

Operating System

Dr. GuoJun LIU

Harbin Institute of Technology

<http://os.guojunhit.cn>

Linux 内核体系结构

Slides-4

Chapter A2

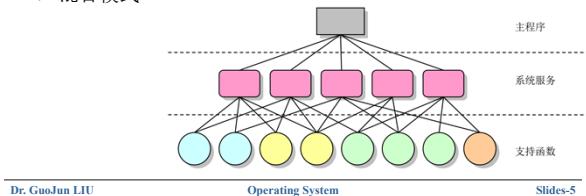
Linux 0.11 Overview

Slides-2

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



Dr. GuoJun LIU Operating System Slides-5

Outline

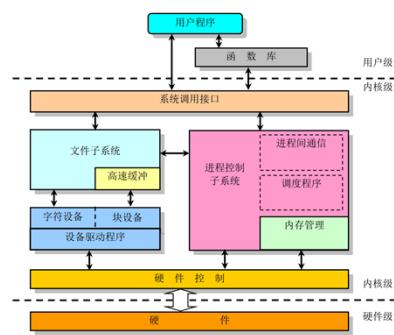
- Linux 内核体系结构
- 引导启动程序
- 初始化程序
- Linux 内核对存的管理和使用
- Linux 进程控制
- 设置描述符
- fork 系统调用
- 信号处理
- 内存管理
- 调度算法
- 电梯算法
- execve 流程解析

Dr. GuoJun LIU

Operating System

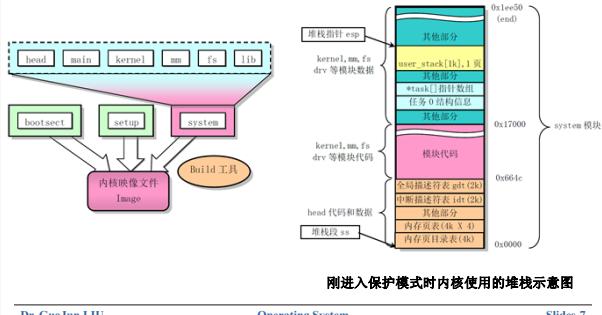
Slides-3

内核结构框图



Dr. GuoJun LIU Operating System Slides-6

内核编译链接/组合结构

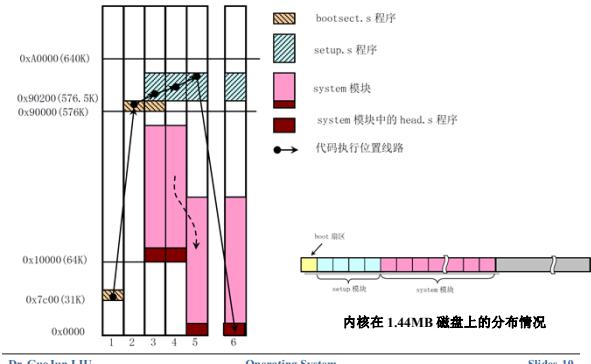


Dr. GuoJun LIU

Operating System

Slides-7

引导时内核在存中的位置和移动情况



Dr. GuoJun LIU

Operating System

Slides-10

引导启动程序

Slides-8

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)

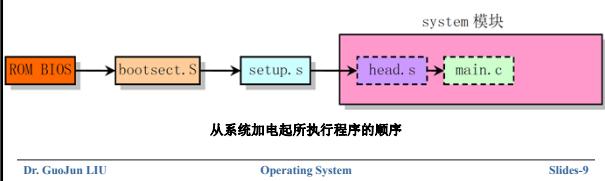
Dr. GuoJun LIU

Operating System

Slides-11

引导启动程序

Linux/boot/目录

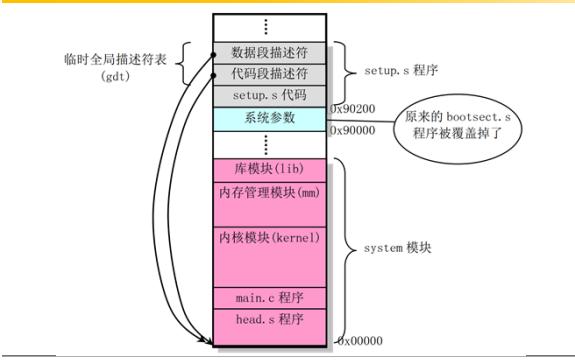


Dr. GuoJun LIU

Operating System

Slides-9

setup.s 程序结束后内存中程序示意图

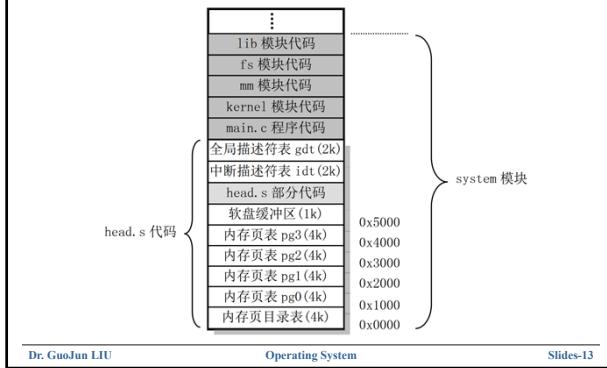


Dr. GuoJun LIU

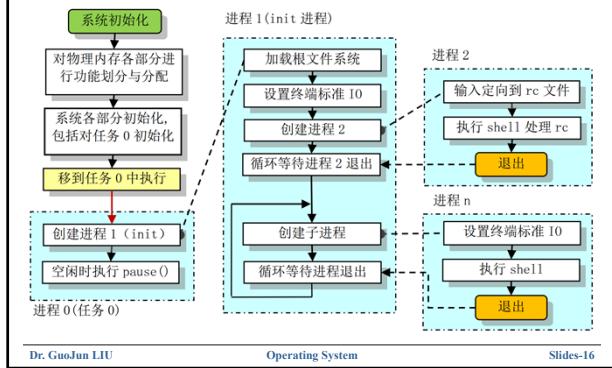
Operating System

Slides-12

system 模块在内存中的映像示意图



内核初始化程序流程示意图



初始化程序

Slides-14

move_to_user_mode()

```
1221 main_memory_start > buffer_memory_end;
1222 #ifdef HAVEIOS
1223 main_memory_start == rd_init(main_memory_start, RNDISK*1024);
1224 #endif
1225 mem_init(main_memory_start, memory_end);
1226 mem_map_init();
1227 blk_dev_init();
1228 ide_init();
1229 tty_init();
1230 timer_init();
1231 sched_init();
1232 buffer_init(buffer_memory_end);
1233 harddisk_init();
1234 floppy_init();
1235 ...
1236 move_to_user_mode();
1237 if (fork()) {
1238     /* we count on this going ok */
1239     init();
1240 }
1241 ...
1242 ..
```

init'main.c

```
000068bf (6)... push dword ptr ds:0x0001dac8
000068c5 (5)... call .+24533 (0x0000c89f)
000068c8 (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 0x000006e3
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 (9)... mov eax, 0x00000002
include\asm\system.h
```

Dr. GuoJun LIU Operating System Slides-17

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

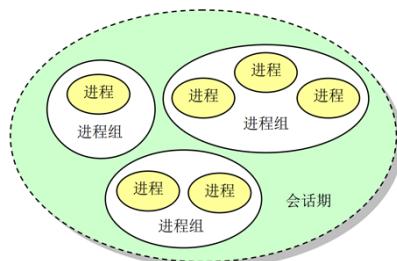
这些内存变量指明了主内存区的开始地址、系统所拥有的内存容量和作为高速缓冲区内存的末端地址

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

Dr. GuoJun LIU Operating System Slides-15

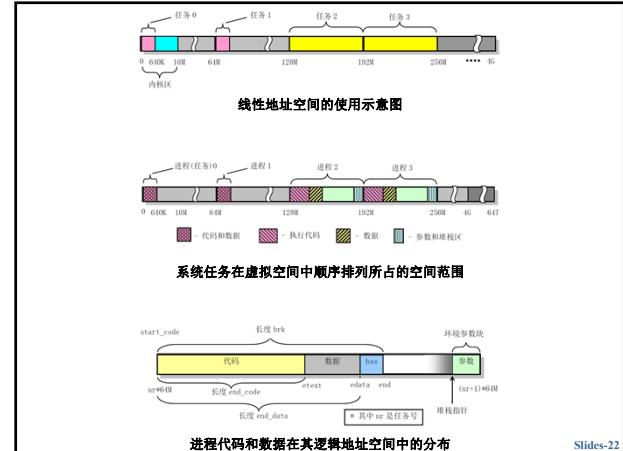
进程、进程组和会话期之间的关系



Dr. GuoJun LIU Operating System Slides-18

Linux 内核对存的管理和使用

Slides-19

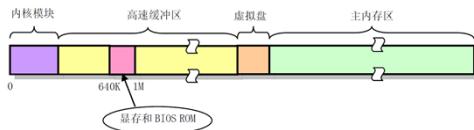


物理内存使用的功能分布

Dr. GuoJun LIU

Operating System

Slides-20

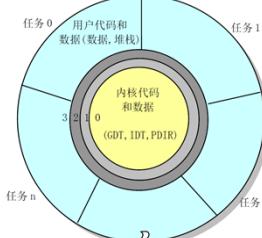


CPU 多任务和保护方式

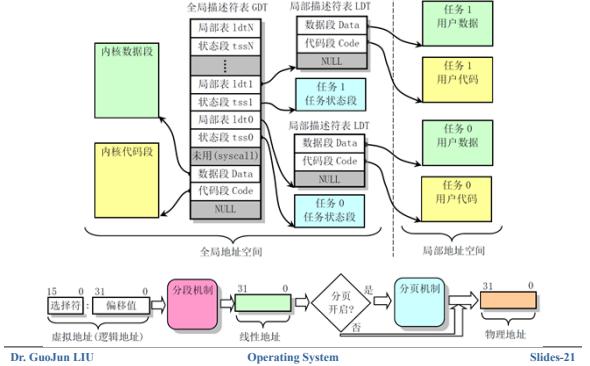
Dr. GuoJun LIU

Operating System

Slides-23



系统中虚拟地址空间分配图



Dr. GuoJun LIU

Operating System

Slides-21

虚拟地址、线性和物理之间的关系

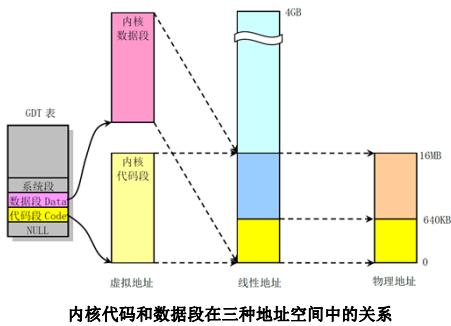
- 内核代码和数据的地址
- 任务 0 在三个地址空间中的相互关系
- 任务 1 在三个地址空间中的相互关系
- 其他任务地址空间中的对应关系

Dr. GuoJun LIU

Operating System

Slides-24

内核代码和数据段在三种地址空间中的关系

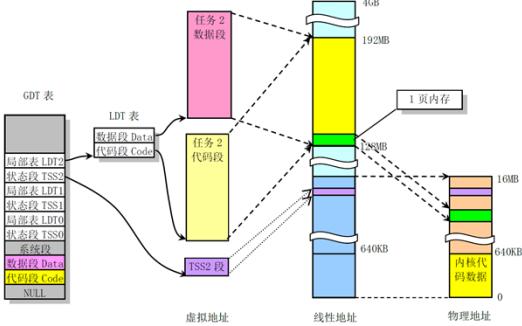


Dr. GuoJun LIU

Operating System

Slides-25

其他任务地址空间中的对应关系

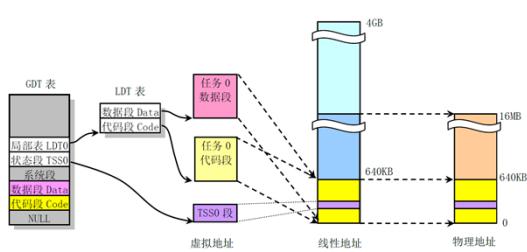


Dr. GuoJun LIU

Operating System

Slides-28

任务 0 在三个地址空间中的相互关系



Dr. GuoJun LIU

Operating System

Slides-26

Linux 进程控制

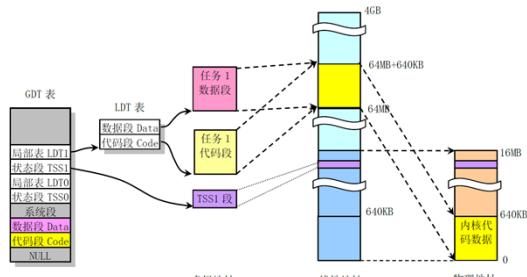
- Task 数据结构 – PCB
- 进程运行状态及转换
- 进程初始化
 - > move_to_user_mode
- 创建新进程
 - > fork()、写时复制 Copy on Write
- 进程调度
 - > 优先级、时间片、进程切换 switch_to()
- 终止进程 exit()

Dr. GuoJun LIU

Operating System

Slides-29

任务 1 在三个地址空间中的相互关系

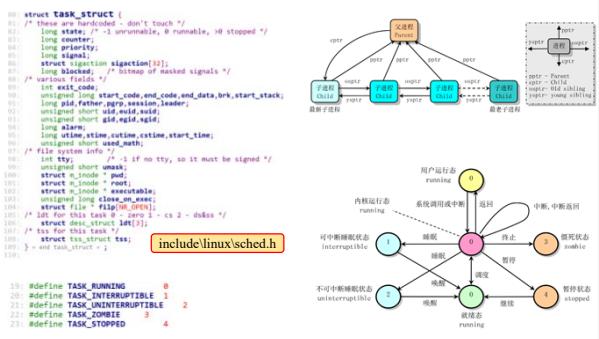


Dr. GuoJun LIU

Operating System

Slides-27

Linux 进程控制



Dr. GuoJun LIU

Operating System

Slides-30

设置描述符

Slides-31

```

kernel/system_calls.h: 72
kernel/system_calls.s: 72
kernel/main.c
kernel/sched.c
include/linux/head.h
include/asm/system.h

```

kernel/system_calls.h: 72

```

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

```

kernel/system_calls.s: 72

```

    ; Ok, I get parallel printer interrupts while using the floppy for some
    ; strange reason. Urgent need to ignore them.
    ; global interrupt, form linear interrupt, recursive
    ; global interrupt, floppy interrupt, parallel interrupt
    ; global device_not_available, coprocessor_error

```

kernel/main.c

```

void sched_init(void)
{
    int i;
    struct dev_struct * p;
    if (total_struct_size != 36) {
        panic("Struct size must be 36 bytes");
    }
    set_kern_desc_gate(0, _TSS_ENTRY, 0, task_list);
    p = (struct dev_struct *)task_list;
    for (i=0; i<total_struct_size/4; i++) {
        p[i] = 0;
        p[i+1] = 0;
        p[i+2] = 0;
        p[i+3] = 0;
    }
}

```

kernel/sched.c

```

/* clear ... so that we won't have troubles with that later on */
clear_dev_struct(&dev_struct, &dev_struct_size, 0);
for (i=0; i<total_struct_size/4; i++) {
    dev_struct[i] = 0;
    dev_struct[i+1] = 0;
    dev_struct[i+2] = 0;
    dev_struct[i+3] = 0;
}

```

include/linux/head.h

```

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

```

include/asm/system.h

```

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

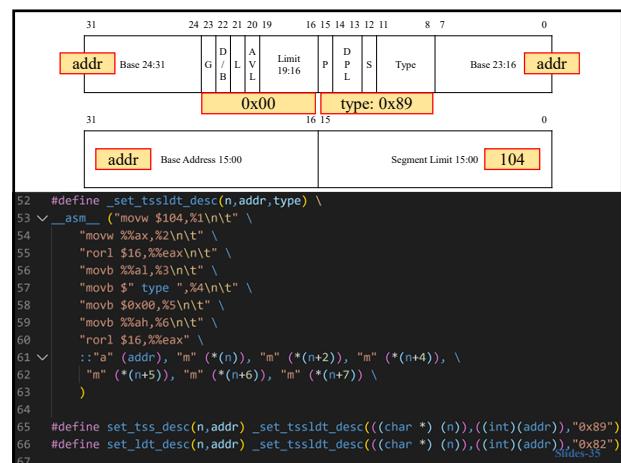
```

```

81     setup_idt:
82         lea ignore_int,%edx
83         movl $0x00000000,%eax
84         movw %dx,%ax /* select segment */
85         movw $0xE00,%dx /* interrupt gate - op1:0, present */
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

```

Slides-32



```

31      24 23 22 21 20 19   16 15 14 13 12 11   8 7   5 4   0
addr Offset 31:16          P D S Type D 1 1 0 0 0 0 0 0
31          16 15          0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                                         edx
                                         eax
                                         dx: addr 15:00
                                         dx: addr15:00

22 #define _set_gate(gate_addr,type,dpl,addr) \
23 __asm__ ("movw %dx,%ax\n\t" \
24     "movl %eax,%dx\n\t" \
25     "movl %edx,%dx\n\t" \
26     : \
27     : "i" ((short) (0x8000+(dpl<<13)+(type<<8))), \
28     "o" (*((char *) (gate_addr))), \
29     "o" (*4+(char *) (gate_addr)), \
30     "d" ((char *) (addr)), "a" (0x00080000))
31 \
32 \
33 #define set_intr_gate(n,addr) \
34     _set_gate(&idt[n],14,0,addr)

```

Slides-33

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		

S=0

System-Segment and Gate-Descriptor Types

Slides-35

fork 系统调用

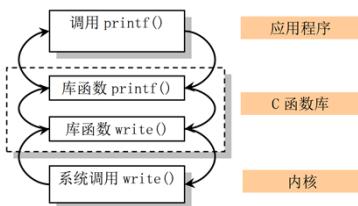
Slides-39

表 8-1 Intel 保留的中断号含义

向量号	名称	类型	信号	说明
0	Device error	故障	SIGPPE	当进行除以零的操作时产生。
1	Debug	陷阱	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	nm	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	寻址到的地址越界时引起。
6	Invalid Opcode	故障	SIGILL	CPU 执行发现一个无效的指令操作码。
7	Device not available	故障	SIGSEGV	设备不存在, 协处理器器。在两种情况下会产生该中断: (a)CPU 遇到一个协处理器并从 EM 置位时。在这种情况下处理程序模拟导致异常的指令。 (b)MP 和 TS 都在置位状态时, CPU 遇到 WAIT 或一个转移指令。在这种情况下, 处理程序在必要时应该更新协处理器器的状态。
8	Double fault	异常中止	SIGSEGV	双故障出错。
9	Coprocessor segment overrun	异常中止	SIGPPE	协处理器段超出。
10	Invalid TSS	故障	SIGSEGV	CPU 切换时发现 TSS 无效。
11	Segment not present	故障	SIGSEGV	描述符所指的段不存在。
12	Stack segment	故障	SIGSEGV	堆栈段不存在或地址越出堆栈段。
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	SIGPPE	协处理器发出的出错信号引起。
17	Alignment check	故障		在应用内存边界检查时, 若用户级数据非边界对齐会产生该异常。
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。

Slides-42

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

Dr. GuoJun LIU

Operating System

Slides-40

Trap.c

// 以下定义了一些函数原型。

```

39 void page_exception(void);
40 void divide_error(void);
41 void debug_error(void);
42 void nm(void);
43 void ram(void);
44 void int3(void);
45 void overflow(void);
46 void bounds(void);
47 void invalid_op(void);
48 void device_not_available(void);
49 void double_fault(void);
50 void coprocessor_segment_overrun(void);
51 void invalid_TSS(void);
52 void segment_not_present(void);
53 void stack_segment(void);
54 void general_protection(void);
55 void page_fault(void);
56 void coprocessor_error(void);
57 void reserved(void);
58 void parallel_interrupt(void);
59 void irq3(void);
60 void alignment_check(void);

```

// 页异常, 实际是 page_fault (mm/page.s, 14)。

```

// int0 (kernel/asms.s, 20) .
// int1 (kernel/asms.s, 51) .
// int2 (kernel/asms.s, 58) .
// int3 (kernel/asms.s, 62) .
// int4 (kernel/asms.s, 66) .
// int5 (kernel/asms.s, 70) .
// int6 (kernel/asms.s, 74) .
// int7 (kernel/sys_call.s, 158) .
// int8 (kernel/sys_call.s, 98) .
// int9 (kernel/sys_call.s, 78) .
// int10 (kernel/asms.s, 132) .
// int11 (kernel/asms.s, 136) .
// int12 (kernel/asms.s, 140) .
// int13 (kernel/asms.s, 144) .
// int14 (mm/page.s, 14) .
// int16 (kernel/sys_call.s, 140) .
// int15 (kernel/asms.s, 82) .
// int39 (kernel/sys_call.s, 295) .
// int45 (kernel/asms.s, 86) 协处理器中断处理。
// int46 (kernel/asms.s, 148) .

```

Dr. GuoJun LIU

Operating System

Slides-43

内核代码

■ 该项目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



内核目录中各文件中函数的调用层次关系

Dr. GuoJun LIU

Operating System

Slides-41

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

Dr. GuoJun LIU

Operating System

Slides-44

copy_process()

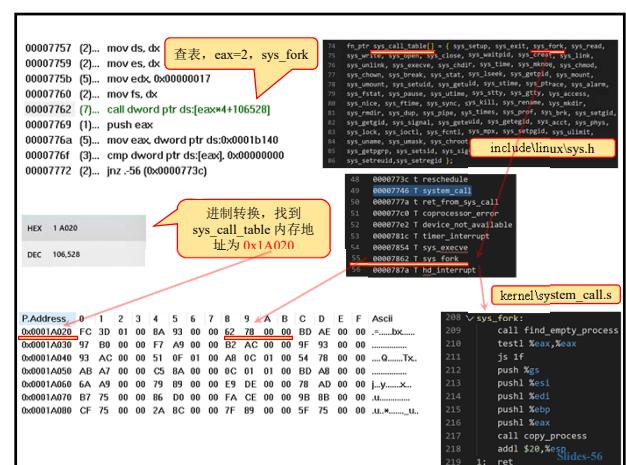
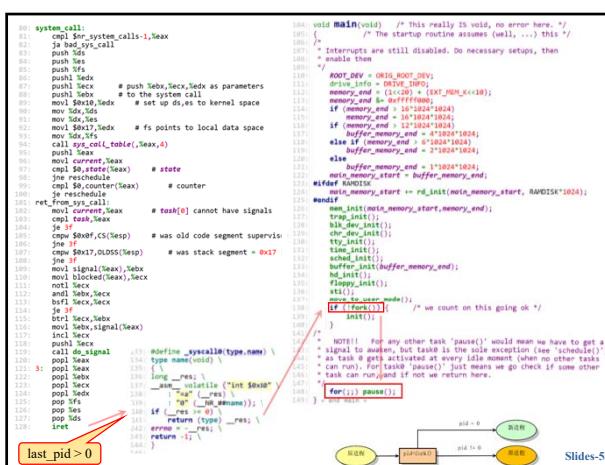
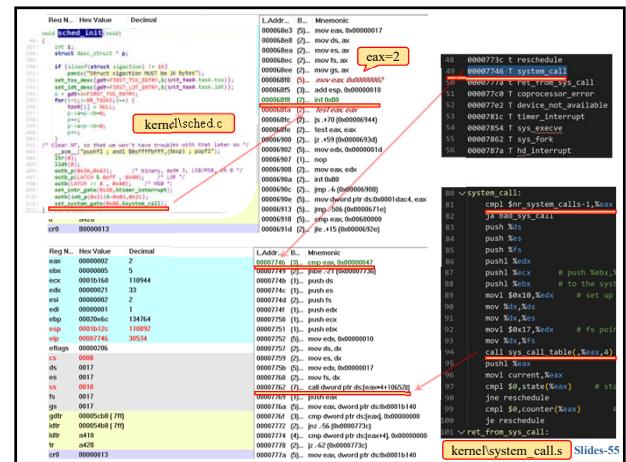
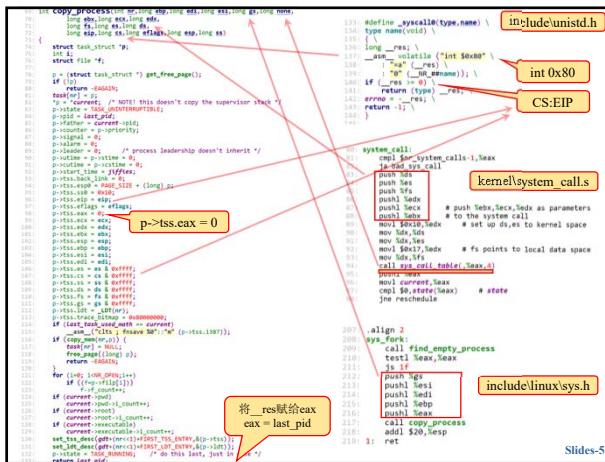
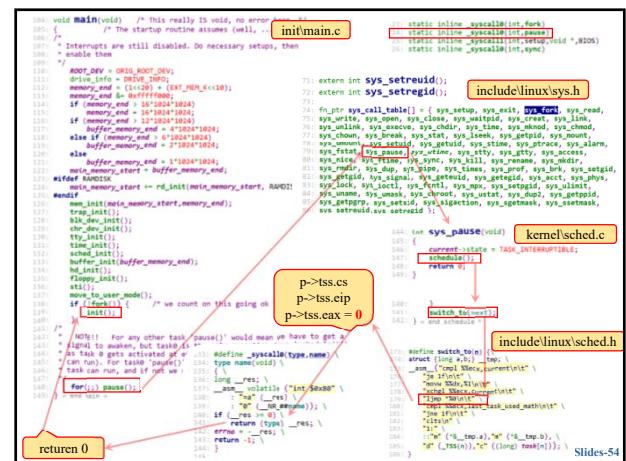
■ 进程0将在copy_process()函数中做非常重要的、体现父子进程创建机制的工作：

- 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

Dr.GuoJun LIU

Operating System

Slides-51



信号处理

Slides-57

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); // 信号处理例程。  
    sigset(SIG_BLOCK, 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）。  
};
```

Dr. GuoJun LIU

Operating System

Slides-60

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

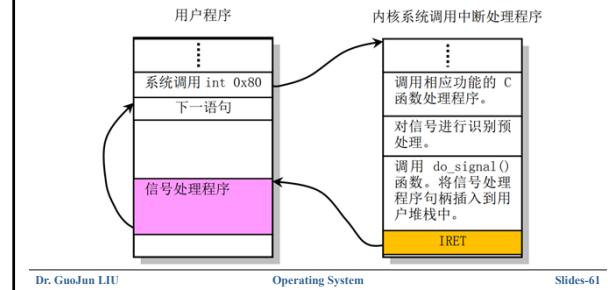
Dr. GuoJun LIU

Operating System

Slides-58

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



Dr. GuoJun LIU

Operating System

Slides-61

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（堆栈错误）

■ 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>  
#define SIGKILL 1  
#define SIGKILL 2  
#define SIGPOLL 3  
#define SIGPOLL 4  
#define SIGPOLL 5  
#define SIGPOLL 6  
#define SIGPOLL 7  
#define SIGPOLL 8  
#define SIGPOLL 9  
#define SIGPOLL 10  
#define SIGPOLL 11  
#define SIGPOLL 12  
#define SIGPOLL 13  
#define SIGPOLL 14  
#define SIGPOLL 15  
#define SIGPOLL 16  
#define SIGPOLL 17  
#define SIGPOLL 18  
#define SIGPOLL 19  
#define SIGPOLL 20  
#define SIGPOLL 21  
#define SIGPOLL 22
```

Dr. GuoJun LIU

Operating System

Slides-59

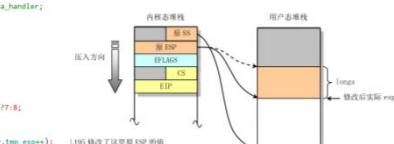
do_signal()函数对用户堆栈的修改

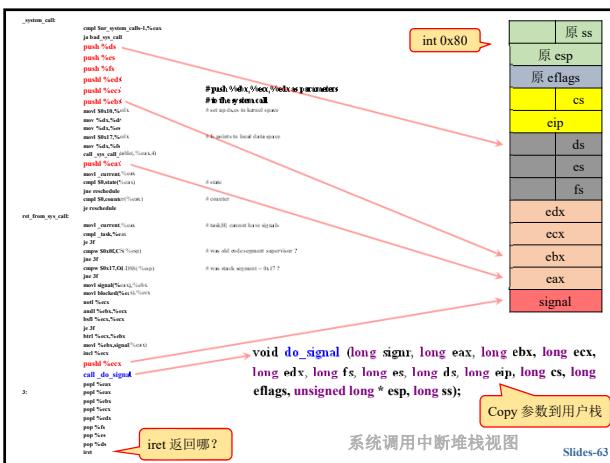
```
82 void do_signal(long signr, long esp, long ebx, long ecx, long edx,  
83                 long fs, long ds,  
84                 long esp, long cs, long eflags,  
85                 unsigned long ip, long ss);  
86 {  
87     if (signr <= 0 || signr > 31)  
88     {  
89         /* Invalid signal number */  
90         long old_ip, esp;  
91         struct sigaction * sa = current->sigaction + signr - 1;  
92         int i;  
93         unsigned long * tmp_esp;  
94         /*  
95          * sa_handler = (unsigned long) sa->sa_handler;  
96          * if (sa->sa_handler == 0)  
97          * {  
98          *     if ((sa->sa_handler & SA_ONESHOT))  
99          *         return;  
100         *     else  
101         *         __do_nexit((long)(signr-1));  
102         * }  
103         * if ((sa->sa_flags & SA_ONESHOT))  
104         *     sa->sa_handler = NULL;  
105         * if ((ip & 0x0000000f) == 0) // 00000000  
106         *     ip = ip + 1;  
107         * if ((ip & 0x0000000f) == 0) // 00000000  
108         *     ip = ip + 1;  
109         * put_fx_long((long)sa->sa_restorer,tmp_esp++);  
110         * put_fx_long((long)sa->sa_handler,tmp_esp++);  
111         * if (((ip->sa)->sa_fnmask) & SA_NOMASK)  
112             ip = ip + 1;  
113         * put_fx_long(ip->sa->sa_handler,tmp_esp++);  
114         * put_fx_long(ip->sa->sa_fnmask,tmp_esp++);  
115         * put_fx_long(ip->sa->sa_handler,tmp_esp++);  
116         * put_fx_long(ip->sa->sa_fnmask,tmp_esp++);  
117         * if (ip->sa->sa_fnmask & SA_NOMASK)  
118             ip = ip + 1;  
119     }  
120 }
```

Dr. GuoJun LIU

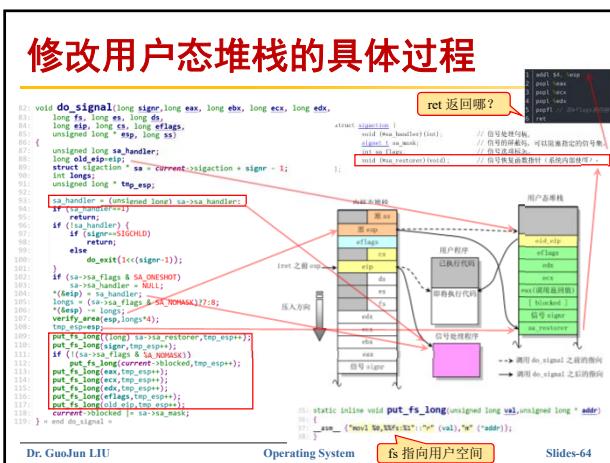
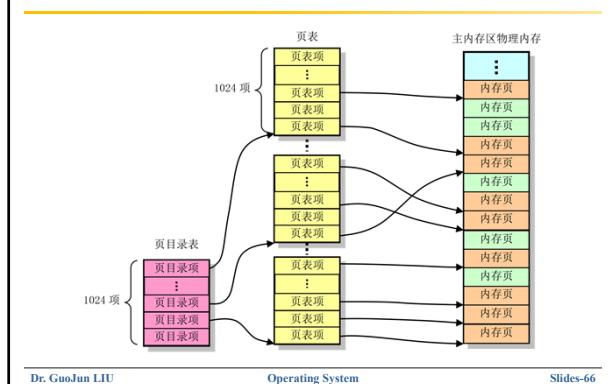
Operating System

Slides-62

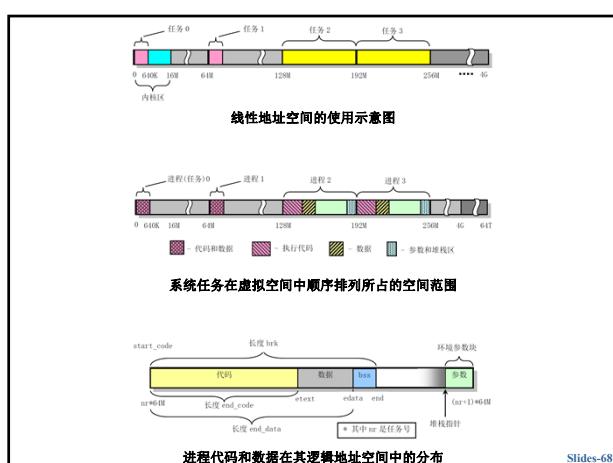
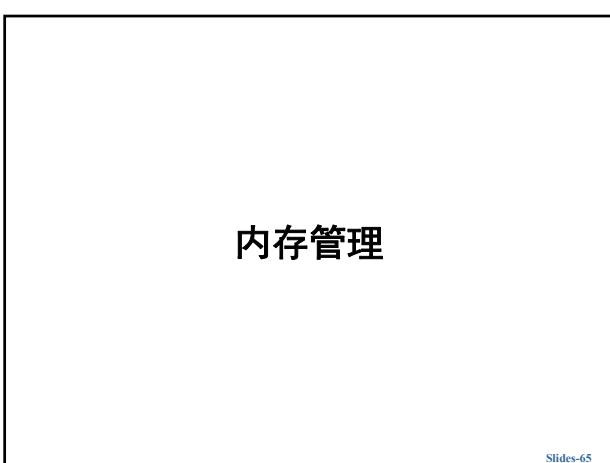
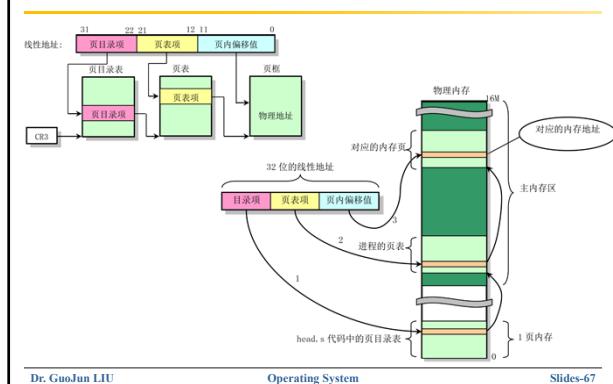




页目录表和页表结构示意图



线性地址对应的物理地址

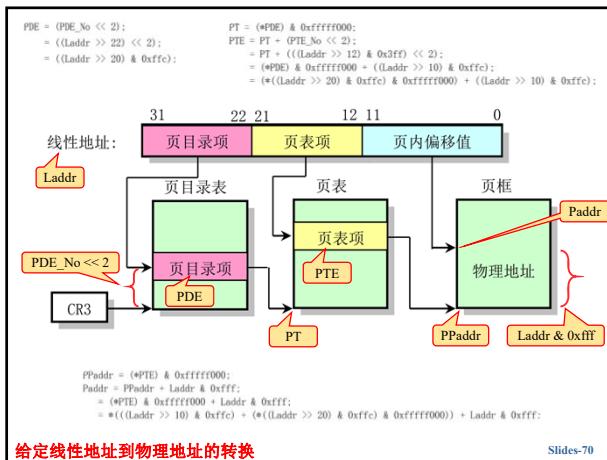
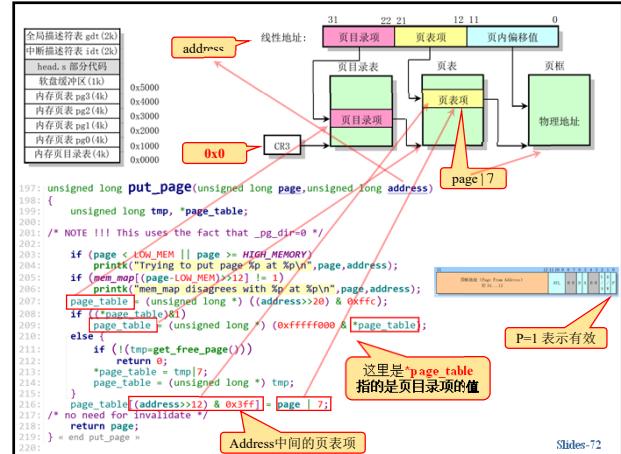


线性地址和逻辑地址的分解

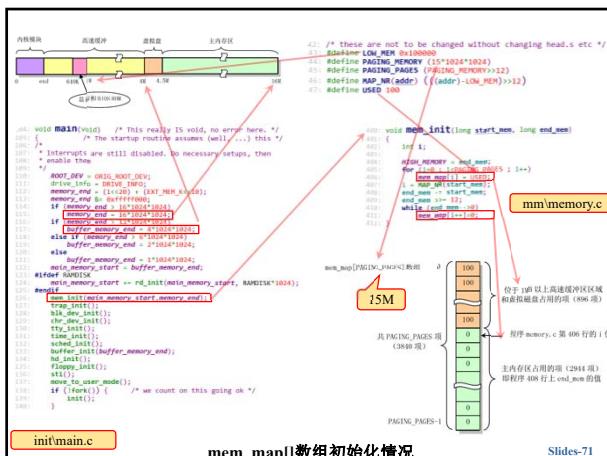
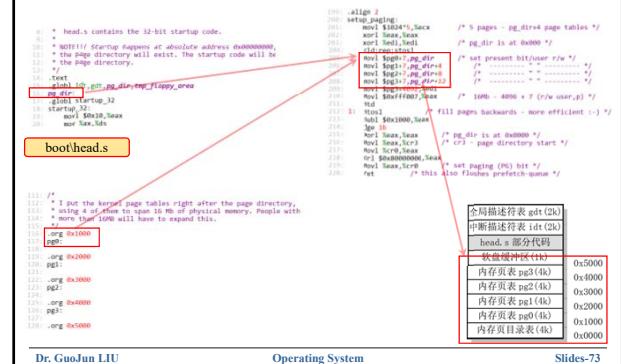
- 根据 CPU 的分页管理机制，一个 32 位的线性地址 addr 可以分解为

- ▶ 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 & 0xffff);



页目录初始化



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

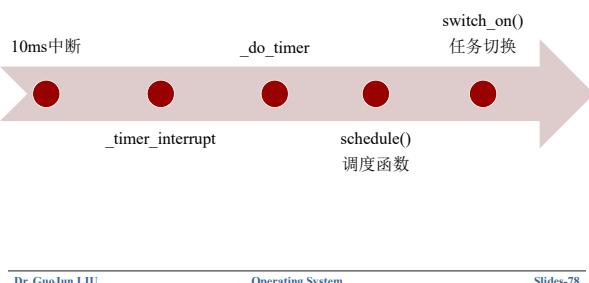
```

180 void trap_init(void) kernel\trap.c
181 {
182     int i;
183
184     set_trap_gate(0,handle_error);
185     set_trap_gate(1,debug);
186     set_trap_gate(2,unavailable);
187     set_trap_gate(3,unavailable); /* 103-3 can be c
188     set_trap_gate(4,unavailable);
189     set_trap_gate(5,unavailable);
190     set_trap_gate(6,unavailable);
191     set_trap_gate(7,unavailable);
192     set_trap_gate(8,unavailable);
193     set_trap_gate(9,unavailable);
194     set_trap_gate(10,unavailable);
195     set_trap_gate(11,unavailable);
196     set_trap_gate(12,unavailable);
197     set_trap_gate(13,unavailable);
198     set_trap_gate(14,unavailable);
199     set_trap_gate(15,unavailable);
200 }
201
202 static inline volatile void OOM(void)
203 {
204     prints("out of memory\n");
205     __asm__ ("movl $0,%eax");
206     __asm__ ("movl %eax,%rcr3");
207     __asm__ ("a"());
208 }
209
210 void do_no_page(unsigned long error_code,unsigned long address)
211 {
212     int err[4];
213     unsigned long tmp;
214     unsigned long page;
215     struct block_desc *block;
216     unsigned long dev;
217     unsigned long offset;
218     unsigned long start_code;
219     if (error_code >= current->current_error) {
220         return;
221     }
222     if (share_page(error_code)) {
223         return;
224     }
225     if ((err[0] = get_free_page()))
226     {
227         /* reservation that we need for header */
228         block = (struct block_desc *)tmp;
229         for (i=0; i<4; i++) {
230             if (!is_executable(block[i].data))
231                 break_page((page_t)current->current_error,
232                             block[i].data, current->current_error);
233             tmp = page + 4096;
234             while (i>0) {
235                 if ((err[i] = get_free_page()))
236                     break_page((page_t)tmp,
237                                block[i].data, err[i]);
238                 if (err[i])
239                     oom();
240             }
241         }
242     }
243 }
244
245 void do_no_page()
246 {
247     pushl %es;
248     pushl %ds;
249     pushl %fs;
250     pushl %gs;
251     pushl %ss;
252     pushl %rdx;
253     pushl %rcx;
254     pushl %rbx;
255     pushl %rbp;
256     pushl %rsi;
257     pushl %rdi;
258     pushl %rsp;
259     pushl %r15;
260     pushl %r14;
261     pushl %r13;
262     pushl %r12;
263     pushl %r11;
264     pushl %r10;
265     pushl %r9;
266     pushl %r8;
267     pushl %r7;
268     pushl %r6;
269     pushl %r5;
270     pushl %r4;
271     pushl %r3;
272     pushl %r2;
273     pushl %r1;
274     pushl %r0;
275     pushl %r15;
276     pushl %r14;
277     pushl %r13;
278     pushl %r12;
279     pushl %r11;
280     pushl %r10;
281     pushl %r9;
282     pushl %r8;
283     pushl %r7;
284     pushl %r6;
285     pushl %r5;
286     pushl %r4;
287     pushl %r3;
288     pushl %r2;
289     pushl %r1;
290     pushl %r0;
291     pushl %r15;
292     pushl %r14;
293     pushl %r13;
294     pushl %r12;
295     pushl %r11;
296     pushl %r10;
297     pushl %r9;
298     pushl %r8;
299     pushl %r7;
300     pushl %r6;
301     pushl %r5;
302     pushl %r4;
303     pushl %r3;
304     pushl %r2;
305     pushl %r1;
306     pushl %r0;
307     pushl %r15;
308     pushl %r14;
309     pushl %r13;
310     pushl %r12;
311     pushl %r11;
312     pushl %r10;
313     pushl %r9;
314     pushl %r8;
315     pushl %r7;
316     pushl %r6;
317     pushl %r5;
318     pushl %r4;
319     pushl %r3;
320     pushl %r2;
321     pushl %r1;
322     pushl %r0;
323     iret
324 }
325
326 mm\memory.c
327
328 static inline volatile void OOM(void)
329 {
330     prints("out of memory\n");
331     __asm__ ("movl $0,%eax");
332     __asm__ ("movl %eax,%rcr3");
333     __asm__ ("a"());
334 }
335
336 void do_no_page(unsigned long error_code,unsigned long address)
337 {
338     int err[4];
339     unsigned long tmp;
340     unsigned long page;
341     struct block_desc *block;
342     unsigned long dev;
343     unsigned long offset;
344     unsigned long start_code;
345     if (error_code >= current->current_error) {
346         return;
347     }
348     if (share_page(error_code)) {
349         return;
350     }
351     if ((err[0] = get_free_page()))
352     {
353         /* reservation that we need for header */
354         block = (struct block_desc *)tmp;
355         for (i=0; i<4; i++) {
356             if (!is_executable(block[i].data))
357                 break_page((page_t)current->current_error,
358                             block[i].data, current->current_error);
359             tmp = page + 4096;
360             while (i>0) {
361                 if ((err[i] = get_free_page()))
362                     break_page((page_t)tmp,
363                                block[i].data, err[i]);
364                 if (err[i])
365                     oom();
366             }
367         }
368     }
369 }
370
371 mm\page.s
372
373

```

Slides-75

进程调度流程



调度算法

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

基于优先级排队的调度策略

Slides-76

设置时钟中断 10ms

```

385: void sched_init(void) kernel\sched.c
386: {
387:     int i;
388:     struct desc_struct *p;
389:
390:     if (sizeof(struct sigaction) != 16)
391:         panic("Struct sigaction must be 16 bytes");
392:     set_tss_desc(gdt-FIRST_TSS_ENTRY,&(init_task.task.tss));
393:     set_ldt_desc(gdt-FIRST_LDT_ENTRY,&(init_task.task.ldt));
394:     p = gdt+2*FIRST_TSS_ENTRY;
395:     for (i=1;i<NR_TASKS;i++) {
396:         p->desc[1] = NULL;
397:         p+=2;
398:         p+=2;
399:         p+=2;
400:         p+=2;
401:     }
402:     /* Clear NT, so that we won't have troubles with that later on */
403:     __asm__ ("pushfl ; andl $0xfffffbff,(%esp) ; popfl");
404:     ltr(0);
405:     lldt(0);
406:     outb_p(LATCH & 0x0f, 0x40); /* Binary mode */
407:     outb_p(LATCH & 0x0f, 0x40); /* LSB */
408:     set_intr_gate(0x20,_timer_interrupt); /* MSB */
409:     outb_p(0x00, 0x40); /* Binary mode */
410:     set_system_gate(0xB0,system_call);
411:     set_system_gate(0xB0,system_call);
412: } end sched_init

```

开启定时器, 10ms

Dr. GuoJun LIU Operating System Slides-79

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。调度程序可以看作是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)任务的运行时间边减边答任务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状态的进程时间片都已经用完，系统就会根据每个进程的优先权值priority，对系统中所有进程(包括正在睡眠的进程)重新计算每个任务需要运行的时间片值counter。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。

Slides-77

中断处理函数 _timer_interrupt

```

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:
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:
184:     pushl %eax
185:     movl $0x10,%eax
186:     movl %eax,%ds
187:     movl %eax,%es
188:     movl $0x17,%eax
189:     movl %eax,%fs
190:     incl $jiffies # EOI to interrupt controller #
191:     movb $0x20,%al # LSB
192:     outb %al,$0x20
193:     movl CS(%esp),%eax
194:     andl $3,%eax # %eax is CPL (0 or 3, 0=supervisor)
195:     pushl %eax
196:     call do_timer # 'do_timer(long CPL)' does everything from
197:     addl $4,%esp # task switching to accounting ...
198:     jmp ret_from_sys_call

```

Dr. GuoJun LIU Operating System Slides-80

do_timer

```

805: void do_timer(long cpi) kernel/sched.c
806: {
807:     extern int beepcount;
808:     extern void sysbeepstop(void);
809:
810:     if (beepcount)
811:         if (!--beepcount)
812:             sysbeepstop();
813:
814:     if (cpl)
815:         current->time++; // 根据当前特权级, 将相应的运行时间递增
816:
817:     if (next_timer) {
818:         next_timer->jiffies--;
819:         while (next_timer && next_timer->jiffies <= 0) { // 如果当前进程时间片不为0, 则退出继续执行当前进程
820:             void (*fn)(void);
821:
822:             fn = next_timer->fn;
823:             next_timer = next_timer->next;
824:             fn(); // 执行调度函数
825:         }
826:         next_timer = next_timer->next;
827:         fn(); // 执行调度函数
828:     }
829:
830:     if (current->ore & ore)
831:         do_floppy_timer();
832:     if (!--current->counter) return; // 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)
833:     schedule();
834: }

```

Dr. GuoJun LIU

Operating System

Slides-81

```

* 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_b; %1 - 指向 __tmp_b 处, 用于存放新 TSS 的选择符;
//          IX - 新任务 n 的 TSS 段选择符; ECX - 新任务 n 的任务结构指针 task[n]。
// 其中地址和 2 字节的段选择符组成。因此, __tmp_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 段选择符会进任务切换到该 TSS 对应的进程。
// 对于造成任务长链接, a 值用 0, 228 行上的间接跳转指令使用 6 字节操作数作为跳转目
// 的绝对长指针, 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__ __volatile__ ("movl %cr0,%a" // 任务 n 是当前任务吗?(current == task[n]?) \
                     "jo 1f[n]" // 是, 则什么都不做, 退出。 \
                     "movw %rdx,%b[n]" // 将新任务 IX 的 16 位选择符存入 __tmp.b 中。 \
                     "xchgl %%ecx,%a[n]" // current = task[n]; ecx = 被切换出的任务。 \
                     "1:jmp %d[n]" // 执行长跳转至 __tmp, 完成任务切换。 \
                     "2:" // 在任务切换后会继续执行下面的语句。 \
                     "cmpl %%ecx, _last_task_used_math\n1:" // 原任务上次使用过协处理器吗? \
                     "jne 1f[n]" // 没有则跳转, 退出。 \
                     "crlts[n]" // 原任务上次使用过协处理器, 则清 cr0 中的任务 \
                     "2:" // 切换标志 TS。 \
                     :: "%r" (__tmp.a), "%r" (__tmp.b), \
                         "%d" (TSS(n)), "%c" ((long)task[n])); \
}

```

Slides-84

schedule()

```

584: void SCHEDULE(void) kernel/sched.c
585: {
586:     struct task_struct ** p;
587:     int i,next;
588:     struct task_struct * ps;
589:
590:     /* check alarm, wake up any interruptible tasks that have got a signal */
591:     for(p=BLAST_TASK; p >= FIRST_TASK; p--) {
592:         if (*p) {
593:             if ((*p)->alarm && (*p)->jiffies < jiffies) {
594:                 (*p)->alarm = 0;
595:                 if ((*p)->signal >= SIG_BLOCK && (*p)->blocked) {
596:                     (*p)->state+=TASK_INTERRUPTIBLE;
597:                     (*p)->state-=TASK_BLOCKED;
598:                 }
599:             }
600:         }
601:     }
602:
603:     /* this is the scheduler proper: */
604:     while(1) {
605:         i = next = 0;
606:         p = BLAST_TASK;
607:         ps = task[0];
608:         for(p=BLAST_TASK; p >= FIRST_TASK; p--) {
609:             if (*p) {
610:                 if ((*p)->state == TASK_RUNNING && (*p)->counter > c) {
611:                     if ((*p)->signal == SIG_BLOCK && (*p)->blocked) {
612:                         (*p)->state+=TASK_INTERRUPTIBLE;
613:                         (*p)->state-=TASK_BLOCKED;
614:                     }
615:                     if ((*p)->state >= FIRST_TASK && (*p)->state <= LAST_TASK) {
616:                         if (*p) {
617:                             if ((*p)->counter >= (counter+priority)/2) {
618:                                 counter = counter + priority;
619:                                 if ((*p)->state == TASK_INTERRUPTIBLE) {
620:                                     if ((*p)->counter >= ((counter+priority)/2+1)) {
621:                                         counter = counter + priority;
622:                                         if ((*p)->state == TASK_INTERRUPTIBLE)
623:                                             switch_to(next);
624:                                         else
625:                                             comment("switched to %p",(*p));
626:                                     }
627:                                 }
628:                             }
629:                         }
630:                     }
631:                 }
632:             }
633:         }
634:         if ((*p)->state == FIRST_TASK && (*p)->state <= LAST_TASK) {
635:             if ((*p)->counter >= ((counter+priority)/2+1)) {
636:                 if ((*p)->state == TASK_INTERRUPTIBLE)
637:                     switch_to(next);
638:             }
639:         }
640:     }
641: }

```

Dr. GuoJun LIU

include/linux/sched.h

Slides-82

电梯算法

```

115: #define INIT_TASK \
116:     .state etc<0 { 0,15,15, \ \
117:         /* signals */ 0,{1},0, \ \
118:         /* ec,brk.. */ 0,0,0,0,0,0,0, \ \
119:         /* pid etc.. */ 0,-1,0,0,0,0,0,0,0, \ \
120:         /* flags */ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, \ \
121:         /* alarm */ 0,0,0,0,0,0, \ \
122:         /* math */ 0,0,0,0,0,0, \ \
123:         /* fs info */ 0,-1,0022,0ULL, \ \
124:         /* flags */ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, \ \
125:         /* brk */ 0,0,0,0,0,0, \ \
126:         /* ld */ 0,0,0,0,0,0,0, \ \
127:         /* idt */ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, \ \
128:         /* msr */ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, \ \
129:         /* ), \ \
130:         /* ts */ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, \ \
131:         /* pd */ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, \ \
132:         /* sss */ 0,0,x17,0x17,0x17,0x17, \ \
133:         /* dt */ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, \ \
134:         /* ld */ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, \ \
135:         /* tss */ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, \ \
136:     };

```

`counter = counter + priority;`

Dr. GuoJun LIU

include/linux/sched.h

Slides-83

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队列以实现顺序地缓存, 一次读写多个逻辑块的操作。

当一个程序需要读取硬盘上一个逻辑块时, 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。如果缓冲区已经有了, 就直接将对应的缓冲区块指针返回给程序员并唤醒等待的进程。若缓冲区中还没有所要求的数据块, 则缓冲区管理程序会调用本章中的低级块设备写函数 `ll_rw_block()`, 向相应的块设备驱动程序发出一个读数据的操作请求, 该函数会为其创建一个请求队列, 并插入请求队列中。`为了提高读写速度的效率, 小概率移动的低频点的请求队列被优化, 并优先级在把请求队列插入请求队列时, 会使用电梯算法将请求插入到磁头从低到高再到低的请求队列位置处。`

若此时对应块设备的请求队列为空, 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, 并调用相对应的读命令之后处理函数, 处理继续读块操作或者结束本次请求的过程。例如对相对块设备进行关闭操作和设置该缓存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。每块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)。

```

Slides-86

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/libexecve.c



Dr. GuoJun LIU

Operating System

Slides-94

do_execve()

```

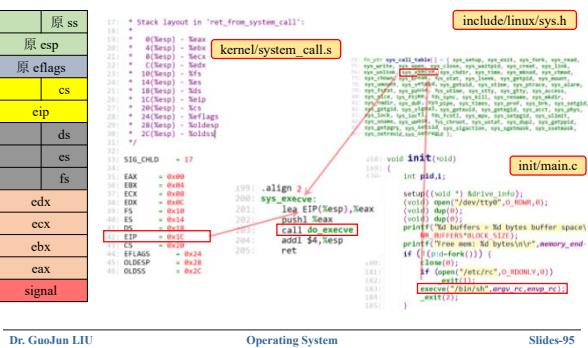
179: /* 'do_execve()' executes a new program. */
180: int do_execve(unsigned long * eip,long tmp,char * filename,
181:             char ** argv,char ** envp,
182:             struct _inode *inode,
183:             struct exec_exi,
184:             struct buffer_head *bb;
185:             struct exec_header;
186:             struct buffer_head *bh;
187:             struct exec_exi;
188:             unsigned long p[MAX_ARG_PAGES];
189:             int i,arg,envc;
190:             int e_uid, e_gid;
191:             int refcnt;
192:             int base, offset = 0;
193:             unsigned long p=PAGE_SIZE*MAX_ARG_PAGES-4;
194:
195:             if ((0xffff & eip[1]) != ax0000)
196:                 panic("execve called from supervisor mode");
197:             for (i=0 ; i<MAX_ARG_PAGES ; i++) /* clear page-table */
198:                 p[i] = 0;
199:             if (!(*filename))
200:                 return -ENODT;
201:             argc = count(argv);
202:             envc = count(envp);
203:
204:             restart_interpreter();
205:             if (IS_ISREG(inode->i_node)) /* * must be regular file */
206:                 goto exec_error;
207:             */
208: 
```

Dr. GuoJun LIU

Operating System

Slides-97

execve本质



Dr. GuoJun LIU

Operating System

Slides-95

shell

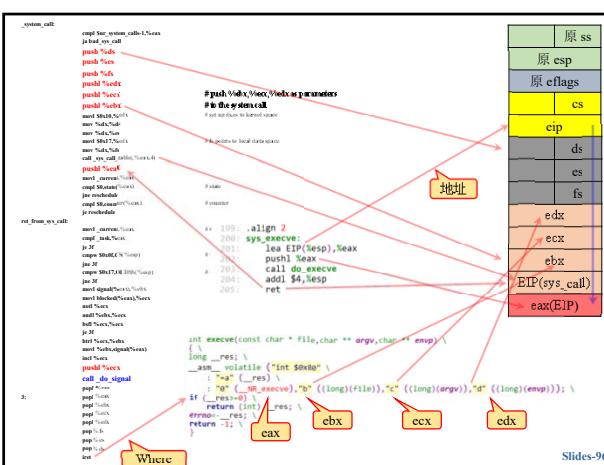
```

204: restart_interpreter();
205:         if (!(_isreg(inode->i_mode)) { /* must be regular file */
206:             retval = -ENOREAD;
207:         }
208:         /* Set up for the interpreter. */
209:         if ((argc+1)*4 <= 1024) {
210:             /* Set up the file area. */
211:             /* Set up for the interpreter. */
212:             /* Set up for the interpreter. */
213:             /* Set up for the interpreter. */
214:             /* Set up for the interpreter. */
215:         }
216:         else {
217:             /* Set up for the interpreter. */
218:             /* Set up for the interpreter. */
219:             /* Set up for the interpreter. */
220:             /* Set up for the interpreter. */
221:             /* Set up for the interpreter. */
222:         }
223:         /* Set up for the interpreter. */
224:         /* Set up for the interpreter. */
225:         /* Set up for the interpreter. */
226:         /* Set up for the interpreter. */
227:         /* Set up for the interpreter. */
228:         /* Set up for the interpreter. */
229:         /* Set up for the interpreter. */
230:         /* Set up for the interpreter. */
231:         /* Set up for the interpreter. */
232:         /* Set up for the interpreter. */
233:         /* Set up for the interpreter. */
234:         /* Set up for the interpreter. */
235:         /* Set up for the interpreter. */
236:         /* Set up for the interpreter. */
237:         /* Set up for the interpreter. */
238:         /* Set up for the interpreter. */
239:         /* Set up for the interpreter. */
240:         /* Set up for the interpreter. */
241:         /* Set up for the interpreter. */
242:         /* Set up for the interpreter. */
243:         /* Set up for the interpreter. */
244:         /* Set up for the interpreter. */
245:         /* Set up for the interpreter. */
246:         /* Set up for the interpreter. */
247:         /* Set up for the interpreter. */
248:         /* Set up for the interpreter. */
249:         /* Set up for the interpreter. */
250:         /* Set up for the interpreter. */
251:         /* Set up for the interpreter. */
252:         /* Set up for the interpreter. */
253:         /* Set up for the interpreter. */
254:         /* Set up for the interpreter. */
255:         /* Set up for the interpreter. */
256:         /* Set up for the interpreter. */
257:         /* Set up for the interpreter. */
258:         /* Set up for the interpreter. */
259:         /* Set up for the interpreter. */
260:         /* Set up for the interpreter. */
261:         /* Set up for the interpreter. */
262:         /* Set up for the interpreter. */
263:         /* Set up for the interpreter. */
264:         /* Set up for the interpreter. */
265:         /* Set up for the interpreter. */
266:         /* Set up for the interpreter. */
267:         /* Set up for the interpreter. */
268:         /* Set up for the interpreter. */
269:         /* Set up for the interpreter. */
270:         /* Set up for the interpreter. */
271:         /* Set up for the interpreter. */
272:         /* Set up for the interpreter. */
273:         /* Set up for the interpreter. */
274:         /* Set up for the interpreter. */
275:         /* Set up for the interpreter. */
276:         /* Set up for the interpreter. */
277:         /* Set up for the interpreter. */
278:         /* Set up for the interpreter. */
279:         /* Set up for the interpreter. */
280:         /* Set up for the interpreter. */
281:         /* Set up for the interpreter. */
282:         /* Set up for the interpreter. */
283:         /* Set up for the interpreter. */
284:         /* Set up for the interpreter. */
285:         /* Set up for the interpreter. */
286:         /* Set up for the interpreter. */
287:         /* Set up for the interpreter. */
288:         /* Set up for the interpreter. */
289:         /* Set up for the interpreter. */
290:         /* Set up for the interpreter. */
291:         /* Set up for the interpreter. */
292:         /* Set up for the interpreter. */
293:         /* Set up for the interpreter. */
294:         /* Set up for the interpreter. */
295:         /* Set up for the interpreter. */
296:         /* Set up for the interpreter. */
297:         /* Set up for the interpreter. */
298:         /* Set up for the interpreter. */
299:         /* Set up for the interpreter. */
300:         /* Set up for the interpreter. */
301:         /* Set up for the interpreter. */
302:         /* Set up for the interpreter. */
303:         /* Set up for the interpreter. */
304:         /* Set up for the interpreter. */
305:         /* Set up for the interpreter. */
306:         /* Set up for the interpreter. */
307:         /* Set up for the interpreter. */
308:         /* Set up for the interpreter. */
309:         /* Set up for the interpreter. */
310:         /* Set up for the interpreter. */
311:         /* Set up for the interpreter. */
312:         /* Set up for the interpreter. */
313:         /* Set up for the interpreter. */
314:         /* Set up for the interpreter. */
315:         /* Set up for the interpreter. */
316:         /* Set up for the interpreter. */
317: 
```

Dr. GuoJun LIU

Operating System

Slides-98



Where

Slides-96

可执行程序

```

298: brelse(bh);
299: if (N_MAGIC(ex) != 2MAGIC || ex.a_trsize || ex.a_drszie || 
300:    ex.a_text+ex.a_data+ex.a_bss>0x30000000 || 
301:    inode->i_size < ex.a_text+ex.a_data+ex.a_sym+~N_TXTOFF(ex)) {
302:     retval = -ENOEXEC;
303:     goto (exec_error2);
304: }
305: if (N_TXTOFF(ex) != BLOCK_SIZE) {
306:     printk("%%: N_TXTOFF != BLOCK_SIZE. See a.out.h.\n", filename);
307:     retval = -ENOEXEC;
308:     goto (exec_error2);
309: }
310: if (!bh->b_text) {
311:     p = copy_strings(argc,argv,page,p,0);
312:     p = copy_strings(argv,page,0,0);
313:     if (!p) {
314:         retval = -ENOMEM;
315:         goto (exec_error2);
316:     }
317: }
```

Dr. GuoJun LIU

Operating System

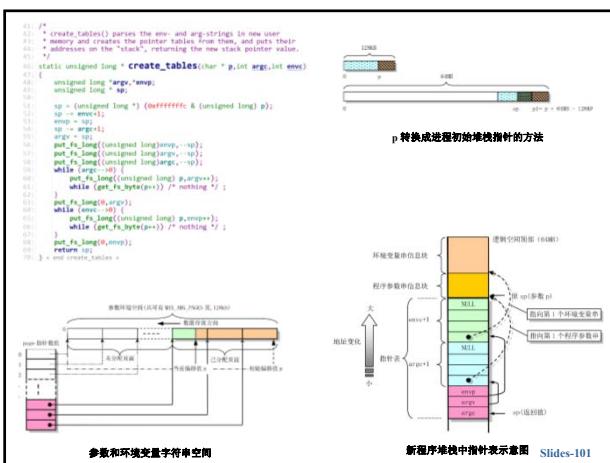
Slides-99

```

318: /* OK, This is the point of no return */
319: if (current->executable)
    释放当前的可执行文件inode节点
320:     out(&(current->executable));
321: current->executable = inode;
322: for (i=0 ; i<32 ; i++)
    Line 199: inode->name(filename)
323:     current->sigtab[i].sa_handler = NULL;
324: for (i=0 ; i<32 ; i++)
325:     if ((current->close_on_exec>>i)&1)
        满空当前进程的页表映射
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(0x1f));
330: if (last_text_usertable == current)
    改变ldt 制作参数表（类似指针数组）
331:     last_text_usertable = NULL;
332: current->used_math = NULL;
333: p += change_ldt(ex_a.text.page)-MAX_ARG_PAGES*PAGE_SIZE;
334: p = (unsigned long)create_tables((char *)p,argc,envc);
    写入代码结束位置、数据结束位置、bss结束位置
335: 
336: 
337:     (current->end_data - ex_a.data +
338:      (current->end_code - ex_a.text));
339: current->start_stack = e_ulp;
340: current->segid = e_gid;
341: current->end_data = ex_a.data;
342: while (1&0fff)
    4KB对齐
343:     put_fs_byte(0,(char *)i++);
344:     eip[0] = ex_a.entry; /* eip, magic happens :- */
345:     eip[1] = p; /* stack pointer */
346:     return 0;
347: exec_error2();
348: input(inode), where
349: exec_error1:
350:     for (i=0 ; i<MAX_ARG_PAGES ; i++)
            将系统调用压栈里的cip, 改为应用程
351:         put_long(page[i]);
352:     return (retval);
353: } = end do_execute =

```

Slides-100



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;
    计算代码段所占页面，并页对齐
160:     data_limit = 0xffffffff00000000;
161:     data_base = 0x40000000;
162:     code_base = get_base(current->ldt[1]);
163:     set_base(current->ldt[1],code_base);
164:     set_base(current->ldt[1],code_limit);
165:     set_base(current->ldt[1],data_base);
166:     set_limit(current->ldt[1],data_limit);
167:     /* make sure the page is aligned to page size */
168:     __asm__ ("pushl $0x17\n\tpop %fs%0");
169:     data_base = 0x40000000;
170:     for (i=MAX_ARG_PAGES-1 ; i>0 ; i--)
    fs 0x17指向用户数据段
171:         data_base -= PAGE_SIZE;
172:         if (i>1)
173:             put_page(page[i],data_base);
174:         else
175:             return data_limit;
176: } = end change_ldt =

```

Slides-102