables(get_base(current->ldt[2]),get_limit(0x17));

```

接下来修改局部表中描述符基址和段限长，并将 128KB 的参数和环境空间页面放置在数据段末端。然后调用内部函数 `create_tables()`，在栈空间中创建环境和参数变量指针表，供程序的 `main()` 作为参数使用，并返回该栈指针。

具体如下：

```

23.     unsigned long p = PAGE_SIZE * MAX_ARG_PAGES - 4;
24.     \\
25.     p = copy_strings(envc, envp, page, p, 0);
26.     p = copy_strings(argc, argv, page, p, 0);
27.     \\
28.     p += change_ldt(ex.a_text, page)-MAX_ARG_PAGES*PAGE_SIZE;
29.     p = (unsigned long) create_tables((char *)p, argc, envc);

```

1. 在进入 `do_execve` 之初，定义 $p = 4096 \times 32 - 4 = 0x20000 - 4 = 128K - 4$ ，作为参数表的开始地址；
2. 接下来两个 `copy_strings` 就是参数表里面存放 `argv` 和 `envp` 等参数信息（实际上就是从 `p` 指针指向的位置向下发展）；

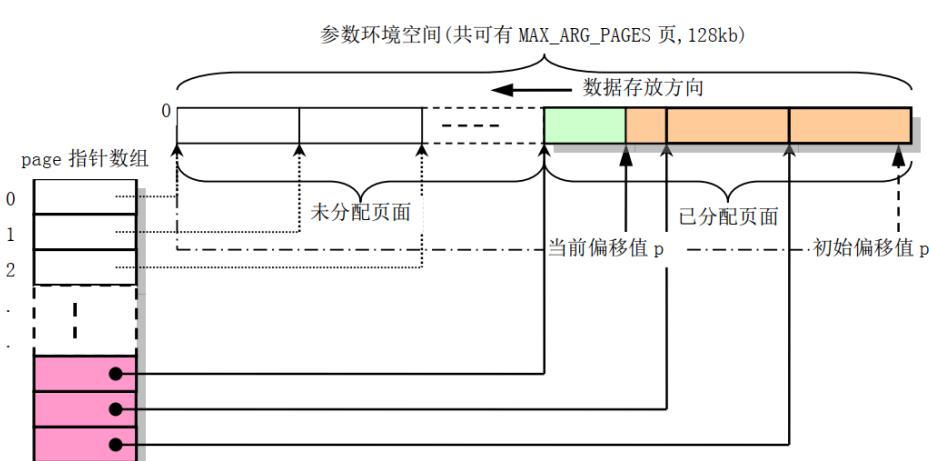


图 42 `copy_string`

3. 更新局部描述符，根据 `ex.a_text` 修改局部描述符中的代码段限长 `code_limit`，并将 128KB 的参数和环境空间页面放置在数据段末端；
4. 更新参数表，`create_tables()` 函数用于根据给定的当前堆栈指针值 `p` 以及参数变量个数值 `argc` 和环境变量个数 `envc`，在新的程序堆栈中创建环境和参数变量指针表，最终得到初始堆栈指针 `sp`，创建完毕后堆栈指针表的形式如下：

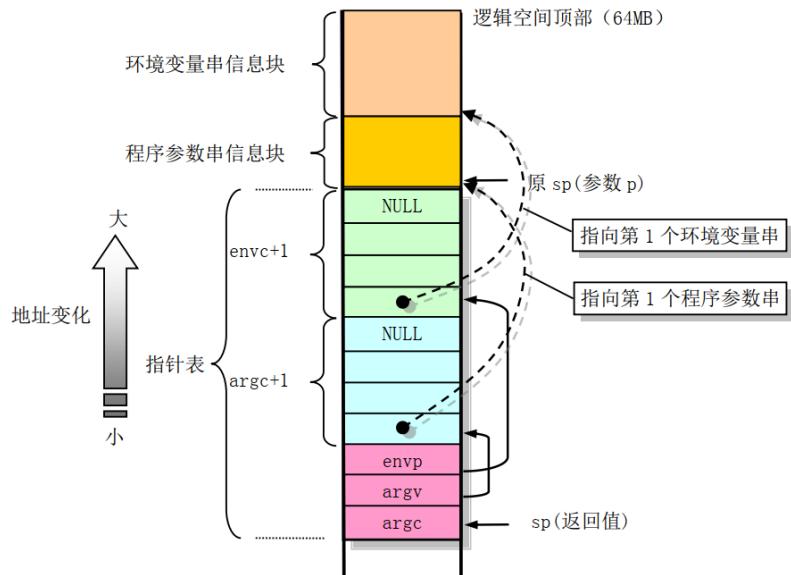


图 42 参数表堆栈

4.3.3 准备出发

接着再修改进程各字段值为新执行文件的信息（如代码尾字段 `end_code`, 数据尾字段 `end_data` 等），然后设置进程栈开始字段为栈指针所在页面（4KB 对齐）。

最后将原调用系统中断的程序在堆栈上的代码指针替换为指向新执行程序的入口点，并将栈指针替换为新执行文件的栈指针。此后返回指令将弹出这些栈数据并使得 CPU 去执行新执行文件，而不会返回到原调用系统中断的程序中去了。

```

30.    \\ .....
31.    eip[0] = ex.a_entry;      \ eip, magic happens :-) \
32.    eip[3] = p;            \ stack pointer \

```

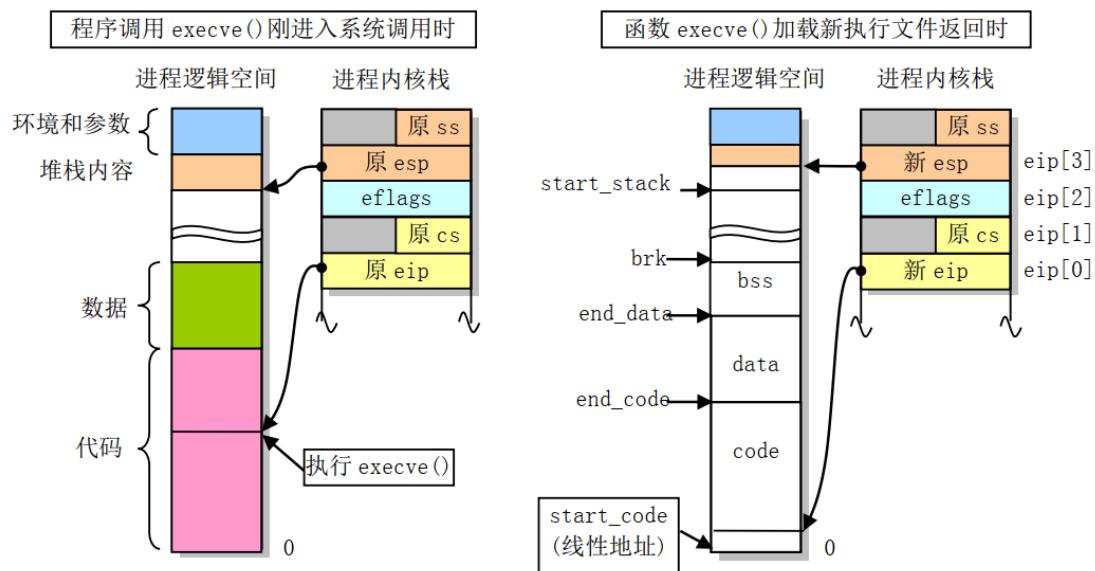


图 43 eip-esp

那么接下来似乎就可以开始运行了，但是在新进程的地址空间。但代码和数据还未拷入。

4.4 缺页中断

在 Linux 0.11 源码中，缺页中断是在陷阱门初始化（`trap_init`）时设置的 `set_trap_gate(14,&page_fault)`。当触发 Page-Fault 中断时，会进入 `page_fault` 方法。根据 `error_code` 的不同分为，缺页引起的页异常中断，调用 `do_no_page` 来处理；由于页写保护引起调用 `do_wp_page` 处理。`do_no_page()` 首先计算出缺页时相对于代码段起始地址的偏移地址 `tmp`，引发缺页中断的地址在缺页发生时由硬件自动保存在 `CR2` 寄存器中。如果缺页是由 `execve` 导致的，为进程注册一页新的物理内存，并读取相应内容到这页物理内存中。申请内存时，调用 `get_free_page` 函数从 `mem_map[]` 中申请一个页框（帧）。接着，根据 `inode` 信息，通过 `tmp\BLOCK_SIZE` 计算出 `tmp` 逻辑地址在可执行文件中的块序号。数据读取时，跳过第一块，使用 `bread_page` 连续读取 4 个数据块到 1 页内存，首先定义四个 `bh` 缓冲块，调用 `getblk` 方法根据设备号 `dev` 和数据块号 `block` 在所有缓冲块中寻找匹配或最空闲的缓冲块。如果高速缓存命中，即在哈希数组中找到对应的数据块，则返回对应缓冲头结构的指针；若不命中，则从空闲链表头开始扫描，寻找一个空闲缓冲块并返回其缓冲头指针。随后，调用 `ll_rw_block` 方法将硬盘中指定数据块中的数据复制到此缓冲块内。最后，调用 `put_page` 将当前进程的 `address` 线性地址空间页面与 `page` 处的物理页面挂接起来，建立页表填充页框。

```

181: void trap_init(void) kernel\trap.c
182: {
183:     int i;
184:
185:     set_trap_gate(0,&divide_error);
186:     set_trap_gate(1,&debug);
187:     set_trap_gate(2,&mi);
188:     set_system_gate(3,&int3); /* int3-5 can be c
189:     set_system_gate(4,&overflow);
190:     set_system_gate(5,&bounds);
191:     set_trap_gate(6,&invalid_op);
192:     set_trap_gate(7,&device_not_available);
193:     set_trap_gate(8,&double_fault);
194:     set_trap_gate(9,&coprocessor_segment_overrun);
195:     set_trap_gate(10,&invalid_TSS);
196:     set_trap_gate(11,&segment_not_present);
197:     set_trap_gate(12,&stack_gard);
198:     set_trap_gate(13,&general_protection);
199:     set_trap_gate(14,&page_fault);
200:     set_trap_gate(15,&reserved);
201:     set_trap_gate(16,&coprocessor_error);

14: page_fault:
15:     xchgl %eax,(%esp)
16:     pushl %ecx
17:     pushl %edx
18:     pushl %ds
19:     pushl %es
20:     pushl %fs
21:     movl $0x10,%edx
22:     movl %dx,%ds
23:     movl %dx,%es
24:     movl %dx,%fs
25:     movl %cr2,%edx
26:     pushl %edx
27:     pushl %eax
28:     testl $1,%eax
29:     jne 1f
30:     call do_no_page
31:     jmp 2f
32: 1: call do_wp_page
33: 2: addl $8,%esp
34:     popl %fs
35:     popl %es
36:     popl %ds
37:     popl %edx
38:     popl %ecx
39:     popl %eax
40:     iret

mm\memory.c
33: static inline volatile void oom(void)
34: {
35:     printk("out of memory\n\r");
36:     do_exit(SIGSEGV);
37: }
38:
39: #define invalidate() \
40: __asm__ ("movl %%eax,%cr3"::"a" (0))

366: void do_no_page(unsigned long error_code,unsigned long address)
367: {
368:     int nr[4];
369:     unsigned long tmp;
370:     unsigned long page;
371:     int block,i;
372:
373:     address &= 0xfffffff000;
374:     tmp = address - current->start_code;
375:     if (!current->executable || tmp >= current->end_data) {
376:         get_empty_page(address);
377:         return;
378:     }
379:     if (share_page(tmp))
380:         return;
381:     if (!(page = get_free_page()))
382:         oom();
383:     /* remember that 1 block is used for header */
384:     block = 1 + tmp/BLOCK_SIZE;
385:     for (i=0 ; i<4 ; block++,i++)
386:         nr[i] = bmap(current->executable,block);
387:     bread_page(page,current->executable->i_dev,nr);
388:     i = tmp + 4096 - current->end_data;
389:     tmp = page + 4096;
390:     while (i - > 0) {
391:         tmp--;
392:         *(char *)tmp = 0;
393:     }
394:     if (put_page(page,address))
395:         return;
396:     free_page(page);
397:     oom();
398: } /* end do_no_page */

得到空闲页
文件操作
1 page是4个block
填入页表项

```

Slides-75

图 44 缺页中断相关代码

我们仅仅将 `hello` 文件的头部加载到了内存，其他部分并没有进行加载，因此在访问 `start_code` 这个线性地址时，会遇到了页表项的存在位 `P` 等于 0 的情况。

一旦遇到了这种情况，CPU 会触发一个中断：页错误（Page-Fault），这在之前的读书笔记中也已经提到过了，这个中断也是在陷阱门初始化（`trap_init`）时被设置的。

当触发这个 `Page-Fault` 中断后，就会进入 `Linux 0.11` 源码中的 `page_fault` 方法

可以看到，这里根据 `error_code` 的不同，主要处理两种情况：

1. 一是由于缺页引起的页异常中断，这需要通过调用 `do_no_page(error_code, address)` 来处理；
2. 二是由于页面写保护引起的页异常，此时调用页写保护处理函数 `do_wp_page(error_code, address)` 进行处理。

而 `do_no_page()` 的逻辑也很简单：

1. 首先计算出发生缺页时的相对于代码段起始地址的偏移地址 `tmp`

```
33.     tmp = address - current->start_code;\\" .....
```

2. 如果当前进程没有可执行文件 (`!current->executable`) 或者缺页的逻辑地址大于进程的代码段和数据段之和 (`tmp >= current->end_data`)，直接调用 `get_empty_page` 为进程申请一页新物理内存即可。(这种情况就是由于 进程压栈（为堆或栈中数据寻找新的页面）造成，而非执行 `execve` 导致。)

```
34.     if (!current->executable || tmp >= current->end_data) {  
35.         get_empty_page(address);  
36.         return;  
37.     }
```

3. 而如果是 `execve` 导致的缺页中断，需要先检查当前进程的 `executable` 是否被其他进程同样引用，于是需要为该进程注册一页新的物理内存，并且读取相应内容到这页物理内存中。

- 申请内存：调用 `get_free_page` 函数去 `mem_map[]` 中申请一个页框（帧）
- 获取位置信息：根据 `inode` 信息，由 `tmp\BLOCK_SIZE` 即得到 `tmp` 逻辑地址在可执行文件中的块序号
 - 数据读取：跳过第一块（上一步已经读取了），把对应一个页面大小的硬盘数据复制到内存

```
38.     if (!(page = get_free_page()))  
39.         oom();  
40.     block = 1 + tmp\BLOCK_SIZE;  
41.     for (i=0 ; i<4 ; block++,i++) \\一个数据块 1024 字节，所以一页内存需要  
        读 4 个数据块  
42.         nr[i] = bmap(current->executable,block);  
43.         bread_page(page,current->executable->i_dev,nr);  
44.         \\ .....  
45.     if (put_page(page,address))  
46.         return;
```

4.4.1 int bmap(struct m_inode inode,int block)

`bmap` 负责将相对于文件的数据块转换为相对于整个硬盘的数据块。

磁盘块号采用两个字节存储：一个间接块占一个磁盘块 `1k`（两个扇区）。`i` 节点中可以访问的总块数：直接块 `7+一次间接 512+二次间接 512 512`。`bmap` 找到 `block` 对应的磁盘块号！放到数组 `nr[i]` 中

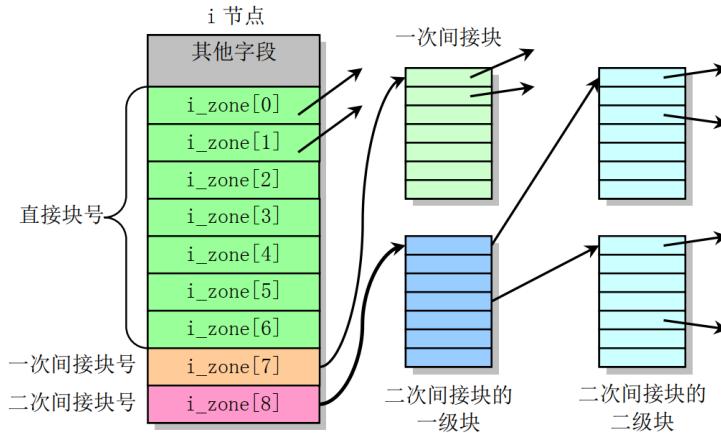


图 45 inode

```

static int _bmap(struct m_inode * inode,int block,int create)
{
    struct buffer_head * bh;
    int i;
    if (block<0)
        panic("_bmap: block<0");
    if (block >= 7*512+512*512)
        panic("_bmap: block>big");
    if (block<7) {
        if (create && !inode->i_zone[block])
            if ((inode->i_zone[block]=new_block(inode->i_dev)) <
                inode->i_ctime=CURRENT_TIME;
            inode->i_dirt=1;
        }
        return inode->i_zone[block];
    }
    block -= 7;
    if (block<512) {
        if (create && !inode->i_zone[7])
            if ((inode->i_zone[7]=new_block(inode->i_dev)) <
                inode->i_dirt=1;
            inode->i_ctime=CURRENT_TIME;
        }
        if (!inode->i_zone[7])
            return 0;
        if (!bh=bread(inode->i_dev,inode->i_zone[7]))
            return 0;
        i = ((unsigned short *) (bh->b_data))[block];
        if (create & & i)
            if ((i=new_block(inode->i_dev)) <
                ((unsigned short *) (bh->b_data))[block]=i;
                bh->b_dirt=1;
            )
            brelse(bh);
        return i;
    }
    108
}

```

```

block -= 512;
if (create && !inode->i_zone[8])
    if ((inode->i_zone[8]=new_block(inode->i_dev)) <
        inode->i_dirt=1;
        inode->i_ctime=CURRENT_TIME;
    )
    if (inode->i_zone[8])
        return 0;
    if (!bh=bread(inode->i_dev,inode->i_zone[8]))
        return 0;
    i = ((unsigned short *)bh->b_data)[block>>9];
    if (create & & i)
        if ((i=new_block(inode->i_dev)) <
            ((unsigned short *) (bh->b_data))[block>>9]=i;
            bh->b_dirt=1;
        )
        brelse(bh);
    if (!i)
        return 0;
    if (!bh=bread(inode->i_dev,i))
        return 0;
    i = ((unsigned short *)bh->b_data)[block&511];
    if (create & & i)
        if ((i=new_block(inode->i_dev)) <
            ((unsigned short *) (bh->b_data))[block&511]=i;
            bh->b_dirt=1;
        )
        brelse(bh);
    return i;
}

```

图 46 fs\inode.c_bmap

4.4.2 void bread_page(unsigned long addr,int dev,int b[4])

`bread_page` 就是连续读取 4 个数据块到 1 页内存的函数，就是定义四个 `bh` 缓冲块，首先调用 `getblk` 方法根据设备号 `dev` 和数据块号 `block`，在所有缓冲块中寻找匹配或最为空闲的缓冲块。

- 若高速缓冲命中，也就是在哈希数组中找到对应的数据块，返回对应缓冲头结构的指针；
- 若不命中，则从空闲链表头开始，对空闲链表进行扫描，寻找一个空闲缓冲块并返回其缓冲头指针。再调用 `ll_rw_block` 方法把硬盘中指定数据块中的数据复制到此缓冲块内

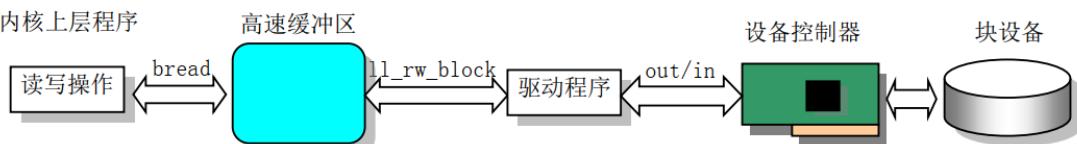


图 47 硬盘访问

4.4.3 unsigned long put_page(unsigned long page,unsigned long address)

最后调用 `put_page` 来将当前进程 `address` 线性地址空间页面与 `page` 处物理页面挂接起来，建立页表填充页框。(返回后会重新执行刚才导致缺页异常的那句代码)

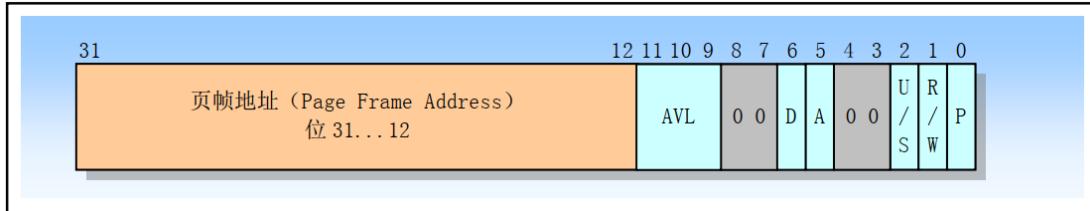


图 48 页表项格式

实际上就是写入页目录项和页表项的过程，这里根据页表项的格式，需要在对应目录项中低 12 位设置相应标志 (7 - User、U\S、R\W)。

```
1  unsigned long put_page(unsigned long page,unsigned long address)
2 {
3     unsigned long tmp, *page_table;
4
5     /* NOTE !!! This uses the fact that _pg_dir=0 */
6
7     if (page < LOW_MEMORY || page >= HIGH_MEMORY)
8         printk("Trying to put page %p at %p\n",page,address);
9     if ((mem_map[(page-LOW_MEMORY)>>12] != 1)
10        printk("mem_map disagrees with %p at %p\n",page,address);
11     page_table = (unsigned long *) ((address>>20) & 0xffffc);
12     if ((*page_table)&1){
13         fprintf(7, "page_catalogue_table(p=1): \t%lx->%lx\n", page_table, *page_table);
14     page_table = (unsigned long *) (0xfffff000 & *page_table);
15     }
16     else {
17         if (!(tmp=get_free_page()))
18             return 0;
19         *page_table = tmp|7;
20         fprintf(7, "page_catalogue_table(p=0, new allocate): \t%lx->%lx\n", page_table, *page_table);
21         page_table = (unsigned long *) tmp;
22     }
23     page_table[(address>>12) & 0x3ff] = page | 7;
24     /* no need for invalidate */
25     return page;
26 }
```

图 49 mm\memory.c\put_page

4.5 进程调度

```
19 #define TASK_RUNNING          0
20 #define TASK_INTERRUPTIBLE    1
21 #define TASK_UNINTERRUPTIBLE  2
22 #define TASK_ZOMBIE          3
23 #define TASK_STOPPED          4
```

图 50 5 种状态

当一个进程的运行时间片用完，系统就会使用调度程序强制切换到其他的进程去执行。另外，如果进程在内核态执行时需要等待系统的某个资源，此时该进程就会调用 `sleep_on()` 或 `interruptible_sleep_on()` 自愿地放弃 CPU 的使用权，而让调度程序去执行其他进程。进程则进入睡眠状态 (`TASK_UNINTERRUPTIBLE` 或 `TASK_INTERRUPTIBLE`)。

只有当进程从“内核运行态”转移到“睡眠状态”时，内核才会进行进程切换操作。在内核态下运行的进程不能被其他进程抢占，而且一个进程不能改变另一个进程的状态。

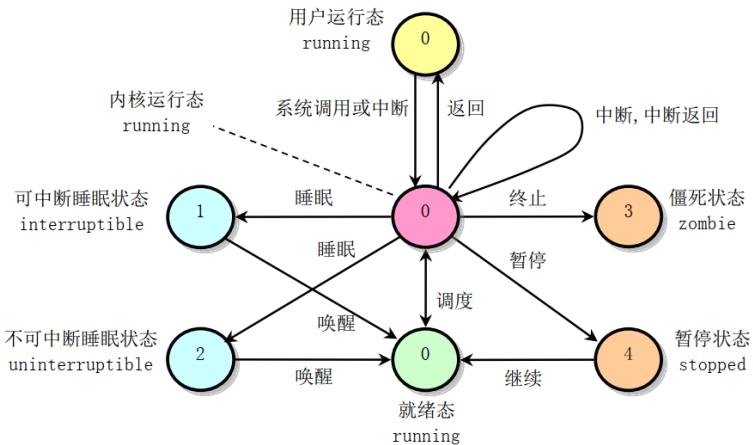


图 51 状态切换图

4.5.1 调度程序 schedule

在 Linux 系统中，内核通过在 `kernel\sched.c` 中的 `sched_init` 函数中调用 `set_intr_gate(0x20,&timer_interrupt)` 设置是时钟中断。其中 `timer_interrupt` 增加 `jiffies` 的值，并调用 `do_timer` 减少当前进程的 `current->counter`。如果为 0 调用 `schedule()` 针对任务数组中的每个任务，检查其报警定时值 `alarm`。如果任务的 `alarm` 时间已经过期，则在它的信号位图中设置 `SIGALRM` 信号，然后重置 `alarm` 值 0。如果进程接收到信号且任务处于可中断睡眠状态，则置任务为就绪状态。随后循环检查任务数组中的所有任务，选取该 `counter` 值最大的一个任务，并利用 `switch_to()` 函数切换到该任务。如果此时所有处于就绪状态的进程时间片都已经用完，系统就会根据每个进程的优先权值 `priority`，对系统中所有进程（包括正在睡眠的进程）重新计算每个任务需要运行的时间片值 `counter` 公式是：`counter=counter\2+priority`。

```

385: void sched_init(void) kernel\sched.c
386: {
387:     int i;
388:     struct desc_struct * p;
389:
390:     if (sizeof(struct sigaction) != 16)
391:         panic("Struct sigaction MUST be 16 bytes");
392:     set_tss_desc(gdt+FIRST_TSS_ENTRY,&(init_task.task.tss));
393:     set_ldt_desc(gdt+FIRST_LDT_ENTRY,&(init_task.task.ldt));
394:     p = gdt+2+FIRST_TSS_ENTRY;
395:     for(i=1;i<NR_TASKS;i++) {
396:         task[i] = NULL;
397:         p->a=p->b=0;
398:         p++;
399:         p->a=p->b=0;
400:         p++;
401:     }
402:     /* Clear NT, so that we won't have troubles with that later on */
403:     __asm__ ("pushfl ; andl $0xfffffff,%esp ; popfl");
404:     ltr(0);
405:     lldt(0);
406:     outb_p(0x36,0x43); /* binary, mode 3, LSB/MSB, ch 0 */
407:     outb_p(LATCH & 0xff, 0x40); /* LSB */
408:     outb(LATCH >> 8, 0x40); /* MSB */
409:     set_intr_gate(0x20,&timer_interrupt);
410:     outb(inb_p(0x21)&~0x01,0x21);
411:     set_system_gate(0x80,&system_call);
412: } « end sched init »
  
```

图 52 开启定时器

```

104: void schedule(void)
105: {
106:     int i,next,c;
107:     struct task_struct ** p;
108:
109: /* check alarm, wake up any interruptible tasks that have got a signal */
110:
111:     for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
112:         if ((*p) {
113:             if ((*p)->alarm && (*p)->alarm < jiffies) {
114:                 (*p)->signal |= (1<<(SIGALRM-1));
115:                 (*p)->alarm = 0;
116:             }
117:             if (((*p)->signal & ~(_BLOCKABLE & (*p)->blocked)) &&
118:                 (*p)->state==TASK_INTERRUPTIBLE)
119:                 (*p)->state=TASK_RUNNING;
120:
121: /* this is the scheduler proper: */
122:
123:     while (1) {
124:         next = 0;
125:         i = NR_TASKS;
126:         p = &task[NR_TASKS];
127:
128:         while (-i) {
129:             if (!*-p)
130:                 continue;
131:             if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
132:                 c = (*p)->counter, next = i;
133:
134:         }
135:         if (c) break;
136:         for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
137:             if (*p)
138:                 (*p)->counter = ((*p)->counter >> 1) +
139:                               (*p)->priority;
140:
141:         switch_to(next);
142:     } /* end schedule */

```

$counter = \frac{counter}{2} + priority$

Dr. GuoJun LIU

Slides-82

图 53 schedule

```

115: #define INIT_TASK \
116: /* state etc */ { 0,15,15, \
117: /* signals */ 0,{},{},0, \
118: /* ec,brk... */ 0,0,0,0,0,0, \
119: /* pid etc.. */ 0,-1,0,0,0, \
120: /* uid etc */ 0,0,0,0,0,0, \
121: /* alarm */ 0,0,0,0,0,0, \
122: /* math */ 0, \
123: /* fs info */ -1,0022,NULL, \
124: /* filp */ {NULL}, \
125: { \
126:     {0,0}, \
127:     /* ldt */ {0x9f,0xc0fa00}, \
128:     {0x9f,0xc0f200}, \
129: }, \
130: /*tss*/ {0,PAGE_SIZE+(long)& \
131:     0,0,0,0,0,0,0, \
132:     0,0,0x17,0x17,0x17,0x17, \
133:     _LDT(0),0x80000000, \
134: }, \
135: }, \
136: }

80: struct task_struct {
81: /* these are hardcoded - don't touch */
82: long state; /* -1 unrunnable, 0 runnable, >0 stopped */
83: long counter;
84: long priority;
85: long signal;
86: struct sigaction sigaction[32];
87: long blocked; /* bitmap of masked signals */
88: /* various fields */
89: int exit_code;
90: unsigned long start_code,end_code,end_data,brk,start_stack;
91: long pid,father,pgid,session,leader;
92: unsigned short uid,euid,suid;
93: unsigned short gid,egid,sgid;
94: long alarm;
95: long utime,stime,cutime,cstime,start_time;
96: unsigned short used_math;
97: /* file system info */
98: int tty; /* -1 if no tty, so it must be signed */
99: unsigned short umask;
100: struct m_inode * pwd;
101: struct m_inode * root;
102: struct m_inode * executable;
103: unsigned long close_on_exec;
104: struct file * filp[NR_OPEN];
105: /* ldt for this task 0 - zero 1 - cs 2 - ds&ss */
106: struct desc_struct ldt[3];
107: /* tss for this task */
108: struct tss_struct tss;
109: } /* end task_struct */;

```

$counter = \frac{counter}{2} + priority$

include/linux/sched.h

图 54 task0 的初始化

4.5.2 阻塞 sleep_on 与唤醒 wake_up

这里有一个有意思的设计，每一个当前任务所在的代码块中的 `tmp` 变量都指向一个正在同样等待一个资源的进程，因此也就形成了一个链表。

那么，当某进程调用了 `wake_up` 函数唤醒 `p` 上指向的第一个任务时，该任务会在 `sleep_on` 函数执行完 `schedule()` 后的状态被唤醒并执行下面的代码，把 `tmp` 指针指向的上一个任务也同样唤醒。

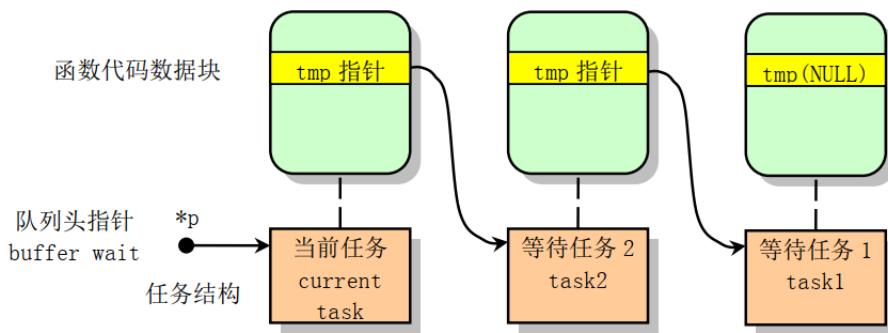


图 55 tmp 初始化

而上一个进程唤醒后，和这个被唤醒的进程一样，也会走过它自己的 `sleep_on` 函数的后半段，把它的上一个进程，也就是上上一个进程唤醒。

这样唤醒的一连串连锁反应，就会让上面的调度程序再次发挥作用。

```

    < void sleep_on(struct task_struct **p)
178 {
179     struct task_struct *tmp;
180
181     if (!p)
182         return;
183     if (current == &(init_task.task))
184         panic("task[0] trying to sleep");
185     tmp = *p;
186     *p = current;
187     current->state = TASK_UNINTERRUPTIBLE;
188     fprintf(3, "%ld\t%c\t%ld\n", current->pid, 'W', jiffies);
189     schedule();
190     if (tmp)
191     {
192         tmp->state = 0;
193         fprintf(3, "%ld\t%c\t%ld\n", tmp->pid, 'J', jiffies);
194     }
195 }
196

```

图 56 kernel\sched.c\sleep_on

```

225 void wake_up(struct task_struct **p)
226 {
227     if (p && *p)
228     {
229         if ((*p).state != 0)
230             fprintf(3, "%ld\t%c\t%ld\n", (*p)->pid, 'J', jiffies);
231         (*p).state = 0;
232         *p = NULL;
233     }
234 }

```

图 56 kernel\sched.c\weak_up

4.6 基于缓冲区的磁盘访问

在 Linux 内核中，`buffer_head` 数据结构用于管理磁盘块的缓存，初始化时通过 `buffer_init()` 设置为双向链表，并通过一个哈希数组进行管理。`getblk` 方法根据设备号和数据块号查找或分配缓冲块，遵循 LRU（最近最少使用）算法优化空闲块的选择。找

到或分配后，`ll_rw_block` 函数创建读盘请求并将其插入到 `request` 链表中，对于硬盘设备，这将触发 `do_hd_request` 函数执行实际的 I/O 操作。当 `do_hd_request` 完成读写操作后，它会发起中断（如 `0x2E`），导致 `hd_interrupt` 调用 `read_intr` 从端口读取数据到内存。如果读取完成，`end_request` 会处理请求项，唤醒等待进程，并更新当前请求项指针至下一个请求。整个过程确保了高效的磁盘 I/O 管理和调度。

在第一部分加载根文件系统中时我们已经初步讨论了 `Linux0.11` 的文件系统设计，同时简单描述了通过一个文件描述符 `fd` 寻找到存储在硬盘中的一个文件。

而在刚刚的缺页中断中，也简单描述了读取硬盘的操作，`Linux0.11` 读操作简化一下其实就三个步骤。

获得 `inode` 信息 -> 对目标存储区域 `buf` 内存进行校验 -> 读取数据到 `buf`

在这里我们再细化一下，加入缓存相关的知识，让这个体系更完整。在 `Linux0.11` 初始化时，会根据内存大小来初始化高速缓冲区大小。缓冲区位于内核 `end` 之后，`buffer_memory_end` 值之前 `4M` 内存处，一共可以划出 3 千多个逻辑块

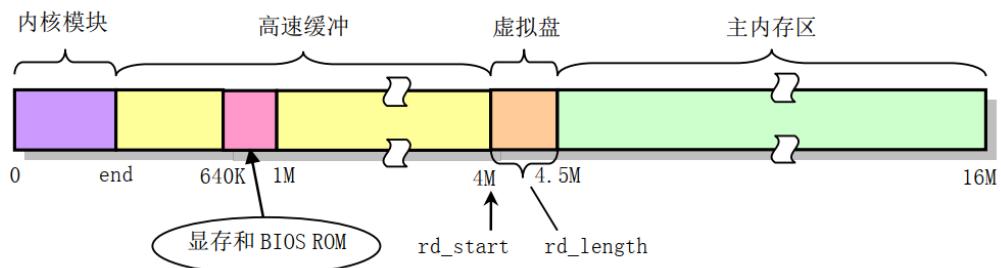


图 57 内存分布

高速缓冲区在提高对块设备的访问效率和增加数据共享方面起着重要的作用。除驱动程序以外，内核其他上层程序对块设备的读写操作都需要经过高速缓冲区管理程序来间接地实现。它们之间的主要联系是通过高速缓冲区管理程序中的 `bread()` 函数和块设备底层接口函数 `ll_rw_block()` 来实现。

上层程序若要访问块设备数据就通过 `bread()` 向缓冲区管理程序申请。如果所需的数据已经在高速缓冲区中，管理程序就会将数据直接返回给程序。如果所需的数据暂时还不在缓冲区中，则管理程序会通过 `ll_rw_block()` 向块设备驱动程序申请，同时让程序对应的进程睡眠等待。

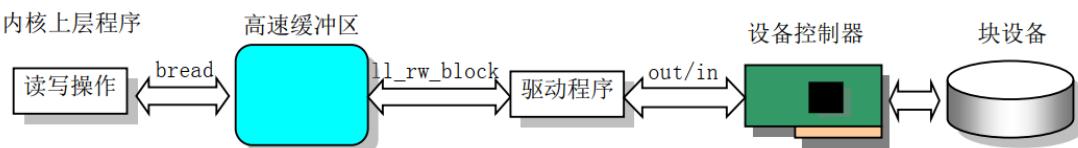


图 58 硬盘访问

回忆一下在缺页中断中首先通过 `bmap` 获取全局的数据块号，然后调用 `bread` 将 `bmap` 获取的数据块号读入到高速缓冲块。

```

6 struct buffer_head {
69     char * b_data;           /* pointer to data block (1024 bytes) */
70     unsigned long b_blocknr; /* block number */
71     unsigned short b_dev;   /* device (0 = free) */
72     unsigned char b_uptodate;
73     unsigned char b_dirty;  /* 0-clean,1-dirty */
74     unsigned char b_count;  /* users using this block */
75     unsigned char b_lock;   /* 0 - ok, 1 - locked */
76     struct task_struct * b_wait;
77     struct buffer_head * b_prev;
78     struct buffer_head * b_next;
79     struct buffer_head * b_prev_free;
80     struct buffer_head * b_next_free;
81 };

```

```

283     struct buffer_head *bread(int dev, int block)
284 {
285     struct buffer_head *bh;
286
287     if (!(bh = getblk(dev, block)))
288         panic("bread: getblk returned NULL\n");
289     if (bh->b_uptodate)
290         return bh;
291     ll_rw_block(READ, bh);
292     wait_on_buffer(bh);
293     if (bh->b_uptodate)
294         return bh;
295     brelse(bh);
296     return NULL;
297 }

```

图 59 fs\buffer.c\bread

4.6.1 缓存管理

这里主要涉及一个重要的数据结构 `buffer_head`（在 `buffer_init()` 中被初始化），当时初始化的时候将他设置为一个双向链表。

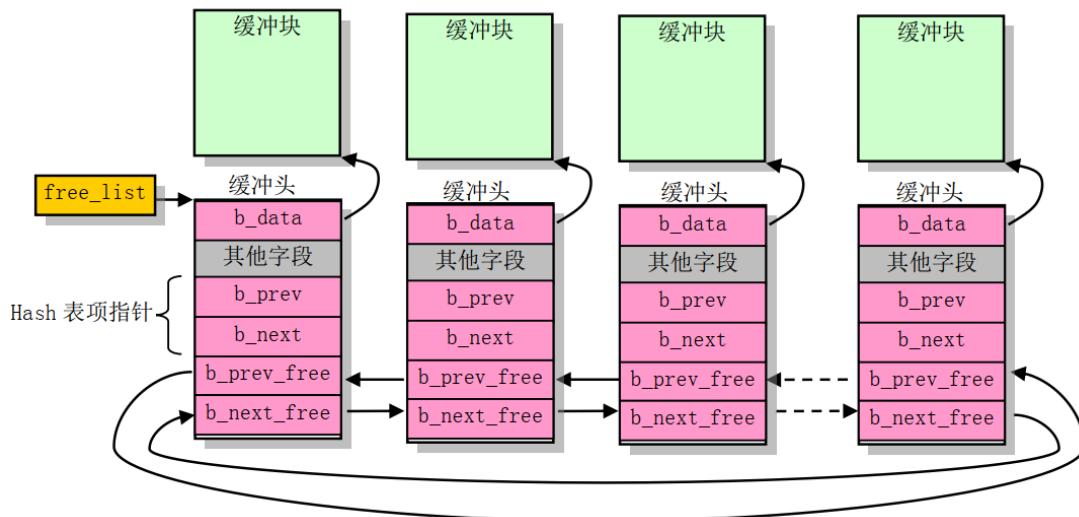


图 60 缓冲头

同时初始化时我们还设置了一个哈希数组对 `buffer_head` 进行管理

而 `getblk` 方法，就是根据设备号 `dev` 和数据块号 `block`，在所有缓冲块中寻找匹配或最为空闲的缓冲块。

- 如果指定的缓冲块存在就立刻返回对应缓冲头结构的指针
- 如果不存在，则从空闲链表头开始，对空闲链表进行扫描，寻找一个空闲缓冲块。

由于搜索空闲块是从空闲队列头开始的，因此这种先从空闲队列中移出并使用最近不常用的缓冲块，然后再重新插入到空闲队列尾部的操作也就实现了最近最少使用 `LRU` 算

法。

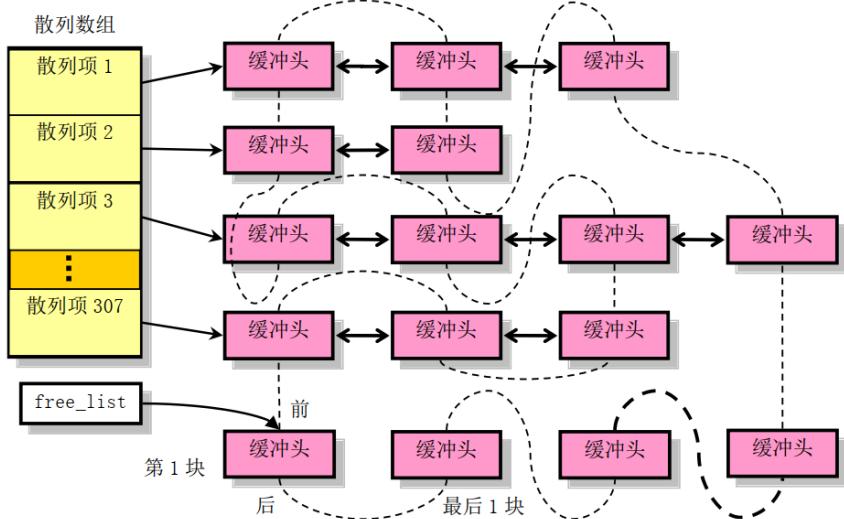


图 61 哈希管理

4.6.2 数据拷贝

接下来 `ll_rw_block` 方法负责把硬盘中指定数据块中的数据，复制到 `getblk` 方法申请到的缓冲块里。

```
160     void ll_rw_block(int rw, struct buffer_head * bh)
161     {
162         unsigned int major;
163
164         if ((major=MAJOR(bh->b_dev)) >= NR_BLK_DEV ||
165             !(blk_dev[major].request_fn)) {
166             printk("Trying to read nonexistent block-device\n\r");
167             return;
168         }
169         make_request(major,rw,bh);
170     }
```

图 62 Kernel\ ll_rw_blk.c

这个函数的逻辑就是根据 `buffer_head` 从 `request` 数组（块设备请求项初始化 `blk_dev_init` 提出）中找到一个空位创建一个新的读盘请求（包括设备号、命令、起始扇区、读\写扇区数、数据缓冲区等信息），然后作为链表项插入到相应设备的 `request` 链表中。

而 `make_request` 函数在初始化一个请求读盘请求参数后调用函数 `add_request`。而 `add_request` 是这样写的，

```

static void add_request(struct blk_dev_struct * dev, struct request * req)
{
    struct request * tmp;
    req->next = NULL;
    cli();
    if (req->bh)
        req->bh->b_dirt = 0;
    if (!(tmp = dev->current_request)) {
        dev->current_request = req;
        // printk(x, "no other request, %lx request can be done right now!\n", req);
        sti();
        (dev->request_fn)();
        return;
    }
    for ( ; tmp->next ; tmp=tmp->next)
        if ((IN_ORDER(tmp,req) ||
            !IN_ORDER(tmp,tmp->next)) &&
            IN_ORDER(req,tmp->next))
            break;
    req->next=tmp->next;
    tmp->next=req;
    // printk(x, "request %lx add in request queue after %lx\n", req, tmp);
    sti();
}

```

图 63 kernel\blk_drv\ll_rw_blk.c\add_request

所以，刚刚的 `request_fn` 背后的具体执行函数，就是这个 `do_hd_request`。最终会根据当前请求是写 (WRITE) 还是读 (READ)，在 `do_hd_request` 中调用 `hd_out` 时传入不同的参数 (磁头、扇区等)，然后向外设端口做读写操作。

```

47. void do_hd_request(void)
48. {
49.     if (CURRENT->cmd == WRITE) {
50.         hd_out(dev,nsect,sec,head,cyl,WIN_WRITE,&write_intr);
51.         \\
52.         port_write(HD_DATA,CURRENT->buffer,256);
53.     } else if (CURRENT->cmd == READ)
54.         hd_out(dev,nsect,sec,head,cyl,WIN_READ,&read_intr);
55. }

56. if (CURRENT->cmd == WRITE) {
530.     hd_out(dev,nsect,sec,head,cyl,WIN_WRITE,&write_intr);
531.     // printk(x, "Writing to disk - dev: %d, nsect: %lx, sec: %lx, head: %lx, cyl: %lx\n", dev, nsect, sec, head, cyl);
532.     for(i=0 ; i<3000 && !(r=inb_p(HD_STATUS)&DRQ_STAT) ; i++)
533.         /* nothing */;
534.     if (!r) {
535.         bad_rw_intr();
536.         goto repeat;
537.     }
538.     port_write(HD_DATA,CURRENT->buffer,256);
539. } else if (CURRENT->cmd == READ) {
540.     hd_out(dev,nsect,sec,head,cyl,WIN_READ,&read_intr);
541.     // printk(7, "Reading from disk - dev: %d, nsect: %lx, sec: %lx, head: %lx, cyl: %lx\n", dev, nsect, sec, head, cyl);
542. } else
543.     panic("unknown hd-command");
544. //	printk(x, "+-----+\n");
545. }

```

图 64 kernel\blk_drv\hd.c\do_hd_request

而如果是读操作，在硬盘读完数据后，又会发起 `0x2E` 中断，便会进入到 `hd_interrupt` 方法里，最终调用 `read_intr` 从数据端口读出数据到内存，

- 如果一次没读完发起读盘请求时所要求的字节数，那么直接返回，等待下次硬盘触发中断
- 否则，调用 `end_request` 方法将请求项清除掉，然后再次调用 `do_hd_request` 方法循环往复。

`end_request` 方法实际上就是唤醒发起请求的进程和因为 `request` 队列满了没有将请求项插进来的进程。随后，将当前设备的当前请求项的 `dev` 置空，并将当前请求项指向链表中的下一个请求项。

总结一下就是：

- 当设备的当前请求项为空时，也就是没有在执行的块设备请求项时，`ll_rw_block` 就会在执行到 `add_request` 方法时，直接执行 `do_hd_request` 方法发起读盘请求。
- 如果已经有在执行的请求项了，就按调度规则插入 `request` 链表中进行等待。
- `do_hd_request` 方法执行完毕后，硬盘发起读或写请求，执行完毕后会发起硬盘中断，进而调用 `read_intr` 中断处理函数。
- `read_intr` 会改变当前请求项指针指向 `request` 链表的下一个请求项，并再次调用 `do_hd_request` 方法。

最后的最后，上层进程通过 `put_fs_byte` 方法就可以一个字节一个字节地，将缓冲区里的数据，复制到用户指定的内存 `buf` 中去了。

4.6.3 磁盘调度

发出请求时，若请求队列不为空，则需要进行磁盘访问调度，按照如下要求将新请求 `request` 加入队列。

- 写请求小于读请求。
- 若请求类型相同，低设备号小于高设备号。
- 若请求同一设备，低扇区号小于高扇区号。

```
56.     for ( ; tmp->next ; tmp=tmp->next)
57.         if ((IN_ORDER(tmp,req) ||
58.             !IN_ORDER(tmp,tmp->next)) &&
59.             IN_ORDER(req,tmp->next))
60.             break;
```

4.7 信号处理

`hello` 程序执行过程中出现的异常可能有中断、陷阱、故障、终止等

如果在 `hello` 程序正在执行时，按下了键盘中的 `CTRL+C`，程序就被迫终止，并再次返回到了 `shell` 等待用户输入命令的状态。

在本部分的第 1 节中我们知道，键盘中断处理函数自然会走到处理字符的 `copy_to_cooked` 函数里。

```
61. void copy_to_cooked (struct tty_struct tty) {
62.     \\
63.     if (c == INTR_CHAR (tty)) {
64.         tty_intr (tty, INTMASK);
65.         continue;
66.     }
67.     \\
68. }
```

就是当 `INTR_CHAR` 发现字符为中断字符时，就调用 `tty_intr` 给进程发送信号。而 `tty_intr` 函数会给组号等于 `tty` 的进程组发送相应的信号（其实也就是将进程 `task_struct` 结构中的 `signal` 的相应位置 1）。

```
69. void tty_intr (struct tty_struct tty, int mask) {
70.     \\ .....
```

```

71.     for (i = 0; i < NR_TASKS; i++) {
72.         if (task[i] && task[i]->pgrp == tty->pgrp) {
73.             task[i]->signal |= mask;
74.         }
75.     }
76. }

```

现在这个进程的 `task_struct` 结构中的 `signal` 就有了对应信号位的值，那么在下次时钟中断到来时，便会通过 `timer_interrupt` 这个时钟中断处理函数，一路调用到 `do_signal` 方法。

```

void do_signal(long signr, long eax, long ebx, long ecx, long edx,
               long fs, long es, long ds,
               long eip, long cs, long eflags,
               unsigned long * esp, long ss)
{
    unsigned long sa_handler;
    long old_eip=eip;
    struct sigaction * sa = current->sigaction + signr - 1;
    int longs;
    unsigned long * tmp_esp;

    sa_handler = (unsigned long) sa->sa_handler;
    if (sa_handler==1)
        return;
    if (!sa_handler) {
        if (signr==SIGCHLD)
            return;
        else
            do_exit(1<<(signr-1));
    }
    if (sa->sa_flags & SA_ONESHOT)
        sa->sa_handler = NULL;
    *(&eip) = sa_handler;
    longs = (sa->sa_flags & SA_NOMASK)?7:8;
    *(&esp) -= longs;
    verify_area(esp,longs*4);
    tmp_esp=esp;
    put_fs_long((long) sa->sa_restorer,tmp_esp++);
    put_fs_long(signr,tmp_esp++);
    if (!(sa->sa_flags & SA_NOMASK))
        put_fs_long(current->blocked,tmp_esp++);
    put_fs_long(eax,tmp_esp++);
    put_fs_long(ecx,tmp_esp++);
    put_fs_long(edx,tmp_esp++);
    put_fs_long(es, tmp_esp++);
    put_fs_long(fs, tmp_esp++);
    put_fs_long(old_eip,tmp_esp++);
    current->blocked |= sa->sa_mask;
}

```

图 65 kernel\signal.c\do_signal

进入 `do_signal` 函数后，如果当前信号 `signr` 对应的信号处理函数 `sa_handler` 为空时，就直接调用 `do_exit` 函数退出

如果信号处理函数不为空，那么就通过将 `sa_handler` 赋值给 `eip` 寄存器，也就是指令寄存器的方式，跳转到相应信号处理函数（注册在每个进程 `task_struct` 中的 `sigaction` 数组中）处运行。

```

48 struct sigaction {
49     void (*sa_handler)(int);
50     sigset_t sa_mask;
51     int sa_flags;
52     void (*sa_restorer)(void);
53 };

```

图 66 include\signal.h\sigaction

```

77. \ union for signal handlers \
78. union __sigaction_u {
79.     void    (__sa_handler)(int);
80.     void    (__sa_sigaction)(int, struct __siginfo ,
81.                             void );
82. };

```

信号是进程间通信的一种方式，`hello` 具体运行过程中，可能产生 `SIGINT`、`SIGKILL`、`SIGSEGV`、`SIGNALARM`、`SIGCHLD` 等信号，比较常见的如：

- `Ctrl+Z`: 进程收到 `SIGSTP` 信号，`hello` 停止，此时进程并未回收，而是后台运行
- `Ctrl+C`: 进程收到 `SIGINT` 信号，`hello` 终止。

更加具体的，定义在 `include\signal.h` 中：

```

#define SIGHUP      1
13 #define SIGINT      2
14 #define SIGQUIT     3
15 #define SIGILL      4
16 #define SIGTRAP     5
17 #define SIGABRT     6
18 #define SIGIOT      6
19 #define SIGUNUSED   7
20 #define SIGFPE      8
21 #define SIGKILL     9
22 #define SIGUSR1    10
23 #define SIGSEGV    11
24 #define SIGUSR2    12
25 #define SIGPIPE    13
26 #define SIGALRM    14
27 #define SIGTERM    15
28 #define SIGSTKFLT  16
29 #define SIGCHLD    17
30 #define SIGCONT    18
31 #define SIGSTOP    19
32 #define SIGTSTP    20
33 #define SIGTTIN    21
34 #define SIGTOU    22

```

图 67 `include\signal.h`

4.7 进程退出

程序退出处理函数 `do_exit()` 会在 `exit` 系统调用的中断处理程序中被调用，其执行流程如下：

1. 首先释放当前进程的代码段和数据段所占用的内存页面。

```

83. free_page_tables(get_base(current->ldt[1]),get_limit(0x0f));
84. free_page_tables(get_base(current->ldt[2]),get_limit(0x17));

```

2. 如果当前进程有子进程，就将子进程的 `father` 字段置为 1，即把子进程的父进程改为进程 1 (`init` 进程)。
 - 如果该子进程已经处于僵死状态，则向进程 1 发送子进程终止信号 `SIGCHLD`。

```

85.     for (i=0 ; i<NR_TASKS ; i++)
86.         if (task[i] && task[i]->father == current->pid) {
87.             task[i]->father = 1;

```

```

88.         if (task[i]->state == TASK_ZOMBIE) (void) send_sig(SIGCHLD,
89.             task[1], 1);

```

3. 接着关闭当前进程打开的所有文件、释放使用的终端设备、协处理器设备。再对当前进程的工作目录 `pwd`、根目录 `root`、执行程序文件的 `i` 节点以及库文件进行同步操作，放回各个 `i` 节点并分别置空（释放）。

- 若当前进程是进程组的首进程，则还需要终止所有相关进程。

```

90.     for (i=0 ; i<NR_OPEN ; i++)
91.         if (current->filp[i]) sys_close(i);

```

4. 随后把当前进程置为僵死状态，设置退出码，并向其父进程发送终止信号 `SIGCHLD`。

5. 最后让内核重新调度任务运行

```

92.     current->state = TASK_ZOMBIE;
93.     current->exit_code = code;
94.     tell_father(current->father);
95.     schedule();
96.     return (-1); \ just to suppress warnings \

```

最终由父进程 `Shell` 等待并回收子进程，内核删除为进程创建的所有资源，进程从系统中被完全删除。

```

int do_exit(long code)
{
    int i;
    printk(4, "\n+-----+\n");
    printk(4, "exit: process %d is exiting... \n", current->pid);
    free_page_tables(get_base(current->ldt[1]),get_limit(0x0f));
    free_page_tables(get_base(current->ldt[2]),get_limit(0x17));
    for (int i = 0; i < 3; ++i) {
        printk(4, "ldt[%d]( 0~31): \t%lx\n", i, current->ldt[i].a);
        printk(4, "ldt[%d](32~63): \t%lx\n", i, current->ldt[i].b);
    }
    for (i=0 ; i<NR_TASKS ; i++)
        if (task[i] && task[i]->father == current->pid) {
            printk(4, "release child process %d\n", task[i]->pid);
            task[i]->father = 1;
            if (task[i]->state == TASK_ZOMBIE)
                /* assumption task[1] is always init */
                (void) send_sig(SIGCHLD, task[1], 1);
        }
    for (i=0 ; i<NR_OPEN ; i++)
        if (current->filp[i])
            sys_close(i);
    input(current->pwd);
    current->pwd=NULL;
    input(current->root);
    current->root=NULL;
    input(current->executable);
    current->executable=NULL;
    if (current->leader && current->tty >= 0)
        tty_table[current->tty].pgrp = 0;
    if (last_task_used_math == current)
        last_task_used_math = NULL;
    if (current->leader)
        kill_session();
    current->state = TASK_ZOMBIE;
    printk(3, "%ld\t%ld\n", current->pid, 'E', jiffies);
    printk(4, "state:\t%lx\n", current->state);
    current->exit_code = code;
    printk(4, "exit_code:\t%lx\n", current->exit_code);
    printk(4, "+-----+\n");
    tell_father(current->father);
    schedule();
    return (-1); /* just to suppress warnings */
}

```

图 68 kernel\exit.c\do_exit

五、实际调试跟踪 —— hello 的一生

首先，为方便观察，小小的修改一下 `hello.c`，加入打印自身 `pid` 信息和延长一下执行时间。

```
97. #include <stdio.h>
98. #include<unistd.h>
99.
100. int main()
101. {
102.     printf("Hello, world! my pid is %d.\n", getpid());
103.     sleep(10);
104.     return 0;
105. }
```

5.1 进程创建

这个过程中，操作系统完成的基本上就是对新进程 `task_struct` 进行赋值的功能，因此我们在创建的最后对父进程和子进程的 `task_struct` 结构体内容进行比较。观察哪些值被修改了，而哪些值没有修改。

在进程 1 调用 `init()` 函数时将文件描述符 4 与文件 `\var\fork.log` 绑定，此后在调用 `find_empty_process` 和 `copy_process` 时进行打印即可

```
106. int copy_process(int nr,long ebp,long edi,.....)
107. {
108.     \\
109.     \\ .....
110.     \\ 打印结构体内容，打印过程中新进程不会被调度.....
111.     p->state = TASK_RUNNING; \\ do this last, just in case \
112.     return last_pid;
113. }
114.
115. int find_empty_process(void)
116. {
117.     \\
118.     for(i=1 ; i<NR_TASKS ; i++)
119.         if (!task[i]){
120.             fprintf(4, "+-----\n");
121.             fprintf(4, "fork: find a empty space %d in task\n", i);
122.             return i;
123.         }
124.     }
125. void print_parent_child(struct task_struct parent, struct task_struct
126. child) {
127.     fprintf(4, "Parent Process:(hex)\tChild Process:(hex)\n");
128.     fprintf(4, "state:\t\t%lx\t\t%lx\n", parent->state, child->state);
129.     fprintf(4, "counter:\t%lx\t\t%lx\n", parent->counter, child-
```

```
>counter);
130.     printk(4, "priority:\t%lx\t\t%lx\n", parent->priority, child-
>priority);
131.     printk(4, "exit_code:\t%x\t\t%x\n", parent->exit_code, child-
>exit_code);
132.     printk(4, "start_code:\t%lx\t\t%lx\n", parent->start_code, child-
>start_code);
133.     printk(4, "end_code:\t%lx\t\t%lx\n", parent->end_code, child-
>end_code);
134.     printk(4, "end_data:\t%lx\t\t%lx\n", parent->end_data, child-
>end_data);
135.     printk(4, "brk:\t\t%lx\t\t%lx\n", parent->brk, child->brk);
136.     printk(4, "start_stack:\t%lx\t\t%lx\n", parent->start_stack,
child->start_stack);
137.     printk(4, "pid:\t\t%lx\t\t%lx\n", parent->pid, child->pid);
138.     printk(4, "father:\t\t%lx\t\t%lx\n", parent->father, child-
>father);
139.     printk(4, "pgrp:\t\t%lx\t\t%lx\n", parent->pgrp, child->pgrp);
140.     printk(4, "session:\t%lx\t\t%lx\n", parent->session, child-
>session);
141.     printk(4, "leader:\t\t%lx\t\t%lx\n", parent->leader, child-
>leader);
142.     printk(4, "uid:\t\t%lx\t\t%lx\n", parent->uid, child->uid);
143.     printk(4, "euid:\t\t%lx\t\t%lx\n", parent->euid, child->euid);
144.     printk(4, "suid:\t\t%lx\t\t%lx\n", parent->suid, child->suid);
145.     printk(4, "gid:\t\t%lx\t\t%lx\n", parent->gid, child->gid);
146.     printk(4, "egid:\t\t%lx\t\t%lx\n", parent->egid, child->egid);
147.     printk(4, "sgid:\t\t%lx\t\t%lx\n", parent->sgid, child->sgid);
148.     printk(4, "alarm:\t\t%lx\t\t%lx\n", parent->alarm, child->alarm);
149.     printk(4, "utime:\t\t%lx\t\t%lx\n", parent->utime, child->utime);
150.     printk(4, "stime:\t\t%lx\t\t%lx\n", parent->stime, child->stime);
151.     printk(4, "cutime:\t\t%lx\t\t%lx\n", parent->cutime, child-
>cutime);
152.     printk(4, "cstime:\t\t%lx\t\t%lx\n", parent->cstime, child-
>cstime);
153.     printk(4, "start_time:\t\t%lx\t\t%lx\n", parent->start_time, child-
>start_time);
154.     printk(4, "used_math:\t\t%lx\t\t%lx\n", parent->used_math, child-
>used_math);
155.     printk(4, "tty:\t\t%d\t\t%d\n", parent->tty, child->tty);
156.     printk(4, "umask:\t\t%lx\t\t%lx\n", parent->umask, child->umask);
157.     printk(4, "pwd:\t\t%lx\t\t%lx\n", parent->pwd, child->pwd);
158.     printk(4, "root:\t\t%lx\t\t%lx\n", parent->root, child->root);
159.     printk(4, "executable:\t\t%lx\t\t%lx\n", parent->executable, child-
>executable);
160.     printk(4, "close_on_exec:\t\t%lx\t\t%lx\n", parent->close_on_exec,
child->close_on_exec);
161.     for (int i = 0; i < 3; ++i) {
162.         printk(4, "ldt[%d]( 0~31):\t%lx\t\t%lx\n", i, parent-
>ldt[i].a, child->ldt[i].a);
163.         printk(4, "ldt[%d](32~63):\t%lx\t\t%lx\n", i, parent-
```

```
        >ldt[i].b, child->ldt[i].b);
164.    }
165.    fprintf(4,
166.            "=====\\n");
166.    fprintf(4, "tss.back_link:\t%lx\t\t%lx\\n", parent->tss.back_link,
167.             child->tss.back_link);
167.    fprintf(4, "tss.esp0:\t%lx\t\t%lx\\n", parent->tss.esp0, child-
168.             >tss.esp0);
168.    fprintf(4, "tss.ss0:\t%lx\t\t%lx\\n", parent->tss.ss0, child-
169.             >tss.ss0);
169.    fprintf(4, "tss.esp1:\t%lx\t\t%lx\\n", parent->tss.esp1, child-
170.             >tss.esp1);
170.    fprintf(4, "tss.ss1:\t%lx\t\t%lx\\n", parent->tss.ss1, child-
171.             >tss.ss1);
171.    fprintf(4, "tss.esp2:\t%lx\t\t%lx\\n", parent->tss.esp2, child-
172.             >tss.esp2);
172.    fprintf(4, "tss.ss2:\t%lx\t\t%lx\\n", parent->tss.ss2, child-
173.             >tss.ss2);
173.    fprintf(4, "tss.cr3:\t%lx\t\t%lx\\n", parent->tss.cr3, child-
174.             >tss.cr3);
174.    fprintf(4, "tss.eip:\t%lx\t\t%lx\\n", parent->tss.eip, child-
175.             >tss.eip);
175.    fprintf(4, "tss.eflags:\t%lx\t\t%lx\\n", parent->tss.eflags, child-
176.             >tss.eflags);
176.    fprintf(4, "tss.eax:\t%lx\t\t%lx\\n", parent->tss.eax, child-
177.             >tss.eax);
177.    fprintf(4, "tss.ecx:\t%lx\t\t%lx\\n", parent->tss.ecx, child-
178.             >tss.ecx);
178.    fprintf(4, "tss.edx:\t%lx\t\t%lx\\n", parent->tss.edx, child-
179.             >tss.edx);
179.    fprintf(4, "tss.ebx:\t%lx\t\t%lx\\n", parent->tss.ebx, child-
180.             >tss.ebx);
180.    fprintf(4, "tss.esp:\t%lx\t\t%lx\\n", parent->tss.esp, child-
181.             >tss.esp);
181.    fprintf(4, "tss.ebp:\t%lx\t\t%lx\\n", parent->tss.ebp, child-
182.             >tss.ebp);
182.    fprintf(4, "tss.esi:\t%lx\t\t%lx\\n", parent->tss.esi, child-
183.             >tss.esi);
183.    fprintf(4, "tss.edi:\t%lx\t\t%lx\\n", parent->tss.edi, child-
184.             >tss.edi);
184.    fprintf(4, "tss.es:\t\t%lx\t\t%lx\\n", parent->tss.es, child-
185.             >tss.es);
185.    fprintf(4, "tss.cs:\t\t%lx\t\t%lx\\n", parent->tss.cs, child-
186.             >tss.cs);
186.    fprintf(4, "tss.ss:\t\t%lx\t\t%lx\\n", parent->tss.ss, child-
187.             >tss.ss);
187.    fprintf(4, "tss.ds:\t\t%lx\t\t%lx\\n", parent->tss.ds, child-
188.             >tss.ds);
188.    fprintf(4, "tss.fs:\t\t%lx\t\t%lx\\n", parent->tss.fs, child-
189.             >tss.fs);
189.    fprintf(4, "tss.gs:\t\t%lx\t\t%lx\\n", parent->tss.gs, child-
```

```

        >tss.gs);
190.      printk(4, "tss.ldt:\t%lx\t\t%lx\n", parent->tss.ldt, child-
        >tss.ldt);
191.      printk(4, "tss.trace_bitmap:\t%lx\t\t%lx\n", parent-
        >tss.trace_bitmap, child->tss.trace_bitmap);
192.      printk(4, "+-----\n");
193.
194. }
195.
196.

```

打印出来提取和 `hello.c` 相关的内容出来如下（省略部分内容，完整内容见附件）：

```

+-----+
fork: find a empty space 4 in task
Parent Process:(hex) Child Process:(hex)
state:      0      2
counter:    c      f
priority:   f      f
exit_code:  0      0
start_code: 8000000  10000000
end_code:   41000    41000
end_data:   45000    45000
brk:       5e400    5e400
start_stack: 3fff000  3fff000
pid:       4      6
father:    1      4
.....
start_time: 55      346
pwd:       2be98    2be98
root:      2ba38    2ba38
executable: 2bdb8    2bdb8
ldt[1]( 0~31): 40      40
ldt[1](32~63): 8c0fb00  10c0fb00
ldt[2]( 0~31): 3fff    3fff
ldt[2](32~63): 8c0f300  10c0f300
=====
tss.esp0:  ffe000    f9d000
tss.ss0:   10      10
.....
tss.eip:   6ffb      398af
tss.eflags: 206      246
tss.eax:   24200    0
.....
tss.cs:    8      f
tss.ss:   10      17
tss.ldt:  48      68

```

```
+-----+
```

可以看到, `hello.c` 的进程的 `task_struct` 与其父进程基本相似, 但存在一些不同, 我们对照第二部分第二节的理论部分进行再验证:

1. 为新进程申请一页内存页, 调用 `get_free_page` 函数遍历 `mem_map[]` 数组, 发现第 5 个位置(从 0 开始计算到 4)可供申请(接下来是在 `copy_process` 中将父进程的 `task_struct` 的全部值都复制给即将创建的进程 `p`, 然后对其中某些值进行修改)
2. 为进程分配一个唯一的进程标识符 6, 把当前进程 4 作为新进程的父进程
3. 清除信号位图并复位新进程各统计值, 并设置初始运行时间片 `counter` 为 15
4. 根据当前进程设置任务状态段(TSS)中各寄存器的值, 主要包括:
 - 由于创建进程时新进程返回值应为 0, 所以需要设置 `tss.eax = 0`
 - 新建进程内核态堆栈指针 `tss.esp0` 被设置成新进程任务数据结构所在内存页面的顶端 `0xf9d000`, 而堆栈段 `tss.ss0` 被设置成内核数据段选择符 `0x10`。
 - `tss.ldt` 被设置为局部表描述符在 `GDT` 中的索引值 `0x68`。
5. 设置新任务的代码和数据段基址、限长, 并复制当前进程内存分页管理的页表。
 - LDT 的复制和改造(主要是 `ldt[1]` 和 `ldt[2]`), 使得不同进程分别映射到了不同的线性地址空间。段基址取决于当前进程号, 为 `nr 64M = 0x1000 0000`
 - `ldt[1]`: 代码段, 段限长就是取自父进程设置好的段限长 `0x40`(`G=1`, 对应 260 KB)
 - `ldt[2]`: 数据段, 段限长就是取自父进程设置好的段限长 `0x3fff`(`G=1`, 对应 64 MB)
 - 页表的复制, 使得不同进程又从不同的线性地址空间被映射到了相同的物理地址空间。也就是让 `0x1000 0000` 和 `0x0800 0000` 的相同偏移位置的页表项内数据一样即可
 - 同时将新老进程的页表都变成只读状态, 为后面写时复制的缺页中断做准备。
6. 把当前进程(父进程)的打开的文件, 如 `pwd`、`root` 和 `executable` 这些 `i` 节点的引用次数都增 1。
7. 随后在 `GDT` 表中设置新任务 TSS 段和 LDT 段描述符项。这两个段的限长均被设置成 104 字节。
8. 将新进程的状态置为 `TASK_RUNNING`, 可以被加入调度队列。

下面验证页表的复制过程, 在 `/mm/memory.c` 的 `copy_page_tables` 函数中加入打印输出到 `var/fork_page_copy.log` 如下:

```
197. int copy_page_tables(unsigned long from,unsigned long to,long size)
198. {
199.     for( ; size-->0 ; from_dir++,to_dir++) {
200.         // .....
201.         to_dir = ((unsigned long) to_page_table) | 7;
202.         fprintf(5, "from_dir:%lx--%lx\t\nto_dir:%lx--%lx\t\t\n",
203.                 from_dir, from_dir, to_dir, to_dir);
204.         nr = (from==0)?0xA0:1024;
205.         for ( ; nr-- > 0 ; from_page_table++,to_page_table++) {
206.             // .....
207.             fprintf(5, "\tfrom_page_table:%lx--%lx\t\t\nto_page_table:%lx--%lx\t\t\n",
208.                     from_page_table, from_page_table, to_page_table,
209.                     to_page_table);
```

最终的输出文件 `/var/fork_page_copy.log` 中如下:

```
from_dir:80--ff5027      to_dir:100--f9b007
from_page_table:ff5000--ff9025      to_page_table:f9b000--ff9025
```

```

from_page_table:ff5004--fe8025      to_page_table:f9b004--fe8025
from_page_table:ff5008--fca025      to_page_table:f9b008--fca025
.....

```

```

1165 +-----+
1166 from_dir:80--ff7027      to_dir:100--fac007
1167   from_page_table:ff7000--ffa025      to_page_table:fac000--ffa025
1168   from_page_table:ff7004--fec025      to_page_table:fac004--fec025
1169   from_page_table:ff7008--fce025      to_page_table:fac008--fce025
1170   from_page_table:ff700c--fcf025      to_page_table:fac00c--fcf025
1171   from_page_table:ff7010--fd0025      to_page_table:fac010--fd0025
1172   from_page_table:ff7014--fcd025      to_page_table:fac014--fcd025
1173   from_page_table:ff7018--fcbb025      to_page_table:fac018--fcbb025
1174   from_page_table:ff701c--ff3025      to_page_table:fac01c--ff3025
1175   from_page_table:ff7020--fdf025      to_page_table:fac020--fdf025
1176   from_page_table:ff7024--fca025      to_page_table:fac024--fca025
1177   from_page_table:ff7028--fc9025      to_page_table:fac028--fc9025
1178   from_page_table:ff702c--fc8025      to_page_table:fac02c--fc8025
1179   from_page_table:ff7030--fd8025      to_page_table:fac030--fd8025
1180   from_page_table:ff7034--fde025      to_page_table:fac034--fde025
1181   from_page_table:ff7038--fdb025      to_page_table:fac038--fdb025
1182   from_page_table:ff703c--fd6025      to_page_table:fac03c--fd6025
1183   from_page_table:ff7040--fe2025      to_page_table:fac040--fe2025
1184   from_page_table:ff7044--fc5025      to_page_table:fac044--fc5025
1185   from_page_table:ff7048--fd5025      to_page_table:fac048--fd5025
1186   from_page_table:ff704c--fc4025      to_page_table:fac04c--fc4025
1187   from_page_table:ff7050--fc3025      to_page_table:fac050--fc3025
1188   from_page_table:ff7054--fdd025      to_page_table:fac054--fdd025
1189   from_page_table:ff7058--fda025      to_page_table:fac058--fda025
1190   from_page_table:ff7064--fe4025      to_page_table:fac064--fe4025
1191   from_page_table:ff7068--ff1025      to_page_table:fac068--ff1025
1192   from_page_table:ff706c--fe1025      to_page_table:fac06c--fe1025

```

图 69 fork_page_copy

可以看到两个页表里的数据一模一样，同时会跳过一些没有使用的页表，大概是这样：

- 如果源目录项无效，即指定的页表不存在（存在位 $P=0$ ），会跳过此项；
- 对于页表项，只复制项内容不为 0 的（存在位 $P=0$ ，则该表项对应的页面可能在交换设备中）。

5.2 进程加载

这个过程中，操作系统完成的是待加载文件的文件头的读取和内存上的调整，我们首先将加载出的 `exec` 文件头打印出来看看，修改的过程和上面一样，绑定到 `var/execve.log` 文件然后标准化输出（修改 `fs/exec.c` 的 `do_execve(.....)` 函数）。

```

207.   ex = ((struct exec ) bh->b_data); / read exec-header /
208.   printk(6, "file %s head information as exec:\n", filename);
209.   printk(6, "\ta_magic: \t%lx\n", ex.a_magic);
210.   // .....

```

根据文件名，按 `inode` 找到并读取文件里的内容，解析开头 1KB 的数据为 `exec` 结构，得到文件头的信息如下（按 16 进制打印 `exec` 结构体）：

- 根据 `a_text` 和 `a_data` 可以得出代码段占 12 KB，数据段占 4 KB
- `a_entry` 字段指定了程序代码开始执行的地址 `0x0`（对应线性地址空间的起始位置）
- 对于可执行文件来说并不需要重定位信息，因此执行文件中的 `a_trsize` 和 `a_drsiz` 字段的值为 `0`

`current pid: 6`

`argc: 1 envc: 10`

```
file head information as exec:  
    a_magic: 10b      // 执行文件魔数。使用 N_MAGIC 等宏访问。  
    a_text: 4000     // 代码长度, 字节数  
    a_data: 1000     // 数据长度, 字节数  
    a_bss: 0        // 文件中的未初始化数据区长度, 字节数  
    a_syms: 7a4 // 文件中的符号表长度, 字节数  
    a_entry: 0      // 执行开始地址  
    a_trsize: 0     // 代码重定位信息长度, 字节数  
    a_drsize: 0     // 数据重定位信息长度, 字节数
```

```
Open [+]
execve.log [Read-Only]
64 MB Volume ~/oslab/hdc/var
Save
☰
×
271 +-----+
272
273 +-----+
274 current pid: 16
275 argc: 1 envc: 10
276 file head information as exec:
277     a_magic:      10b
278     a_text:       4000
279     a_data:       1000
280     a_bss:        0
281     a_syms:      7a4
282     a_entry:      0
283     a_trsize:     0
284     a_drsize:     0
285 ****
286 current task information after modified:
287     ldt[1]( 0-31): 3
288     ldt[1](32-63): 10c0fb00
289     ldt[2]( 0-31): 3fff
290     ldt[2](32-63): 10c0f300
291     current->brk: 5000
292     current->end_data: 5000
293     current->end_code: 4000
294     current->start_stack: 3ffff000
295     old eip:        3986c 3ffffd1c
296     new eip:        0      3fffff00
297
298 +-----+
```

图 70 execve.log

最后将原调用系统中断的程序在堆栈上的代码指针替换为指向新执行程序的入口点，并将栈指针替换为新执行文件的栈指针。此后返回指令将弹出这些栈数据并使得 CPU 去执行新执行文件，而不会返回到原调用系统中断的程序中去了。

也就是设置 `eip` 和 `esp`, 代码指针 `eip` 决定了 CPU 将执行哪一段指令, 栈指针 `esp` 决定了 CPU 压栈操作的位置。

- 重新设置了代码指针 `eip` 的值，指向文件的头结构 `exec` 中的 `a_entry` 字段 `0x0`，表示该程序的入口地址。
 - 重新设置了栈指针 `esp` 的值，指向了刚刚得到的进程栈开始处 `0x3fff000`。

中断返回后，也就是 `do_execve` 这个函数 `return` 之后，就会寻找中断返回前的这几个值（包括 `eip` 和 `esp`）进行恢复吗，由此程序便不会返回到原调用系统中断的程序中去了。

5.3 缺页中断

我们仅仅将 `hello` 文件的头部加载到了内存，其他部分并没有进行加载，因此在访

问 `start_code` 这个线性地址时，会遇到了页表项的存在位 `P` 等于 `0` 的情况。

一旦遇到了这种情况，CPU 会触发一个中断：页错误（`Page-Fault`），而由于缺页引起的页异常中断，会通过调用 `do_no_page(error_code, address)` 来处理。

这里我们暂时忽略磁盘读写的内容，先来看看地址转换和发生缺页中断后如何将新的页建立起映射的，在 `mm/memory.c` 的 `do_no_page(error_code, address)` 函数和 `put_page(page, address)` 函数（`do_no_page` 中调用）在 `var/page_fault.log` 中进行打印，修改如下：

```
211. unsigned long put_page(unsigned long page,unsigned long address)
212. {
213.     // .....
214.     page_table = (unsigned long ) ((address>>20) & 0xffc);
215.     if ((page_table)&1){
216.         printk(7, "page_catalogue_table(p=1): \t%lx->%lx\n", page_table,
217.             page_table);
218.         page_table = (unsigned long ) (0xfffff000 & page_table);
219.     }
220.     else {
221.         if (!(tmp=get_free_page()))    return 0;
222.         page_table = tmp|7;
223.         printk(7, "page_catalogue_table(p=0, new allocate): \t%lx-
224.             >%lx\n", page_table, page_table);
225.         page_table = (unsigned long ) tmp;
226.     }
227.     page_table[(address>>12) & 0x3ff] = page | 7;
228.     printk(7, "page_table: \t%lx->%lx\n", page_table,
229.         page_table[(address>>12) & 0x3ff]);
230.     return page;
231. }
232. void do_no_page(unsigned long error_code,unsigned long address)
233. {
234.     // .....
235.     if (!(page = get_free_page()))
236.         oom();
237.     printk(7, "pid: %d\n", current->pid);
238.     printk(7, "page fault happened at address: %lx\n", address_temp);
239. }
```

于是我们便可以打印发生缺页中断的地方和如何申请内存并建立页表映射的：

- 进入 `hello` 代码的第一行便会中断（代码和数据还未拷入）
- 首先申请一个页框地址为 `0xf9a000`
- 然后根据发生缺页的地址，按照线性地址转换为物理地址的方式去设定页目录项和页表项
 - 若对应页目录项不存在 (`p=0`)，需要额外为此页表申请一页内存
 - 若对应页目录项存在 (`p=1`)，直接取存储在该地址的数据即可

`pid: 6`

`page fault happened at address: 10000000`

```

new allocate page address: f9a000
page_catalogue_table(p=0, new allocate): 100->f99007
page_table: f99000->f9a007

```

```

2922
2923 page_catalogue_table(p=1): 17c->f90027
2924 page_table: f90000->f63007
2925 page_catalogue_table(p=1): 13c->fa7027
2926 page_table: fa7000->f8b007
2927 page_catalogue_table(p=1): 100->fa9027
2928 page_table: fa9000->f85007
2929 page_catalogue_table(p=0, new allocate): 13c->fcc007
2930 page_table: fcc000->f90007
2931 +-----+
2932 pid: 16
2933 page fault happened at address: 10000000
2934 new allocate page address: fac000
2935 page_catalogue_table(p=0, new allocate): 100->faa007
2936 page_table: faa000->fac007
2937 address_convert: page_catalogue_table 100 -> faa000[0] -> (fac007 & 0xffffffff000) = fac000 + 0 = fac000
2938 +-----+
2939
2940 +-----+
2941 pid: 16
2942 page fault happened at address: 10004000
2943 new allocate page address: fa9000
2944 page_catalogue_table(p=1): 100->faa027
2945 page_table: faa000->fa9007
2946 address_convert: page_catalogue_table 100 -> faa000[4] -> (fa9007 & 0xffffffff000) = fa9000 + 0 = fa9000
2947 +-----+

```

图 71 page_fault.log

那么我们接下来再来打印一下线性地址转换为物理地址的转换过程（Linux0.11 采用的二级页表）：

```

240. void address_convert(unsigned long address)
241. {
242.     unsigned long page_table, page_catalogue_table;
243.     unsigned long offset = address & 0x00000fff;
244.     page_catalogue_table = (unsigned long ) ((address>>20) & 0xfffc);
245.     page_table = (unsigned long ) (0xfffff000 & page_catalogue_table);
246.     unsigned long page = page_table[(address>>12) & 0x3ff];
247.     unsigned long final = (page & 0xfffff000) + offset;
248.     printk(7, "address_convert: page_catalogue_table %lx -> %lx[%lx] ->
(%lx & 0xffffffff000) = %lx + %lx = %lx\n", page_catalogue_table,
page_table, ((address>>12) & 0x3ff), page, page & 0xfffff000, offset,
final);
249. }

```

对于地址 0x10000000，则其的转化过程如下（对照转化图来看）：

```

address_convert: page_catalogue_table 100 -> f99000[0] -> (f9a007 &
0xffffffff000) = f9a000 + 0 = f9a000

```

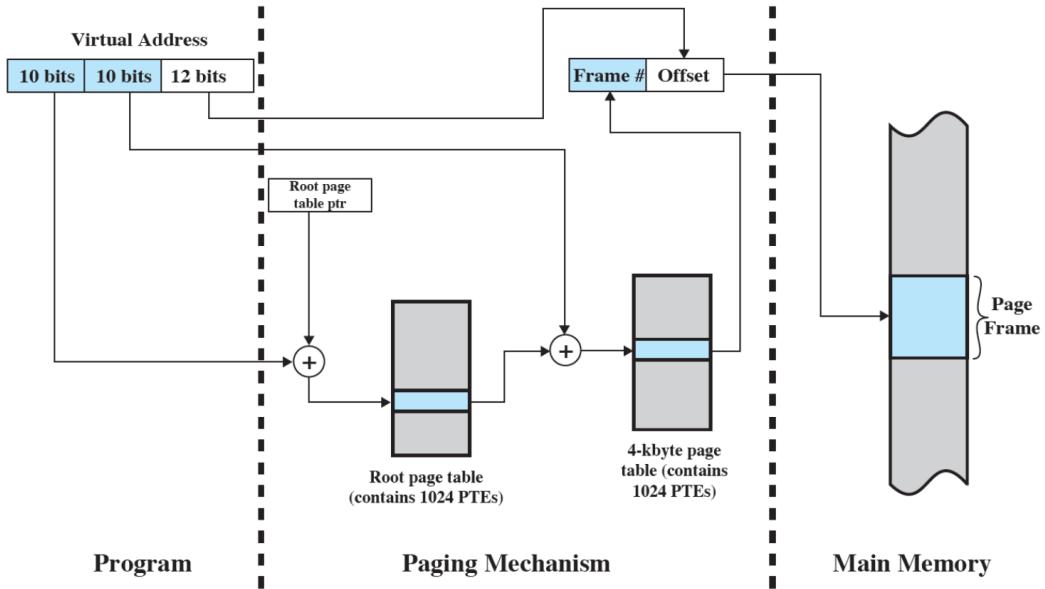


图 72 二级页表

而在 `hello` 的总执行过程中，一共进行了 4 次缺页中断，尽管在 `execve` 阶段加载了一页 4 KB 的文件头，加载的总页面大小仍小于其 25 KB 的文件大小，这也侧面验证了为什么操作系统不必一次将所有内容加载进内存，因为有的页可能根本不会访问到（局部性原理：程序倾向于在相对较小的区域内集中访问数据，而不是在整个地址空间中随机访问）。

5.4 进程调度

进程 1 调用 `init()`。在 `init()` 中：

```

250. setup((void ) &drive_info);           //加载文件系统
251. (void) open("/dev/tty0",O_RDWR,0);    //打开/dev/tty0, 建立文件描述符 0 和
   /dev/tty0 的关联
252. (void) dup(0);                      //让文件描述符 1 也和/dev/tty0 关联
253. (void) dup(0);                      //让文件描述符 2 也和/dev/tty0 关联
254. (void) open("/var/process.log", O_CREAT|O_TRUNC|O_WRONLY, 0666);

```

打开 `log` 文件的参数的含义是建立只写文件，如果文件已存在则清空已有内容。文件的权限是所有人可读可写。

修改 `kernel\printk.c\fprintf()` 函数：

```

255. {
256.     // if (!(file=task[0]->filp[fd]))      / 从进程 0 的文件描述符表中
   得到文件句柄 /
257.     //      return 0;
258.
259.     // 修改为如下:
260.     // task[1]->filp is not ready or f_inode->i_dev is not ready
261.     if (!(file=task[1]->filp[fd]) || !task[1]->filp[fd]->f_inode-
   >i_dev) {    / 从进程 1 的文件描述符表中得到文件句柄 /
262.         return 0;
263.     }
264.     inode=file->f_inode;

```

```

265.
266.         __asm__ ("push %%fs\n\t"
267.                     "push %%ds\n\t"
268.                     "pop %%fs\n\t"
269.                     "pushl %0\n\t"
270.                     "pushl $logbuf\n\t"
271.                     "pushl %1\n\t"
272.                     "pushl %2\n\t"
273.                     "call file_write\n\t"
274.                     "addl $12,%esp\n\t"
275.                     "popl %0\n\t"
276.                     "pop %%fs"
277.                     ::"r" (count), "r" (file), "r" (inode):"ax","cx","dx");
278.     }
279.     return count;

```

从进程 1 的文件描述符表中获取一个文件句柄，如果该文件句柄或其对应的设备 inode 未准备好，则返回 0；否则，代码会使用汇编语言调用 file_write 函数进行写入操作，同时设置相关的寄存器和栈，以便正确处理写入过程，并在完成后返回写入的字节数。

对 copy_process() 增加创建态“N”和就绪态“J”

```

280.     struct task_struct p;
281.     // .....
282.     p = (struct task_struct ) get_free_page(); //获得一个 task_struct
结构体空间
283.     // .....
284.     p->pid = last_pid;
285.     // .....
286.     p->start_time = jiffies; //设置 start_time 为 jiffies
287.     / 进程被创建 /
288.     printk(3, "%ld\t%c\t%ld\n", last_pid, 'N', jiffies);
289.     // .....
290.     / 进程已经准备好了相应的 TSS 段中的内容，进入就绪态 /
291.     p->state = TASK_RUNNING;
292.     printk(3, "%ld\t%c\t%ld\n", last_pid, 'J', jiffies);
293.     return last_pid;

```

在 schedule 中增加运行态“R”和就绪态“J”

```

294. void schedule(void)
295. {
296.     int i,next,c;
297.     struct task_struct p;
298.
299.     / check alarm, wake up any interruptible tasks that have got a signal
/
300.     for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
301.         if (p) {
302.             if ((p)->alarm && (p)->alarm < jiffies) {
303.                 (p)->signal |= (1<<(SIGALRM-1));
304.                 (p)->alarm = 0;
305.             }
306.             if (((p)->signal & ~(_BLOCKABLE & (p)->blocked)) &&

```

```

307.         (p)->state==TASK_INTERRUPTIBLE){
308.             (p)->state=TASK_RUNNING;
309.             / 唤醒接收到信号且处于可中断睡眠状态的任务 /
310.             fprintf(3, "%ld\t%c\t%ld\n", (p)->pid, 'J', jiffies);
311.         }
312.     }
313.
314.     / this is the scheduler proper: /
315.     while (1) {
316.         c = -1;
317.         next = 0;
318.         i = NR_TASKS;
319.         p = &task[NR_TASKS];
320.         while (--i) {
321.             if (!--p)
322.                 continue;
323.             if ((p)->state == TASK_RUNNING && (p)->counter > c)
324.                 c = (p)->counter, next = i;
325.         }
326.         if (c) break;
327.         for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
328.             if (p)
329.                 (p)->counter = ((p)->counter >> 1) + (p)->priority;
330.     }
331.     / 如果下一个进程不是当前进程，则将当前进程切换至就绪态，调度选择的下一个进
程切换至运行态 /
332.     if (task[next]->pid != current->pid) {
333.         if (current->state == TASK_RUNNING)
334.             fprintf(3, "%ld\t%c\t%ld\n", current->pid, 'J', jiffies);
335.         fprintf(3, "%ld\t%c\t%ld\n", task[next]->pid, 'R', jiffies);
336.     }
337.     switch_to(next);
338. }
```

在 `sleep_on` 中增加阻塞态 “W” 和就绪态 “J”

```

339. void sleep_on(struct task_struct p)
340. {
341.     struct task_struct tmp;
342.
343.     if (!p)
344.         return;
345.     if (current == &(init_task.task))
346.         panic("task[0] trying to sleep");
347.     tmp = p;
348.     p = current;
349.     current->state = TASK_UNINTERRUPTIBLE;
350.     / 进程切换到非中断睡眠态 /
351.     fprintf(3, "%ld\t%c\t%ld\n", current->pid, 'W', jiffies);
352.     schedule();
353.     if (tmp){
354.         tmp->state=0;
355.         / 将队列中的上一个 (tmp) 睡眠进程唤醒，切换到就绪态 /
```

```
356.     fprintf(3, "%ld\t%c\t%ld\n", tmp->pid, 'J', jiffies);
357. }
358. }
```

在 `interruptible_sleep_on` 中增加阻塞态 “W” 和就绪态 “J”

```
359. void interruptible_sleep_on(struct task_struct p)
360. {
361.     struct task_struct tmp;
362.
363.     if (!p)
364.         return;
365.     if (current == &(init_task.task))
366.         panic("task[0] trying to sleep");
367.     tmp=p;
368.     p=current;
369.     repeat: current->state = TASK_INTERRUPTIBLE;
370.             / 进程切换到可中断睡眠态 /
371.             fprintf(3, "%ld\t%c\t%ld\n", current->pid, 'W', jiffies);
372.             schedule();
373.             if (p && p != current) {
374.                 (p).state=0;
375.                 / 唤醒队列头 /
376.                 fprintf(3, "%ld\t%c\t%ld\n", (p)->pid, 'J', jiffies);
377.                 goto repeat;
378.             }
379.             p=NULL;
380.             if (tmp){
381.                 tmp->state=0;
382.                 / 将队列中的上一个 (tmp) 睡眠进程切换到就绪态 /
383.                 fprintf(3, "%ld\t%c\t%ld\n", tmp->pid, 'J', jiffies);
384.             }
385. }
```

在 `wake_up` 中增加就绪态 “J”

```
386. void wake_up(struct task_struct p)
387. {
388.     if (p && p) {
389.         / 将处于不可中断睡眠状态的任务唤醒 /
390.         if((p).state != 0)
391.             fprintf(3, "%ld\t%c\t%ld\n", (p)->pid, 'J', jiffies);
392.         (p).state=0;
393.         p=NULL;
394.     }
395. }
```

在 `sys_pause` 中增加阻塞态 “W”

```
396. int sys_pause(void)
397. {
398.     current->state = TASK_INTERRUPTIBLE;
399.     / 当前进程进入睡眠状态 /
400.     if (current->pid != 0)
401.         fprintf(3, "%ld\t%c\t%ld\n", current->pid, 'W', jiffies);
402.     schedule();
```

```
403.     return 0;
404. }
```

在 `sys_waitpid` 中增加阻塞态 “W”

```
405. int sys_waitpid(pid_t pid,unsigned long stat_addr, int options)
406. {
407.     // .....
408.     if (flag) {
409.         if (options & WNOHANG)
410.             return 0;
411.         current->state=TASK_INTERRUPTIBLE;
412.         / 挂起当前进程 /
413.         fprintf(3, "%ld\t%c\t%ld\n", current->pid, 'W', jiffies);
414.         schedule();
415.         if (!(current->signal &=~(1<<(SIGCHLD-1))))
416.             goto repeat;
417.         else
418.             return -EINTR;
419.     }
420.     return -ECHILD;
421. }
```

在 `do_exit` 中增加退出态 “E”

```
422. int do_exit(long code)
423. {
424.     // .....
425.     current->state = TASK_ZOMBIE;
426.     fprintf(3, "%ld\t%c\t%ld\n", current->pid, 'E', jiffies);
427.     current->exit_code = code;
428.     tell_father(current->father);
429.     schedule();
430.     return (-1);    / just to suppress warnings /
431. }
```

Line	Process ID	Status	Time
1	1	N	45
2	2	N	46
3	2	J	47
4	1	W	47
5	2	R	47
6	3	N	64
7	3	J	65
8	2	E	70
9	1	J	70
10	1	R	70
11	4	N	70
12	4	J	71
13	1	W	71
14	3	R	71
15	3	W	77
16	4	R	77
17	5	N	113
18	5	J	114
19	4	W	115
20	5	R	115
21	4	J	117
22	5	E	118
23	4	R	118
24	4	W	124
25	0	R	124

图 73 process.log

```

guojun@guojunos:~/oslab/files$ python stat_log.py process.log -g |more
(Unit: tick)
Process   Turnaround   Waiting   CPU Burst   I/O Burst
 0        8407         0         0          0
 1        33           0         1          30
 2        31           6         24          0
 3       6017          4         12         6000
 4       8381          54         53         8273
 5         4           0         3          0
 6      5523          329        38          38
 8      5525          4523        900         101
 9      5418          4408        800         209
10      5217          4206        701         309
11      4920          3904        600         416
12      4523          3501        500         521
13      4006          2979        400         626
14      3409          2377        300         731
15      2706          1672        200         834
16      1916          870         100         945
17         4           0         3          0
18         2           1         0          0
Average:    3767.32    1745.89
Throughput: 0.23/s
--More--

```

图 74 stat_log.py

可以看到在实际执行过程中，操作系统会高速切换轮转运行多个进程（控制权转移），对于长进程来说因为因时间片超时而产生的进程调度较多，因而被调度次数会大大增加。

```

923 #
924 #
925 #
926 #
927 #
928 #
929 #
930 #
931 #
932 #
933 #
934 #
935 #
936 #
937 #
938 #
939 #

```

图 75 切换过程

但是，具体跟踪一下会发现在进程 6 执行过程中，有一段时间进程 4 和 6 会同时处于等待状态，这里应该是进程 6 在执行 `sleep(10)` 这句话，而进程 4 在等待进程 6 执行完毕，此时系统又没有其他进程处于就绪态，因而系统不进行任何调度进行空转，同时保持对新到来的就绪态进程的执行准备。

5.5 文件系统和硬盘访问

根据 `task_struct` 的结构，执行文件 `i` 节点结构指针保存在当前进程 `current->executable` 中，当前工作目录 `i` 节点结构指针保存在 `current->pwd` 中

获得磁盘数据大致分为 3 步：（这里也调用了 `put_page` 就解释了为什么在第三步

缺页中断的日志文件中会有一些其他的信息打印)

定位 `bmap` -> 读盘 `bread`(涉及基于缓冲区的读写) -> 装页 `put_page`
那么同样在下列函数之中打印我们想要的信息

- `fs/inode.c` 中的 `bmap()` 函数打印寻找磁盘块号过程;
- `fs/buffer.c` 中的 和 `getblk()` (在 `bread()` 中被调用) 打印查询缓冲过程;
- `kernel/blk_drv/ll_rw_blk.c` 中的 `make_request()` 和 `add_request()` 打印创建磁盘访问请求并加入队列相关信息;
- `kernel/blk_drv/hd.c` 中的 `do_hd_request()` 打印磁盘读写相关信息;

这里一部分函数加上 `fprintk` 打印后会报错 (有时不报错, 但是会卡死在初始化的过程), 反复调试仍无法解决, 推测可能是由于在文件读写过程中加入 `fprintk` 继续进行文件读写操作从而导致自身的递归调用。

`Kernel panic: free_page_tables called with wrong alignment`

那么这里对照源码较为详尽的梳理一下结合文件系统和缓冲区的磁盘访问过程, 作为实验替代:

1. 根据 `i` 节点结构指针, 调用 `bmap` 获取全局的数据块号, 这一步涉及直接、一次间接、二次间接的 `i` 节点寻址, 从而得到 `block` 对应的磁盘块号
2. 而后调用 `bread()` 将 `bmap` 获取的数据块号读入到高速缓冲块。首先调用 `getblk` 方法根据设备号 `dev` 和数据块号 `block`, 在所有缓冲块中寻找匹配或最为空闲的缓冲块
 - 若高速缓冲命中, 也就是在哈希数组中找到对应的数据块, 返回对应缓冲头结构的指针;
 - 若不命中, 则从空闲链表头开始, 对空闲链表进行扫描, 寻找一个空闲缓冲块并返回其缓冲头指针。
3. 对于得到的缓冲块, 若数据有效 (`b_uptodate = 1`) 则直接返回即可; 而若数据无效, 则调用 `ll_rw_block` 方法把硬盘中指定数据块中的数据, 复制到 `getblk` 方法申请到的缓冲块里
 - 调用 `make_request` 根据 `buffer_head` 从 `request` 数组中找到一个空位创建一个新的读盘请求 (包括设备号、命令、起始扇区、读/写扇区数、数据缓冲区等信息)
 - 调用 `add_request` 将请求加入队列
 - 收到 `request` 请求且没有其他请求时, 会立即执行该设备的 `request_fn` 方法。
 - 否则, 就按照之前提过的磁盘调度规则将 `request` 请求加入队列
4. 硬盘请求函数 `do_hd_request` 根据当前请求是写 (`WRITE`) 还是读 (`READ`), 在 `do_hd_request` 中调用 `hd_out` 时传入不同的参数 (磁头、扇区等), 然后向外设端口做读写操作
5. 在硬盘读完数据后发起 `0x2E` 中断, 便会进入到 `hd_interrupt` 方法里, 最终调用 `read_intr` 从数据端口读出数据到内存,
 - 如果一次没读完发起读盘请求时所要求的字节数, 那么直接返回, 等待下次硬盘触发中断;
 - 否则, 调用 `end_request` 方法将请求项清除掉, 然后再次调用 `do_hd_request` 方法循环往复。

最后, 上层进程通过 `put_fs_byte` 方法就可以将缓冲区里的数据, 复制到用户指定的内存 `buf` 中去了。

5.6 屏幕输出

这一步我们不去打印输出了，做一个稍微有意思的实验，来探究一下键盘中断和屏幕输出的原理……

修改 Linux 0.11 的终端设备处理代码，对键盘输入和字符显示进行非常规的控制。在初始状态，一切如常。用户按一次 F12 后，把应用程序向终端输出所有字母都替换为""。用户再按一次 F12，又恢复正常。第三次按 F12，再进行输出替换。依此类推。

5.6.1 添加 F12 功能键盘处理

- 修改 kernel/chr_drv/tty_io.c 文件，在文件末尾添加如下代码以实现对 F12 的识别：

```
432. int switch_show_char_flag = 0;
433. void press_f12_handle(void)
434. {
435.     if (switch_show_char_flag == 0)
436.         switch_show_char_flag = 1;
437.     else if (switch_show_char_flag == 1)
438.         switch_show_char_flag = 0;
439. }
```

- 修改 /include/linux/tty.h 文件，在末尾添加函数声明引用：

```
440. extern int switch_show_char_flag;
441. void press_f12_handle(void);
```

- 修改 kernel/chr_drv/keyboard.s 文件，修改 key_table 代码(359 行)如下：

```
442. .long press_f12_handle,none,none,none
```

5.6.2 添加字符 显示处理

- 修改 kernel/chr_drv/console.c 文件，修改 con_write 函数如下：即当 switch_show_char_flag 标签为 1 时就将所有字符都替换为""。

```
443. void con_write(struct tty_struct *tty)
444. {
445.     // .....
446.     case 0:
447.         if (c>31 && c<127) {
448.             // .....
449.             if (switch_show_char_flag == 1)
450.                 if((c>='A'&&c<='Z')||(c>='a'&&c<='z')||(c>='0'&&c<='9'))
451.                     c = ' ';
452.             // .....
453.             pos += 2;
454.             x++;
455. }
```

5.6.3 实验结果

可以看到，按下 F12 后，所有屏幕输出字符（键盘中断指令输入和程序屏幕打印）都变为了""

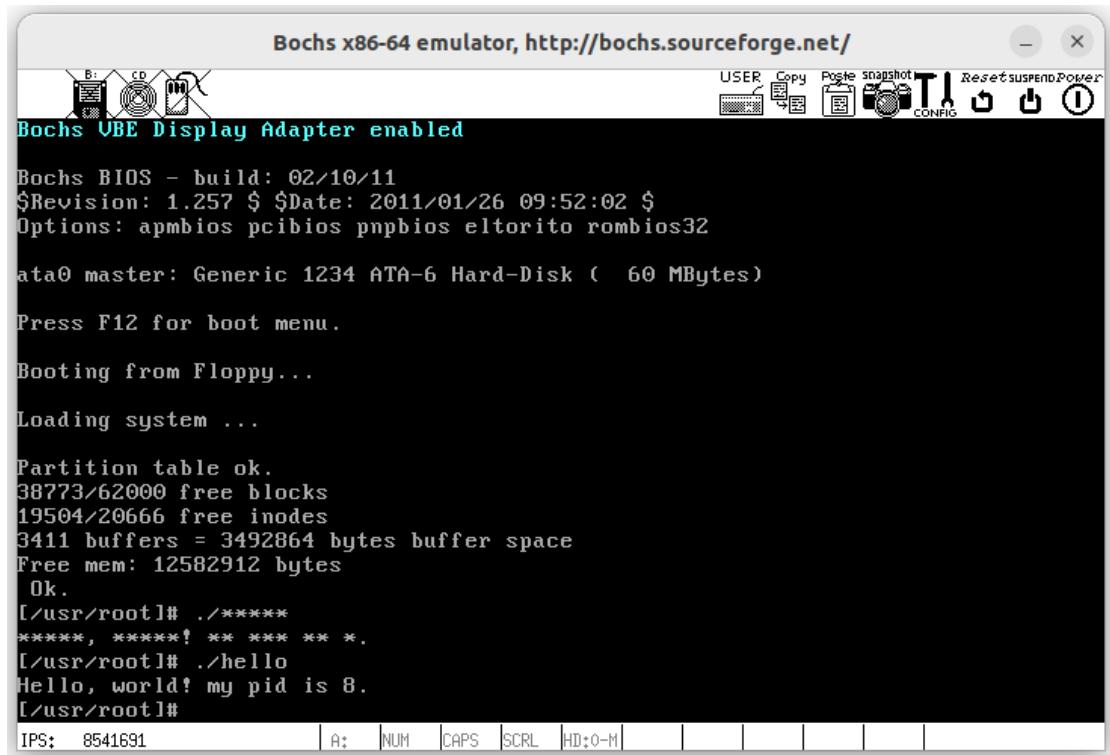


图 76 屏幕输出截图

5.7 信号

信号是进程间通信的一种方式，在 `hello` 的一生中虽然没有直接涉及，但是这也是操作系统很重要的一个部分。这里我们重新编写一个程序 `test.c`（直接在 `hello.c` 上修改会导致前面的结果有细微变化）去捕获 `SIGINT` 这个信号。

`test.c` 通过 `signal` 注册了 `SIGINT` 的信号处理函数，里面做的事情仅仅是打印一下信号值。捕获一次后，解除信号处理函数 `int_handler` 与 `SIGINT` 信号的绑定。这样当没有为 `SIGINT` 注册信号处理函数时，程序接收到 `CTRL+C` 的 `SIGINT` 信号时便会退出。

```
455. #include <stdio.h>
456. #include <signal.h>
457.
458. void int_handler(int signal_num) {
459.     printf("signal receive %d\n", signal_num);
460.     signal(SIGINT, NULL);
461. }
462.
463. int main(int argc, char argv) {
464.     signal(SIGINT, int_handler);
465.     printf("hello world!\n");
466.     for(;;)
467.         pause();
468.     return 0;
469. }
```

我们发现，第一次按下 `CTRL+C` 程序会正常输出，而按下第二次 `CTRL+C` 程序就会

退出了。

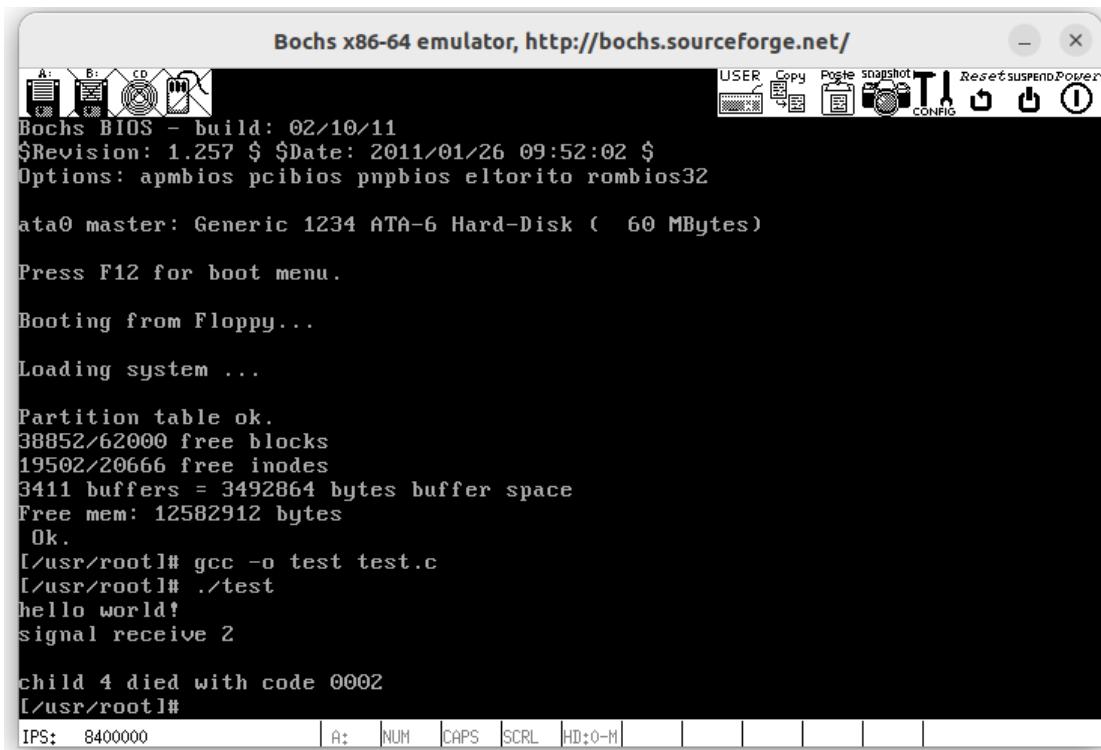


图 77 信号截图

5.8 进程退出

好了现在 `hello` 已经执行完毕了（通过执行完所有的任务和代码，达到了预定的终止条件），操作系统应该怎么为他“善后”呢？我们在程序退出处理函数 `kernel/exit.c` 中的 `do_exit()` 函数中进行加入对其结构体打印如下（代码见附件）：

```
exit: process 6 is exiting...
ldt[0]( 0~31):    0
ldt[0](32~63):    0
ldt[1]( 0~31):    3
ldt[1](32~63):    10c0fb00
ldt[2]( 0~31):    3fff
ldt[2](32~63):    10c0f300
state:           3
exit_code:        0
```

可以看到，进程退出比进程创建要简单许多，我们也可以通过 `var/fork.log` 进程创建和退出的顺序与进程调度中互为验证：

- 首先释放当前进程代码段和数据段所占的内存页，
- 如果当前进程有子进程，则让进程 1（`init` 进程）成为其所有子进程的父进程；如果子进程已经处于僵死状态，则向进程 1 发送子进程终止信号 `SIGCHLD`（这里没有子进程，跳过）

- 然后关闭当前进程打开着的所有文件。再对当前进程的工作目录 `pwd`、根目录 `root`、执行程序文件的 `i` 节点以及 - 库文件进行同步操作，放回各个 `i` 节点并分别置空（释放）
- 随后把当前进程置为僵死状态 `3`，设置退出码 `0`，并向其父进程发送子进程终止信号 `SIGCHLD`

最后由父进程 `Shell` 等待并回收子进程，内核删除为进程创建的所有资源。最终的状态就是进程相关的表和其他信息已完全从内存中删除，仿佛没有来过一样。

六、总结回顾

简要总结一下，`hello` 的一生包含如下阶段：

1. 进程创建与载入：通过 `Shell` 键入命令 `./hello`，操作系统为程序 `fork` 新进程并通过 `execve` 加载文件头信息到为其提供的私有虚拟内存空间，程序开始执行。
2. 进程控制：由进程调度器对进程进行时间片调度，并通过上下文切换实现 `hello` 的执行，程序计数器(`PC`)更新，CPU 按顺序取指，执行程序控制逻辑。
3. 内存管理：通过缺页中断将程序需要的代码和数据段载入内存。内存管理单元 `MMU` 将逻辑地址逐步转换成物理地址。
4. 磁盘管理：结合文件系统找到对应磁盘区域，而后基于缓冲区访问物理内存/磁盘中的数据
5. 屏幕输出：调用 `tty_write` 函数，最终调用 `con_write` 函数将字符输出，实际上是将内存的数据换了个地方写入显存
6. 信号处理：进程接收信号，调用相应的信号处理函数对信号进行终止、停止、前/后台运行等处理。
7. 进程回收：`Shell` 等待并回收子进程，内核删除为进程创建的所有资源。