Operating System

Dr. GuoJun LIU

Harbin Institute of Technology

http://os.guojunhit.cn

Chapter A2

Linux 0.11 Overview

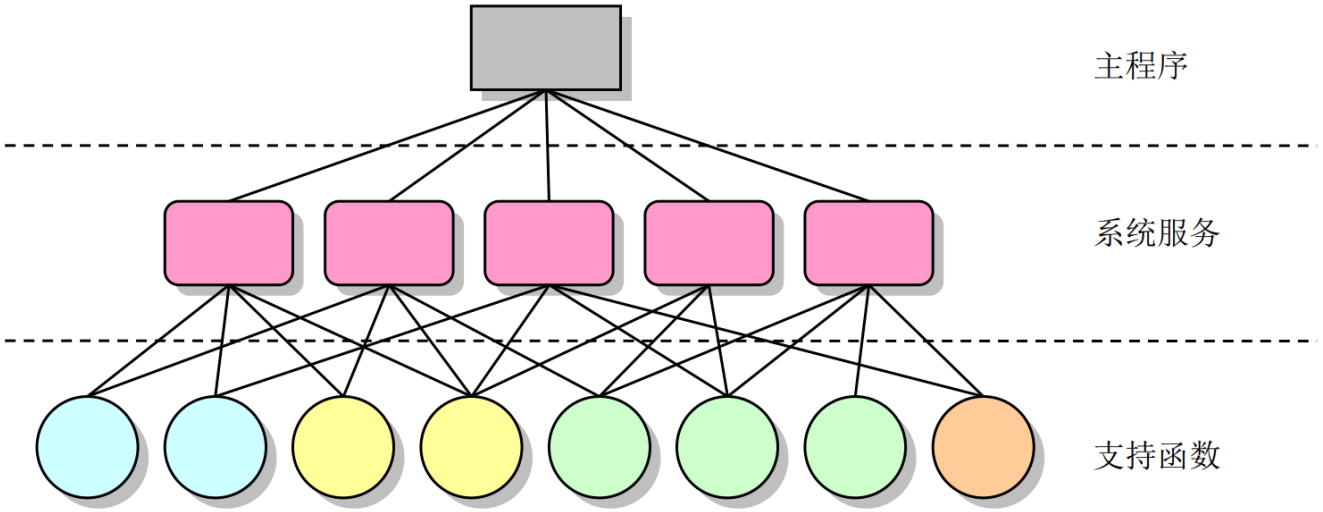
Outline

- Linux 内核体系结构
- 引导启动程序
- 初始化程序
- Linux 内核对存的管理和使用
- Linux 进程控制
- 设置描述符
- fork系统调用
- 信号处理
- 内存管理
- 调度算法
- 电梯算法
- execve流程解析

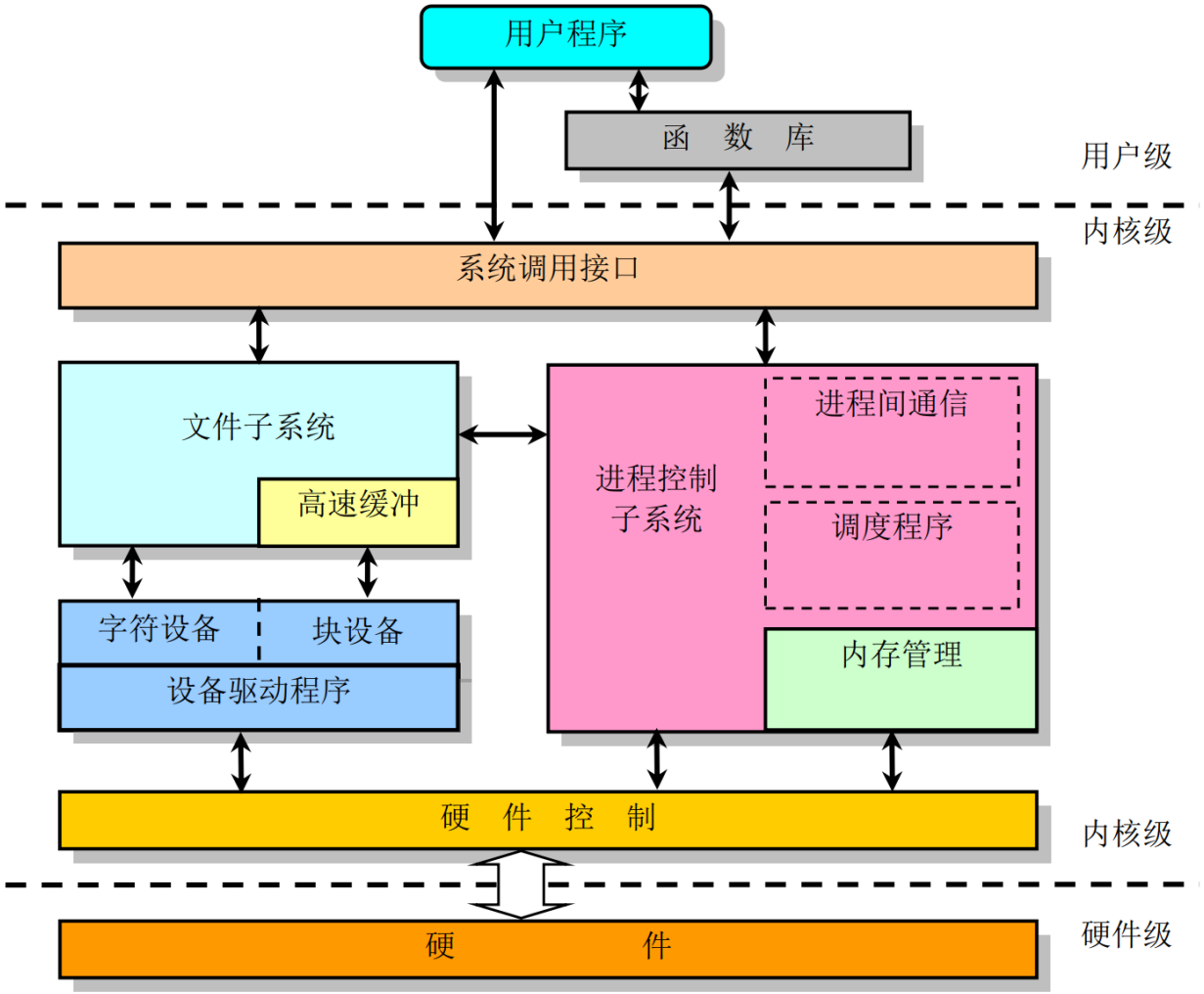
Linux 内核体系结构

单内核模式的简单结构模型

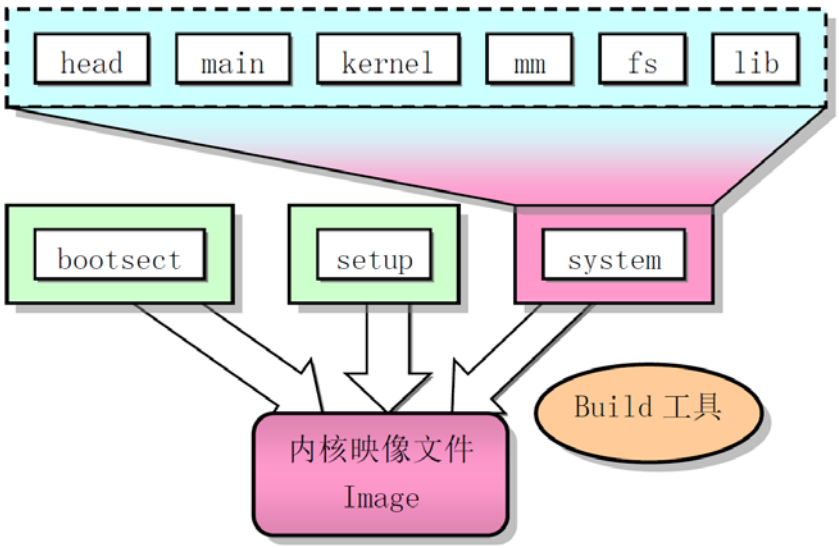
- 操作系统内核的结构模式主要可分为
  - 整体式的单内核模式
    - 优点是内核代码结构紧凑、执行速度快
    - 不足之处主要是层次结构性不强
  - 层次式的微内核模式
  - 混合模式

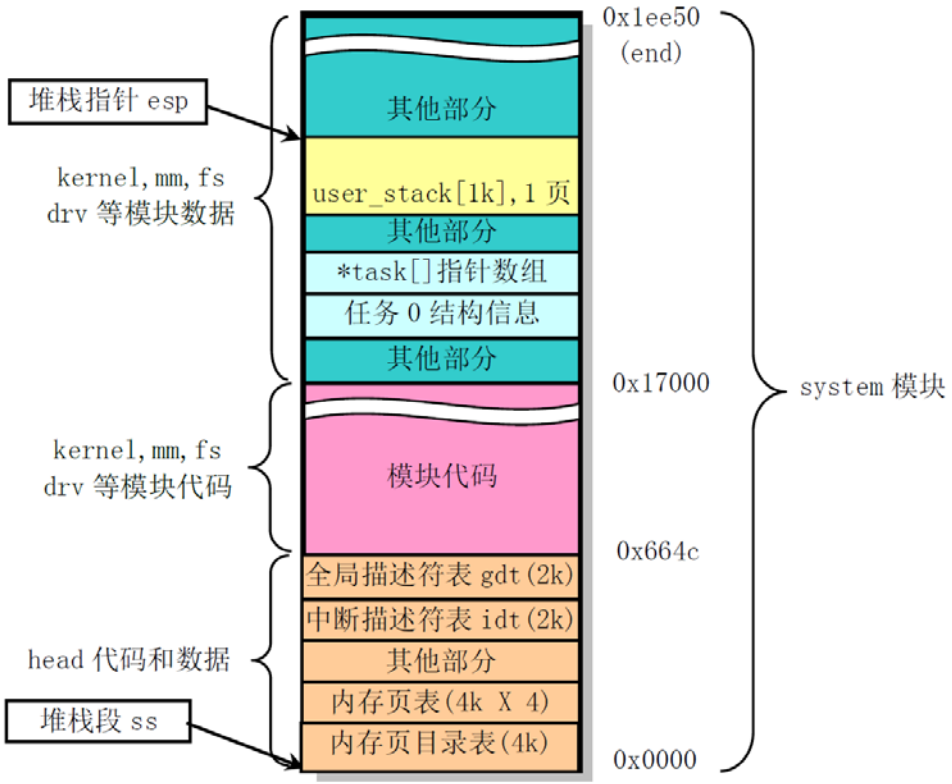


内核结构框图


用户程序
函 数 库
用户级
内核级
系统调用接口
文件子系统
高速缓冲
字符设备
块设备
设备驱动程序
进程控制
子系统
进程间通信
调度程序
内存管理
硬 件 控 制
内核级
硬 件
硬件级

内核编译链接/组合结构





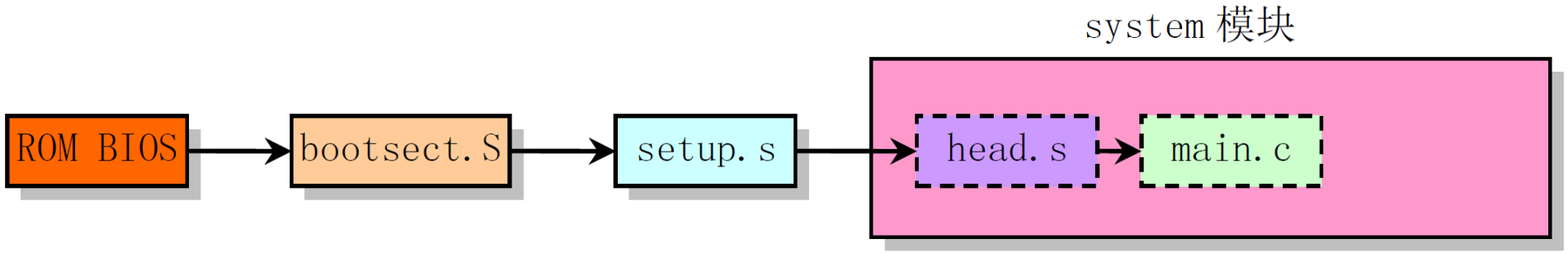
刚进入保护模式时内核使用的堆栈示意图

引导启动程序

引导启动程序

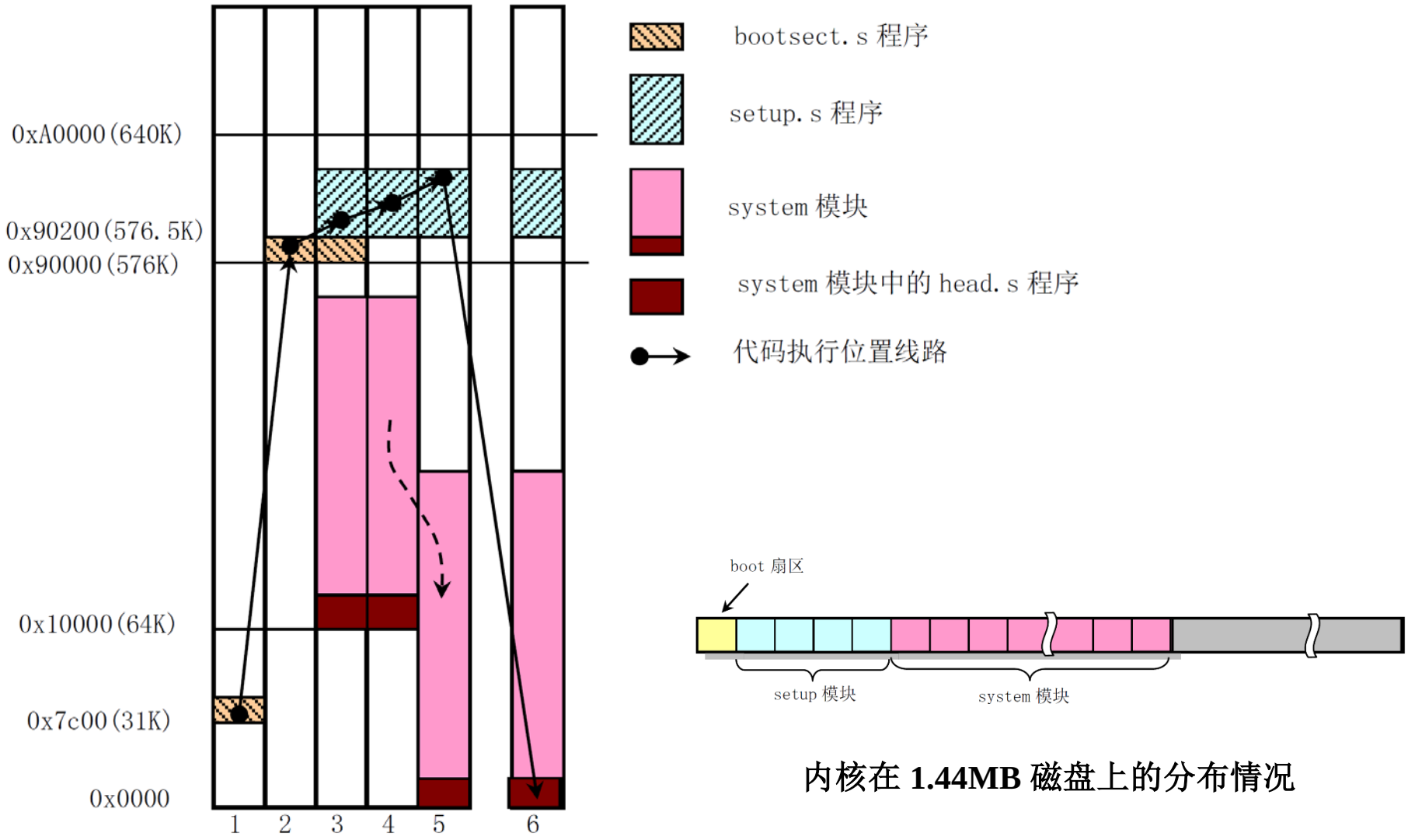
文件名	长度（字节）	最后修改时间（GMT）	说明
bootsect.S	7574 bytes	1992-01-14 15:45:22	
head.s	5938 bytes	1992-01-11 04:50:17	
setup.S	12144 bytes	1992-01-11 18:10:18	

Linux/boot/目录



从系统加电起所执行程序的顺序

引导时内核在存中的位置和移动情况

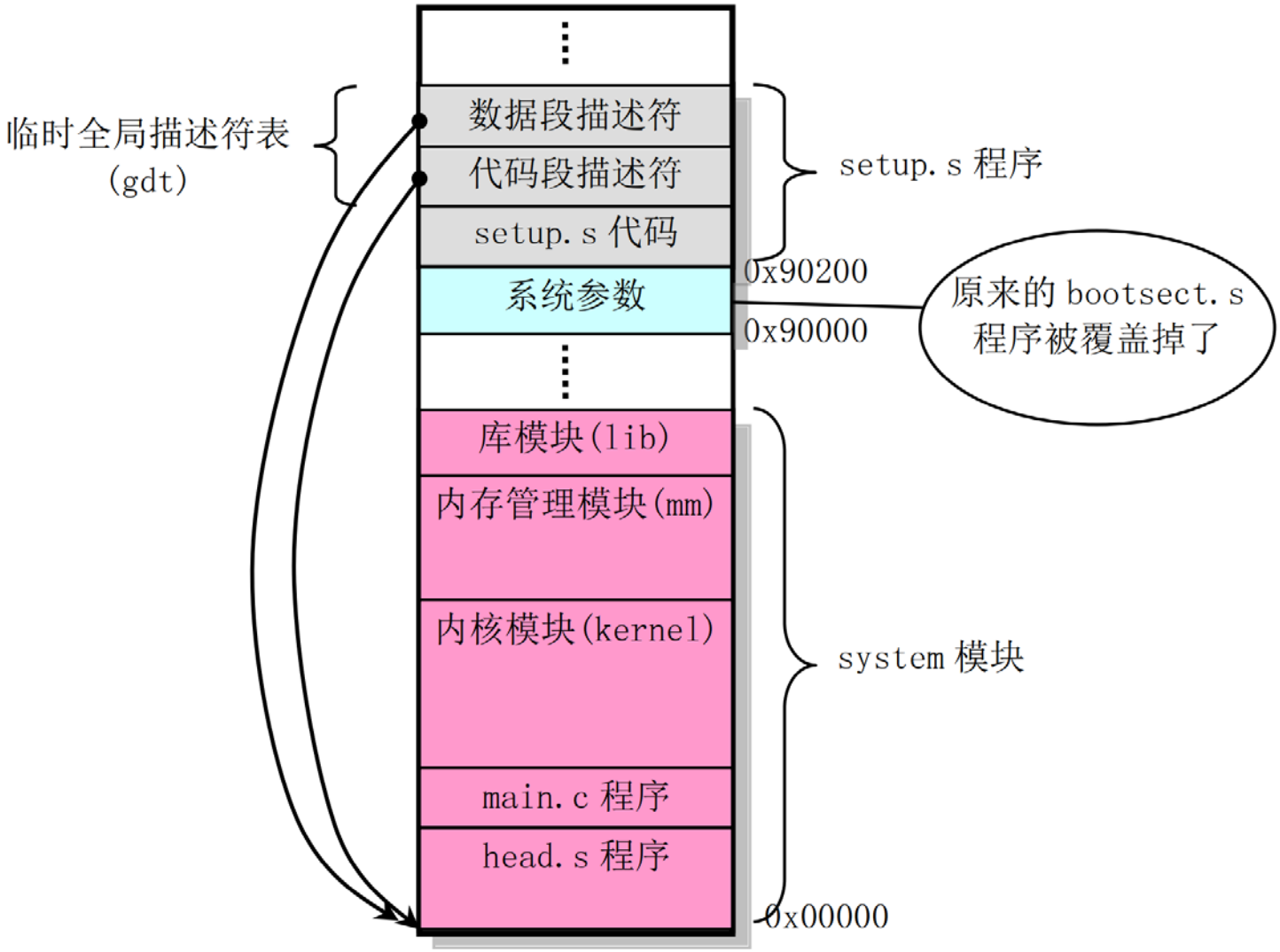


内核在 **1.44MB** 磁盘上的分布情况

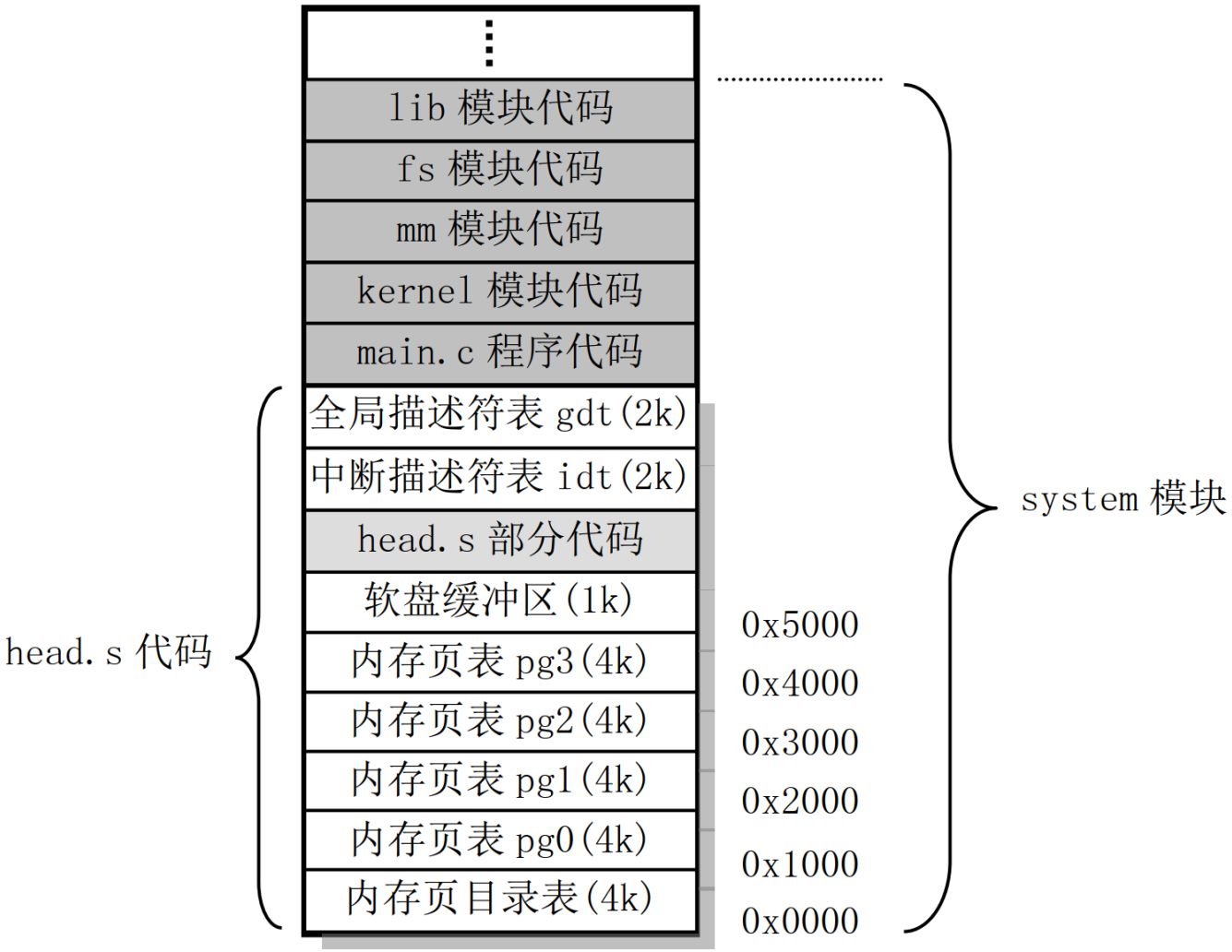
setup 程序读取并保留的参数

内存地址	长度(字节)	名称	描述
0x90000	2	光标位置	列号（0x00-最左端），行号（0x00-最顶端）
0x90002	2	扩展内存数	系统从 1MB 开始的扩展内存数值（KB）。
0x90004	2	显示页面	当前显示页面
0x90006	1	显示模式	
0x90007	1	字符列数	
0x90008	2	??	
0x9000A	1	显示内存	显示内存(0x00-64k,0x01-128k,0x02-192k,0x03=256k)
0x9000B	1	显示状态	0x00-彩色,I/O=0x3dX；0x01-单色,I/O=0x3bX
0x9000C	2	特性参数	显示卡特性参数
0x9000E	1	屏幕行数	屏幕当前显示行数。
0x9000F	1	屏幕列数	屏幕当前显示列数。
...			
0x90080	16	硬盘参数表	第 1 个硬盘的参数表
0x90090	16	硬盘参数表	第 2 个硬盘的参数表（如果没有，则清零）
0x901FC	2	根设备号	根文件系统所在的设备号（bootsec.s 中设置）

setup.s 程序结束后内存中程序示意图



system 模块在内存中的映像示意图



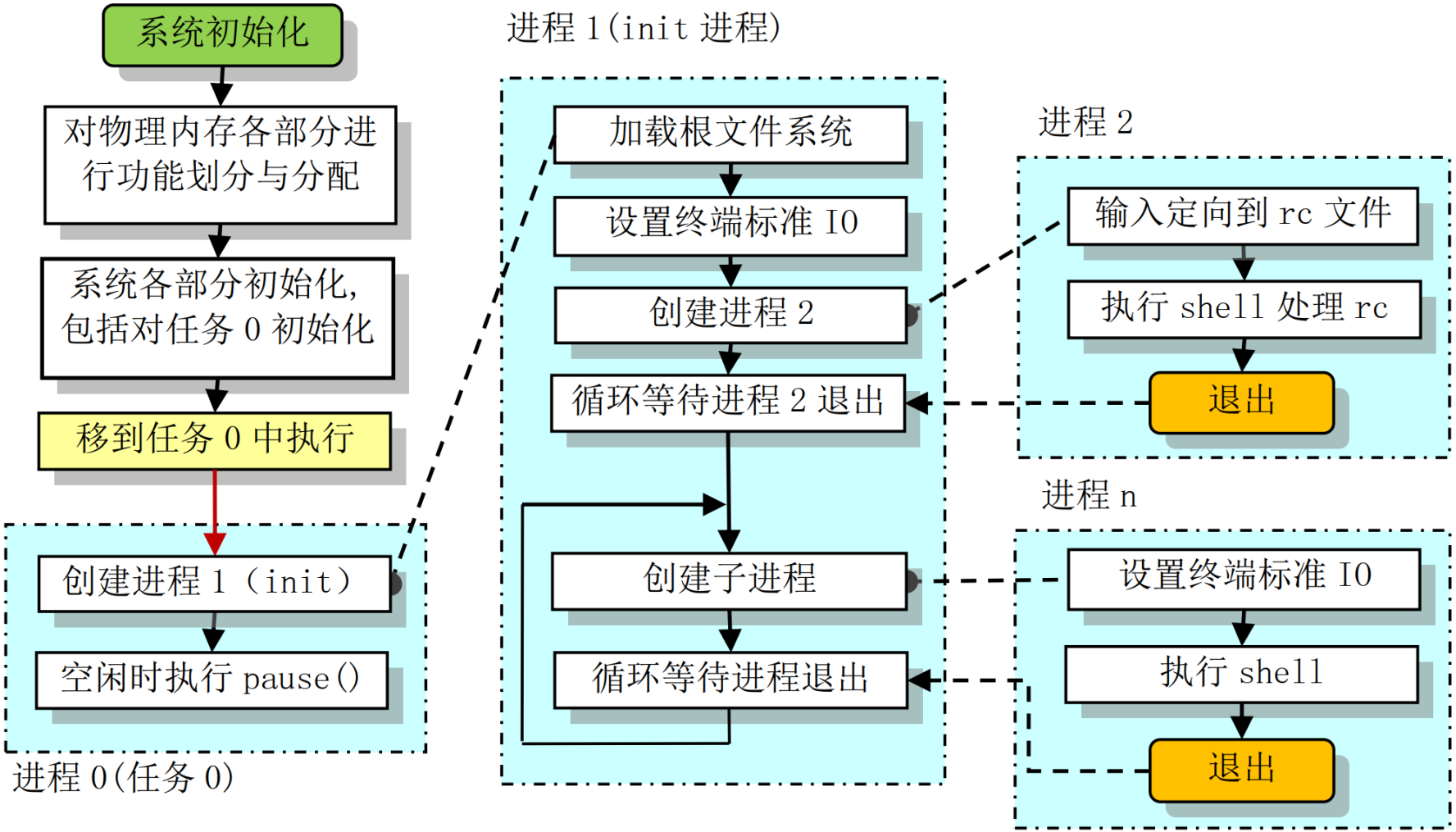
初始化程序

main.c 程序

- 利用前面 setup.s 程序取得的机器参数设置系统的根文件设备号以及一些内存全局变量
  - 这些内存变量指明了主内存区的开始地址、系统所拥有的内存容量和作为高速缓冲区内存的末端地址

- 内核进行各方面的硬件初始化工作
  - 包括陷阱门、块设备、字符设备和 tty，还包括人工设置第一个任务（ task 0）
  - 待所有初始化工作完成后， 程序就设置中断允许标志以开启中断，并切换到任务0 中运行
  - 内核会通过任务 0 创建几个最初的任务，运行 shell 程序并显示命令行提示符，从而 Linux 系统进入正常运行阶段

内核初始化程序流程示意图



move_to_user_mode()

```
122:     main_memory_start = buffer_memory_end;
123: #ifdef RAMDISK
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:     }
```

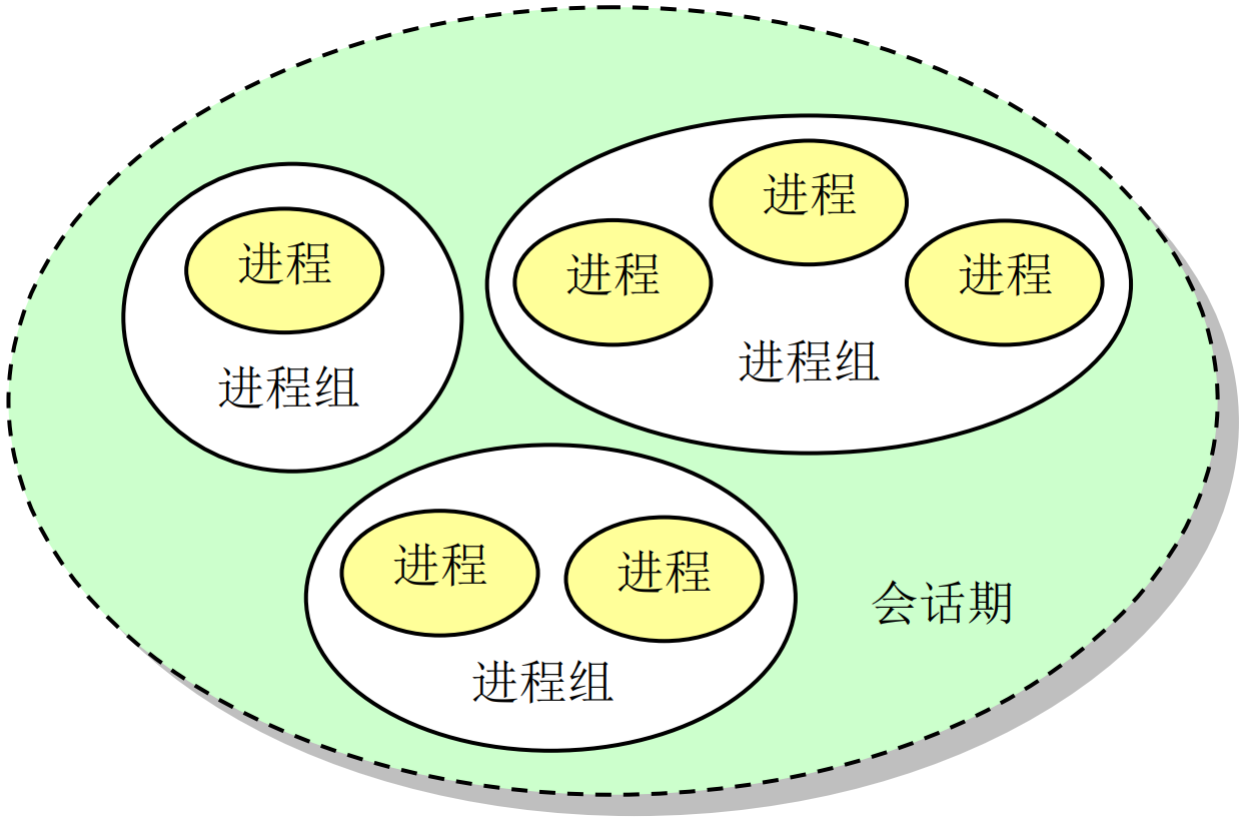
init\main.c

```
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" \
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")
15:
16: #define sti() __asm__ ("sti"::)
```

include\asm\system.h

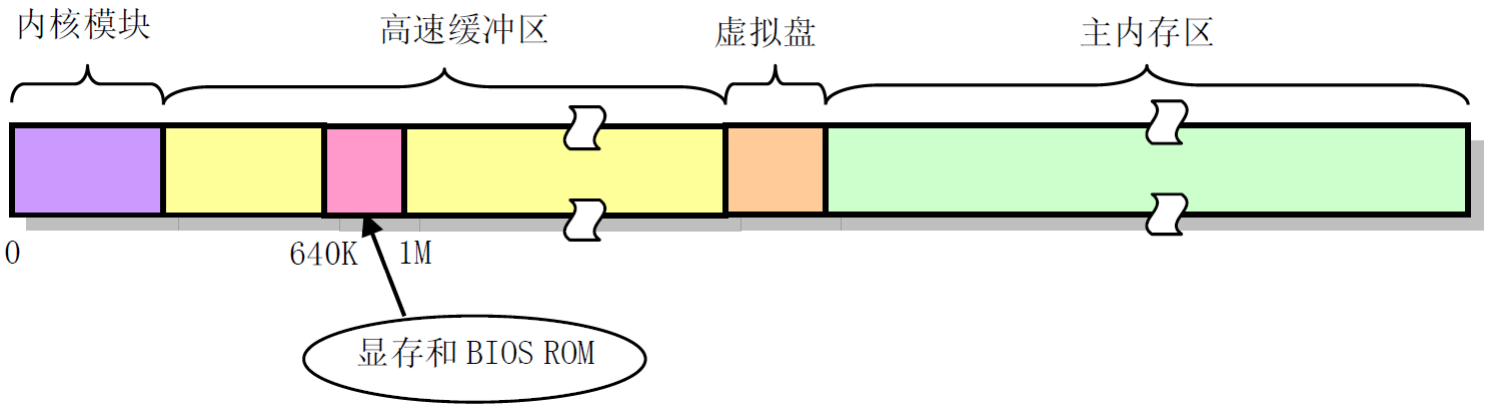
```
000068bf  (6)...  push dword ptr ds:0x0001dac8
000068c5  (5)...  call .+24533 (0x0000c89f)
000068ca  (5)...  call .+58386 (0x00014ce1)
000068cf  (5)...  call .+54513 (0x00013dc5)
000068d4  (1)...  sti
000068d5  (2)...  mov eax, esp
000068d7  (2)...  push 0x00000017
000068d9  (1)...  push eax
000068da  (1)...  pushf
000068db  (2)...  push 0x0000000f
000068dd  (5)...  push 0x000068e3
000068e2  (1)...  iret
000068e3  (5)...  mov eax, 0x00000017
000068e8  (2)...  mov ds, ax
000068ea  (2)...  mov es, ax
000068ec  (2)...  mov fs, ax
000068ee  (2)...  mov gs, ax
000068f0  (5)...  mov eax, 0x00000002
```

进程、进程组和会话期之间的关系

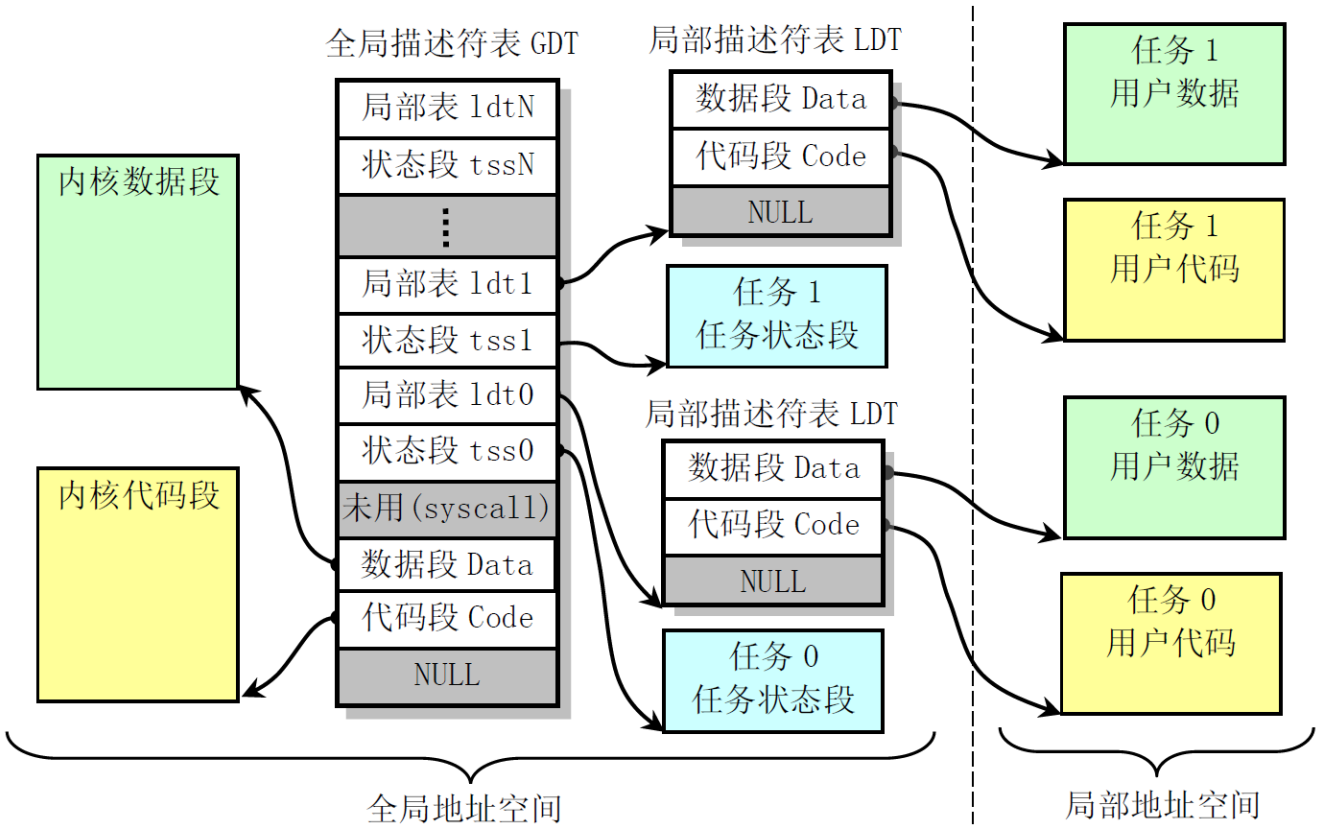


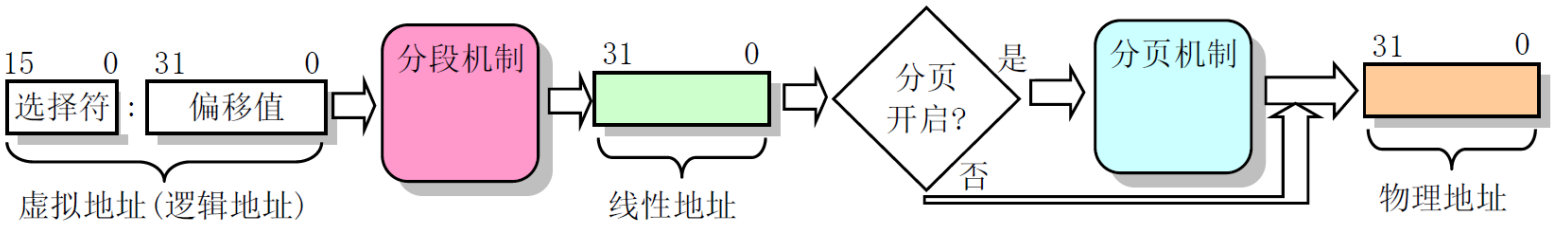
Linux 内核对存的管理和使用

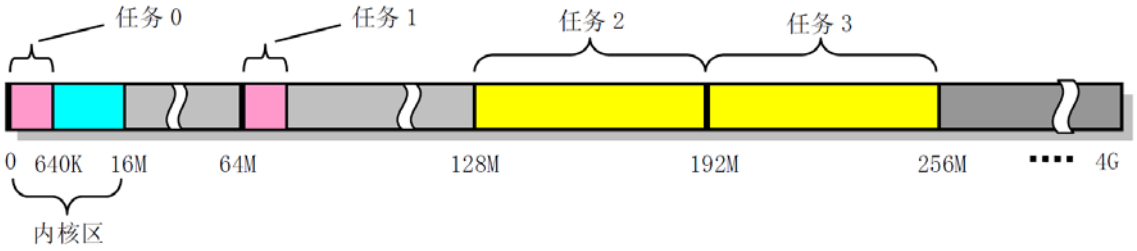
物理内存使用的功能分布



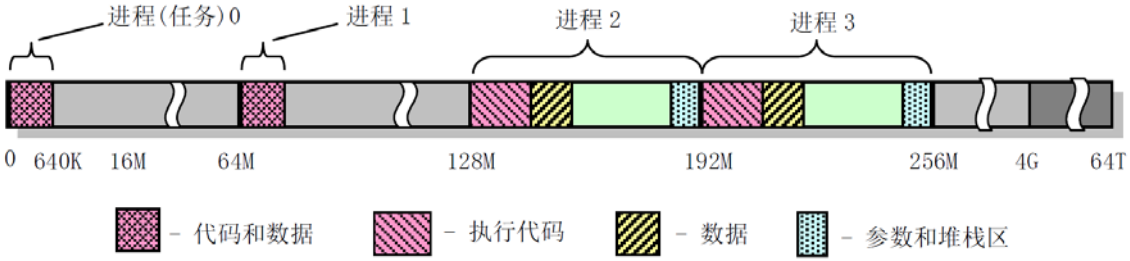
系统中虚拟地址空间分配图



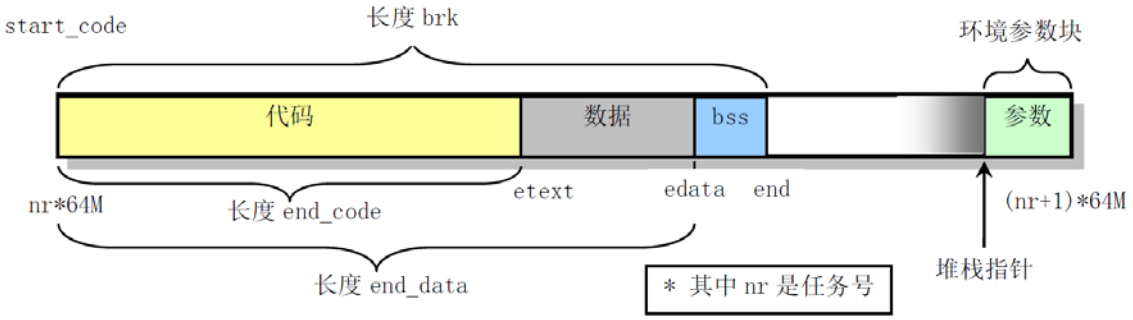




线性地址空间的使用示意图

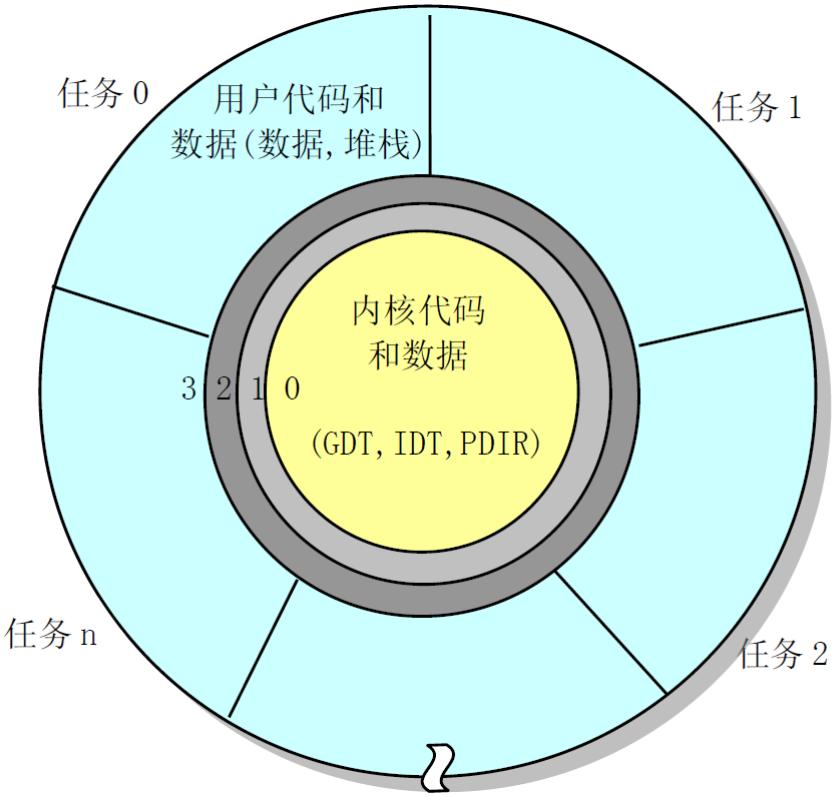


系统任务在虚拟空间中顺序排列所占的空间范围



进程代码和数据在其逻辑地址空间中的分布

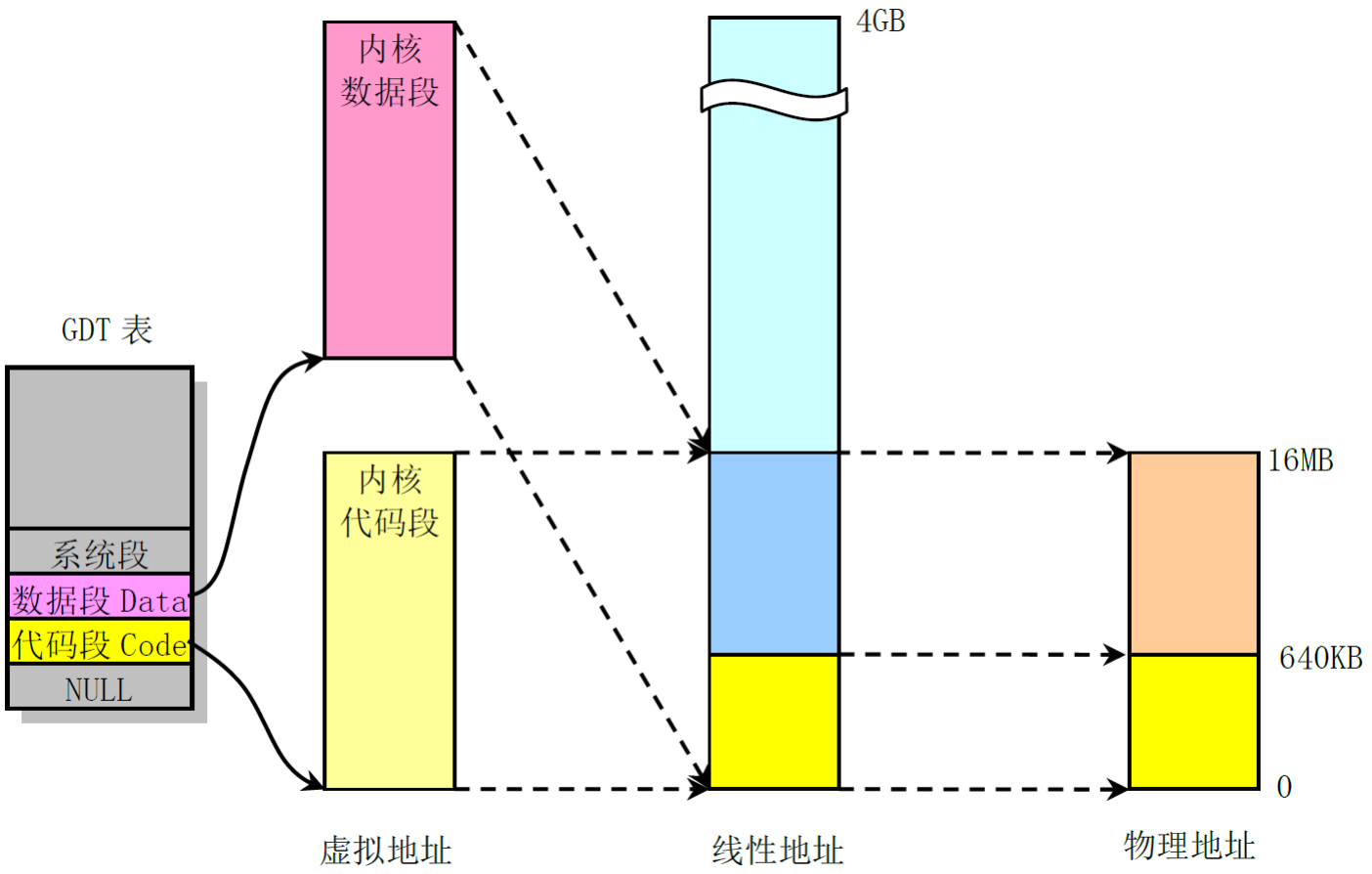
CPU 多任务和保护方式



虚拟地址、线性和物理之间的关系

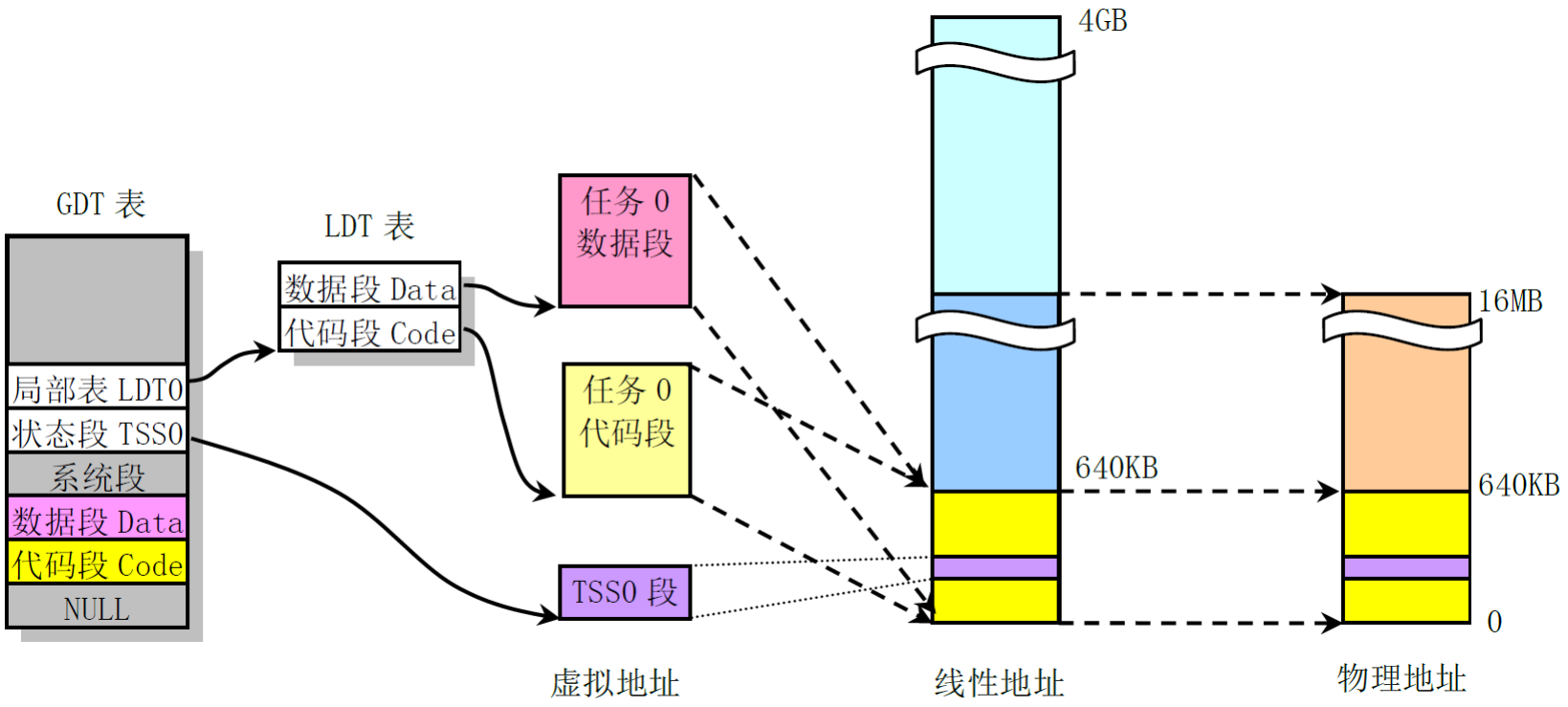
- 内核代码和数据的地址
- 任务 0 在三个地址空间中的相互关系
- 任务 1 在三个地址空间中的相互关系
- 其他任务地址空间中的对应关系

内核代码和数据的地址

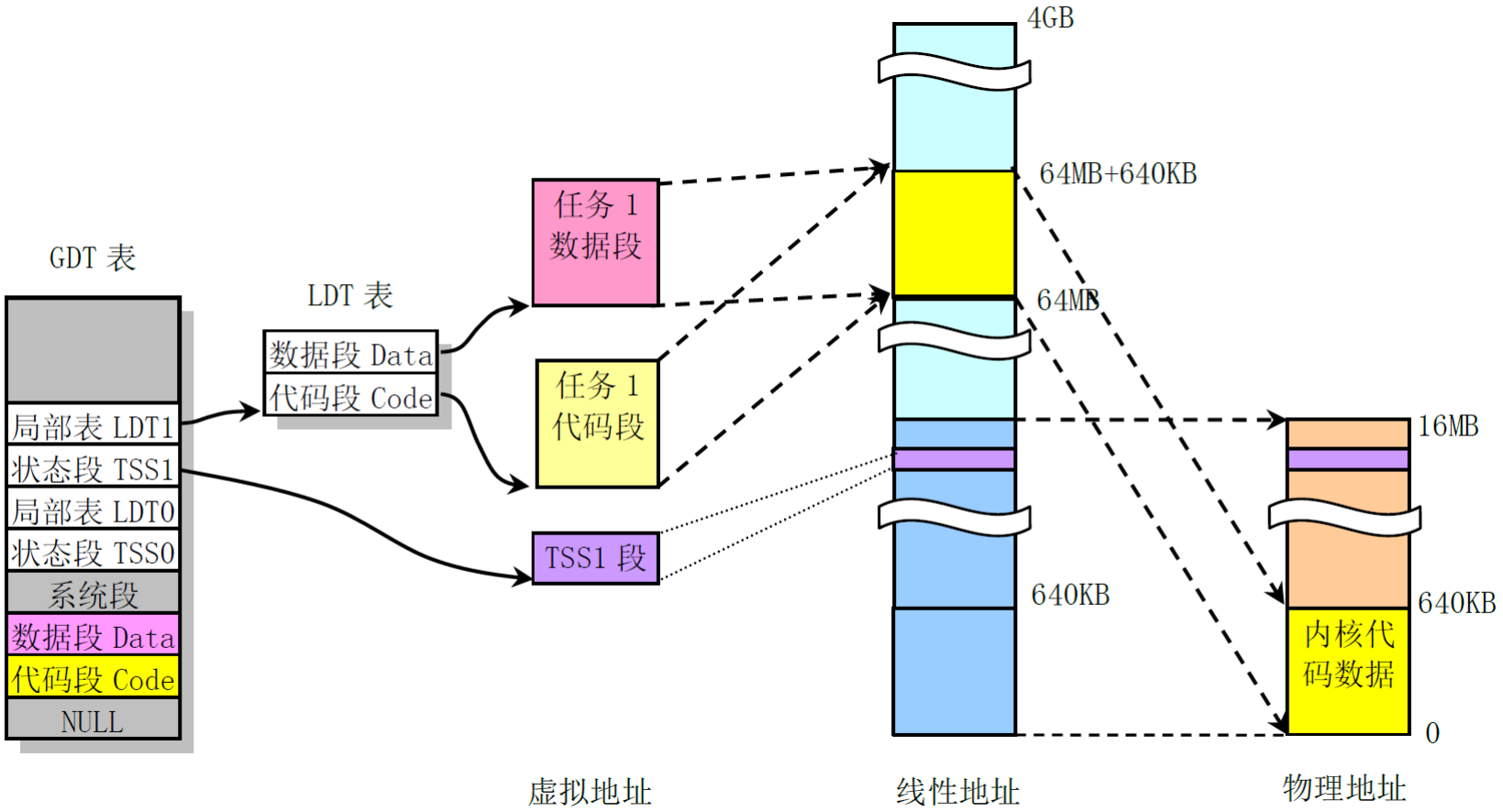


内核代码和数据段在三种地址空间中的关系

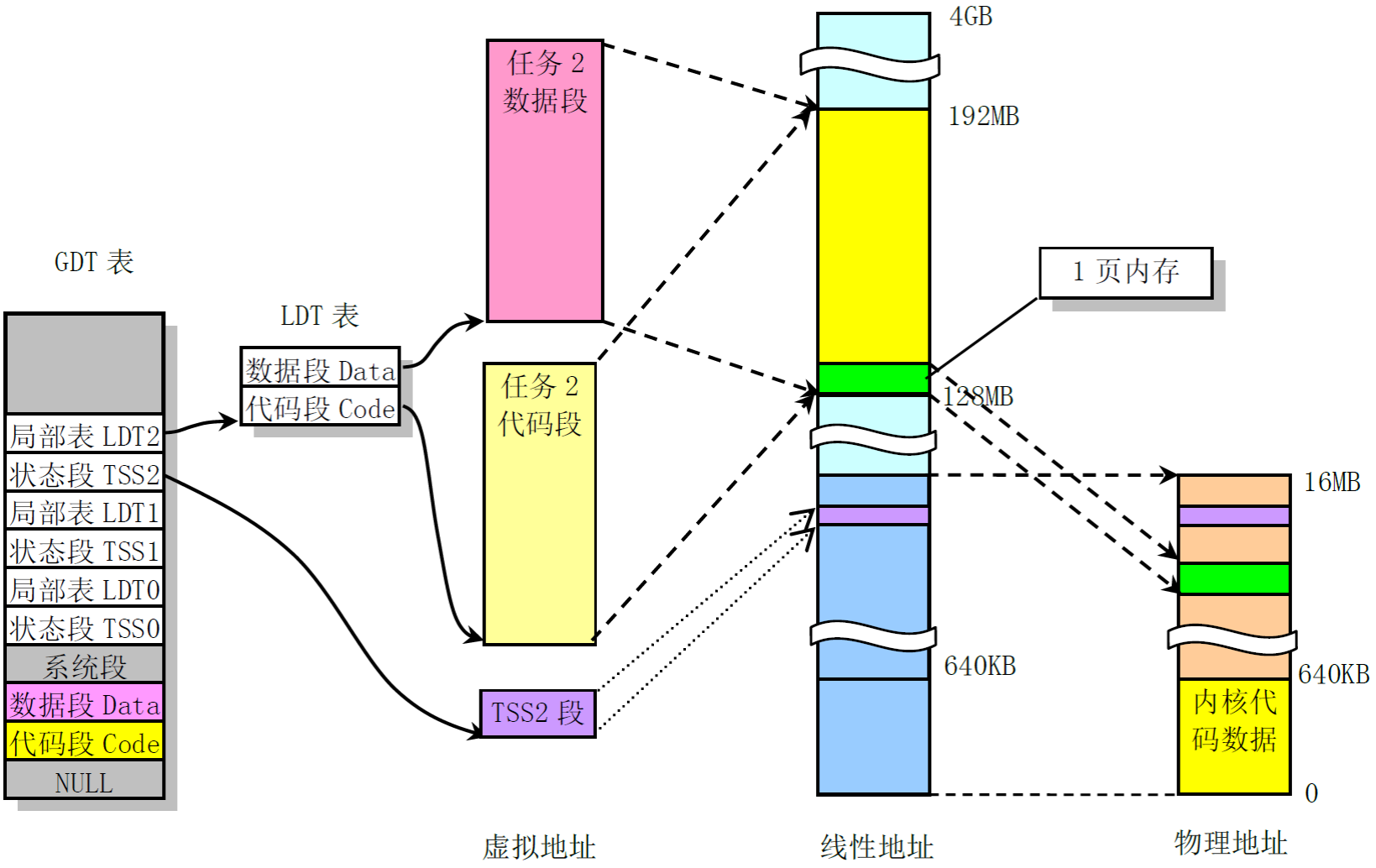
任务 0 在三个地址空间中的相互关系



任务 1 在三个地址空间中的相互关系



其他任务地址空间中的对应关系



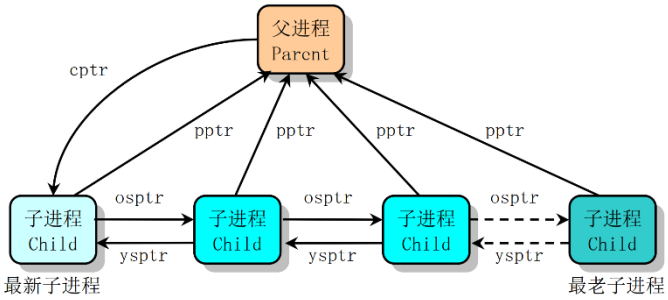
Linux 进程控制

- **Task 数据结构 – PCB**
- **进程运行状态及转换**
- **进程初始化**
  - move_to_user_mode
- **创建新进程**
  - fork()、写时复制 Copy on Write
- **进程调度**
  - 优先级、时间片、进程切换 switch_to()
- **终止进程 exit()**

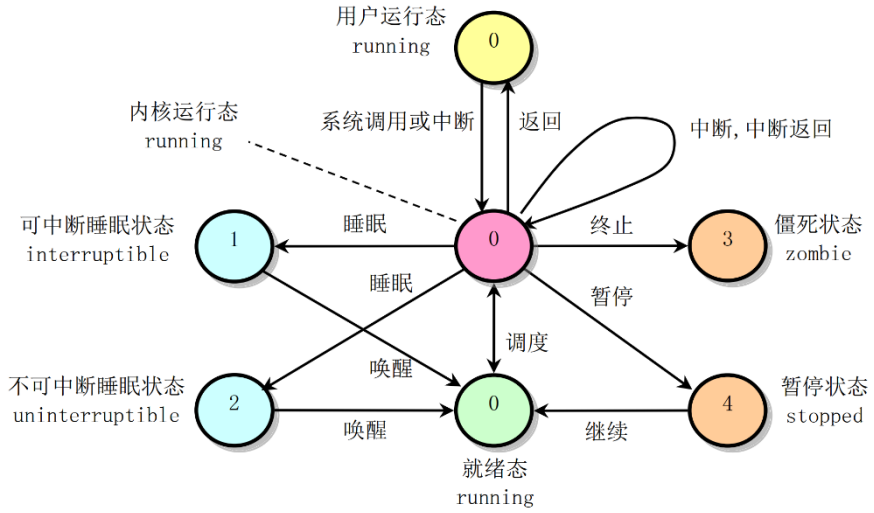
Linux 进程控制

```
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,pgrp,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 » ;
110:
```

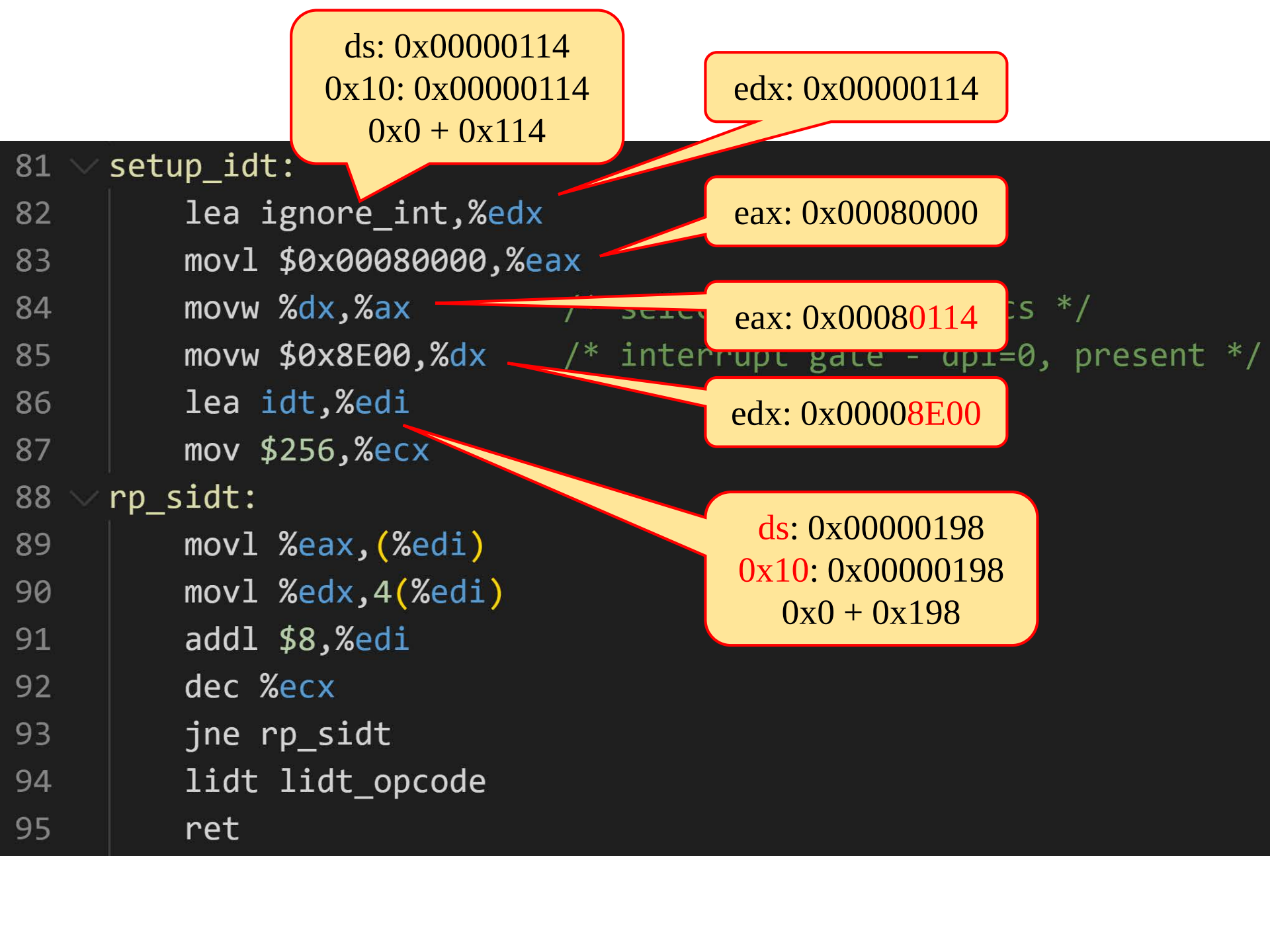
include\linux\sched.h

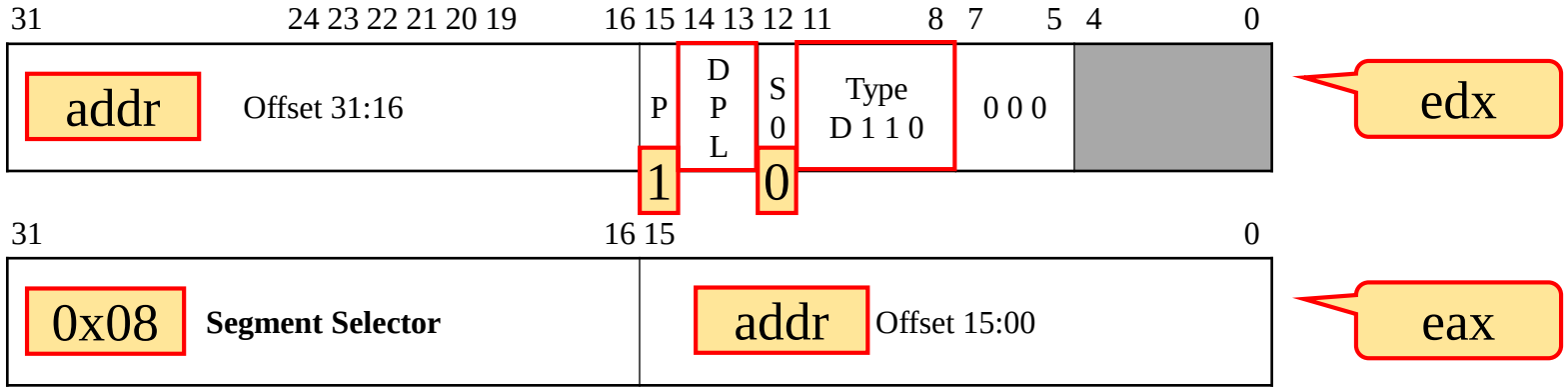



```
19: #define TASK_RUNNING        0
20: #define TASK_INTERRUPTIBLE  1
21: #define TASK_UNINTERRUPTIBLE    2
22: #define TASK_ZOMBIE     3
23: #define TASK_STOPPED        4
```



设置描述符


ds: 0x00000114
0x10: 0x00000114
0x0 + 0x114
edx: 0x00000114
eax: 0x00080000
eax: 0x00080114
edx: 0x00008E00
ds: 0x00000198
0x10: 0x00000198
0x0 + 0x198
81 setup_idt:
82 lea ignore_int,%edx
83 movl $0x00080000,%eax
84 movw %dx,%ax
85 movw $0x8E00,%dx
86 lea idt,%edi
87 mov $256,%ecx
88 rp_sidt:
89 movl %eax,(%edi)
90 movl %edx,4(%edi)
91 addl $8,%edi
92 dec %ecx
93 jne rp_sidt
94 lidt lidt_opcode
95 ret




```
22  #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" (0x00080000))
32
33  #define set_intr_gate(n,addr) \
34      _set_gate(&idt[n],14,0,addr)
```

eax: 0x00080000
dx: addr 15:00

dx: ~~addr 15:00~~

init\main.c

```
#ifdef RAMDISK
    main_memory_start += rd_init(main_memory_start, RAMDISK*1024);
#endif
    mem_init(main_memory_start,memory_end);
    trap_init();
    blk_dev_init();
    chr_dev_init();
    tty_init();
    time_init();
    sched_init();
    buffer_init(buffer_memory_end);
    hd_init();
    floppy_init();
    sti();
    move_to_user_mode();
    if (!fork()) {      /* we count on this going ok */
        init();
    }
```

kernel\sched.c

```
void sched_init(void)
{
    int i;
    struct desc_struct * p;

    if (sizeof(struct sigaction) != 16)
        panic("Struct sigaction MUST be 16 bytes");
    set_tss_desc(gdt+FIRST_TSS_ENTRY,&(init_task.task.tss));
    set_ldt_desc(gdt+FIRST_LDT_ENTRY,&(init_task.task.ldt));
    p = gdt+2+FIRST_TSS_ENTRY;
    for(i=1;i<NR_TASKS;i++) {
        task[i] = NULL;
        p->a=p->b=0;
        p++;
        p->a=p->b=0;
        p++;
    }
/* Clear NT, so that we won't have troubles with that later on */
    __asm__("pushfl ; andl $0xffffbfff,(%esp) ; popfl");
    ltr(0);
    lldt(0);
    outb_p(0x36,0x43);      /* binary, mode 3, LSB/MSB, ch 0 */
    outb_p(LATCH & 0xff , 0x40);    /* LSB */
    outb(LATCH >> 8 , 0x40);    /* MSB */
    set_intr_gate(0x20,&timer_interrupt);
    outb(inb_p(0x21)&0x01,0x21);
    set_system_gate(0x80,&system_call);
} « end sched_init »
```

include\asm\system.h

```
#define _set_gate(gate_addr,type,dpl,addr) \
__asm__ ("movw %%dx,%%ax\n\t" \
    "movw %0,%%dx\n\t" \
    "movl %%eax,%1\n\t" \
    "movl %%edx,%2" \
    : \
    : "i" ((short) (0x8000+(dpl<<13)+(type<<8))), \
    "o" (*((char *) (gate_addr))), \
    "o" (*(4+(char *) (gate_addr))), \
    "d" ((char *) (addr)),"a" (0x00080000))

#define set_intr_gate(n,addr) \
    _set_gate(&idt[n],14,0,addr)

#define set_trap_gate(n,addr) \
    _set_gate(&idt[n],15,0,addr)
```

include\linux\head.h

```
#ifndef _HEAD_H
#define _HEAD_H

typedef struct desc_struct {
    unsigned long a,b;
} desc_table[256];

extern unsigned long pg_dir[1024];
extern desc_table idt,gdt;

#define GDT_NUL 0
#define GDT_CODE 1
#define GDT_DATA 2
```

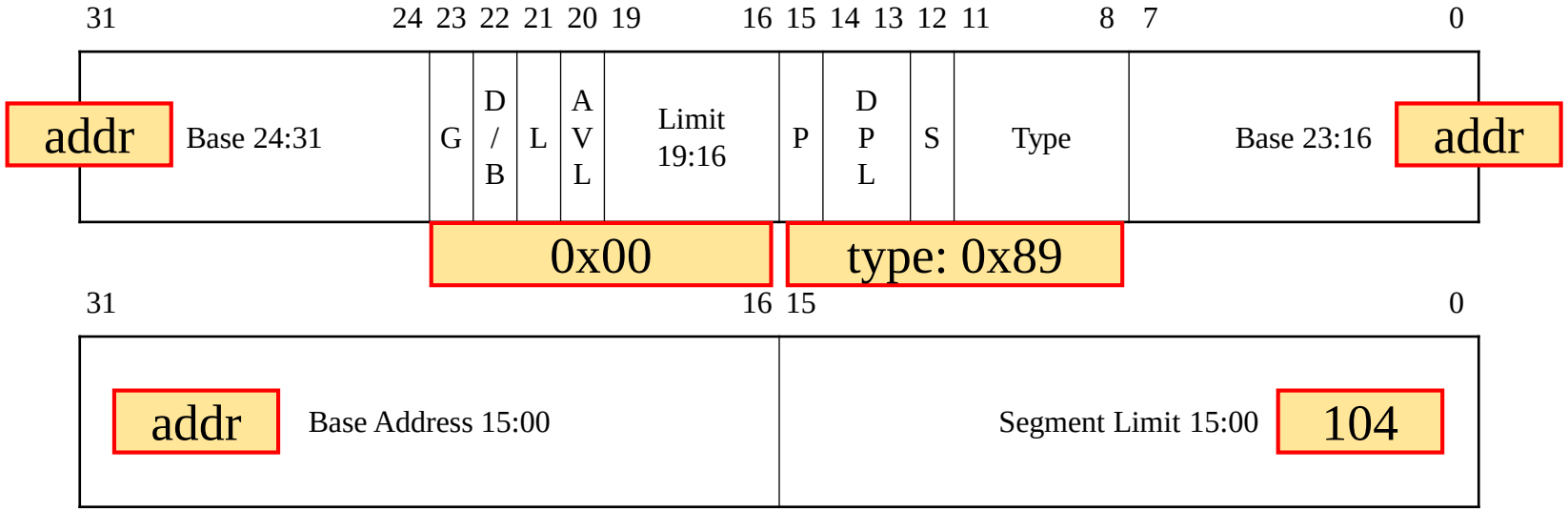
kernel\system_call.s

```
nr_system_calls = 72

/*
 * Ok, I get parallel printer interrupts while using the floppy for some
 * strange reason. Urgel. Now I just ignore them.
 */
.globl system_call,sys_fork,timer_interrupt,sys_execve
.globl hd_interrupt,floppy_interrupt,parallel_interrupt
.globl device_not_available, coprocessor_error
```

```
.align 2
timer_interrupt:
        push %ds                # save ds,es and put kernel data space
        push %es                # into them. %fs is used by _system_call
        push %fs
        pushl %edx              # we save %eax,%ecx,%edx as gcc doesn't
        pushl %ecx              # save those across function calls. %ebx
        pushl %ebx              # is saved as we use that in ret_sys_call
        pushl %eax
        movl $0x10,%eax
        mov %ax,%ds
        mov %ax,%es
        movl $0x17,%eax
        mov %ax,%fs
        incl jiffies
        movb $0x20,%al          # EOI to interrupt controller #1
        outb %al,$0x20
        movl CS(%esp),%eax
        andl $3,%eax            # %eax is CPL (0 or 3, 0=supervisor)
        pushl %eax
        call do_timer           # 'do_timer(long CPL)' does everything from
        addl $4,%esp            # task switching to accounting ...
        jmp ret_from_sys_call
```

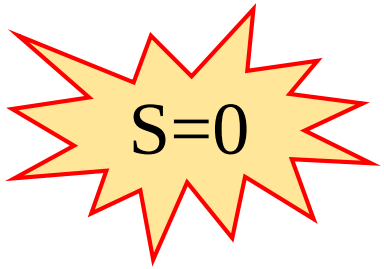
```
51   000077c0 T coprocessor_error
52   000077e2 T device_not_available
53   0000781c T timer_interrupt
54   00007854 T sys_execve
55   00007862 T sys_fork
56   0000787a T hd_interrupt
```

```
000076e8  (2)...  jmp .+0 (0x000076ea)
000076ea  (2)...  mov al, 0x2e
000076ec  (1)...  out dx, al
000076ed  (5)...  mov edx, 0x0000781c
000076f2  (5)...  mov eax, 0x00080000
000076f7  (3)...  mov ax, dx
000076fa  (4)...  mov dx, 0x8e00
000076fe  (5)...  mov dword ptr ds:0x000055b8, eax
00007703  (6)...  mov dword ptr ds:0x000055bc, edx
00007709  (5)...  mov edx, 0x00000021
0000770e  (1)...  in al, dx
0000770f  (2)...  jmp .+0 (0x00007711)
00007711  (2)...  jmp .+0 (0x00007713)
00007713  (5)...  and eax, 0x000000fe
00007718  (1)...  out dx, al
```




```
52  #define _set_tssldt_desc(n,addr,type) \
53  __asm__ ("movw $104,%1\n\t" \
54      "movw %%ax,%2\n\t" \
55      "rorl $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      "rorl $16,%%eax" \
61      ::"a" (addr), "m" (*(n)), "m" (*(n+2)), "m" (*(n+4)), \
62       "m" (*(n+5)), "m" (*(n+6)), "m" (*(n+7)) \
63      )
64
65  #define set_tss_desc(n,addr) _set_tssldt_desc(((char *) (n)),((int)(addr)),"0x89")
66  #define set_ldt_desc(n,addr) _set_tssldt_desc(((char *) (n)),((int)(addr)),"0x82")
67
```

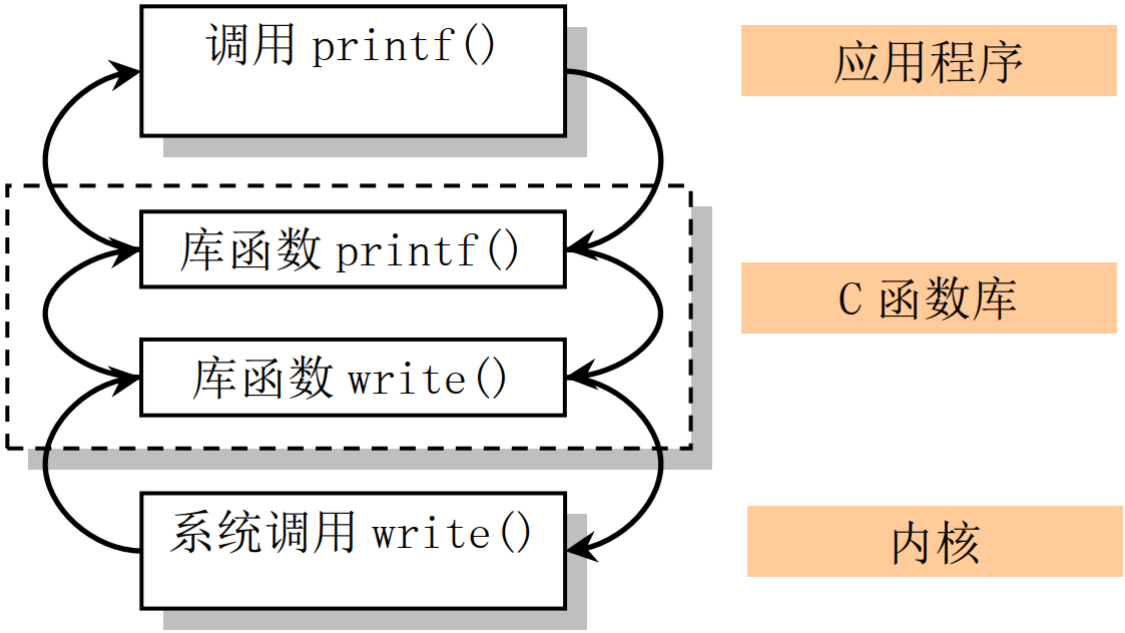
Type Field					Description
Decimal	11	10	9	8	32-Bit Mode
0	0	0	0	0	Reserved
1	0	0	0	1	16-bit TSS (Available)
2	0	0	1	0	LDT
3	0	0	1	1	16-bit TSS (Busy)
4	0	1	0	0	16-bit Call Gate
5	0	1	0	1	Task Gate
6	0	1	1	0	16-bit Interrupt Gate
7	0	1	1	1	16-bit Trap Gate
8	1	0	0	0	Reserved
9	1	0	0	1	32-bit TSS (Available)
10	1	0	1	0	Reserved
11	1	0	1	1	32-bit TSS (Busy)
12	1	1	0	0	32-bit Call Gate
13	1	1	0	1	Reserved
14	1	1	1	0	32-bit Interrupt Gate
15	1	1	1	1	32-bit Trap Gate



System-Segment and Gate-Descriptor Types

fork 系统调用

系统调用接口

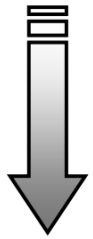


应用程序、库函数和内核系统调用之间的关系

内核代码

- 该目录下的代码文件从功能上可以分为三类
  - 硬件（异常）中断处理程序文件
  - 系统调用服务处理程序文件
  - 进程调度等通用功能文件

<table>
<tr><th colspan="2">通用程序</th><th>硬件中断程序</th><th>系统调用程序</th></tr>
<tr><td>schedule.c</td><td rowspan="3">mktime</td><td rowspan="2">asm.s</td><td rowspan="2">sys_call.s</td></tr>
<tr><td>panic.c</td></tr>
<tr><td>pintk, vsprintf</td><td>traps.c</td><td>fork.c, sys.c, exit.c, signal.c</td></tr>
</table>



调用关系

内核目录中各文件中函数的调用层次关系

表 8－1 Intel 保留的中断号含义

向量号	名称	类型	信号	说明
0	Devide error	故障	SIGFPE	当进行除以零的操作时产生。
1	Debug	陷阱 故障	SIGTRAP	当进行程序单步跟踪调试时，设置了标志寄存器 eflags 的 T 标志时产生这个中断。
2	nmi	硬件		由不可屏蔽中断 NMI 产生。
3	Breakpoint	陷阱	SIGTRAP	由断点指令 int3 产生，与 debug 处理相同。
4	Overflow	陷阱	SIGSEGV	eflags 的溢出标志 OF 引起。
5	Bounds check	故障	SIGSEGV	寻址到有效地址以外时引起。
6	Invalid Opcode	故障	SIGILL	CPU 执行时发现一个无效的指令操作码。
7	Device not available	故障	SIGSEGV	设备不存在，指协处理器。在两种情况下会产生该中断：(a)CPU 遇到一个转意指令并且 EM 置位时。在这种情况下处理程序应该模拟导致异常的指令。 (b)MP 和 TS 都在置位状态时，CPU 遇到 WAIT 或一个转意指令。在这种情况下，处理程序在必要时应该更新协处理器的状态。
8	Double fault	异常中止	SIGSEGV	双故障出错。
9	Coprocessor segment overrun	异常中止	SIGFPE	协处理器段超出。
10	Invalid TSS	故障	SIGSEGV	CPU 切换时发觉 TSS 无效。
11	Segment not present	故障	SIGBUS	描述符所指的段不存在。
12	Stack segment	故障	SIGBUS	堆栈段不存在或寻址越出堆栈段。
13	General protection	故障	SIGSEGV	没有符合 80386 保护机制（特权级）的操作引起。
14	Page fault	故障	SIGSEGV	页不在内存。
15	Intel reserved			
16	Coprocessor error	故障	SIGFPE	协处理器发出的出错信号引起。
17	Alignment check	故障		在启用内存边界检查时，若用户级数据非边界对齐时会产生该异常。
20-31	Intel reserved			
32-255	User Defined interrupts	中断		用户定义的外部中断，或使用中断指令 INT n.

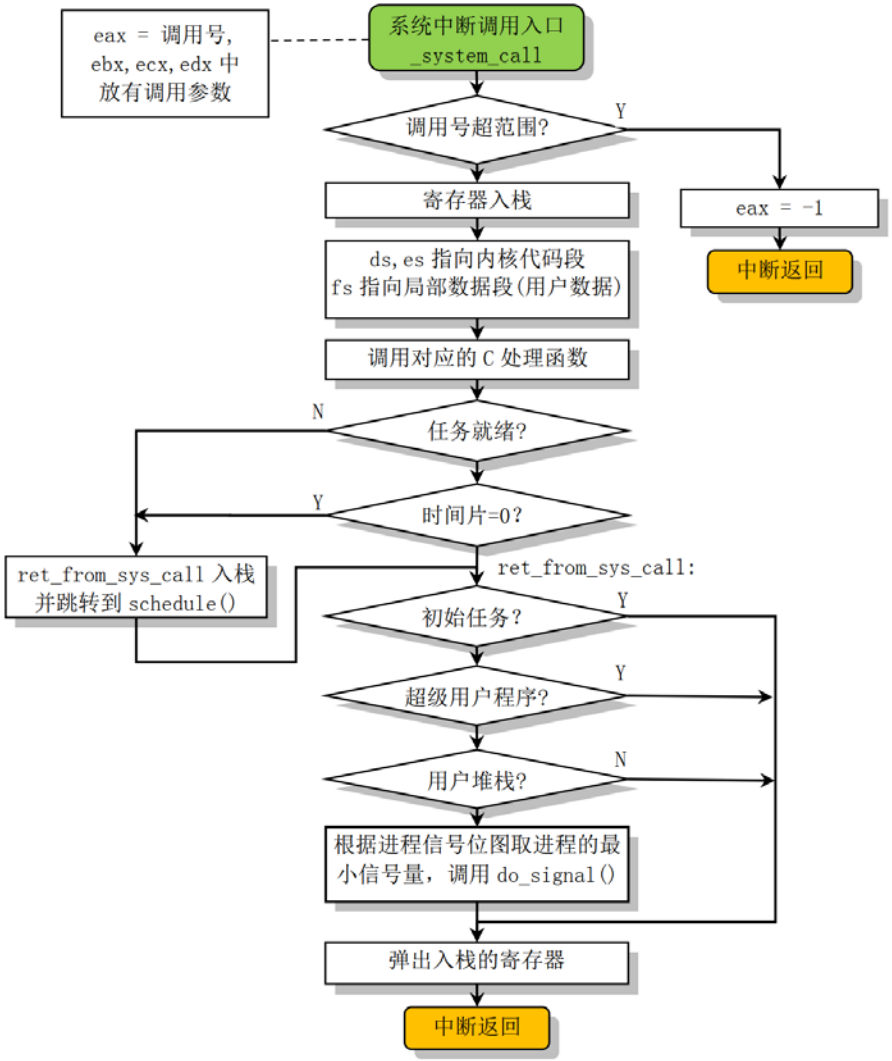
Trap.c

```
    // 以下定义了一些函数原型。
39  void page_exception(void);                  // 页异常。实际是 page_fault（mm/page.s，14）。
40
41  void divide_error(void);                    // int0（kernel/asm.s，20）。
42  void debug(void);                           // int1（kernel/asm.s，54）。
43  void nmi(void);                             // int2（kernel/asm.s，58）。
44  void int3(void);                            // int3（kernel/asm.s，62）。
45  void overflow(void);                        // int4（kernel/asm.s，66）。
46  void bounds(void);                          // int5（kernel/asm.s，70）。
47  void invalid_op(void);                      // int6（kernel/asm.s，74）。
48  void device_not_available(void);            // int7（kernel/sys_call.s，158）。
49  void double_fault(void);                    // int8（kernel/asm.s，98）。
50  void coprocessor_segment_overrun(void);     // int9（kernel/asm.s，78）。
51  void invalid_TSS(void);                     // int10（kernel/asm.s，132）。
52  void segment_not_present(void);             // int11（kernel/asm.s，136）。
53  void stack_segment(void);                   // int12（kernel/asm.s，140）。
54  void general_protection(void);              // int13（kernel/asm.s，144）。
55  void page_fault(void);                      // int14（mm/page.s，14）。
56  void coprocessor_error(void);               // int16（kernel/sys_call.s，140）。
57  void reserved(void);                        // int15（kernel/asm.s，82）。
58  void parallel_interrupt(void);              // int39（kernel/sys_call.s，295）。
59  void irq13(void);                           // int45（kernel/asm.s，86）协处理器中断处理。
60  void alignment_check(void);                 // int46（kernel/asm.s，148）。
```

系统调用处理相关程序

- **Linux 中应用程序使用系统资源时需要利用中断调用 INT 0x80 进行，并且需在寄存器 EAX 中放入调用号**
  - 如果需要给中断处理程序传递参数，则可用寄存器 EBX、ECX 和 EDX 来存放参数
- **该中断调用被称为系统调用**
- **实现系统调用的相关文件包括**
  - sys_call.s、 fork.c、 signal.c、 sys.c 和 exit.c
    - 通常以'do_'开头的中断处理过程中调用的 C 函数，要么是系统调用处理过程中通用的函数，要么是某个系统调用专用的函数
    - 而以'sys_'开头的则是指定的系统调用的专用处理函数

系统中断调用处理流程


eax = 调用号,
ebx, ecx, edx 中
放有调用参数
系统中断调用入口
_system_call
调用号超范围?
Y
eax = -1
中断返回
寄存器入栈
ds, es 指向内核代码段
fs 指向局部数据段(用户数据)
调用对应的 C 处理函数
N
任务就绪?
Y
时间片=0?
ret_from_sys_call 入栈
并跳转到 schedule()
ret_from_sys_call:
初始任务?
Y
超级用户程序?
Y
用户堆栈?
N
根据进程信号位图取进程的最
小信号量，调用 do_signal()
弹出入栈的寄存器
中断返回

fork()系统调用用于创建子进程

- **Linux 中所有进程都是进程 0（任务 0）的子进程**
- **fork.c 程序是sys_fork() 系统调用的辅助处理函数集**
  - 给出了 sys_fork()系统调用中使用的两个 C 函数：
    - find_empty_process()
    - copy_process()
  - 还包括进程内存区域验证与内存分配函数
    - verify_area()
    - copy_mem()

```
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,pgrp,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 » ;
110:
```

include\linux\sched.h

```
53: struct tss_struct {
54:     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: } « end tss_struct » ;
```

```
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,NULL,NULL,0, \
124: /* filp */  {NULL,}, \
125:     { \
126:         {0,0}, \
127: /* ldt */   {0x9f,0xc0fa00}, \
128:         {0x9f,0xc0f200}, \
129:     }, \
130: /*tss*/ {0,PAGE_SIZE+(long)&init_task,0x10,0,0,0,0,(long)&pg_dir,\
131:      0,0,0,0,0,0,0,0, \
132:      0,0,0x17,0x17,0x17,0x17,0x17,0x17, \
133:      _LDT(0),0x80000000, \
134:         {} \
135:     }, \
136: }
```

```
138: extern struct task_struct *task[NR_TASKS];
139: extern struct task_struct *last_task_used_math;
140: extern struct task_struct *current;
141: extern long volatile jiffies;
142: extern long startup_time;
```

```
4: #define NR_TASKS 64
5: #define HZ 100
6:
7: #define FIRST_TASK task[0]
8: #define LAST_TASK task[NR_TASKS-1]
```

Linux的系统调用 -- fork()

init\main.c

```
23: static inline _syscall0(int,fork)
24: static inline _syscall0(int,pause)
25: static inline _syscall1(int,setup,void *,BIOS)
26: static inline _syscall0(int,sync)
27:

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:     }
```

int fork(void)

include\unistd.h

```
58: #ifdef __LIBRARY__
59:
60: #define __NR_setup  0
61: #define __NR_exit   1
62: #define __NR_fork   2
63: #define __NR_read   3
64: #define __NR_write  4

133: #define _syscall0(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: }
145:
```

系统调用

输出：将_res赋给eax

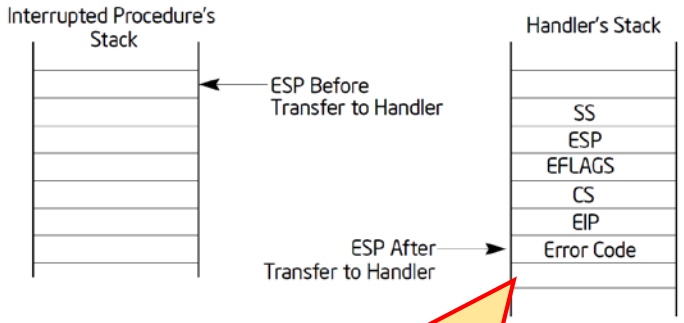
输入：__NR_fork 是2，赋给eax

“**0**” 同上寄存器，即eax

kernel\sched.c

```
void sched_init(void)
{
    int i;
    struct desc_struct * p;

    if (sizeof(struct sigaction) != 16)
        panic("Struct sigaction MUST be 16 bytes");
    set_tss_desc(gdt+FIRST_TSS_ENTRY,&(init_task.task.tss));
    set_ldt_desc(gdt+FIRST_LDT_ENTRY,&(init_task.task.ldt));
    p = gdt+2+FIRST_TSS_ENTRY;
    for(i=1;i<NR_TASKS;i++) {
        task[i] = NULL;
        p->a=p->b=0;
        p++;
        p->a=p->b=0;
        p++;
    }
/* Clear NT, so that we won't have troubles with that later on */
    __asm__("pushfl ; andl $0xffffbfff,(%esp) ; popfl");
    ltr(0);
    lldt(0);
    outb_p(0x36,0x43);      /* binary, mode 3, LSB/MSB, ch 0 */
    outb_p(LATCH & 0xff , 0x40);    /* LSB */
    outb(LATCH >> 8 , 0x40);    /* MSB */
    set_intr_gate(0x20,&timer_interrupt);
    outb(inb_p(0x21)&~0x01,0x21);
    set_system_gate(0x80,&system_call);
}
```

int 0x80



Privilege-Level Change

kernel\system_call.s

```
system_call:
    cmpl $nr_system_calls-1,%eax
    ja bad_sys_call
    push %ds
    push %es
    push %fs
    pushl %edx
    pushl %ecx      # push %ebx,%ecx,%edx as parameters
    pushl %ebx      # to the system call
    movl $0x10,%edx     # set up ds,es to kernel space
    mov %dx,%ds
    mov %dx,%es
    movl $0x17,%edx     # fs points to local data space
    mov %dx,%fs
    call sys_call_table(,%eax,4)
    pushl %eax
    movl current,%eax
    cmpl $0,state(%eax)     # state
    jne reschedule
    cmpl $0,counter(%eax)       # counter
    je reschedule
ret_from_sys_call:
    movl current,%eax       # task[0] cannot have signals
    cmpl task,%eax
    je 3f
    cmpw $0x0f,CS(%esp)     # was old code segment supervisor ?
    jne 3f
    cmpw $0x17,OLDSS(%esp)      # was stack segment = 0x17 ?
    jne 3f
    movl signal(%eax),%ebx
    movl blocked(%eax),%ecx
    notl %ecx
    andl %ebx,%ecx
    bsfl %ecx,%ecx
    je 3f
    btrl %ecx,%ebx
    movl %ebx,signal(%eax)
    incl %ecx
    pushl %ecx
    call do_signal
    popl %eax
3:  popl %eax
    popl %ebx
    popl %ecx
    popl %edx
    pop %fs
    pop %es
    pop %ds
    iret
```

kernel\system_call.s

int 0x80

include\linux\sys.h

```
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 %ebx      # 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
```

```
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_ftime, 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_acct, 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_getpgrp, sys_setsid, sys_sigaction, sys_sgetmask, 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
```

kernel\fork.c

```
136: int find_empty_process(void)
137: {
138:     int i;
139:
140:     repeat:
141:         if ((++last_pid)<0) last_pid=1;
142:         for(i=0 ; i<NR_TASKS ; i++)
143:             if (task[i] && task[i]->pid == last_pid) goto repeat;
144:     for(i=1 ; i<NR_TASKS ; i++)
145:         if (!task[i])
146:             return i;
147:     return -EAGAIN;
148: }
```

copy_process()

- **进程0将在copy_process（）函数中做非常重要的、体现父子进程创建机制的工作：**
  - 为进程1创建task_struct，将进程0的task_struct的内容复制给进程1
  - 为进程1的task_struct、tss做个性化设置
  - 为进程1创建第一个页表，将进程0的页表项内容赋给这个页表
  - 进程1共享进程0的文件
  - 设置进程1的GDT项
  - 最后将进程1设置为就绪态，使其可以参与进程间的轮转

```
int copy_process(int nr,long ebp,long edi,long esi,long gs,long none,
        long ebx,long ecx,long edx,
        long fs,long es,long ds,
        long eip,long cs,long eflags,long esp,long ss)
{
    struct task_struct *p;
    int i;
    struct file *f;

    p = (struct task_struct *) get_free_page();
    if (!p)
        return -EAGAIN;
    task[nr] = p;
    *p = *current;  /* NOTE! this doesn't copy the supervisor stack */
    p->state = TASK_UNINTERRUPTIBLE;
    p->pid = last_pid;
    p->father = current->pid;
    p->counter = p->priority;
    p->signal = 0;
    p->alarm = 0;
    p->leader = 0;      /* process leadership doesn't inherit */
    p->utime = p->stime = 0;
    p->cutime = p->cstime = 0;
    p->start_time = jiffies;
    p->tss.back_link = 0;
    p->tss.esp0 = PAGE_SIZE + (long) p;
    p->tss.ss0 = 0x10;
    p->tss.eip = eip;
    p->tss.eflags = eflags;
    p->tss.eax = 0;
    p->tss.ecx = ecx;
    p->tss.edx = edx;
    p->tss.ebx = ebx;
    p->tss.esp = esp;
    p->tss.ebp = ebp;
    p->tss.esi = esi;
    p->tss.edi = edi;
    p->tss.es = es & 0xffff;
    p->tss.cs = cs & 0xffff;
    p->tss.ss = ss & 0xffff;
    p->tss.ds = ds & 0xffff;
    p->tss.fs = fs & 0xffff;
    p->tss.gs = gs & 0xffff;
    p->tss.ldt = _LDT(nr);
    p->tss.trace_bitmap = 0x80000000;
    if (last_task_used_math == current)
        __asm__("clts ; fnsave %0"::"m" (p->tss.i387));
    if (copy_mem(nr,p)) {
        task[nr] = NULL;
        free_page((long) p);
        return -EAGAIN;
    }
    for (i=0; i<NR_OPEN;i++)
        if ((f=p->filp[i]))
            f->f_count++;
    if (current->pwd)
        current->pwd->i_count++;
    if (current->root)
        current->root->i_count++;
    if (current->executable)
        current->executable->i_count++;
    set_tss_desc(gdt+(nr<<1)+FIRST_TSS_ENTRY,&(p->tss));
    set_ldt_desc(gdt+(nr<<1)+FIRST_LDT_ENTRY,&(p->ldt));
    p->state = TASK_RUNNING;    /* do this last, just in case */
    return last_pid;
}
```

include\unistd.h

```
#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; \
}
```

int 0x80

CS:EIP

kernel\system_call.s

```
system_call:
    cmpl $nr_system_calls-1,%eax
    ja bad_sys_call
    push %ds
    push %es
    push %fs
    pushl %edx
    pushl %ecx      # push %ebx,%ecx,%edx as parameters
    pushl %ebx      # to the system call
    movl $0x10,%edx     # set up ds,es to kernel space
    mov %dx,%ds
    mov %dx,%es
    movl $0x17,%edx     # fs points to local data space
    mov %dx,%fs
    call sys_call_table(,%eax,4)
    pushl %eax
    movl current,%eax
    cmpl $0,state(%eax)     # state
    jne reschedule
```

include\linux\sys.h

```
.align 2
sys_fork:
    call find_empty_process
    testl %eax,%eax
    js 1f
    push %gs
    pushl %esi
    pushl %edi
    pushl %ebp
    pushl %eax
    call copy_process
    addl $20,%esp
1:  ret
```

p->tss.eax = 0

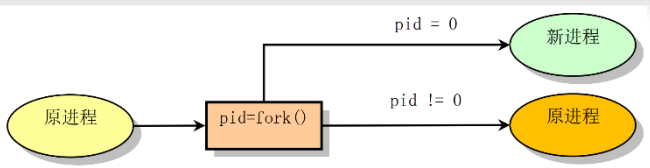
将__res赋给eax
eax = last_pid

```
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 %ebx      # 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 supervisc
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
```

```
104: void main(void)    /* This really IS void, no error here. */
105: {          /* The startup routine assumes (well, ...) this */
106: /*
107:  * Interrupts are still disabled. Do necessary setups, then
108:  * enable them
109:  */
110:     ROOT_DEV = ORIG_ROOT_DEV;
111:     drive_info = DRIVE_INFO;
112:     memory_end = (1<<20) + (EXT_MEM_K<<10);
113:     memory_end &= 0xfffff000;
114:     if (memory_end > 16*1024*1024)
115:         memory_end = 16*1024*1024;
116:     if (memory_end > 12*1024*1024)
117:         buffer_memory_end = 4*1024*1024;
118:     else if (memory_end > 6*1024*1024)
119:         buffer_memory_end = 2*1024*1024;
120:     else
121:         buffer_memory_end = 1*1024*1024;
122:     main_memory_start = buffer_memory_end;
123: #ifdef RAMDISK
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:     }
141: /*
142:  *   NOTE!!   For any other task 'pause()' would mean we have to get a
143:  * signal to awaken, but task0 is the sole exception (see 'schedule()')
144:  * as task 0 gets activated at every idle moment (when no other tasks
145:  * can run). For task0 'pause()' just means we go check if some other
146:  * task can run, and if not we return here.
147:  */
148:     for(;;) pause();
149: } « end main »
```

```
133: #define _syscall0(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: }
```

last_pid > 0



init\main.c

```
104: void main(void)    /* This really IS void, no error here. */
105: {          /* The startup routine assumes (well, ...
106: /*
107:  * Interrupts are still disabled. Do necessary setups, then
108:  * enable them
109:  */
110:     ROOT_DEV = ORIG_ROOT_DEV;
111:     drive_info = DRIVE_INFO;
112:     memory_end = (1<<20) + (EXT_MEM_K<<10);
113:     memory_end &= 0xfffff000;
114:     if (memory_end > 16*1024*1024)
115:         memory_end = 16*1024*1024;
116:     if (memory_end > 12*1024*1024)
117:         buffer_memory_end = 4*1024*1024;
118:     else if (memory_end > 6*1024*1024)
119:         buffer_memory_end = 2*1024*1024;
120:     else
121:         buffer_memory_end = 1*1024*1024;
122:     main_memory_start = buffer_memory_end;
123: #ifdef RAMDISK
124:     main_memory_start += rd_init(main_memory_start, RAMDIS
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:     }
141: /*
142:  *   NOTE!!   For any other task 'pause()' would mean we have to get a
143:  * signal to awaken, but task0 is th
144:  * as task 0 gets activated at e
145:  * can run). For task0 'pause()
146:  * task can run, and if not we
147:  */
148:     for(;;) pause();
149: } « end main »
```

```
23: static inline _syscall0(int,fork)
24: static inline _syscall0(int,pause)
25: static inline _syscall1(int,setup,void *,BIOS)
26: static inline _syscall0(int,sync)
```

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_ftime, 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_acct, 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_getpgrp, sys_setsid, sys_sigaction, sys_sgetmask, sys_ssetmask,
86: sys_setreuid,sys_setregid };
```

kernel\sched.c

```
144: int sys_pause(void)
145: {
146:     current->state = TASK_INTERRUPTIBLE;
147:     schedule();
148:     return 0;
149: }
```

```
140:         }
141:         switch_to(next);
142: } « end schedule »
```

p->tss.cs
p->tss.eip
p->tss.eax = **0**

include\linux\sched.h

```
173: #define switch_to(n) {\
174: struct {long a,b;} __tmp; \
175: __asm__("cmpl %%ecx,current\n\t" \
176:     "je 1f\n\t" \
177:     "movw %%dx,%1\n\t" \
178:     "xchgl %%ecx,current\n\t" \
179:     "ljmp *%0\n\t" \
180:     "cmpl %%ecx,last_task_used_math\n\t" \
181:     "jne 1f\n\t" \
182:     "clts\n" \
183:     "1:" \
184:     ::"m" (*&__tmp.a),"m" (*&__tmp.b), \
185:     "d" (_TSS(n)),"c" ((long) task[n])); \
186: }
```

```
133: #define _syscall0(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: }
```

returen 0

```
void sched_init(void)
{
    int i;
    struct desc_struct * p;

    if (sizeof(struct sigaction) != 16)
        panic("Struct sigaction MUST be 16 bytes");
    set_tss_desc(gdt+FIRST_TSS_ENTRY,&(init_task.task.tss));
    set_ldt_desc(gdt+FIRST_LDT_ENTRY,&(init_task.task.ldt));
    p = gdt+2+FIRST_TSS_ENTRY;
    for(i=1;i<NR_TASKS;i++) {
        task[i] = NULL;
        p->a=p->b=0;
        p++;
        p->a=p->b=0;
        p++;
    }
/* Clear NT, so that we won't have troubles with that later on */
    __asm__("pushfl ; andl $0xffffbfff,(%esp) ; popfl");
    ltr(0);
    lldt(0);
    outb_p(0x36,0x43);      /* binary, mode 3, LSB/MSB, ch 0 */
    outb_p(LATCH & 0xff , 0x40);    /* LSB */
    outb(LATCH >> 8 , 0x40);    /* MSB */
    set_intr_gate(0x20,&timer_interrupt);
    outb(inb_p(0x21)&~0x01,0x21);
    set_system_gate(0x80,&system_call);
}
```

kernel\sched.c

Reg N...	Hex Value	Decimal
cr0	80000013	

L.Addr...	B...	Mnemonic
000068e3	(5)...	mov eax, 0x00000017
000068e8	(2)...	mov ds, ax
000068ea	(2)...	mov es, ax
000068ec	(2)...	mov fs, ax
000068ee	(2)...	mov gs, ax
000068f0	(5)...	*mov eax, 0x00000002*
000068f5	(3)...	add esp, 0x00000010
000068f8	(2)...	int 0x80
000068fa	(2)...	*test eax, eax*
000068fc	(2)...	js .+70 (0x00006944)
000068fe	(2)...	test eax, eax
00006900	(2)...	jz .+59 (0x0000693d)
00006902	(5)...	mov edx, 0x0000001d
00006907	(1)...	nop
00006908	(2)...	mov eax, edx
0000690a	(2)...	int 0x80
0000690c	(2)...	jmp .-6 (0x00006908)
0000690e	(5)...	mov dword ptr ds:0x0001dac4, eax
00006913	(5)...	jmp .-506 (0x0000671e)
00006918	(5)...	cmp eax, 0x00600000
0000691d	(2)...	jle .+15 (0x0000692e)

eax=2

```
0000773c t reschedule
00007746 T system_call
0000777a t ret_from_sys_call
000077c0 T coprocessor_error
000077e2 T device_not_available
0000781c T timer_interrupt
00007854 T sys_execve
00007862 T sys_fork
0000787a T hd_interrupt
```

Reg N...	Hex Value	Decimal
eax	00000002	2
ebx	00000005	5
ecx	0001b160	110944
edx	00000021	33
esi	00000002	2
edi	00000001	1
ebp	00020e6c	134764
esp	0001b12c	110892
eip	00007746	30534
eflags	00000206	
cs	0008	
ds	0017	
es	0017	
ss	0010	
fs	0017	
gs	0017	
gdtr	00005cb8 (7ff)	
idtr	000054b8 (7ff)	
ldtr	a410	
tr	a428	
cr0	80000013	

L.Addr...	B...	Mnemonic
00007746	(3)...	cmp eax, 0x00000047
00007749	(2)...	jnbe .-21 (0x00007736)
0000774b	(1)...	push ds
0000774c	(1)...	push es
0000774d	(2)...	push fs
0000774f	(1)...	push edx
00007750	(1)...	push ecx
00007751	(1)...	push ebx
00007752	(5)...	mov edx, 0x00000010
00007757	(2)...	mov ds, dx
00007759	(2)...	mov es, dx
0000775b	(5)...	mov edx, 0x00000017
00007760	(2)...	mov fs, dx
00007762	(7)...	call dword ptr ds:[eax*4+106528]
00007769	(1)...	push eax
0000776a	(5)...	mov eax, dword ptr ds:0x0001b140
0000776f	(3)...	cmp dword ptr ds:[eax], 0x00000000
00007772	(2)...	jnz .-56 (0x0000773c)
00007774	(4)...	cmp dword ptr ds:[eax+4], 0x00000000
00007778	(2)...	jz .-62 (0x0000773c)
0000777a	(5)...	mov eax, dword ptr ds:0x0001b140

```
system_call:
    cmpl $nr_system_calls-1,%eax
    ja bad_sys_call
    push %ds
    push %es
    push %fs
    pushl %edx
    pushl %ecx      # push %ebx,%
    pushl %ebx      # to the syst
    movl $0x10,%edx     # set up
    mov %dx,%ds
    mov %dx,%es
    movl $0x17,%edx     # fs poir
    mov %dx,%fs
    call sys_call_table(,%eax,4)
    pushl %eax
    movl current,%eax
    cmpl $0,state(%eax)     # sta
    jne reschedule
    cmpl $0,counter(%eax)       #
    je reschedule
ret_from_sys_call:
```

kernel\system_call.s

```
00007757  (2)...  mov ds, dx
00007759  (2)...  mov es, dx
0000775b  (5)...  mov edx, 0x00000017
00007760  (2)...  mov fs, dx
00007762  (7)...  call dword ptr ds:[eax*4+106528]
00007769  (1)...  push eax
0000776a  (5)...  mov eax, dword ptr ds:0x0001b140
0000776f  (3)...  cmp dword ptr ds:[eax], 0x00000000
00007772  (2)...  jnz .-56 (0x0000773c)
```

查表，eax=2，sys_fork

```
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_ftime, 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_acct, sys_phys,
83  sys_lock, sys_ioctl, sys_fcntl, sys_mpx, sys_setpgid, sys_ulimit,
84  sys_uname, sys_umask, sys_chroot,
85  sys_getpgrp, sys_setsid, sys_sig
86  sys_setreuid,sys_setregid };
```

include\linux\sys.h

```
48  0000773c t reschedule
49  00007746 T system_call
50  0000777a t ret_from_sys_call
51  000077c0 T coprocessor_error
52  000077e2 T device_not_available
53  0000781c T timer_interrupt
54  00007854 T sys_execve
55  00007862 T sys_fork
56  0000787a T hd_interrupt
```

HEX	1 A020
DEC	106,528

进制转换，找到 sys_call_table 内存地址为 0x1A020

P.Address	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Ascii
0x0001A020	FC	3D	01	00	8A	93	00	00	62	78	00	00	BD	AE	00	00	.=.....bx.....
0x0001A030	97	B0	00	00	F7	A9	00	00	B2	AC	00	00	9F	93	00	00	.....
0x0001A040	93	AC	00	00	51	0F	01	00	A8	0C	01	00	54	78	00	00	....Q.....Tx..
0x0001A050	AB	A7	00	00	C5	8A	00	00	0C	01	01	00	BD	A8	00	00	.....
0x0001A060	6A	A9	00	00	79	89	00	00	E9	DE	00	00	78	AD	00	00	j...y.....x...
0x0001A070	B7	75	00	00	86	D0	00	00	FA	CE	00	00	9B	8B	00	00	.u.....
0x0001A080	CF	75	00	00	2A	8C	00	00	7F	89	00	00	5F	75	00	00	.u..*....._u..

kernel\system_call.s

```
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
```

信号处理

信号处理

- 信号是一种“软件中断”处理机制
  - 有许多较为复杂的程序会使用到信号
- 信号机制提供了一种处理异步事件的方法
  - 可用作进程之间通信的一种简单消息机制，使得一个进程可以向另一个进程发送信号
- 信号通常是一个正整数值
  - 它除了指明自己的信号类别，不携带任何其他信息

Linux 中的信号

- 通常使用一个无符号长整数（ 32 位）中的比特位来表示各种不同信号
  - 因此系统中最多可有 32 个不同的信号
- 本版 Linux 定义了 22 种不同的信号
  - 其中 20 种信号是POSIX.1 标准中规定的信号
  - 另外 2 种是 Linux 的专用信号
    - SIGUNUSED（未定义）和 SIGSTKFLT（堆栈错）
- 收到信号三种不同的处理或操作方法
  - 忽略信号 （SIGKILL 和 SIGSTOP除外）
  - 进程定义自己的信号处理程序来处理信号
  - 执行系统的默认信号处理操作

include\signal.h

```
 9: #define _NSIG        32
10: #define NSIG         _NSIG
11:
12: #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 SIGTTOU      22
```

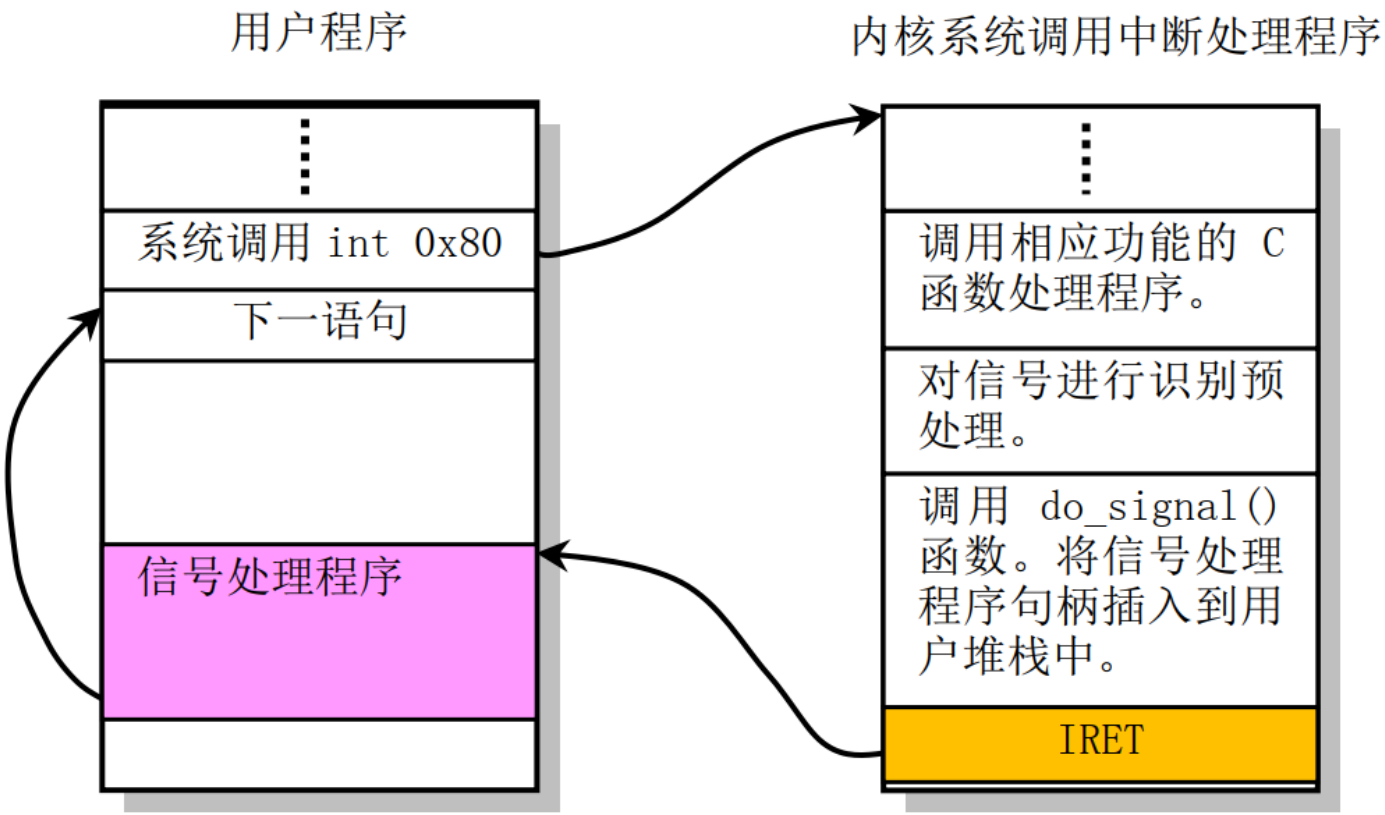
信号处理实现

- 当用户想使用自己的信号处理程序（信号句柄）时
  - 需要使用 signal()或 sigaction()系统调用首先在进程任务数据结构中设置sigaction[]结构数组项
  - 把自身信号处理程序的指针和一些属性“记录”在该结构项中
- 当内核在退出一个系统调用和某些中断过程时会检测当前进程是否收到信号
  - 若收到特定信号，内核就会根据进程任务数据结构中sigaction[]中对应信号的结构项执行用户自己定义的信号处理服务程序

```
struct sigaction {
        void (*sa_handler)(int);          // 信号处理句柄。
        sigset_t sa_mask;                 // 信号的屏蔽码，可以阻塞指定的信号集。
        int sa_flags;                     // 信号选项标志。
        void (*sa_restorer)(void);        // 信号恢复函数指针（系统内部使用）。
};
```

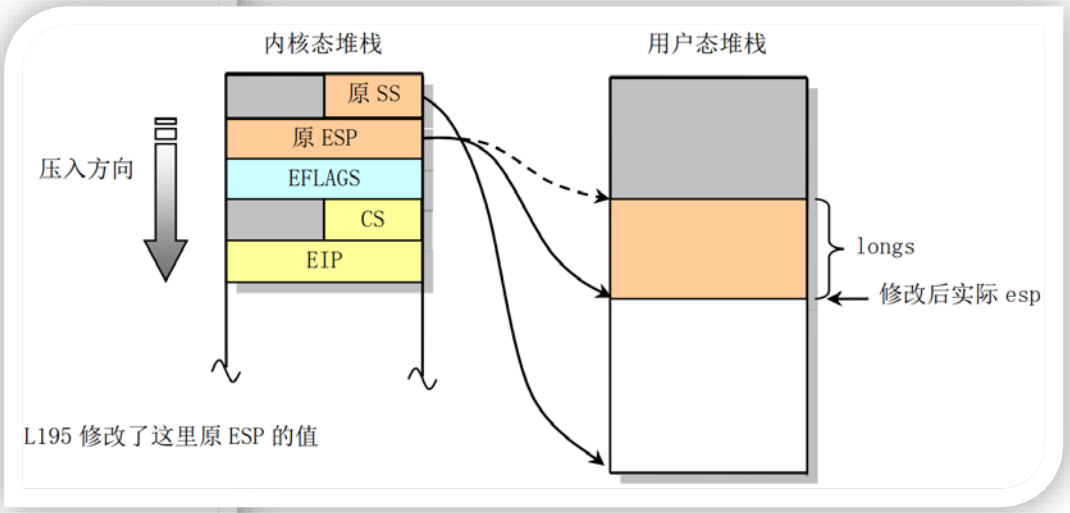
do_signal()函数

- do_signal()函数是内核系统调用(int 0x80)中断处理程序中对信号的预处理程序



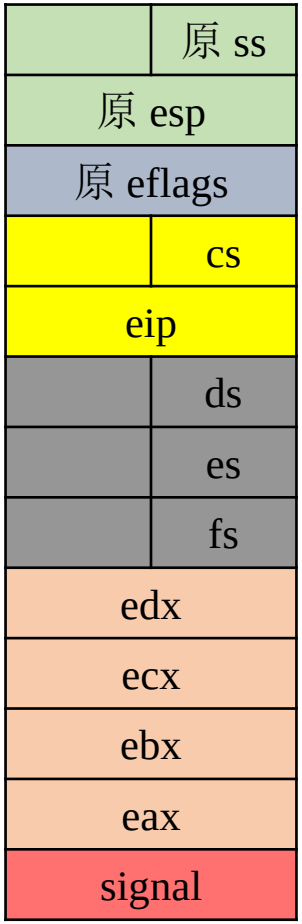
do_signal()函数对用户堆栈的修改

```
82: void do_signal(long signr,long eax, long ebx, long ecx, long edx,
83:     long fs, long es, long ds,
84:     long eip, long cs, long eflags,
85:     unsigned long * esp, long ss)
86: {
87:     unsigned long sa_handler;
88:     long old_eip=eip;
89:     struct sigaction * sa = current->sigaction + signr - 1;
90:     int longs;
91:     unsigned long * tmp_esp;
92:
93:     sa_handler = (unsigned long) sa->sa_handler;
94:     if (sa_handler==1)
95:         return;
96:     if (!sa_handler) {
97:         if (signr==SIGCHLD)
98:             return;
99:         else
100:            do_exit(1<<(signr-1));
101:    }
102:    if (sa->sa_flags & SA_ONESHOT)
103:        sa->sa_handler = NULL;
104:    *(&eip) = sa_handler;
105:    longs = (sa->sa_flags & SA_NOMASK)?7:8;
106:    *(&esp) -= longs;
107:    verify_area(esp,longs*4);
108:    tmp_esp=esp;
109:    put_fs_long((long) sa->sa_restorer,tmp_esp++);
110:    put_fs_long(signr,tmp_esp++);
111:    if (!(sa->sa_flags & SA_NOMASK))
112:        put_fs_long(current->blocked,tmp_esp++);
113:    put_fs_long(eax,tmp_esp++);
114:    put_fs_long(ecx,tmp_esp++);
115:    put_fs_long(edx,tmp_esp++);
116:    put_fs_long(eflags,tmp_esp++);
117:    put_fs_long(old_eip,tmp_esp++);
118:    current->blocked |= sa->sa_mask;
119: } « end do_signal »
```



```
_system_call:
        cmpl $nr_system_calls-1,%eax
        ja bad_sys_call
        push %ds
        push %es
        push %fs
        pushl %edx
        pushl %ecx              # push %ebx,%ecx,%edx as parameters
        pushl %ebx              # to the system call
        movl $0x10,%edx         # set up ds,es to kernel space
        mov %dx,%ds
        mov %dx,%es
        movl $0x17,%edx         # fs points to local data space
        mov %dx,%fs
        call _sys_call_table(,%eax,4)
        pushl %eax
        movl _current,%eax
        cmpl $0,state(%eax)     # state
        jne reschedule
        cmpl $0,counter(%eax)   # counter
        je reschedule
ret_from_sys_call:
        movl _current,%eax      # task[0] cannot have signals
        cmpl _task,%eax
        je 3f
        cmpw $0x0f,CS(%esp)     # was old code segment supervisor ?
        jne 3f
        cmpw $0x17,OLDSS(%esp)  # was stack segment = 0x17 ?
        jne 3f
        movl signal(%eax),%ebx
        movl blocked(%eax),%ecx
        notl %ecx
        andl %ebx,%ecx
        bsfl %ecx,%ecx
        je 3f
        btrl %ecx,%ebx
        movl %ebx,signal(%eax)
        incl %ecx
        pushl %ecx
        call _do_signal
        popl %eax
3:      popl %eax
        popl %ebx
        popl %ecx
        popl %edx
        pop %fs
        pop %es
        pop %ds
        iret
```

int 0x80




```
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);
```

Copy 参数到用户栈

iret 返回哪？

系统调用中断堆栈视图

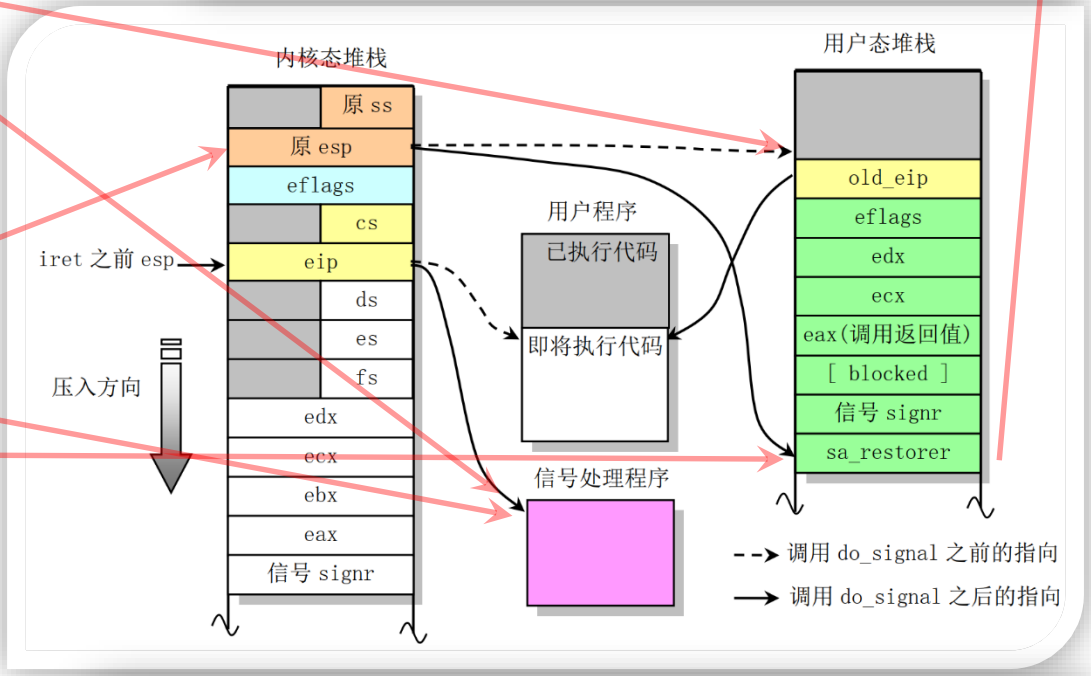
修改用户态堆栈的具体过程

```
82: void do_signal(long signr,long eax, long ebx, long ecx, long edx,
83:     long fs, long es, long ds,
84:     long eip, long cs, long eflags,
85:     unsigned long * esp, long ss)
86: {
87:     unsigned long sa_handler;
88:     long old_eip=eip;
89:     struct sigaction * sa = current->sigaction + signr - 1;
90:     int longs;
91:     unsigned long * tmp_esp;
92:
93:     sa_handler = (unsigned long) sa->sa_handler;
94:     if (sa_handler==1)
95:         return;
96:     if (!sa_handler) {
97:         if (signr==SIGCHLD)
98:             return;
99:         else
100:            do_exit(1<<(signr-1));
101:    }
102:    if (sa->sa_flags & SA_ONESHOT)
103:        sa->sa_handler = NULL;
104:    *(&eip) = sa_handler;
105:    longs = (sa->sa_flags & SA_NOMASK)?7:8;
106:    *(&esp) -= longs;
107:    verify_area(esp,longs*4);
108:    tmp_esp=esp;
109:    put_fs_long((long) sa->sa_restorer,tmp_esp++);
110:    put_fs_long(signr,tmp_esp++);
111:    if (!(sa->sa_flags & SA_NOMASK))
112:        put_fs_long(current->blocked,tmp_esp++);
113:    put_fs_long(eax,tmp_esp++);
114:    put_fs_long(ecx,tmp_esp++);
115:    put_fs_long(edx,tmp_esp++);
116:    put_fs_long(eflags,tmp_esp++);
117:    put_fs_long(old_eip,tmp_esp++);
118:    current->blocked |= sa->sa_mask;
119: } « end do_signal »
```

ret 返回哪？

```
1  addl $4, %esp
2  popl %eax
3  popl %ecx
4  popl %edx
5  popfl // 即eflags寄存器
6  ret
```

```
struct sigaction {
        void (*sa_handler)(int);          // 信号处理句柄。
        sigset_t sa_mask;                 // 信号的屏蔽码，可以阻塞指定的信号集。
        int sa_flags;                     // 信号选项标志。
        void (*sa_restorer)(void);        // 信号恢复函数指针（系统内部使用）。
};
```

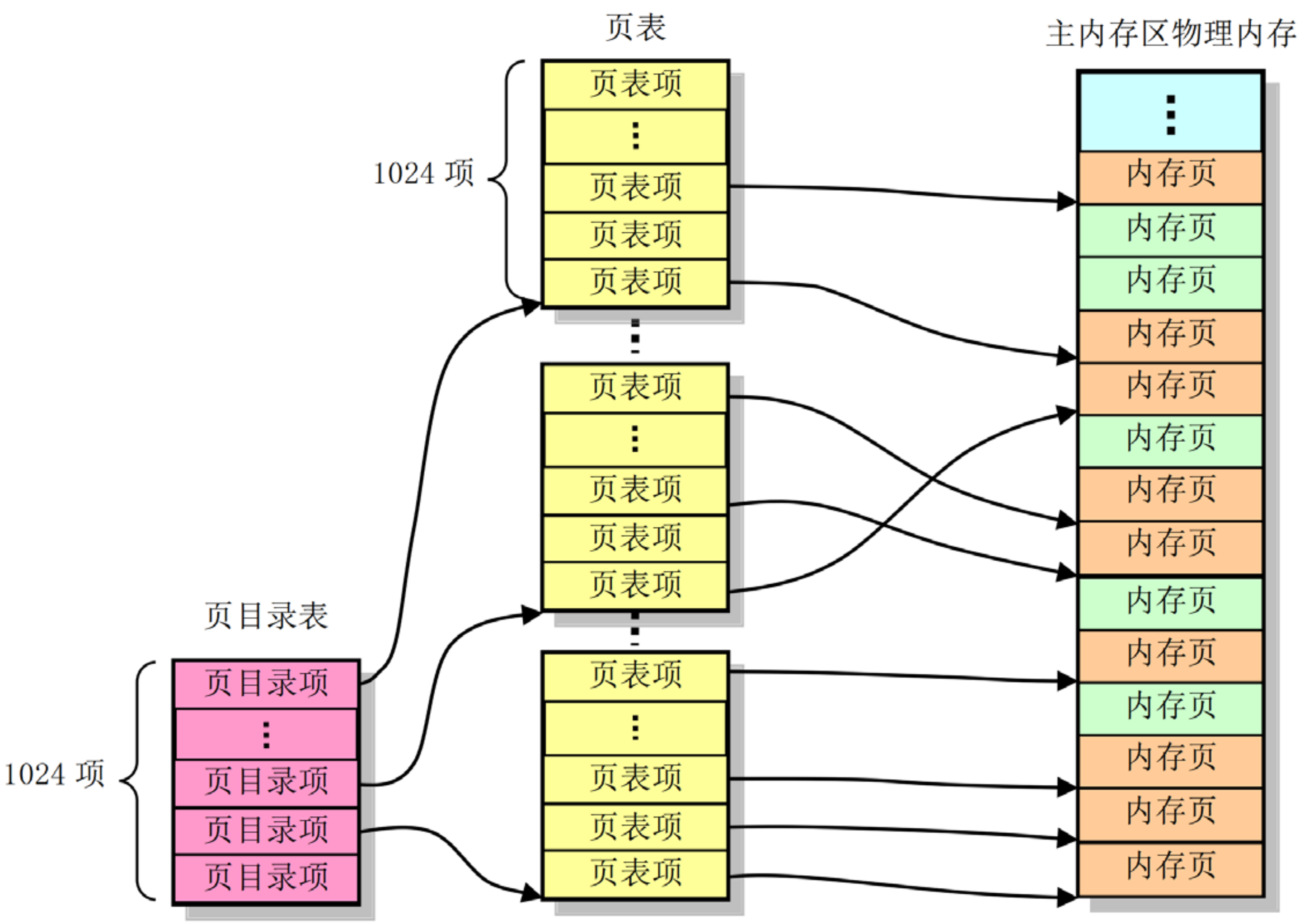



```
35: static inline void put_fs_long(unsigned long val,unsigned long * addr)
36: {
37: __asm__ ("movl %0,%%fs:%1"::"r" (val),"m" (*addr));
38: }
```

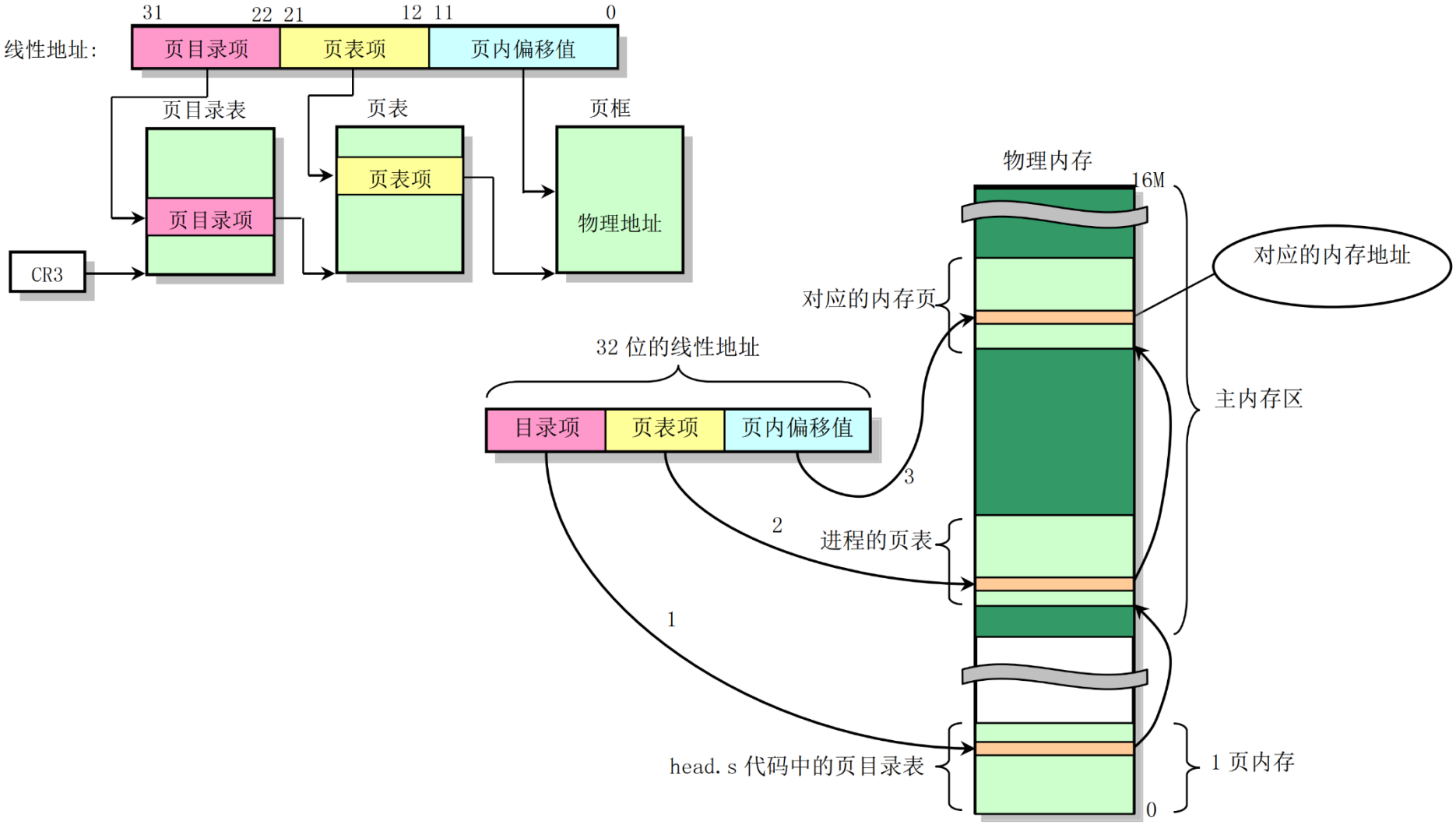
fs 指向用户空间

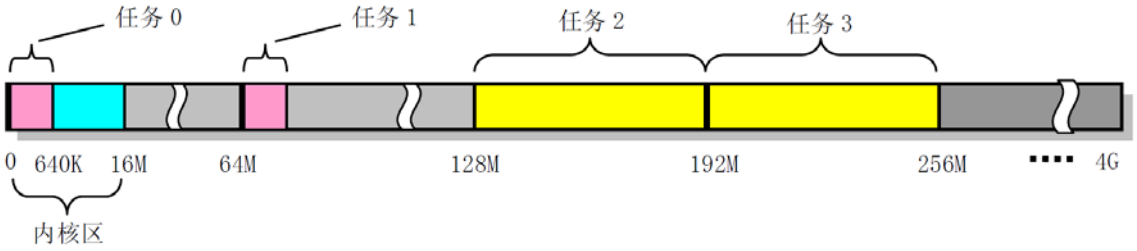
内存管理

页目录表和页表结构示意图

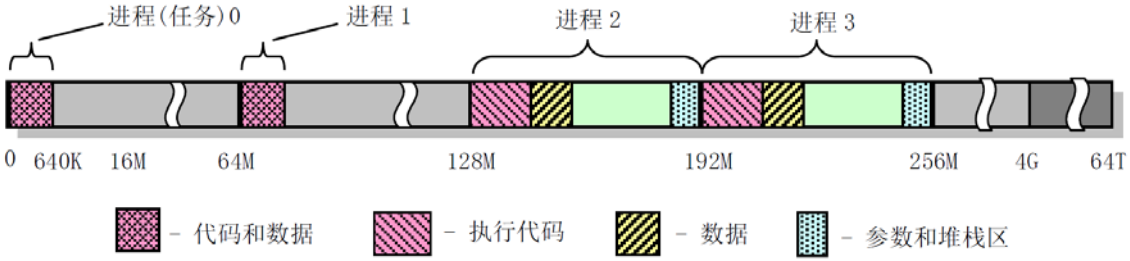


线性地址对应的物理地址

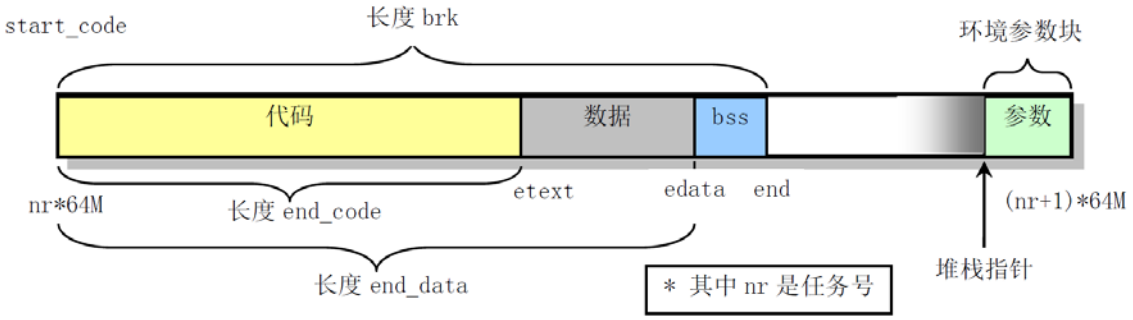




线性地址空间的使用示意图



系统任务在虚拟空间中顺序排列所占的空间范围



进程代码和数据在其逻辑地址空间中的分布

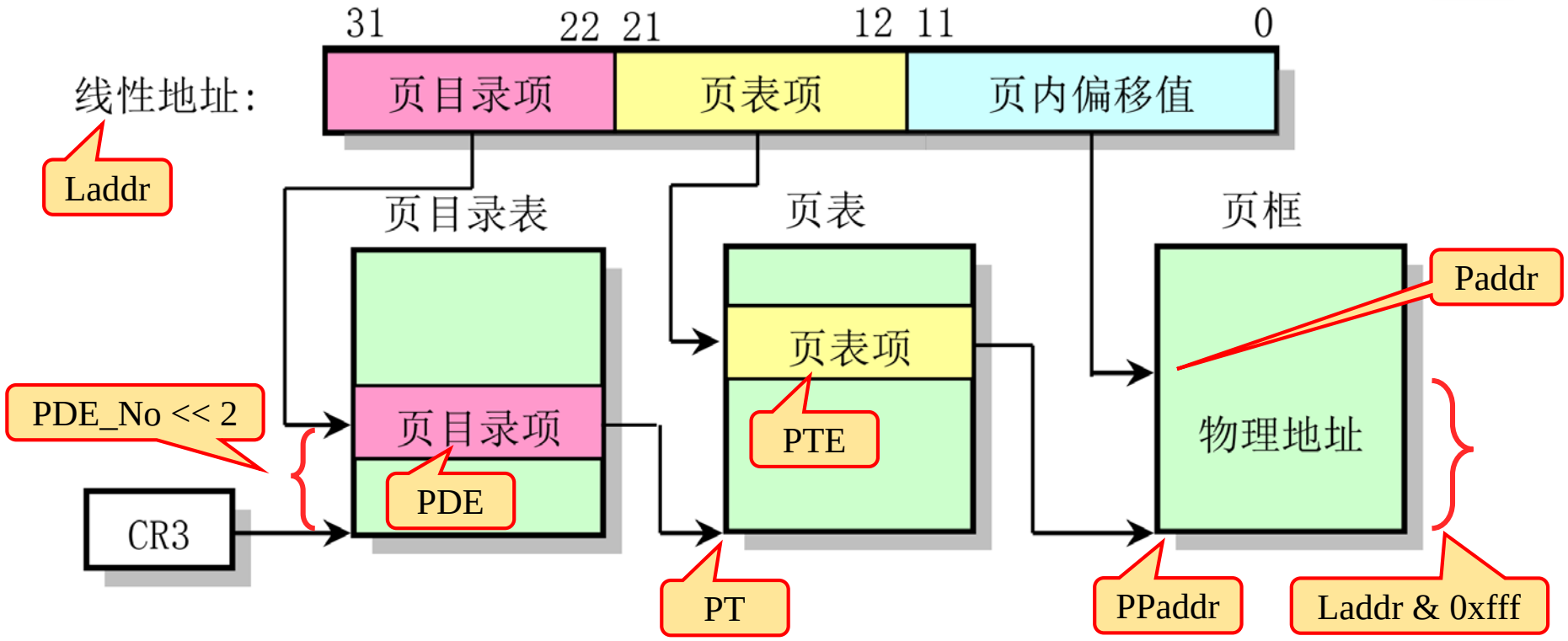
线性地址和逻辑地址的分解

- **根据 CPU的分页管理机制， 一个32位的线性地址addr可以分解为**
  - 页目录项PDE（ Page Directory Entry）号（位 31-22）
  - 页表项 PTE（ Page Table Entry） 号（位 21-12）
  - 页内偏移（位 11-0）

```
目录项号：PDE_No = (addr >> 22);
页表项号：PTE_No = (addr >> 12) & 0x3ff;
页内偏移：Offset = (addr & 0xfff);
```

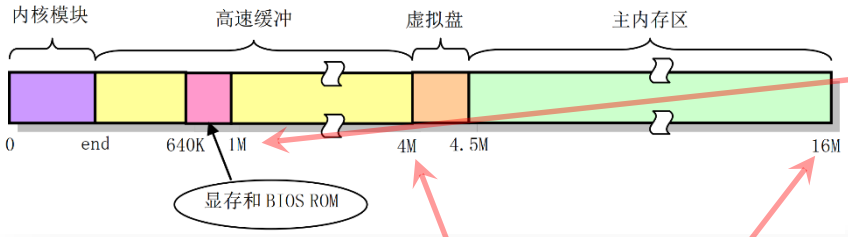
```
PDE = (PDE_No << 2);
    = ((Laddr >> 22) << 2);
    = ((Laddr >> 20) & 0xffc);
```

```
PT = (*PDE) & 0xfffff000;
PTE = PT + (PTE_No << 2);
    = PT + (((Laddr >> 12) & 0x3ff) << 2);
    = (*PDE) & 0xfffff000 + ((Laddr >> 10) & 0xffc);
    = (*((Laddr >> 20) & 0xffc) & 0xfffff000) + ((Laddr >> 10) & 0xffc);
```




```
PPaddr = (*PTE) & 0xfffff000;
Paddr = PPaddr + Laddr & 0xfff;
   = (*PTE) & 0xfffff000 + Laddr & 0xfff;
   = *(((Laddr >> 10) & 0xffc) + (*((Laddr >> 20) & 0xffc) & 0xfffff000)) + Laddr & 0xfff;
```

给定线性地址到物理地址的转换



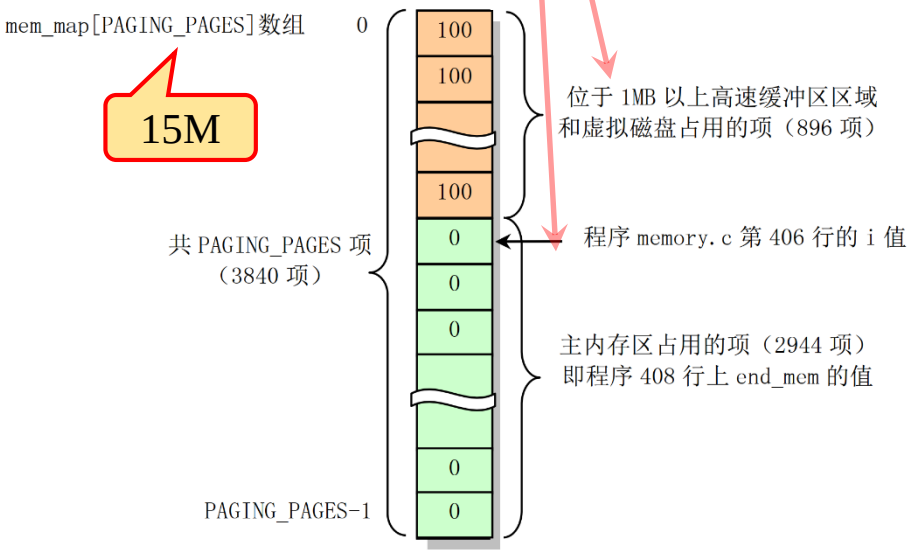

```
42: /* these are not to be changed without changing head.s etc */
43: #define LOW_MEM 0x100000
44: #define PAGING_MEMORY (15*1024*1024)
45: #define PAGING_PAGES (PAGING_MEMORY>>12)
46: #define MAP_NR(addr) (((addr)-LOW_MEM)>>12)
47: #define USED 100
```

```
104: void main(void)    /* This really IS void, no error here. */
105: {          /* The startup routine assumes (well, ...) this */
106: /*
107:  * Interrupts are still disabled. Do necessary setups, then
108:  * enable them
109:  */
110:     ROOT_DEV = ORIG_ROOT_DEV;
111:     drive_info = DRIVE_INFO;
112:     memory_end = (1<<20) + (EXT_MEM_K<<10);
113:     memory_end &= 0xfffff000;
114:     if (memory_end > 16*1024*1024)
115:         memory_end = 16*1024*1024;
116:     if (memory_end > 12*1024*1024)
117:         buffer_memory_end = 4*1024*1024;
118:     else if (memory_end > 6*1024*1024)
119:         buffer_memory_end = 2*1024*1024;
120:     else
121:         buffer_memory_end = 1*1024*1024;
122:     main_memory_start = buffer_memory_end;
123: #ifdef RAMDISK
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:     }
```

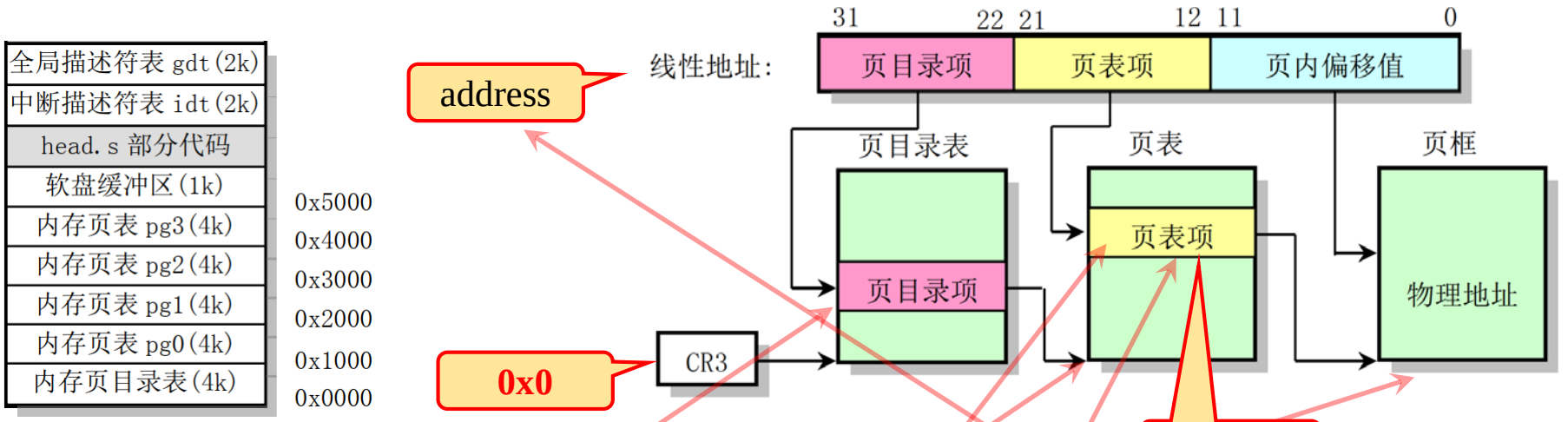
init\main.c

```
400: void mem_init(long start_mem, long end_mem)
401: {
402:     int i;
403:
404:     HIGH_MEMORY = end_mem;
405:     for (i=0 ; i<PAGING_PAGES ; i++)
406:         mem_map[i] = USED;
407:     i = MAP_NR(start_mem);
408:     end_mem -= start_mem;
409:     end_mem >>= 12;
410:     while (end_mem-->0)
411:         mem_map[i++]=0;
412: }
```

mm\memory.c



mem_map[]数组初始化情况




```
197: unsigned long put_page(unsigned long page,unsigned long address)
198: {
199:     unsigned long tmp, *page_table;
200:
201: /* NOTE !!! This uses the fact that _pg_dir=0 */
202:
203:     if (page < LOW_MEM || page >= HIGH_MEMORY)
204:         printk("Trying to put page %p at %p\n",page,address);
205:     if (mem_map[(page-LOW_MEM)>>12] != 1)
206:         printk("mem_map disagrees with %p at %p\n",page,address);
207:     page_table = (unsigned long *) ((address>>20) & 0xffc);
208:     if ((*page_table)&1)
209:         page_table = (unsigned long *) (0xfffff000 & *page_table);
210:     else {
211:         if (!(tmp=get_free_page()))
212:             return 0;
213:         *page_table = tmp|7;
214:         page_table = (unsigned long *) tmp;
215:     }
216:     page_table[(address>>12) & 0x3ff] = page | 7;
217: /* no need for invalidate */
218:     return page;
219: } « end put_page »
220:
```

这里是*page_table
指的是页目录项的值

P=1 表示有效

Address中间的页表项

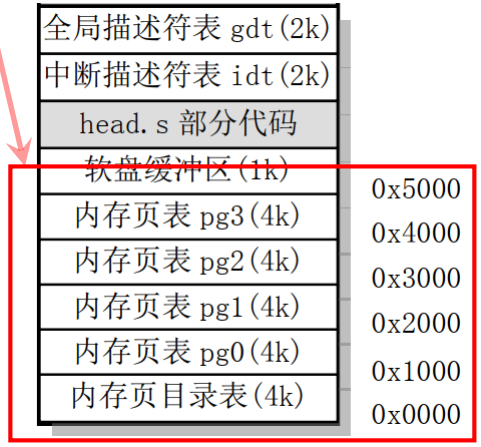
页目录初始化

```
8:  *  head.s contains the 32-bit startup code.
9:  *
10: * NOTE!!! Startup happens at absolute address 0x00000000,
11: * the page directory will exist. The startup code will be
12: * the page directory.
13: */
14: .text
15: .globl idt,gdt,pg_dir,tmp_floppy_area
16: pg_dir:
17: .globl startup_32
18: startup_32:
19:     movl $0x10,%eax
20:     mov %ax,%ds
```

boot\head.s

```
199: .align 2
200: setup_paging:
201:     movl $1024*5,%ecx       /* 5 pages - pg_dir+4 page tables */
202:     xorl %eax,%eax
203:     xorl %edi,%edi          /* pg_dir is at 0x000 */
204:     cld;rep;stosl
205:     movl $pg0+7,pg_dir      /* set present bit/user r/w */
206:     movl $pg1+7,pg_dir+4        /*  --------- " " --------- */
207:     movl $pg2+7,pg_dir+8        /*  --------- " " --------- */
208:     movl $pg3+7,pg_dir+12       /*  --------- " " --------- */
209:     movl $pg3+4092,%edi
210:     movl $0xfff007,%eax     /*  16Mb - 4096 + 7 (r/w user,p) */
211:     std
212: 1:  stosl           /* fill pages backwards - more efficient :-) */
213:     subl $0x1000,%eax
214:     jge 1b
215:     xorl %eax,%eax      /* pg_dir is at 0x0000 */
216:     movl %eax,%cr3      /* cr3 - page directory start */
217:     movl %cr0,%eax
218:     orl $0x80000000,%eax
219:     movl %eax,%cr0          /* set paging (PG) bit */
220:     ret         /* this also flushes prefetch-queue */
```

```
111: /*
112:  * I put the kernel page tables right after the page directory,
113:  * using 4 of them to span 16 Mb of physical memory. People with
114:  * more than 16MB will have to expand this.
115:  */
116: .org 0x1000
117: pg0:
118:
119: .org 0x2000
120: pg1:
121:
122: .org 0x3000
123: pg2:
124:
125: .org 0x4000
126: pg3:
127:
128: .org 0x5000
```



缺页处理 do_no_page

- 执行缺页处理
  - 访问不存在页面的处理函数，页异常中断处理过程中调用此函数。在 page.s 程序中被调用
  - 函数参数 error_code 和 address 是进程在访问页面时因缺页产生异常而由 CPU 自动生成
  - 该函数首先查看所缺页是否在交换设备中，若是则交换进来
  - 否则尝试与已加载的相同文件进行页面共享
  - 或者只是由于进程动态申请内存页面而只需映射一页物理内存页即可
  - 若共享操作不成功，那么只能从相应文件中读入所缺的数据页面到指定线性地址处

kernel\trap.c

```
181: void trap_init(void)
182: {
183:     int i;
184:
185:     set_trap_gate(0,&divide_error);
186:     set_trap_gate(1,&debug);
187:     set_trap_gate(2,&nmi);
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_segment);
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);
```

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))
```

mm\page.s

```
14: page_fault:
15:     xchgl %eax,(%esp)
16:     pushl %ecx
17:     pushl %edx
18:     push %ds
19:     push %es
20:     push %fs
21:     movl $0x10,%edx
22:     mov %dx,%ds
23:     mov %dx,%es
24:     mov %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:     pop %fs
35:     pop %es
36:     pop %ds
37:     popl %edx
38:     popl %ecx
39:     popl %eax
40:     iret
```

```
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 &= 0xfffff000;
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

调度算法

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

基于优先级排队的调度策略

5.7.5进程调度

内核中的调度程序用于选择系统中下一个要运行的进程。这种选择运行机制是多任务操作系统的基础。调度程序可以看作为在所有处于运行状态的进程之间分配 CPU 运行时间的管理代码。由前面描述可知，Linux 进程是抢占式的，但被抢占的进程仍然处于 TASK_RUNNING 状态，只是暂时没有被 CPU 运行。进程的抢占发生在进程处于用户态执行阶段，在内核态执行时是不能被抢占的。

为了能让进程有效地使用系统资源，又能使进程有较快的响应时间，就需要对进程的切换调度采用一定的调度策略。在 Linux 0.12 中采用了基于优先级排队的调度策略。

调度程序

schedule()函数首先扫描任务数组。通过比较每个就绪态（TASK_RUNNING）任务的运行时间递减滴答计数 counter 的值来确定当前哪个进程运行的时间最少。哪一个的值大，就表示运行时间还不长，于是就选中该进程，并使用任务切换宏函数切换到该进程运行。

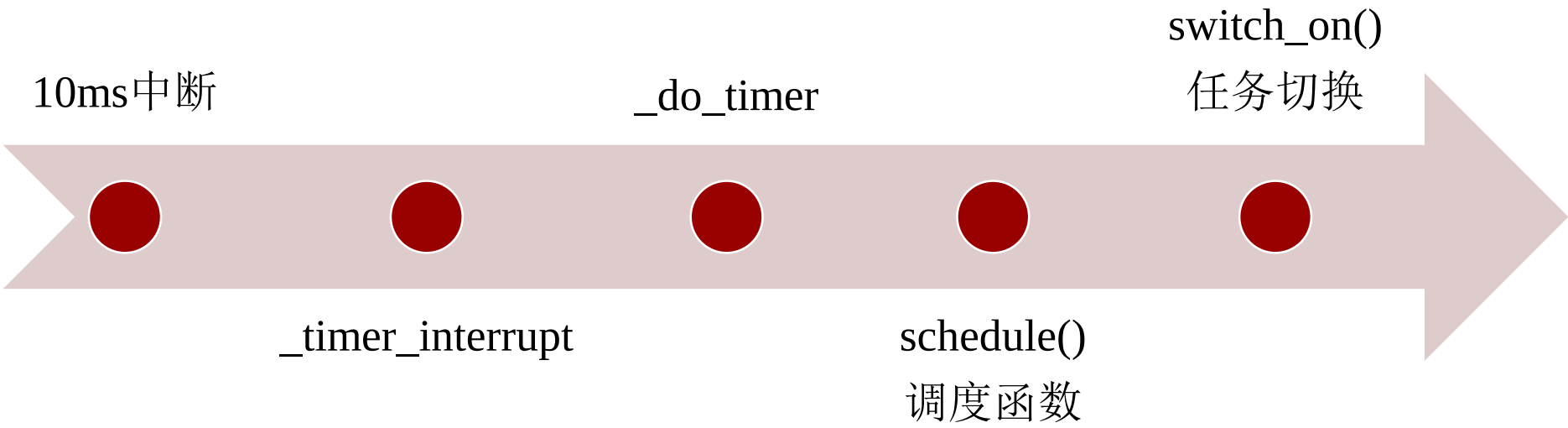
如果此时所有处于 TASK_RUNNING 状态进程的时间片都已经用完，系统就会根据每个进程的优先权值 priority，对系统中所有进程（包括正在睡眠的进程）重新计算每个任务需要运行的时间片值 counter。计算的公式是：

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

这样对于正在睡眠的进程当它们被唤醒时就具有较高的时间片 counter 值。然后 schedule()函数重新扫描任务数组中所有处于 TASK_RUNNING 状态的进程，并重复上述过程，直到选择出一个进程为止。最后调用 switch_to()执行实际的进程切换操作。

如果此时没有其他进程可运行，系统就会选择进程 0 运行。对于 Linux 0.12 来说，进程 0 会调用 pause()把自己置为可中断的睡眠状态并再次调用 schedule()。不过在调度进程运行时，schedule()并不在意进程 0 处于什么状态。只要系统空闲就调度进程 0 运行。

进程调度流程


switch_on()
任务切换
10ms中断
_do_timer
_timer_interrupt
schedule()
调度函数

设置时钟中断 10ms

kernel\sched.c

```
385: void sched_init(void)
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 $0xffffbfff,(%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 »
```

```
46: #define LATCH (1193180/HZ)
```

开启定时器，10ms

中断处理函数_timer_interrupt

kernel\system_call.s

```
176: timer_interrupt:
177:     push %ds        # save ds,es and put kernel data space
178:     push %es        # into them. %fs is used by _system_call
179:     push %fs
180:     pushl %edx      # we save %eax,%ecx,%edx as gcc does not
181:     pushl %ecx      # save those across function calls. %ebx
182:     pushl %ebx      # is saved as we use that in ret_sys_call
183:     pushl %eax
184:     movl $0x10,%eax
185:     mov %ax,%ds
186:     mov %ax,%es
187:     movl $0x17,%eax
188:     mov %ax,%fs
189:     incl jiffies
190:     movb $0x20,%al      # EOI to interrupt controller #1
191:     outb %al,$0x20
192:     movl CS(%esp),%eax
193:     andl $3,%eax        # %eax is CPL (0 or 3, 0=supervisor)
194:     pushl %eax
195:     call do_timer       # 'do_timer(long CPL)' does everything from
196:     addl $4,%esp        # task switching to accounting ...
197:     jmp ret_from_sys_call
```

do_timer

kernel\sched.c

```
305: void do_timer(long cpl)
306: {
307:     extern int beepcount;
308:     extern void sysbeepstop(void);
309:
310:     if (beepcount)
311:         if (!--beepcount)
312:             sysbeepstop();
313:
314:     if (cpl)
315:         current->utime++;
316:     else
317:         current->stime++;
318:
319:     if (next_timer) {
320:         next_timer->jiffies--;
321:         while (next_timer && next_timer->jiffies <= 0) {
322:             void (*fn)(void);
323:
324:             fn = next_timer->fn;
325:             next_timer->fn = NULL;
326:             next_timer = next_timer->next;
327:             (fn)();
328:         }
329:     }
330:     if (current_DOR & 0xf0)
331:         do_floppy_timer();
332:     if ((--current->counter)>0) return;
333:     current->counter=0;
334:     if (!cpl) return;
335:     schedule();
336: } « end do_timer »
```

根据当前特权级，将相应的运行时间递增

如果当前进程时间片不为0，则退出继续执行当前进程

如果当前特权级表示发生中断时正在内核态运行，则返回(内核任务不可被抢占)

执行调度函数

schedule()

kernel\sched.c

```
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:
122: /* this is the scheduler proper: */
123:
124:     while (1) {
125:         c = -1;
126:         next = 0;
127:         i = NR_TASKS;
128:         p = &task[NR_TASKS];
129:         while (--i) {
130:             if (!*--p)
131:                 continue;
132:             if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
133:                 c = (*p)->counter, next = i;
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 »
```

include/linux/sched.h

```
1: #ifndef _SCHED_H
2: #define _SCHED_H
3:
4: #define NR_TASKS 64
5: #define HZ 100
6:
7: #define FIRST_TASK task[0]
8: #define LAST_TASK task[NR_TASKS-1]
```

保存选出的任务的任务号

遍历任务数组，选出就绪态的时间片最大的任务

如果所有任务时间片都是0，则执行下面代码为所有任务重新分配时间片，否则跳出当前while(1)循环

遍历任务数组重新分配时间片，新分配的时间片为counter/2+优先级，所以优先级越高分配到的时间片越大

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

```
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:     0,0,0,0,0,0,0,0, \
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,pgrp,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 » ;
```

include/linux/sched.h

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

include/linux/sched.h

```
 * switch_to(n)将切换当前任务到任务 nr，即 n。首先检测任务 n 不是当前任务，
 * 如果是则什么也不做退出。如果我们切换到的任务最近（上次运行）使用过数学
 * 协处理器的话，则还需复位控制寄存器 cr0 中的 TS 标志。
 */
// 跳转到一个任务的 TSS 段选择符组成的地址处会造成 CPU 进行任务切换操作。
// 输入：%0 - 指向__tmp；                %1 - 指向__tmp.b 处，用于存放新 TSS 的选择符；
//       DX - 新任务 n 的 TSS 段选择符；   ECX - 新任务 n 的任务结构指针 task[n]。
// 其中临时数据结构__tmp 用于组建 228 行远跳转（FAR JUMP）指令的操作数。该操作数由 4 字节
// 偏移地址和 2 字节的段选择符组成。因此__tmp 中 a 的值是 32 位偏移值，而 b 的低 2 字节是新
// TSS 段的选择符（高 2 字节不用）。跳转到 TSS 段选择符会造成任务切换到该 TSS 对应的进程。
// 对于造成任务切换的长跳转，a 值无用。228 行上的间接跳转指令使用 6 字节操作数作为跳转目
// 的地的长指针，其格式为： JMP 16 位段选择符：32 位偏移值。
// 任务切换回来之后，在判断原任务上次执行是否使用过协处理器时，是通过将原任务指针与保存
// 在 last_task_used_math 变量中的上次使用过协处理器任务指针进行比较而作出的，参见文件
// kernel/sched.c 中有关 math_state_restore()函数的说明。
#define switch_to(n) {\
struct {long a,b;} __tmp; \
__asm__("cmpl %%ecx,_current\n\t" \      // 任务 n 是当前任务吗?(current ==task[n]?)
        "je 1f\n\t" \                    // 是，则什么都不做，退出。
        "movw %%dx,%1\n\t" \             // 将新任务 TSS 的 16 位选择符存入__tmp.b 中。
        "xchgl %%ecx,_current\n\t" \     // current = task[n]; ecx = 被切换出的任务。
        "ljmp %0\n\t" \                  // 执行长跳转至*&__tmp，造成任务切换。
                                         // 在任务切换回来后才会继续执行下面的语句。
        "cmpl %%ecx,_last_task_used_math\n\t" \      // 原任务上次使用过协处理器吗？
        "jne 1f\n\t" \                   // 没有则跳转，退出。
        "clts\n" \                       // 原任务上次使用过协处理器，则清 cr0 中的任务
        "1:" \                           // 切换标志 TS。
        ::"m" (*&__tmp.a),"m" (*&__tmp.b), \
        "d" (_TSS(n)),"c" ((long) task[n])); \
}
```

电梯算法

9.1 总体功能

对硬盘和软盘块设备上数据的读写操作是通过中断处理程序进行的。内核每次读写的数据量以一个逻辑块（1024 字节）为单位，而块设备控制器则是以扇区（512 字节）为单位访问块设备。在处理过程中，内核使用了读写请求项等待队列来顺序地缓冲一次读写多个逻辑块的操作。

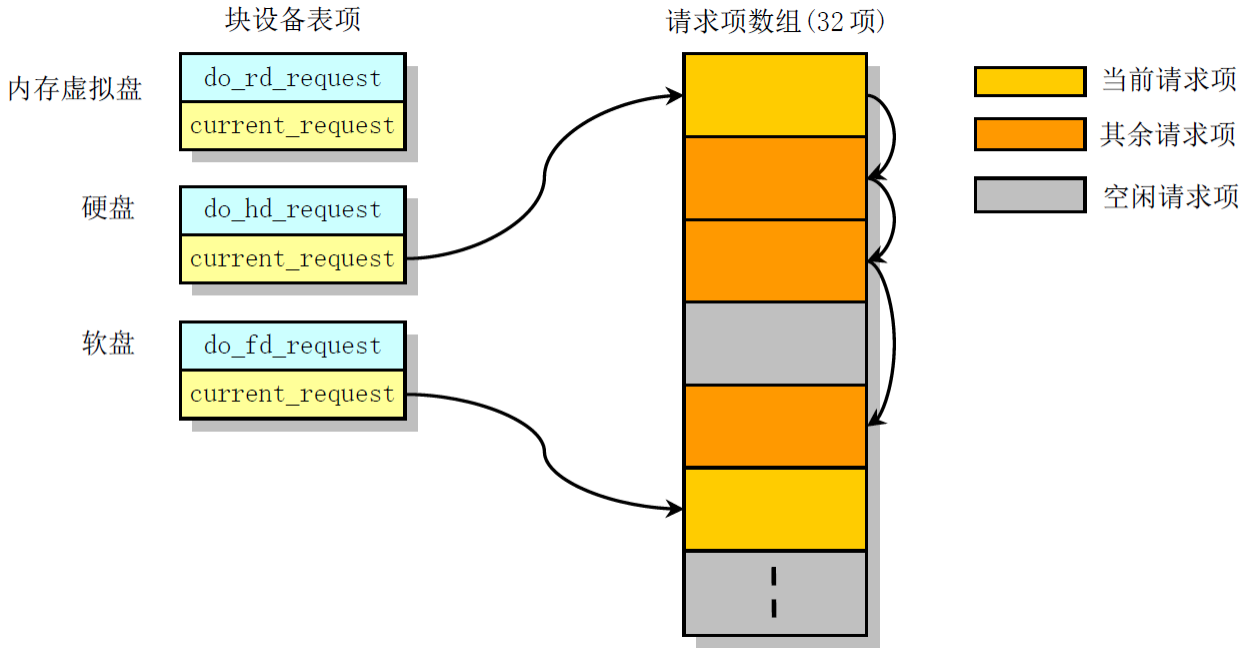
当一个程序需要读取硬盘上的一个逻辑块时，就会向缓冲区管理程序提出申请。而请求读写的程序进程则进入睡眠等待状态。缓冲区管理程序首先在缓冲区中寻找以前是否已经读取过这块数据。如果缓冲区中已经有了，就直接将对应的缓冲区块头指针返回给程序并唤醒等待的进程。若缓冲区中还不存在所要求的数据块，则缓冲管理程序就会调用本章中的低级块读写函数 ll_rw_block()，向相应的块设备驱动程序发出一个读数据块的操作请求。该函数会为此创建一个请求结构项，并插入请求队列中。为了提高读写磁盘的效率，减小磁头移动的距离，内核代码在把请求项插入请求队列时，会使用电梯算法将请求项插入到磁头移动距离最小的请求队列位置处。

若此时对应块设备的请求项队列为空，则表明此刻该块设备不忙。于是内核就会立刻向该块设备的控制器发出读数据命令。当块设备的控制器将数据读入到指定的缓冲块后，就会发出中断请求信号，并调用相应的读命令后处理函数，处理继续读扇区操作或者结束本次请求项的过程。例如对相应块设备进行关闭操作和设置该缓冲块数据已经更新标志，最后唤醒等待该块数据的进程。

9.1.1块设备请求项和请求队列

根据上面描述，我们知道低级读写函数 ll_rw_block()是通过请求项来与各种块设备建立联系并发出读写请求。对于各种块设备，内核使用了一张块设备表 blk_dev[]来进行管理。每种块设备都在块设备表中占有一项。块设备表中每个块设备项的结构为（见后面的 blk.h 文件）：

```
struct blk_dev_struct {
        void (*request_fn)(void);              // 请求项操作的函数指针。
        struct request * current_request;      // 当前请求项指针。
};
extern struct blk_dev_struct blk_dev[NR_BLK_DEV]; // 块设备表（数组）（NR_BLK_DEV = 7）。
```



设备表项与请求项

```
struct request {
        int dev;                        // 使用的设备号（若为-1，表示该项空闲）。
        int cmd;                        // 命令(READ 或 WRITE)。
        int errors;                     // 操作时产生的错误次数。
        unsigned long sector;           // 起始扇区。(1 块=2 扇区)
        unsigned long nr_sectors;       // 读/写扇区数。
        char * buffer;                  // 数据缓冲区。
        struct task_struct * waiting;   // 任务等待操作执行完成的地方。
        struct buffer_head * bh;        // 缓冲区头指针(include/linux/fs.h,68)。
        struct request * next;          // 指向下一请求项。
};
extern struct request request[NR_REQUEST];     // 请求项数组（NR_REQUEST = 32）。
```

sys_read->file_read

fs/read_write.c

```
55: int sys_read(unsigned int fd,char * buf,int count)
56: {
57:     struct file * file;
58:     struct m_inode * inode;
59:
60:     if (fd>=NR_OPEN || count<0 || !(file=current->filp[fd]))
61:         return -EINVAL;
62:     if (!count)
63:         return 0;
64:     verify_area(buf,count);
65:     inode = file->f_inode;
66:     if (inode->i_pipe)
67:         return (file->f_mode&1)?read_pipe(inode,buf,count):-EIO;
68:     if (S_ISCHR(inode->i_mode))
69:         return rw_char(READ,inode->i_zone[0],buf,count,&file->f_pos);
70:     if (S_ISBLK(inode->i_mode))
71:         return block_read(inode->i_zone[0],&file->f_pos,buf,count);
72:     if (S_ISDIR(inode->i_mode) || S_ISREG(inode->i_mode)) {
73:         if (count+file->f_pos > inode->i_size)
74:             count = inode->i_size - file->f_pos;
75:         if (count<=0)
76:             return 0;
77:         return file_read(inode,file,buf,count);
78:     }
79:     printk("(Read)inode->i_mode=%06o\n\r",inode->i_mode);
80:     return -EINVAL;
81: } « end sys_read »
```

include/linux/sys.h

```
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_ftime, sys_sync, sys_kill, sys_rename, sys_mkdir,
    pipe, sys_times, sys_prof, sys_brk, sys_setgid,
    ys_geteuid, sys_getegid, sys_acct, sys_phys,
    fcntl, sys_mpx, sys_setpgid, sys_ulimit,
    _chroot, sys_ustat, sys_dup2, sys_getppid,
    sys_sigaction, sys_sgetmask, sys_ssetmask,
    };
```

file_read->bread->ll_rw_block

fs/file_dev.c

```
17: int file_read(struct m_inode * inode, struct file * filp, char * buf, int count)
18: {
19:     int left,chars,nr;
20:     struct buffer_head * bh;
21:
22:     if ((left=count)<=0)
23:         return 0;
24:     while (left) {
25:         if ((nr = bmap(inode,(filp->f_pos)/BLOCK_SIZE))) {
26:             if (!(bh=bread(inode->i_dev,nr)))
27:                 break;
28:         } else
29:             bh = NULL;
30:         nr = filp->f_pos % BLOCK_SIZE;
31:         chars = MIN( BLOCK_SIZE-nr , left );
32:         filp->f_pos += chars;
33:         left -= chars;
34:         if (bh) {
35:             char * p = nr + bh->b_data;
36:             while (chars-->0)
37:                 put_fs_byte(*(p++),buf++);
38:             brelse(bh);
39:         } else {
40:             while (chars-->0)
41:                 put_fs_byte(0,buf++);
42:         }
43:     } « end while left »
44:     inode->i_atime = CURRENT_TIME;
45:     return (count-left)?(count-left):-ERROR;
46: } « end file_read »
```

fs/buffer.c

```
270: struct buffer_head * bread(int dev,int block)
271: {
272:     struct buffer_head * bh;
273:
274:     if (!(bh=getblk(dev,block)))
275:         panic("bread: getblk returned NULL\n");
276:     if (bh->b_uptodate)
277:         return bh;
278:     ll_rw_block(READ,bh);
279:     wait_on_buffer(bh);
280:     if (bh->b_uptodate)
281:         return bh;
282:     brelse(bh);
283:     return NULL;
284: }
```

make_request -> add_request

```c
static void make_request(int major,int rw, struct buffer_head * bh)
{
    struct request * req;
    int rw_ahead;

/* WRITEA/READA is special case - it is not really needed, so if the */
/* buffer is locked, we just forget about it, else it's a normal read */
    if ((rw_ahead = (rw == READA || rw == WRITEA))) {
        if (bh->b_lock)
            return;
        if (rw == READA)
            rw = READ;
        else
            rw = WRITE;
    }
    if (rw!=READ && rw!=WRITE)
        panic("Bad block dev command, must be R/W/RA/WA");
    lock_buffer(bh);
    if ((rw == WRITE && !bh->b_dirt) || (rw == READ && bh->b_uptodate)) {
        unlock_buffer(bh);
        return;
    }
repeat:
/* we don't allow the write-requests to fill up the queue completely:
 * we want some room for reads: they take precedence. The last third
 * of the requests are only for reads.
 */
    if (rw == READ)
        req = request+NR_REQUEST;
    else
        req = request+((NR_REQUEST*2)/3);
/* find an empty request */
    while (--req >= request)
        if (req->dev<0)
            break;
/* if none found, sleep on new requests: check for rw_ahead */
    if (req < request) {
        if (rw_ahead) {
            unlock_buffer(bh);
            return;
        }
        sleep_on(&wait_for_request);
        goto repeat;
    }
/* fill up the request-info, and add it to the queue */
    req->dev = bh->b_dev;
    req->cmd = rw;
    req->errors=0;
    req->sector = bh->b_blocknr<<1;
    req->nr_sectors = 2;
    req->buffer = bh->b_data;
    req->waiting = NULL;
    req->bh = bh;
    req->next = NULL;
    add_request(major+blk_dev,req);
}
```

```c
void ll_rw_block(int rw, struct buffer_head * bh)
{
    unsigned int major;

    if ((major=MAJOR(bh->b_dev)) >= NR_BLK_DEV ||
    !(blk_dev[major].request_fn)) {
        printk("Trying to read nonexistent block-device\n\r");
        return;
    }
    make_request(major,rw,bh);
}
```

kernel/blk_drv.c

```c
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;
        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;
    sti();
} « end add_request »
```

排序

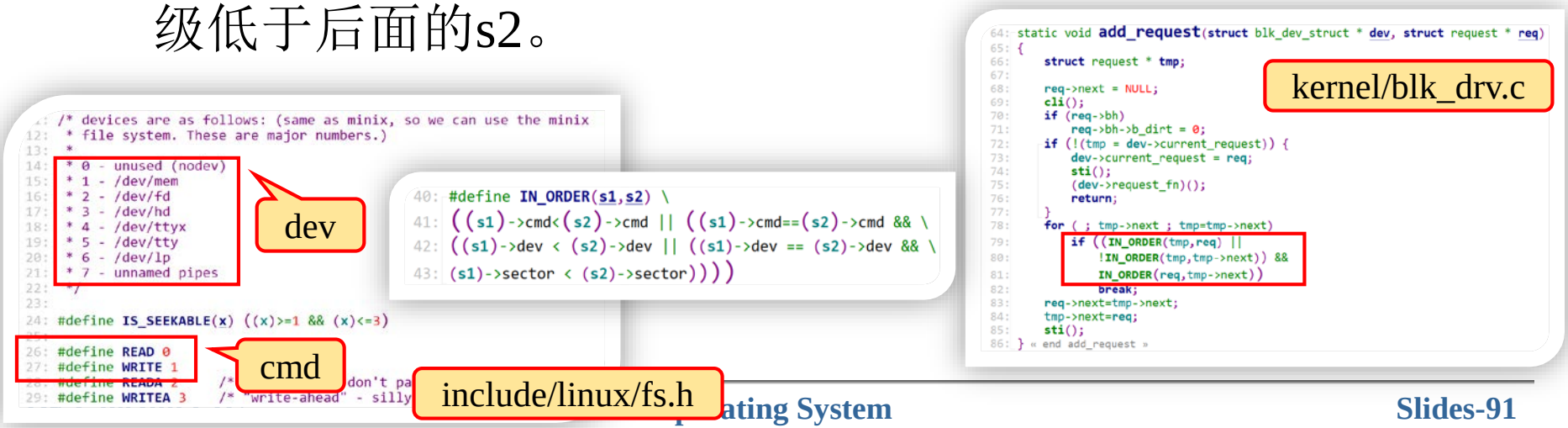
链表插入

```c
struct request {
    int dev;        /* -1 if no r
    int cmd;        /* READ or WR
    int errors;
    unsigned long sector;
    unsigned long nr_sectors;
    char * buffer;
    struct task_struct * waiting;
    struct buffer_head * bh;
    struct request * next;
};
```

```c
#define IN_ORDER(s1,s2) \
((s1)->cmd<(s2)->cmd || ((s1)->cmd==(s2)->cmd && \
((s1)->dev < (s2)->dev || ((s1)->dev == (s2)->dev && \
(s1)->sector < (s2)->sector))))
```

电梯算法

- **IN_ORDER的内部比较思想是：**
  - 先比较操作类型，读操作优先于写操作；
  - 如果操作类型相同，则比较设备号，设备号小的设备优先于设备号大的；
  - 如果设备号也相同，则比较扇区号，先处理扇区号小的扇区，意思就是从磁头从里向外读写了；
  - 如果扇区号也相同，那么返回false，也就是前面的s1优先级低于后面的s2。



execve流程解析

execve功能介绍

- 用于运行用户程序（a.out）或shell脚本的函数
- 是linux编程中常用的一个系统调用类函数
- 在linux命令行下运行用户程序本质其实就是执行execve系统调用

linux/lib/execve.c

```
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; \
}
```

系统调用号

eax

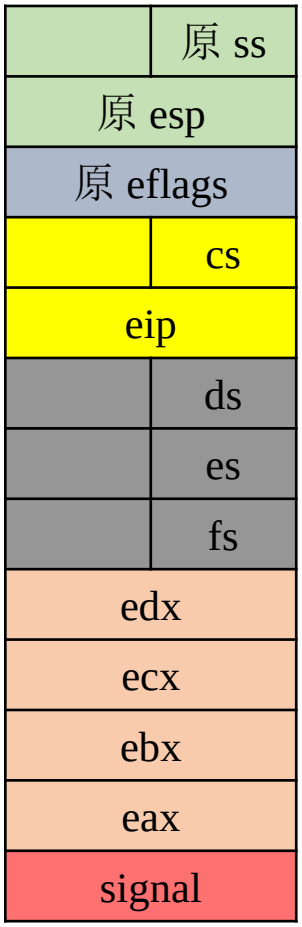
return

ebx

ecx

edx

execve本质



kernel/system_call.s

```
17:  * Stack layout in 'ret_from_system_call':
18:  *
19:  *    0(%esp) - %eax
20:  *    4(%esp) - %ebx
21:  *    8(%esp) - %ecx
22:  *    C(%esp) - %edx
23:  *   10(%esp) - %fs
24:  *   14(%esp) - %es
25:  *   18(%esp) - %ds
26:  *   1C(%esp) - %eip
27:  *   20(%esp) - %cs
28:  *   24(%esp) - %eflags
29:  *   28(%esp) - %oldesp
30:  *   2C(%esp) - %oldss
31:  */
32:
33: SIG_CHLD     = 17
34:
35: EAX      = 0x00
36: EBX      = 0x04
37: ECX      = 0x08
38: EDX      = 0x0C
39: FS       = 0x10
40: ES       = 0x14
41: DS       = 0x18
42: EIP      = 0x1C
43: CS       = 0x20
44: EFLAGS       = 0x24
45: OLDESP       = 0x28
46: OLDSS        = 0x2C
```

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

include/linux/sys.h

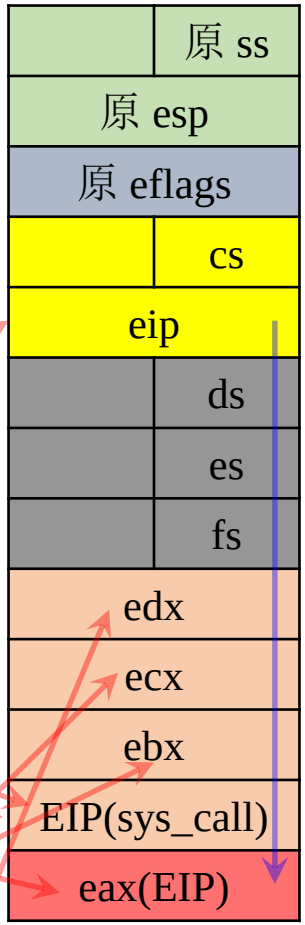
```
74: fn_ptr sys_call_table[] = { sys_setup, sys_exit, sys_fork, sys_read,
75: sys_write, sys_open, sys_close, sys_waitpid, sys_creat, sys_link,
76: sys_unlink, sys_execve, sys_chdir, sys_time, sys_mknod, sys_chmod,
77: sys_chown, sys_break, sys_stat, sys_lseek, sys_getpid, sys_mount,
78: sys_umount, 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_ftime, 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_acct, 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_getpgrp, sys_setsid, sys_sigaction, sys_sgetmask, sys_ssetmask,
86: sys_setreuid,sys_setregid };
```

init/main.c

```
168: void init(void)
169: {
170:     int pid,i;
171:
172:     setup((void *) &drive_info);
173:     (void) open("/dev/tty0",O_RDWR,0);
174:     (void) dup(0);
175:     (void) dup(0);
176:     printf("%d buffers = %d bytes buffer space\
177:         NR_BUFFERS*BLOCK_SIZE);
178:     printf("Free mem: %d bytes\n\r",memory_end-
179:     if (!(pid=fork())) {
180:         close(0);
181:         if (open("/etc/rc",O_RDONLY,0))
182:             exit(1);
183:         execve("/bin/sh",argv_rc,envp_rc);
184:         _exit(2);
185:     }
```

```
_system_call:
        cmpl $nr_system_calls-1,%eax
        ja bad_sys_call
        push %ds
        push %es
        push %fs
        pushl %edx
        pushl %ecx              # push %ebx,%ecx,%edx as parameters
        pushl %ebx              # to the system call
        movl $0x10,%edx         # set up ds,es to kernel space
        mov %dx,%ds
        mov %dx,%es
        movl $0x17,%edx         # fs points to local data space
        mov %dx,%fs
        call _sys_call_table(,%eax,4)
        pushl %eax
        movl _current,%eax
        cmpl $0,state(%eax)     # state
        jne reschedule
        cmpl $0,counter(%eax)   # counter
        je reschedule
ret_from_sys_call:
        movl _current,%eax      # t
        cmpl _task,%eax
        je 3f
        cmpw $0x0f,CS(%esp)     # w
        jne 3f
        cmpw $0x17,OLDSS(%esp)  # w
        jne 3f
        movl signal(%eax),%ebx
        movl blocked(%eax),%ecx
        notl %ecx
        andl %ebx,%ecx
        bsfl %ecx,%ecx
        je 3f
        btrl %ecx,%ebx
        movl %ebx,signal(%eax)
        incl %ecx
        pushl %ecx
        call _do_signal
        popl %eax
3:      popl %eax
        popl %ebx
        popl %ecx
        popl %edx
        pop %fs
        pop %es
        pop %ds
        iret
```

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



地址

```
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; \
}
```

eax

ebx

ecx

edx

Where

do_execve()

fs/exec.c

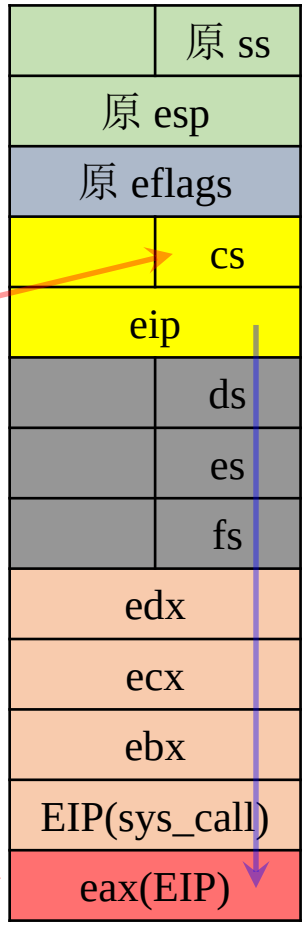
```
179: /*
180:  * 'do_execve()' executes a new program.
181:  */
182: int do_execve(unsigned long * eip,long tmp,char * filename,
183:     char ** argv, char ** envp)
184: {
185:     struct m_inode * inode;
186:     struct buffer_head * bh;
187:     struct exec ex;
188:     unsigned long page[MAX_ARG_PAGES];
189:     int i,argc,envc;
190:     int e_uid, e_gid;
191:     int retval;
192:     int sh_bang = 0;
193:     unsigned long p=PAGE_SIZE*MAX_ARG_PAGES-4;
194:
195:     if ((0xffff & eip[1]) != 0x000f)
196:         panic("execve called from supervisor mode");
197:     for (i=0 ; i<MAX_ARG_PAGES ; i++)    /* clear page-table */
198:         page[i]=0;
199:     if (!(inode=namei(filename)))        /* ge
200:         return -ENOENT;
201:     argc = count(argv);
202:     envc = count(envp);
203:
204: restart_interp:
205:     if (!S_ISREG(inode->i_mode)) {  /* must be regular file */
206:         retval = -EACCES;
207:         goto ↓exec_error2;
208:     }
```

eip[1]指向cs

将32个存放物理页地址的数组清零

获取filename可执行文件的inode

计算参数个数



shell

```
204: restart_interp:
205:     if (!S_ISREG(inode->i_mode)) {  /* must be regular file */
206:         retval = -EACCES;
207:         goto ↓exec_error2;
208:     }
209:     i = inode->i_mode;
210:     e_uid = (i & S_ISUID) ? inode->i_uid : current->euid;
211:     e_gid = (i & S_ISGID) ? inode->i_gid : current->egid;
212:     if (current->euid == inode->i_uid)
213:         i >>= 6;
214:     else if (current->egid == inode->i_gid)
215:         i >>= 3;
216:     if (!(i & 1) &&
217:         !((inode->i_mode & 0111) && suser())) {
218:         retval = -ENOEXEC;
219:         goto ↓exec_error2;
220:     }
221:     if (!(bh = bread(inode->i_dev,inode->i_zone[0]))) {
222:         retval = -EACCES;
223:         goto ↓exec_error2;
224:     }
225:     ex = *((struct exec *) bh->b_data); /* read exec-header */
226:     if ((bh->b_data[0] == '#') && (bh->b_data[1] == '!') && (!sh_bang)) {
227:         /*
228:          * This section does the #! interpretation.
229:          * Sorta complicated, but hopefully it will work.  -TYT
230:          */
231:
232:         char buf[1023], *cp, *interp, *i_name, *i_arg;
233:         unsigned long old_fs;
234:
235:         strncpy(buf, bh->b_data+2, 1022);
236:         brelse(bh);
237:         iput(inode);
238:         buf[1022] = '\0';
239:         if ((cp = strchr(buf, '\n'))) {
240:             *cp = '\0';
241:             for (cp = buf; (*cp == ' ') || (*cp == '\t'); cp++);
242:         }
243:         if (!cp || *cp == '\0') {
244:             retval = -ENOEXEC; /* No interpreter name found */
```

```
6:  struct exec {
7:    unsigned long a_magic;    /* Use macros N_MAGIC, etc for access */
8:    unsigned a_text;      /* length of text, in bytes */
9:    unsigned a_data;      /* length of data, in bytes */
10:   unsigned a_bss;       /* length of uninitialized data area for file, in bytes */
11:   unsigned a_syms;      /* length of symbol table data in file, in bytes */
12:   unsigned a_entry;     /* start address */
13:   unsigned a_trsize;        /* length of relocation info for text, in bytes */
14:   unsigned a_drsize;        /* length of relocation info for data, in bytes */
15: };
16:
17: #ifndef N_MAGIC
18: #define N_MAGIC(exec) ((exec).a_magic)
19: #endif
```

可执行程序

```
298:     brelse(bh);
299:     if (N_MAGIC(ex) != ZMAGIC || ex.a_trsize || ex.a_drsize ||
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）

```
318: /* OK, This is the point of no return */
319:     if (current->executable)
320:         iput(current->executable);
321:     current->executable = inode;
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);
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;
332:     current->used_math = 0;
333:     p += change_ldt(ex.a_text,page)-MAX_ARG_PAGES*PAGE_SIZE;
334:     p = (unsigned long) create_tables((char *)p,argc,envc);
335:     current->brk = ex.a_bss +
336:         (current->end_data = ex.a_data +
337:         (current->end_code = ex.a_text));
338:     current->start_stack = p & 0xfffff000;
339:     current->euid = e_uid;
340:     current->egid = e_gid;
341:     i = ex.a_text+ex.a_data;
342:     while (i&0xfff)
343:         put_fs_byte(0,(char *) (i++));
344:     eip[0] = ex.a_entry;        /* eip, magic happens :-) */
345:     eip[3] = p;             /* stack pointer */
346:     return 0;
347: exec_error2:
348:     iput(inode);
349: exec_error1:
350:     for (i=0 ; i<MAX_ARG_PAGES ; i++)
351:         free_page(page[i]);
352:     return(retval);
353: } « end do_execve »
```

释放当前的可执行文件inode节点

Line 199： inode=namei(filename)

清空当前进程的页表映射

改变ldt

制作参数表（类似指针数组）

写入代码结束位置、数据结束位置、bss结束位置

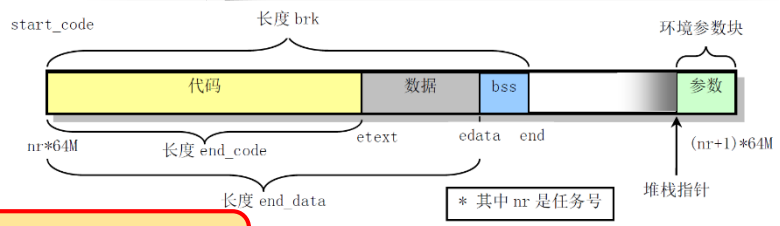
4KB对齐

where

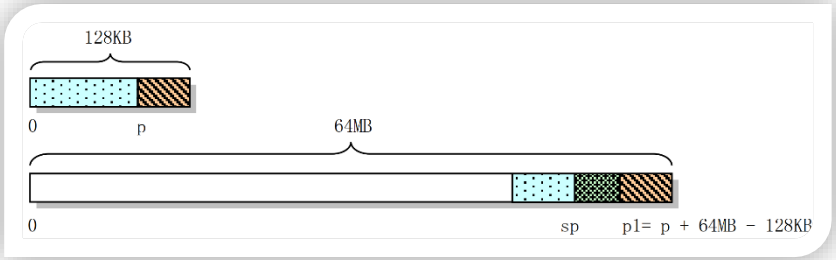
将系统调用压栈里的eip，改为应用程序的入口地址。同时将栈顶esp设为p

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

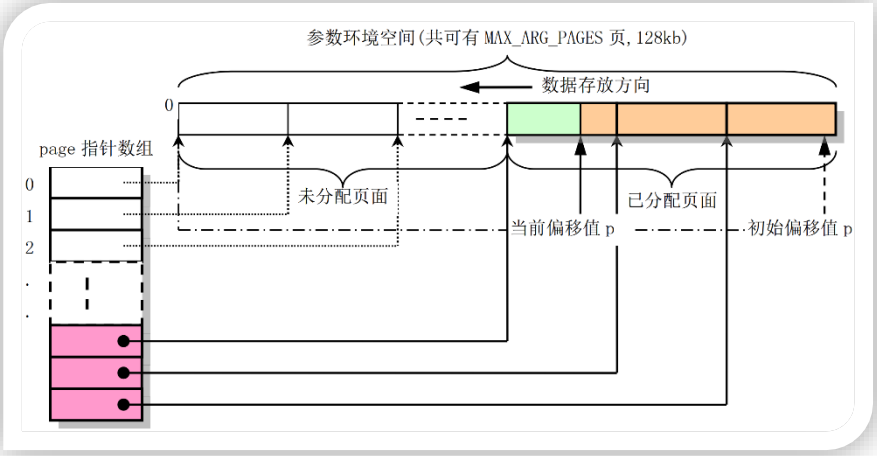
```
6:  struct exec {
7:    unsigned long a_magic;    /* Use macros N_MAGIC, etc for access */
8:    unsigned a_text;      /* length of text, in bytes */
9:    unsigned a_data;      /* length of data, in bytes */
10:   unsigned a_bss;       /* length of uninitialized data area for file, in bytes */
11:   unsigned a_syms;      /* length of symbol table data in file, in bytes */
12:   unsigned a_entry;     /* start address */
13:   unsigned a_trsize;        /* length of relocation info for text, in bytes */
14:   unsigned a_drsize;        /* length of relocation info for data, in bytes */
15: };
16:
17: #ifndef N_MAGIC
18: #define N_MAGIC(exec) ((exec).a_magic)
19: #endif
```



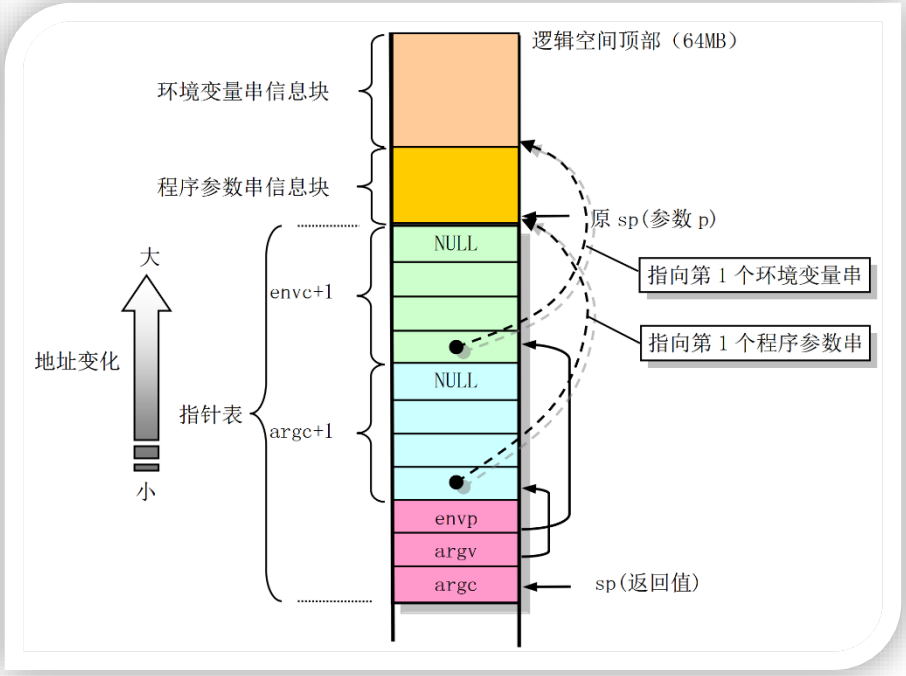
```c
/*
 * create_tables() parses the env- and arg-strings in new user
 * memory and creates the pointer tables from them, and puts their
 * addresses on the "stack", returning the new stack pointer value.
 */
static unsigned long * create_tables(char * p,int argc,int envc)
{
    unsigned long *argv,*envp;
    unsigned long * sp;

    sp = (unsigned long *) (0xfffffffc & (unsigned long) p);
    sp -= envc+1;
    envp = sp;
    sp -= argc+1;
    argv = sp;
    put_fs_long((unsigned long)envp,--sp);
    put_fs_long((unsigned long)argv,--sp);
    put_fs_long((unsigned long)argc,--sp);
    while (argc-->0) {
        put_fs_long((unsigned long) p,argv++);
        while (get_fs_byte(p++)) /* nothing */ ;
    }
    put_fs_long(0,argv);
    while (envc-->0) {
        put_fs_long((unsigned long) p,envp++);
        while (get_fs_byte(p++)) /* nothing */ ;
    }
    put_fs_long(0,envp);
    return sp;
} « end create_tables »
```



p 转换成进程初始堆栈指针的方法



参数和环境变量字符串空间



新程序堆栈中指针表示意图

change_ldt

```
154: static unsigned long change_ldt(unsigned long text_size,unsigned long * page)
155: {
156:     unsigned long code_limit,data_limit,code_base,data_base;
157:     int i;
158:
159:     code_limit = text_size+PAGE_SIZE -1;
160:     code_limit &= 0xFFFFF000;
161:     data_limit = 0x4000000;
162:     code_base = get_base(current->ldt[1]);
163:     data_base = code_base;
164:     set_base(current->ldt[1],code_base);
165:     set_limit(current->ldt[1],code_limit);
166:     set_base(current->ldt[2],data_base);
167:     set_limit(current->ldt[2],data_limit);
168: /* make sure fs points to the NEW data segment */
169:     __asm__("pushl $0x17\n\tpop %%fs"::);
170:     data_base += data_limit;
171:     for (i=MAX_ARG_PAGES-1 ; i>=0 ; i--) {
172:         data_base -= PAGE_SIZE;
173:         if (page[i])
174:             put_page(page[i],data_base);
175:     }
176:     return data_limit;
177: } « end change_ldt »
```

计算代码段所占页面，并页对齐

64M

设置代码段起始和限长

fs 0x17 指向用户数据段

指向末端

这里是填充参数页，
将存有参数的物理页
填入页表形成映射