Lab3 introduction

2010-5

Lab3任务清单

- 用户环境的建立
- 实现中断处理
- 实现系统调用

概要

- 引言
- **Part A**
- **Part B**

引言

- 新的debug命令
- 内联汇编简介
- 重要文件说明
- 重要数据结构
- **Struct task_struct v.s. struct Env**

引言

- 新的debug命令
- 内联汇编简介
- 重要文件说明
- 重要数据结构
- **Struct task_struct v.s. struct Env**

Bochs debug 命令

- **info idt**
  - 查看idt的内容
  - 在lab3中，用于检查idt的设置是否正确
- **vb**
  - 在逻辑地址上设断点
  - 注意参数—— **selector:offset**
    - 内核地址 **vb 8:offset**
    - 用户地址 **vb 0x1b:offset**

```
// Global descriptor numbers
#define GD_KT    0x08    // kernel text
#define GD_KD    0x10    // kernel data
#define GD_UT    0x18    // user text
#define GD_UD    0x20    // user data
#define GD_TSS   0x28    // Task segment selector
```

(why 0x1b?)

Bochs debug 命令——举例(1)

- 在obj/kernel.asm中找到lidt的位置
  - 0xf0103a4e(各人可能不同)
- 在lidt的上一条指令地址设置0xf0103a4b处设置断点
  - vb 8: 0xf0103a4b
- 运行到断点处，在lidt前查看idt内容
  - info idt
- 在执行完lidt之后查看idt的内容

Bochs debug 命令——举例(2)

```
00000000000i[     ] reading configuration from .bochsrc
Next at t=0
(0) [0xfffffff0] f000:fff0 (unk. ctxt): jmp far f000:e05b         ; ea5be000f0
<bochs:1> vb 8:0xf0103a4b
<bochs:2> c
(0) Breakpoint 1, 0xf0103a4b (0x0008:0xf0103a4b)
Next at t=10204580
(0) [0x00103a4b] 0008:0xf0103a4b (unk. ctxt): ltr ax               ; 0f00d8
<bochs:3> info idt
Interrupt Descriptor Table (0x00000000):
error: IDTR+8*0 points to invalid linear address 0x0
<bochs:4> s
Next at t=10204581
(0) [0x00103a4e] 0008:0xf0103a4e (unk. ctxt): lidt ds:0xf0119358   ; 0f011d589311f0
<bochs:5> s
Next at t=10204582
(0) [0x00103a55] 0008:0xf0103a55 (unk. ctxt): pop ebx              ; 5b
<bochs:6> info idt
Interrupt Descriptor Table (0xf0174f80):
IDT[0x00]=32-Bit Interrupt Gate target=0x0008:0xf0103dc4, DPL=0
IDT[0x01]=32-Bit Interrupt Gate target=0x0008:0xf0103dca, DPL=0
IDT[0x02]=32-Bit Interrupt Gate target=0x0008:0xf0103dd0, DPL=0
IDT[0x03]=32-Bit Interrupt Gate target=0x0008:0xf0103dd6, DPL=3
```

引言

- 新的debug命令
- 内联汇编简介
- 重要文件说明
- 重要数据结构
- Struct task_struct v.s. struct Env

GCC内联汇编回顾

- GCC支持在C/C++代码中嵌入汇编代码，这些汇编代码被称作GCC内联汇编(GCC Inline Assembly)
- 功能：
  - 可以使用内联汇编表达一些C/C++语言中无法表达的指令
  - 允许我们直接在C/C++代码中使用汇编语言编写简洁高效的代码

GCC内联汇编-基本格式

- asm (“statements”);
- 说明：
  - __asm或asm 用来声明一个内联汇编表达式，所以任何一个内联汇编表达式都是以它开头的，是必不可少的
  - statements是指令序列，每条指令都必须被双引号括起来
- 示例：
  - asm ("pushl %eax\n\t“
    "movl $0, %eax\n\t“
    "popl %eax");
- 注意：如果修改了寄存器，可能会造成灾难性后果

GCC内联汇编-扩展格式

- asm ("statements"
  : output_registers
  : input_registers
  : clobbered_registers);
- output_registers：用来指定当前内联汇编语句的输出
- input_registers：用来指定当前内联汇编语句的输入
- clobbered_registers：声明当前内联汇编在statements中对某些寄存器或内存进行修改

示例-1

```
asm ("cld\n\t"
     "rep\n\t"
     "stosl"
     : /* no output registers */
     : "c" (count), "a" (fill_value), "D" (dest)
     : "%ecx", "%edi"
   );
```

示例-2

- asm ("leal (%%ebx,%%ebx,4), %%ebx"
  : "=b" (x)
  : "b" (x));
- 指令中寄存器前必须使用两个百分号(%%)，而不是像基本汇编格式一样在寄存器前只使用一个百分号(%)

引言

- 新的debug命令
- 内联汇编简介
- 重要文件说明
- 重要数据结构
- Struct task_struct v.s. struct Env

重要文件说明

- **env.c：** env的初始化、创建、释放等
- **trap.c：** 中断处理相关，包含dispatch
- **trapentry.S：** 中断入口的生成及中断相关的栈操作
- **syscall.c：** 系统调用相关
- ***.h：** ……

引言

- 新的debug命令
- 内联汇编简介
- 重要文件说明
- 重要数据结构
- Struct task_struct v.s. struct Env

重要数据结构（1/2）

- NENV：宏定义的可被创建的最大的环境数量
- struct Env *envs：全部环境的数组
- struct Env *curenv：当前运行环境
- static struct Env_list env_free_list: 环境的空闲链表
- struct Env：环境结构

重要数据结构（2/2）

```
struct  Env {
    struct Trapframe env_tf;          // 切换时，在此保存寄存器
    LIST_ENTRY(Env) env_link;         //空闲链表指针
    envid_t env_id;                   //环境id，唯一辨识
    envid_t env_parent_id;            // 父亲的id
    unsigned env_status;              // 环境的状态
    uint32_t env_runs;                //环境已运行次数
          // Address space
    pde_t *env_pgdir;                 // 页目录的虚拟地址
    physaddr_t env_cr3;               //页目录的物理地址
};
```

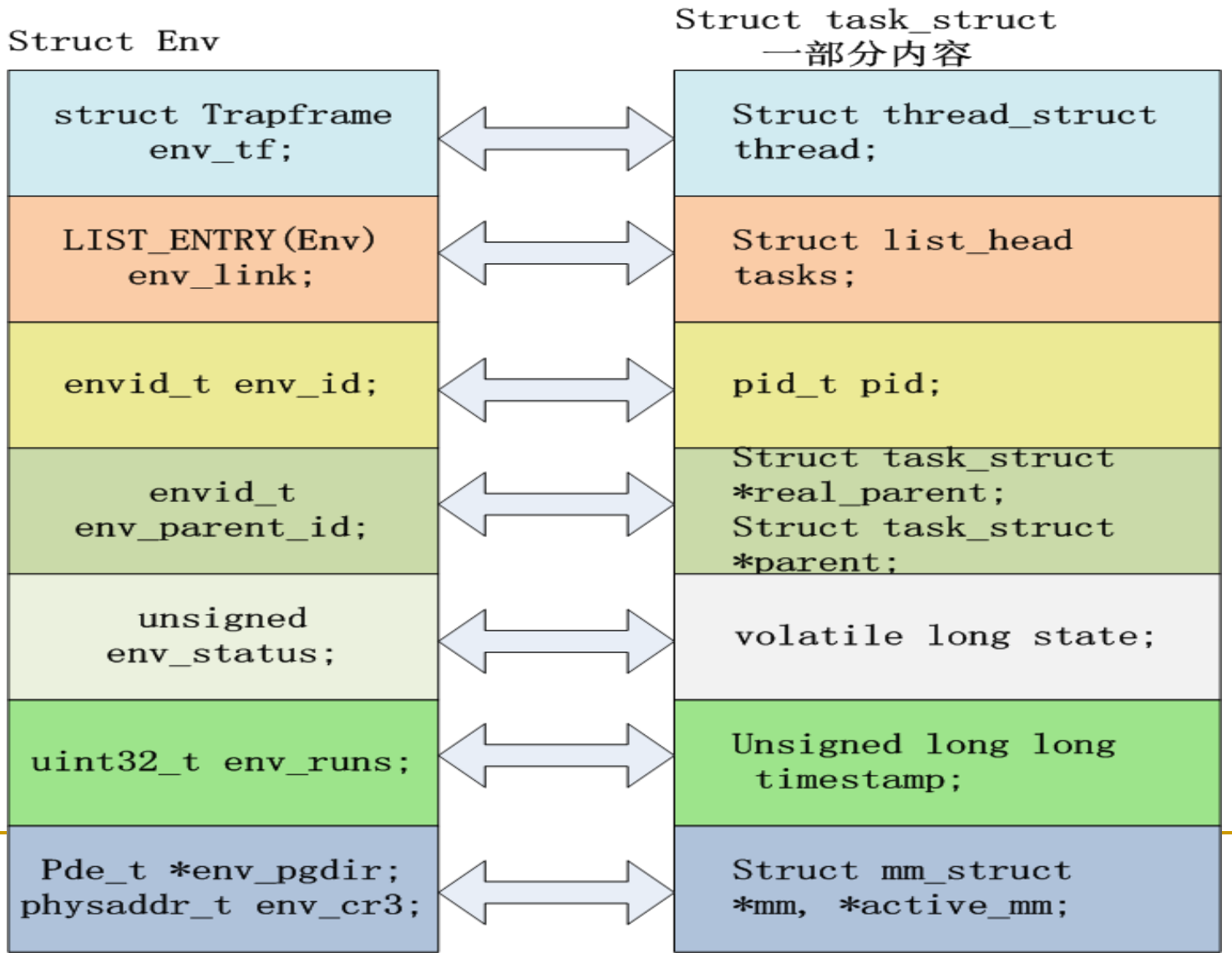
引言

- 新的debug命令
- 内联汇编简介
- 重要文件说明
- 重要数据结构
- Struct task_struct v.s. struct Env

Linux中的进程描述符

- Linux 内核2.6.11版本用 struct task_struct 来描述进程
- Struct Task_struct比struct Env 复杂得多。Struct Env里的各个结构成员在struct Task_struct中都可以找到对应的部分

Struct task_struct v.s. struct Env



完整的task_struct

```
struct task_struct {
    volatile long state;
    struct thread_info *thread_info;
    atomic_t usage;
    unsigned long flags;
    unsigned long ptrace;

    int lock_depth;

    int prio, static_prio;
    struct list_head run_list;
    prio_array_t *array;

    unsigned long sleep_avg;
    unsigned long long timestamp, last_ran;
    int activated;

    unsigned long policy;
    cpumask_t cpus_allowed;
    unsigned int time_slice, first_time_slice;

#ifdef CONFIG_SCHEDSTATS
    struct sched_info sched_info;
#endif

    struct list_head tasks;
    struct list_head ptrace_children;
    struct list_head ptrace_list;

    struct mm_struct *mm, *active_mm;

    struct linux_binfmt *binfmt;
    long exit_state;
    int exit_code, exit_signal;
    int pdeath_signal;
    unsigned long personality;
    unsigned did_exec:1;
    pid_t pid;
    pid_t tgid;
    struct task_struct *real_parent;
    struct task_struct *parent;
    struct list_head children;
    struct list_head sibling;
    struct task_struct *group_leader;

    struct pid pids[PIDTYPE_MAX];

    struct completion *vfork_done;
    int __user *set_child_tid;
    int __user *clear_child_tid;

    unsigned long rt_priority;
    unsigned long it_real_value, it_real_incr;
    cputime_t it_virt_value, it_virt_incr;
    cputime_t it_prof_value, it_prof_incr;
    struct timer_list real_timer;
    cputime_t utime, stime;
    unsigned long nvcsw, nivcsw;
    struct timespec start_time;
    unsigned long min_flt, maj_flt;
    uid_t uid,euid,suid,fsuid;
    gid_t gid,egid,sgid,fsgid;
    struct group_info *group_info;
    kernel_cap_t    cap_effective,\
        cap_inheritable, cap_permitted;
    unsigned keep_capabilities:1;
    struct user_struct *user;
#ifdef CONFIG_KEYS
    struct key *session_keyring;
    struct key *process_keyring;
    struct key *thread_keyring;
#endif
    int oomkilladj;
    char comm[TASK_COMM_LEN];
    int link_count, total_link_count;
    struct sysv_sem sysvsem;
    struct thread_struct thread;
    struct fs_struct *fs;
    struct files_struct *files;
    struct namespace *namespace;
    struct signal_struct *signal;
    struct sighand_struct *sighand;

    sigset_t blocked, real_blocked;
    struct sigpending pending;

    unsigned long sas_ss_sp;
    size_t sas_ss_size;
    int (*notifier)(void *priv);

    void *notifier_data;
    sigset_t *notifier_mask;

    void *security;
    struct audit_context *audit_context;

    u32 parent_exec_id;
    u32 self_exec_id;
    spinlock_t alloc_lock;
    spinlock_t proc_lock;
    spinlock_t switch_lock;

    void *journal_info;

    struct reclaim_state *reclaim_state;

    struct dentry *proc_dentry;
    struct backing_dev_info *backing_dev_info;

    struct io_context *io_context;

    unsigned long ptrace_message;
    siginfo_t *last_siginfo;
    wait_queue_t *io_wait;
    u64 rchar, wchar, syscr, syscw;
#if defined(CONFIG_BSD_PROCESS_ACCT)
    u64 acct_rss_mem1;
    u64 acct_vm_mem1;
    clock_t acct_stimexpd;
#endif
#ifdef CONFIG_NUMA
    struct mempolicy *mempolicy;
    short il_next;
#endif
};
```

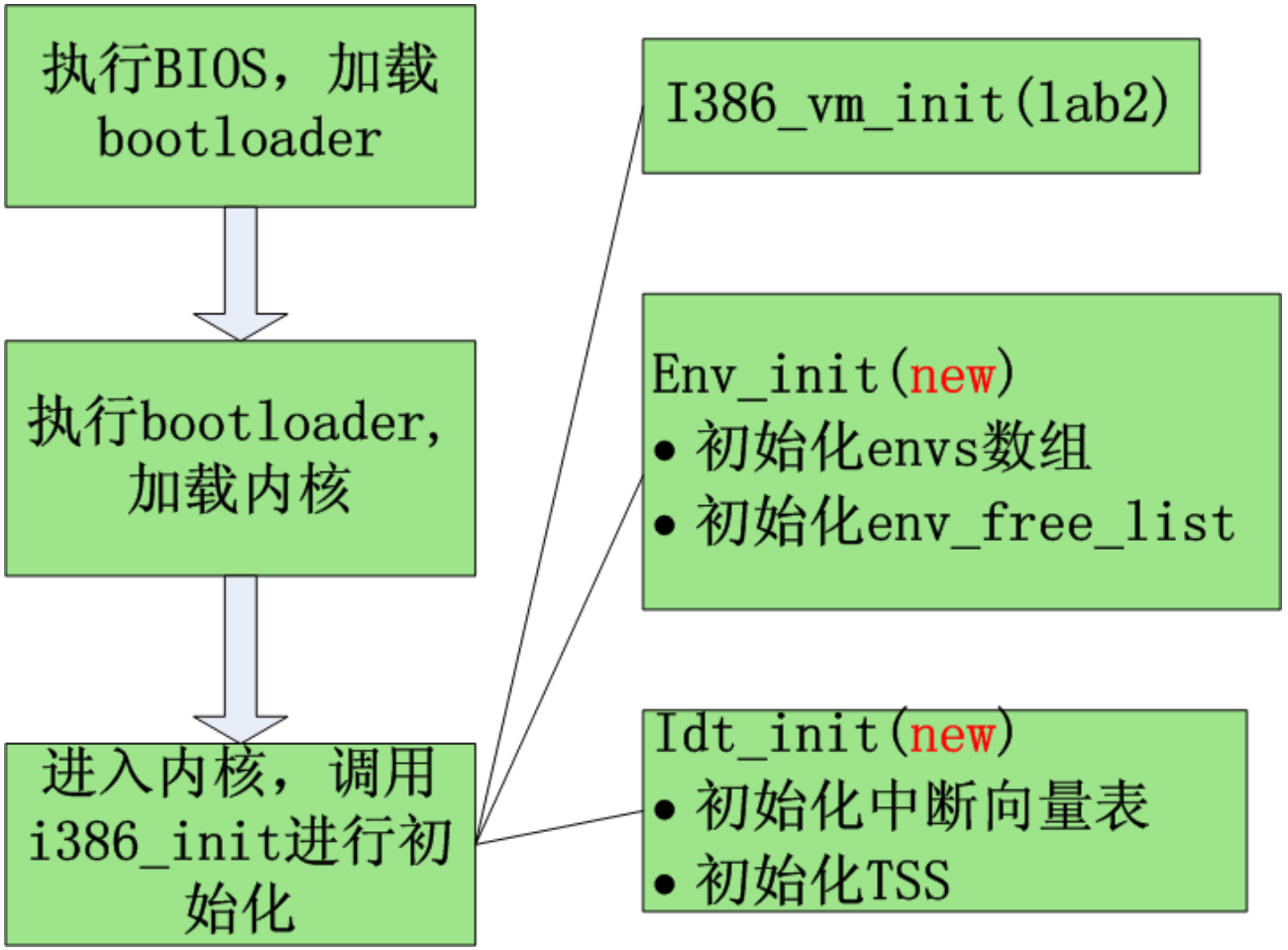
Part A

- 新的PC启动初始化过程
- load_icode-ELF文件的加载
- interrupt VS exception
- 中断处理流程

Part A

- 新的PC启动初始化过程
- load_icode-ELF文件的加载
- interrupt VS exception
- 中断处理流程

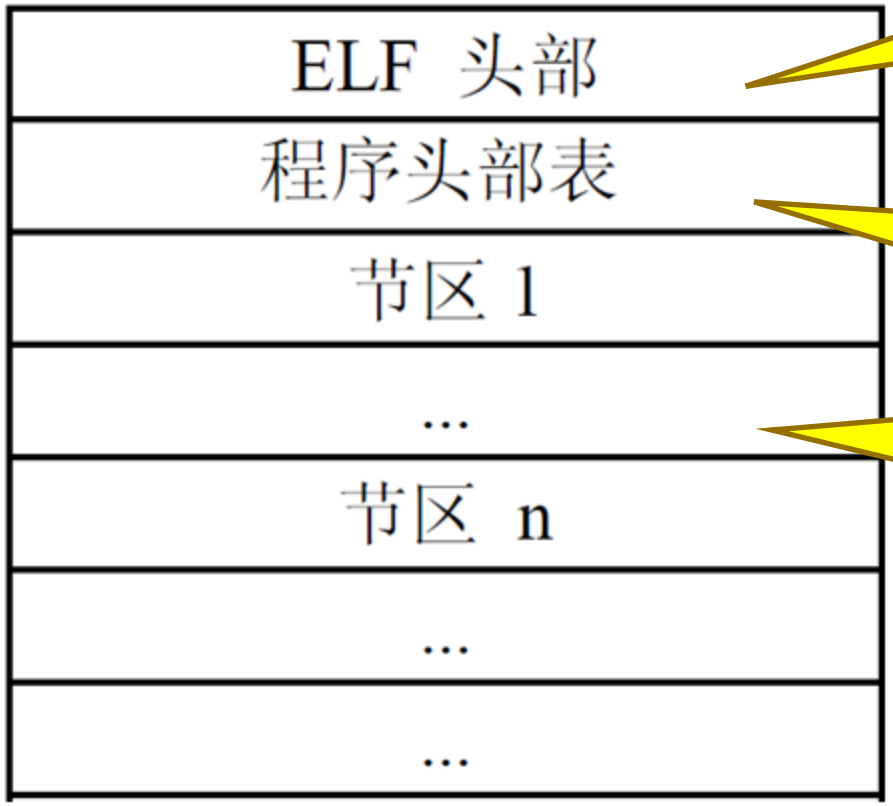
初始化流程


执行BIOS，加载bootloader
执行bootloader，加载内核
进入内核，调用i386_init进行初始化
I386_vm_init(lab2)
Env_init(new)
• 初始化envs数组
• 初始化env_free_list
Idt_init(new)
• 初始化中断向量表
• 初始化TSS

Part A

- 新的PC启动初始化过程
- load_icode-ELF文件的加载
- interrupt VS exception
- 中断处理流程

ELF文件格式（回顾）



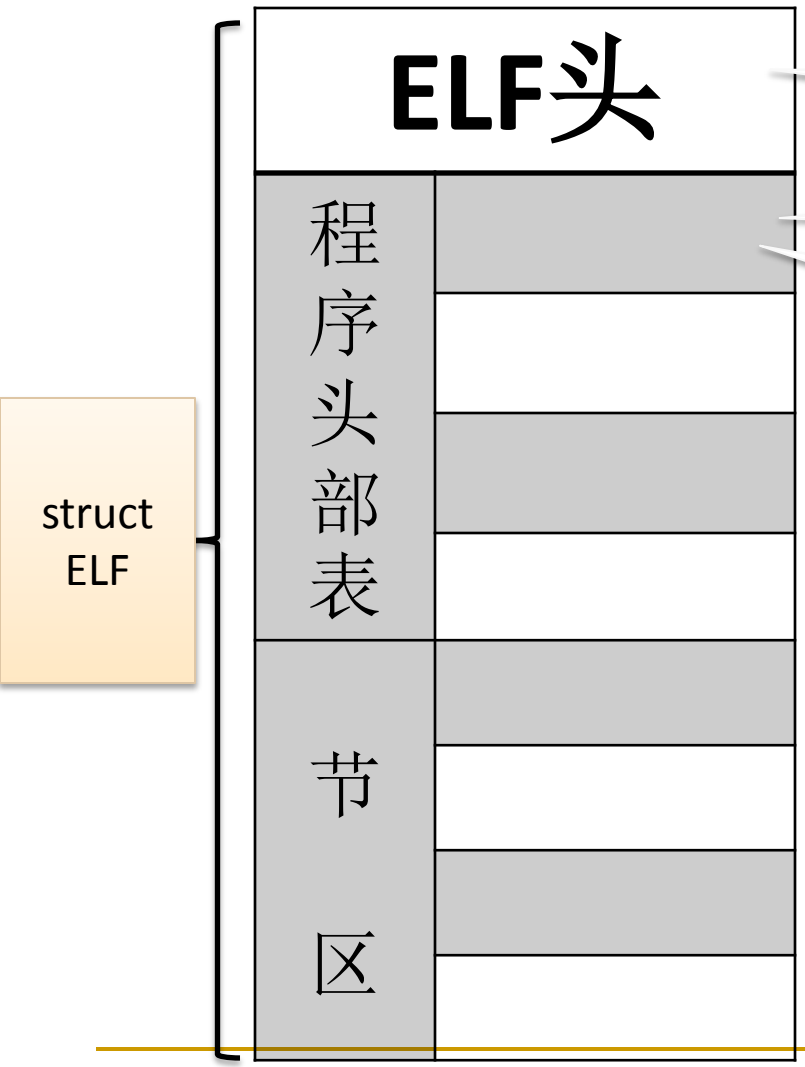
描述整个文件的组织

告诉系统如何创建进程映像。用来构造进程映像的目标文件必须具有程序头部表

节区部分包含链接视图的大量信息：指令、数据、符号表、重定位信息等等

Section VS Sector VS Segment

与ELF文件相关的数据结构



长度为ELF->e_phoff

和节(section)一一对应

```
struct proghdr{
    p_type; //段的类型
    p_va;//段的第一个字节将被放
           到内存中的虚拟地址。
    p_offset;//从文件头到该段第一个字
           节的偏移
    p_filesz;//段在文件映像中所占的字
           节数
    p_memsz;//段在内存映像中占用的
           字节数
}
```

加载ELF文件的流程

1. 略过ELF头；
2. 找到第一个section对应的struct proghdr；
3. while(还有section没有加载), do
4. 　分配p_memsz大小的内存
5. 　根据p_offset找到section；
6. 　把section开始的p_filesz导入内存；
7. 　将剩余部分的内存设置为0；
8. end while

** ELF_PROG_LOAD

Part A

- 新的PC启动初始化过程
- load_icode-ELF文件的加载
- interrupt VS exception
- 中断处理流程

中断与异常

- **Exceptions and Interrupts**
  - 区别在于：中断处理异步事件（相对处理器是外部的），而异常是指处理器自己检测到的情况
  - 中断的分类：可屏蔽、不可屏蔽
  - 异常的分类：处理器检测、程序触发(bp,int 0x30)

中断号对应表

Table 9-1. Interrupt and Exception ID Assignments

Identifier	Description
0	Divide error
1	Debug exceptions
2	Nonmaskable interrupt
3	Breakpoint (one-byte INT 3 instruction)
4	Overflow (INTO instruction)
5	Bounds check (BOUND instruction)
6	Invalid opcode
7	Coprocessor not available
8	Double fault
9	(reserved)
10	Invalid TSS
11	Segment not present
12	Stack exception
13	General protection
14	Page fault
15	(reserved)
16	Coprecessor error
17-31	(reserved)
32-255	Available for external interrupts via INTR pin

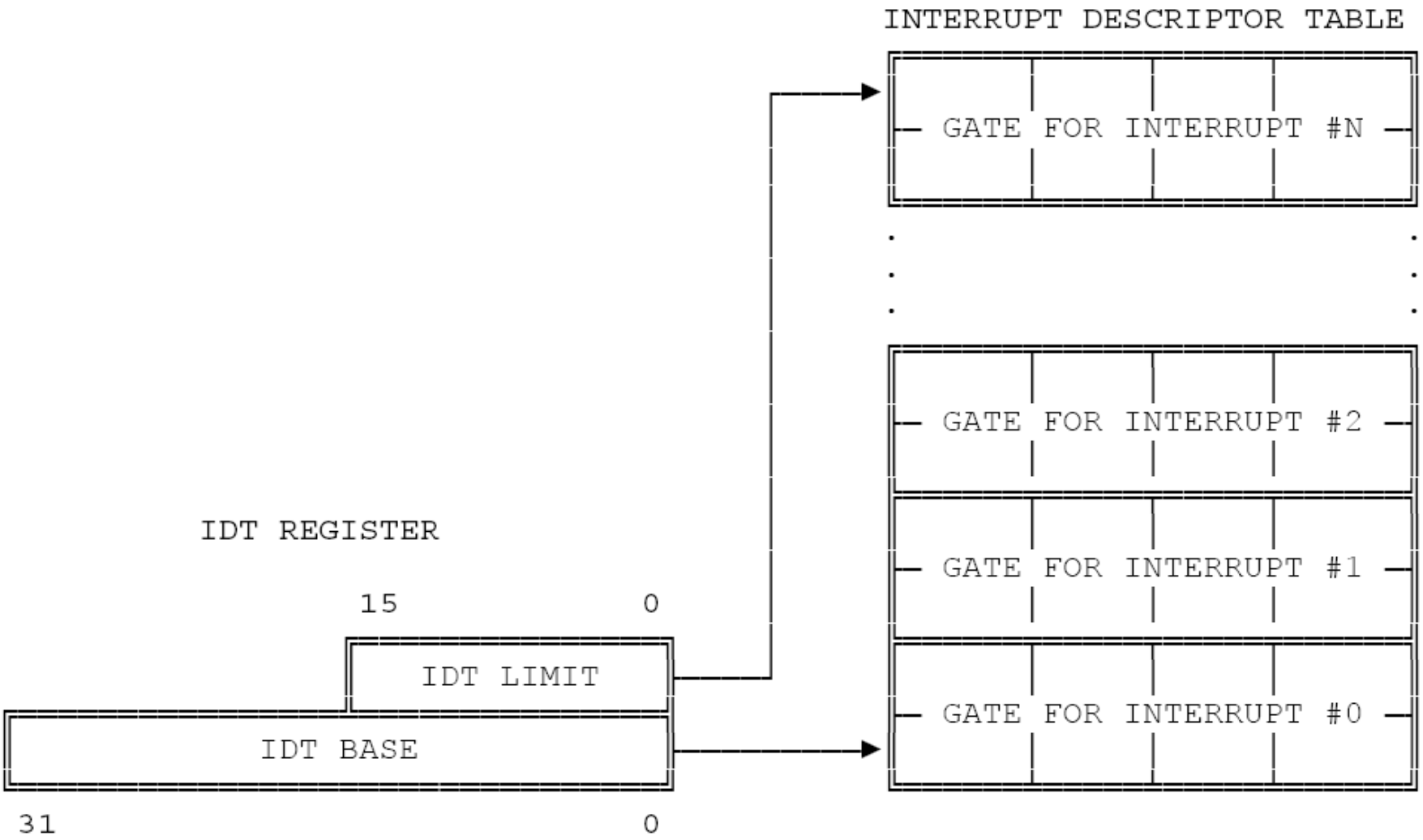
Part A

- 新的PC启动初始化过程
- load_icode-ELF文件的加载
- interrupt VS exception
- 中断处理流程

中断处理过程概览

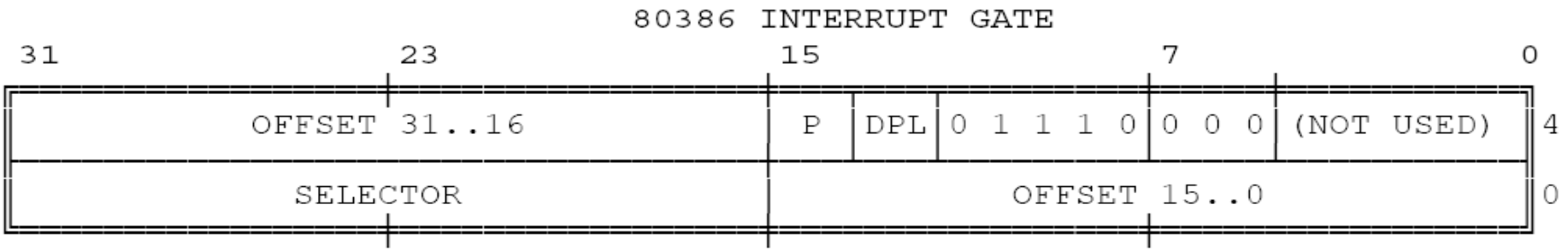
- 寄存器idtr保存了中断描述符表的基址和长度
- 每个中断描述符保存了一个中断处理程序入口的cs和eip
- 发生中断时，硬件用获得的中断号作为中断描述符表的下标，找到相应的中断处理程序入口，执行中断处理程序

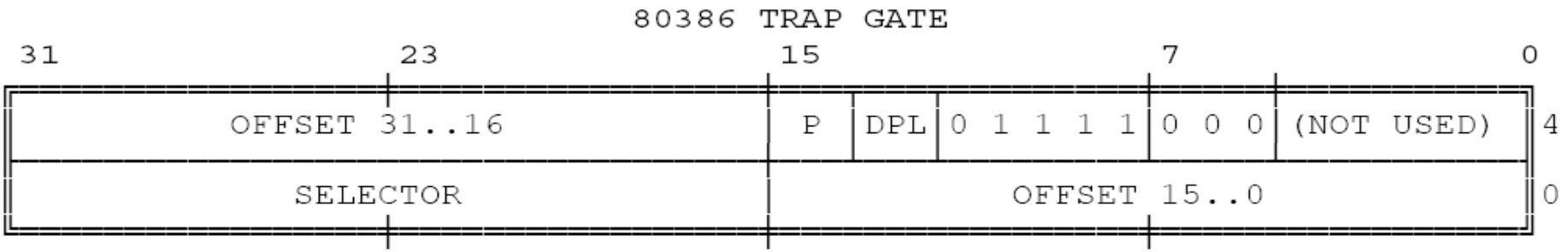
IDTR和idt(interrupt descriptor table)



中断描述符

- **Interrupt gate**
  - 定义中断处理程序的入口
- **Trap gate**
  - 定义异常处理程序的入口





Idt的初始化

- 在kern/trap.c中定义了全局数组

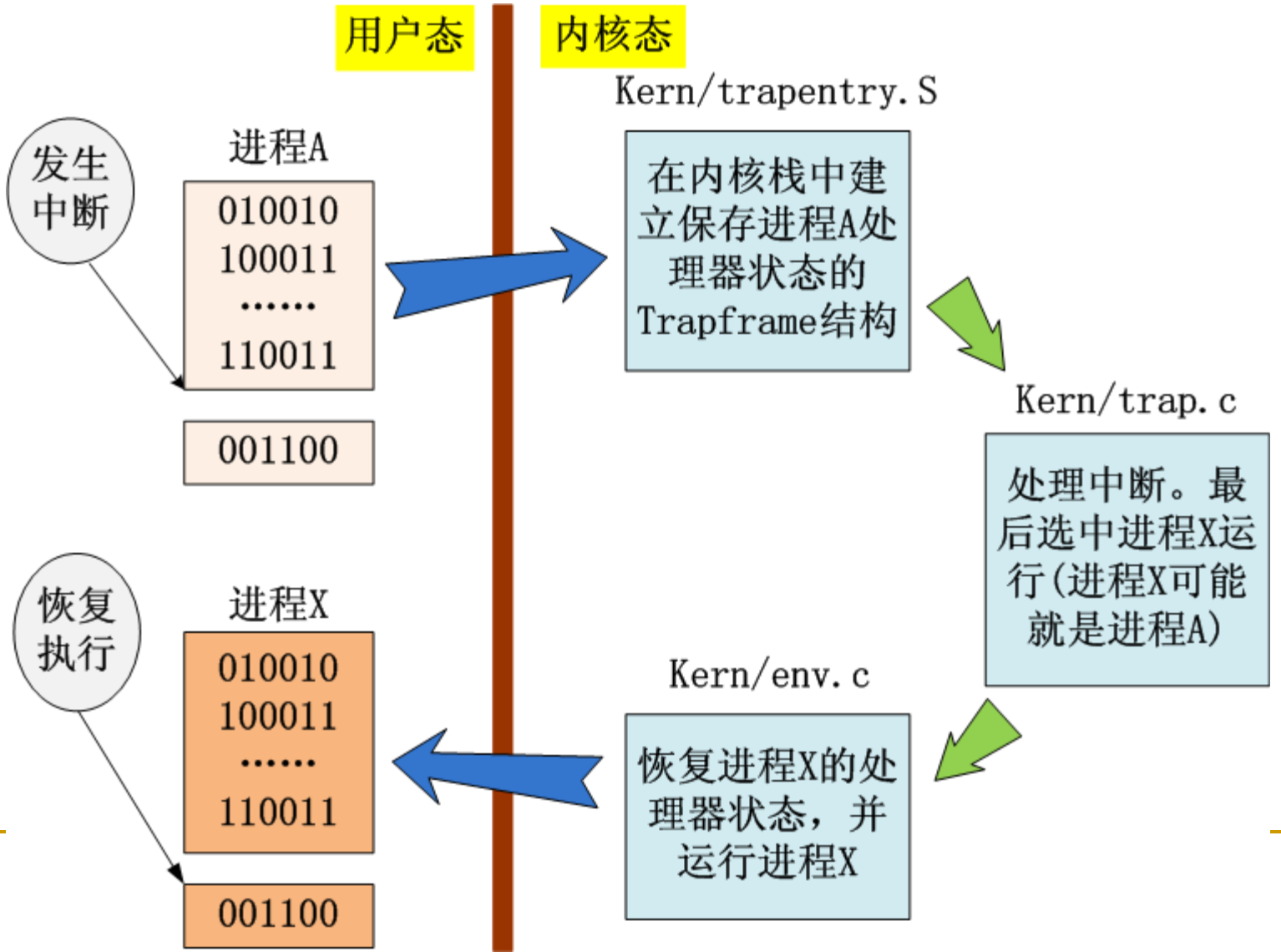
  struct Gatedesc idt[256] = { { 0 } };

  作为中断描述符表
- 在idt_init中使用宏SETGATE设置中断描述符
  - 中断处理程序入口在kern/trapentry.S中定义。所以要想引用中断处理程序入口的eip，需要使用 C中的extern关键字

中断处理程序

- 在kern/trapentry.S中，使用宏TRAPHANDLER和TRAPHANDLER_NOEC定义中断处理程序的入口
- 中断处理程序先在栈中设置好struct Trapframe，然后调用kern/trap.c 里面的trap函数进行中断处理

图示


用户态
内核态
Kern/trapentry.S
发生中断
进程A
010010
100011
……
110011
001100
在内核栈中建立保存进程A处理器状态的Trapframe结构
Kern/trap.c
处理中断。最后选中进程X运行(进程X可能就是进程A)
恢复执行
进程X
010010
100011
……
110011
001100
Kern/env.c
恢复进程X的处理器状态，并运行进程X

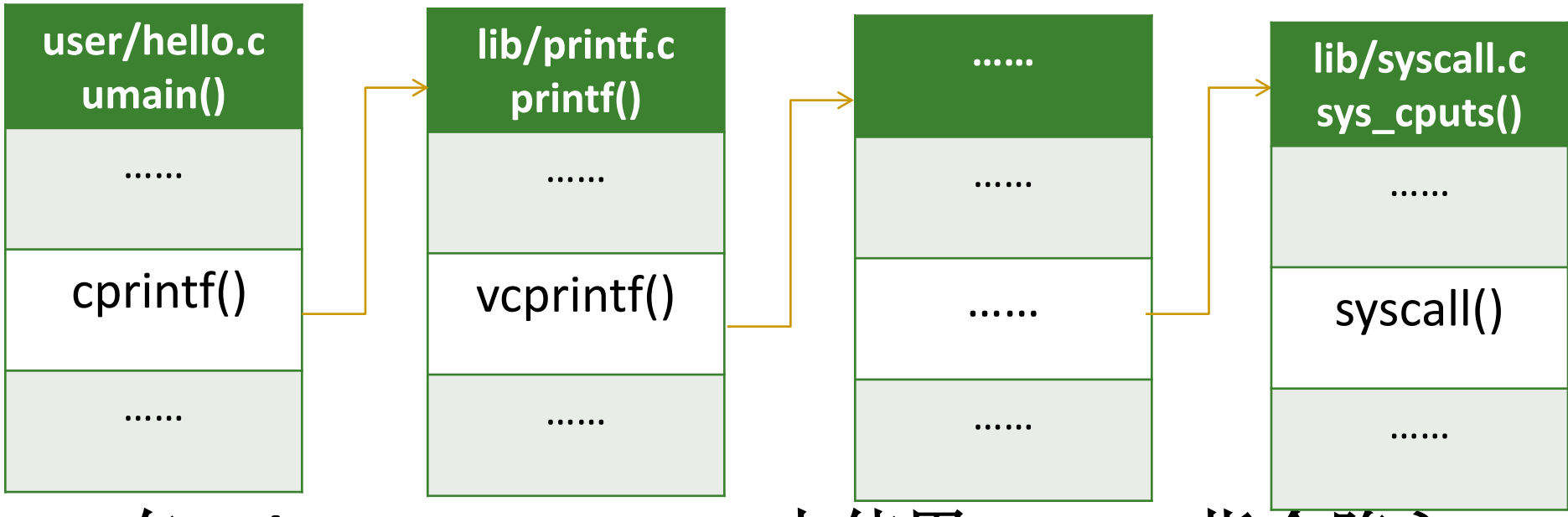
Part B

- 系统调用原理与简单流程

系统调用

- 系统调用通过软件中断实现
  - int 0x30
- 设置idt
  - 需要在idt中进行相应的设置
- 参数传递：
  - 传递系统调用号:EAX ； 5个参数:EDX, ECX, EBX, EDI, ESI
  - 返回值:EAX
- 思考：为什么用寄存器传参？

系统调用—流程实例1



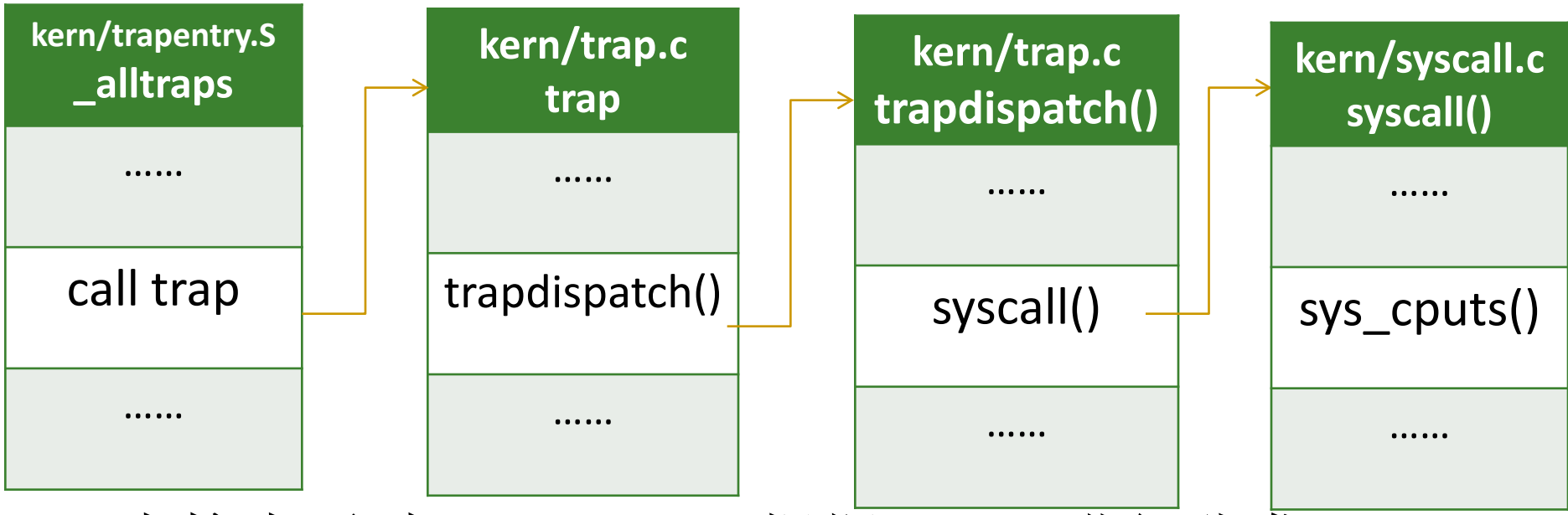
- 在lib/syscall.c--syscall()中使用int 0x30指令陷入到内核态

syscall()

```
asm volatile("int %1\n"
        : "=a" (ret)
        : "i" (T_SYSCALL),
          "a" (num),
          "d" (a1),
          "c" (a2),
          "b" (a3),
          "D" (a4),
          "S" (a5)
        : "cc", "memory");
```

- 要求：阅读参考资料中的Inline Assembly with DJGPP.mht

系统调用—流程实例2



- 内核中通过**trapdispatch**根据**trapno**进行分发
- **kern/syscall.c/syscall()**是实现系统调用指定功能的函数
- 注意与**lib**目录中**syscall()**函数的区别

Q&A