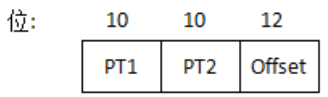
Lab3：进程环境

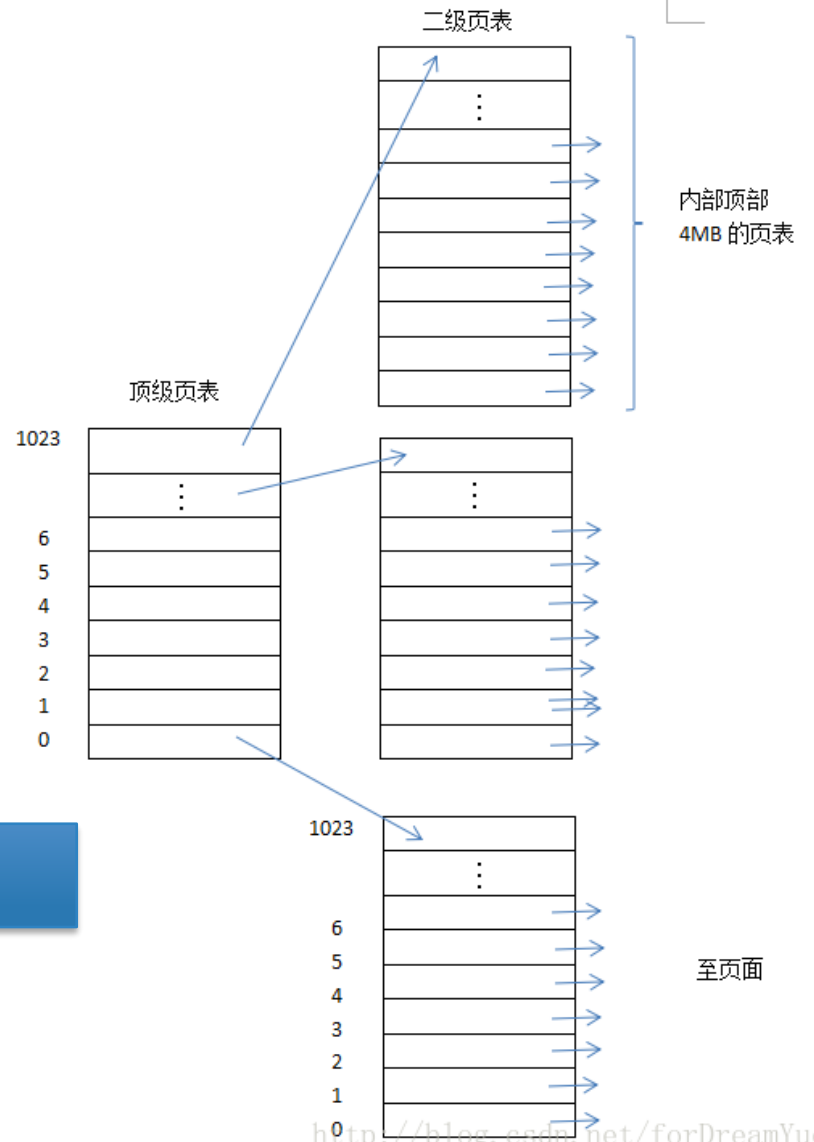
Lab 3

- Process
- Exception Handling
- Page Faults
- System Calls

回忆：页表自映射

- 方便在虚拟地址空间直接访问页表和页表项
- 节省4KB的空间（32位，两级页表的系统）
  - 把所有的页表（4KB）放到连续的4MB 虚拟地址空间中
  - 4MB对齐
  - 会有一张页表的内容与页目录的内容完全相同





10位一级索引	10位二级索引	12位页内偏移

JOS中PCB包括那些内容？是怎么组织的？

进程关系
进程状态
地址空间
方便内核管理的一些额外信息

```
46  struct Env {
47      struct Trapframe env_tf;    // Saved registers
48      struct Env *env_link;       // Next free Env
49      envid_t env_id;           // Unique environment identifier
50      envid_t env_parent_id;      // env_id of this env's parent
51      enum EnvType env_type;      // Indicates special system environments
52      unsigned env_status;        // Status of the environment
53      uint32_t env_runs;      // Number of times environment has run
54
55      // Address space
56      pde_t *env_pgdir;       // Kernel virtual address of page dir
57  };
```

JOS中PCB包括那些内容？是怎么组织的？

- PCB的组织方法
  - envs数组维护所有的PCB
  - curenv指向当前运行中的进程PCB
  - env_free_list链表维护所有可用的PCB
- 一个进程能否查看其他进程状态？
  - 取决于一个进程能不能拿到其他进程的PCB
  - 存储PCB的内存映射时对所有进程可读（memlayout中的ENVS部分）

JOS的进程管理

1.还要映射内核最高256MB部分的二级页表

- JOS创建一个新进程需要分配哪些东西?
  - 一个PCB条目
  - 一个页目录
- JOS能容纳的最大进程数? 受哪些因素限制?
  - 为ENV分配了一个PTSIZE　　**PTSIZE / sizeof(PCB)**
- 进程能否访问kernbase以上?
  - kernbase以上，Kernel是R/W，user是-/-

用户进程的启动

- 用户程序的链接地址和加载地址如何指定?
- 进程的第一条指令在什么位置，是什么，如何指定?

user/user.ld

```
/* Simple linker script for JOS user-level programs.
   See the GNU ld 'info' manual ("info ld") to learn the syntax. */

OUTPUT_FORMAT("elf32-i386", "elf32-i386", "elf32-i386")
OUTPUT_ARCH(i386)
ENTRY(_start)

SECTIONS
{
    /* Load programs at this address: "." means the current address */
    . = 0x800020;

    .text : {
        *(.text .stub .text.* .gnu.linkonce.t.*)
    }
```

lib/entry.S

```
// Entrypoint - this is where the kernel (or our parent environment)
// starts us running when we are initially loaded into a new environment.
.text
.globl _start
_start:
    // See if we were started with arguments on the stack
    cmpl $USTACKTOP, %esp
    jne args_exist

    // If not, push dummy argc/argv arguments.
    // This happens when we are loaded by the kernel,
    // because the kernel does not know about passing arguments.
    pushl $0
    pushl $0

args_exist:
    call libmain
1:  jmp 1b
```

解读env_pop_tf()

- 唯一调用env_pop_tf()的过程是env_run()
  - 这个函数的目的是上下文切换
- popal: 恢复PushRegs里的8个寄存器
- popl %es
- popl %ds
- addl $0x8, %esp

- iret (Interrupt Return)
  - 和普通的ret比起来，多恢复了几个寄存器

  也就是TrapFrame里面最后几个
- env_pop_tf()不返回（panic）
  - 会用TrapFrame里的tf_eip覆盖当前的eip

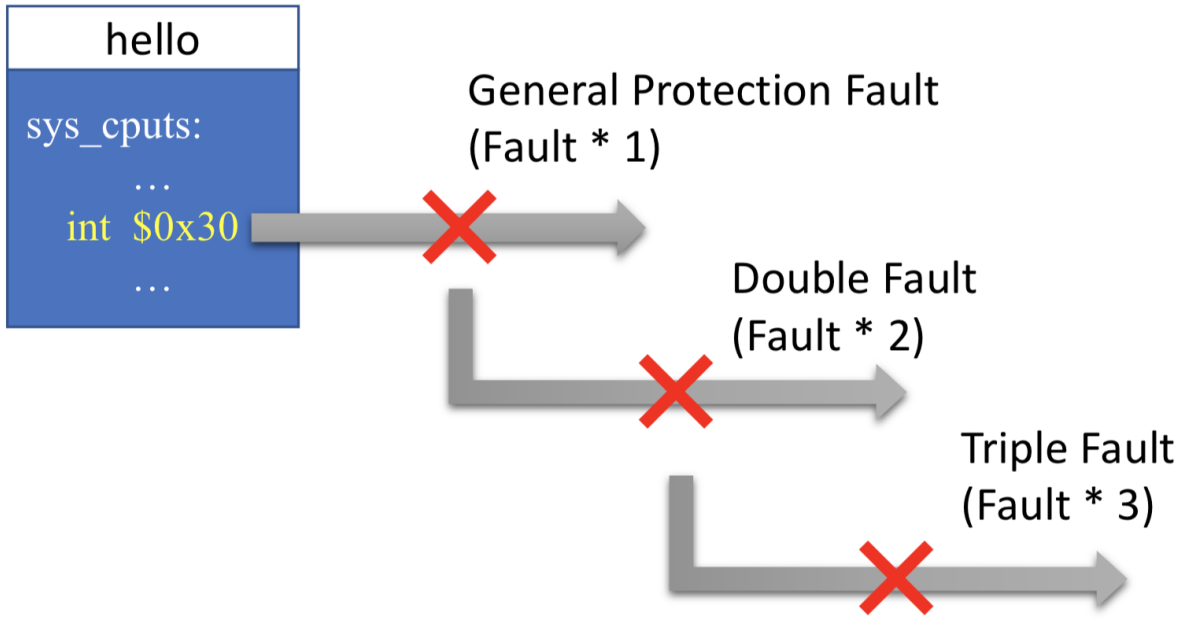
```
struct PushRegs {
    /* registers as pushed by pusha */
    uint32_t reg_edi;
    uint32_t reg_esi;
    uint32_t reg_ebp;
    uint32_t reg_oesp;      /* Useless */
    uint32_t reg_ebx;
    uint32_t reg_edx;
    uint32_t reg_ecx;
    uint32_t reg_eax;
} __attribute__((packed));
```

```
struct Trapframe {
    struct PushRegs tf_regs;
    uint16_t tf_es;
    uint16_t tf_padding1;
    uint16_t tf_ds;
    uint16_t tf_padding2;
    uint32_t tf_trapno;
    /* below here defined by x86 hardware */
    uint32_t tf_err;
    uintptr_t tf_eip;
    uint16_t tf_cs;
    uint16_t tf_padding3;
    uint32_t tf_eflags;
    /* below here only when crossing rings, such as from user to kernel */
    uintptr_t tf_esp;
    uint16_t tf_ss;
    uint16_t tf_padding4;
} __attribute__((packed));
void
env_pop_tf(struct Trapframe *tf)
{
    asm volatile(
        "\tmovl %0,%%esp\n"
        "\tpopal\n"
        "\tpopl %%es\n"
        "\tpopl %%ds\n"
        "\taddl $0x8,%%esp\n" /* skip tf_trapno and tf_errcode */
        "\tiret\n"
        : : "g" (tf) : "memory");
    panic("iret failed");  /* mostly to placate the compiler */
}
```

Questions：用户进程如何结束执行？为什么在实现用户进程环境后发生了triple fault?

- 一般使用exit(0)这个函数。
  - 在exit之后，用户进程发起系统调用进入内核态
  - 内核态把这个进程回收，所以再也不会返回用户进程。
- 实现用户进程环境后为什么会发生triple fault?

系统调用不存在->Fault
异常程序不存在->Fault
异常异常程序不存在->Fault
自动重启



中断和异常的区别，Fault、trap和abort的区别

- 中断（interrupts）：外部设备引起的
- 异常（exceptions）：CPU 内部事件所引起的
- fault、trap、abort是三种异常
  - fault是可纠正的异常，返回地址指向错误指令，而不是指向错误指令之后的指令（page fault）
  - trap的返回地址指向要在陷阱指令之后执行的指令（syscall）
  - abort不允许重新启动导致异常的程序或任务（除0异常）

IDT和TSS都包含什么内容?

"Task State Segment"（任务状态段）

- IDT保存有中断向量到中断处理程序的映射，每个映射都包含中断处理程序所在位置的EIP和CS值
- TSS定义从用户态切换到内核态处理器保护现场的栈的位置。TSS的地址存放在GDT中，TR寄存器存放TSS Selector
- 在trap_init_percpu中初始化TSS，并且有一个ltr操作，即把TSS段描述符的选择子装入在tr寄存器中。

```
void trap_init(void) {
    extern struct Segdesc gdt[];
    extern void* (*vectors[])();

    for (int i = 0; i < 20; ++i)
        if (i != 9 && i != 15)
            SETGATE(idt[i], 0, GD_KT, vectors[i], 0);
    SETGATE(idt[T_BRKPT], 0, GD_KT, vectors[T_BRKPT], 3);
    SETGATE(idt[T_SYSCALL], 0, GD_KT, vectors[T_SYSCALL], 3);

    trap_init_percpu();
}
```

允许在用户态发起

这里为什么单独设置为3?

```
void
trap_init_percpu(void)
{
    // Setup a TSS so that we get the right stack
    // when we trap to the kernel.
    ts.ts_esp0 = KSTACKTOP;
    ts.ts_ss0 = GD_KD;
    ts.ts_iomb = sizeof(struct Taskstate);

    // Initialize the TSS slot of the gdt.
    gdt[GD_TSS0 >> 3] = SEG16(STS_T32A, (uint32_t) (&ts),
                    sizeof(struct Taskstate) - 1, 0);
    gdt[GD_TSS0 >> 3].sd_s = 0;

    // Load the TSS selector (like other segment selectors, the
    // bottom three bits are special; we leave them 0)
    ltr(GD_TSS0);

    // Load the IDT
    lidt(&idt_pd);
}
```

从中断发生到进入中断处理程序，中间发生了什么？

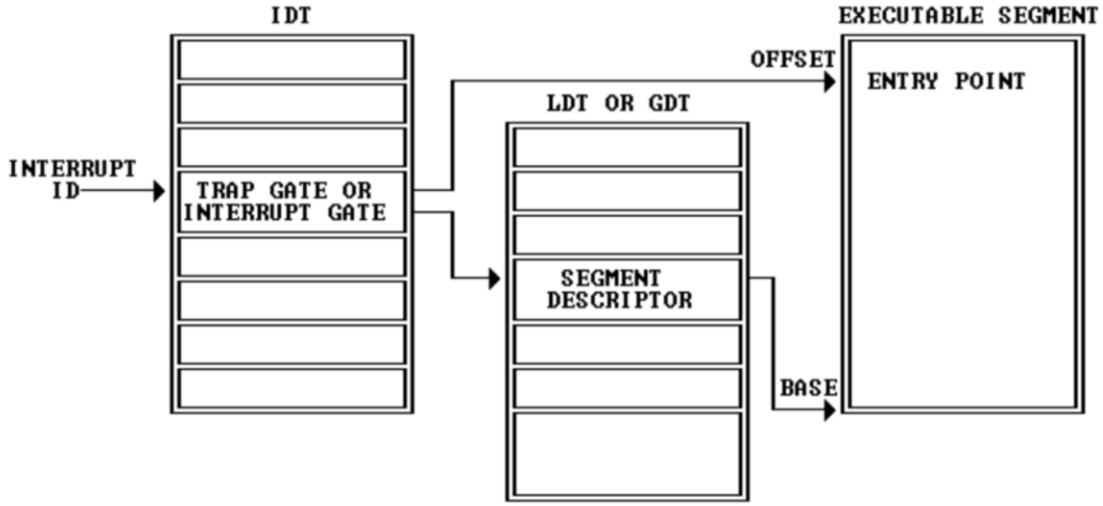
- 处理器根据TSS找到内核栈的位置
- 处理器将上下文保存到栈中
- 处理器根据中断向量从IDT中读取中断处理程序的CS和EIP
- 处理程序处理中断

TSS 的结构
- TSS 是一个固定格式的段，包含多个字段，例如：
- ESP0 和 SS0：内核模式栈的栈指针和段选择子。
- CR3：页表基地址（用于任务的虚拟内存）。
- EIP 和 ESP：保存任务执行时的指令指针和栈指针。
- I/O 位图：定义任务的 I/O 访问权限

在中断处理过程中，软件和硬件分别做了什么？

- 硬件：查找入口
- 中断发生，CPU获取中断向量号n
- CPU获取IDT基址，由n计算偏移，获取门描述符(中断/陷阱/任务)
- 根据描述符中的选择子，在GDT中找到中断处理程序所在段的基址，再加上描述符中提供的段内偏移，得到中断处理程序入口CS：IP

Figure 9-4. Interrupt Vectoring for Procedures



在中断处理过程中，软件和硬件分别做了什么？

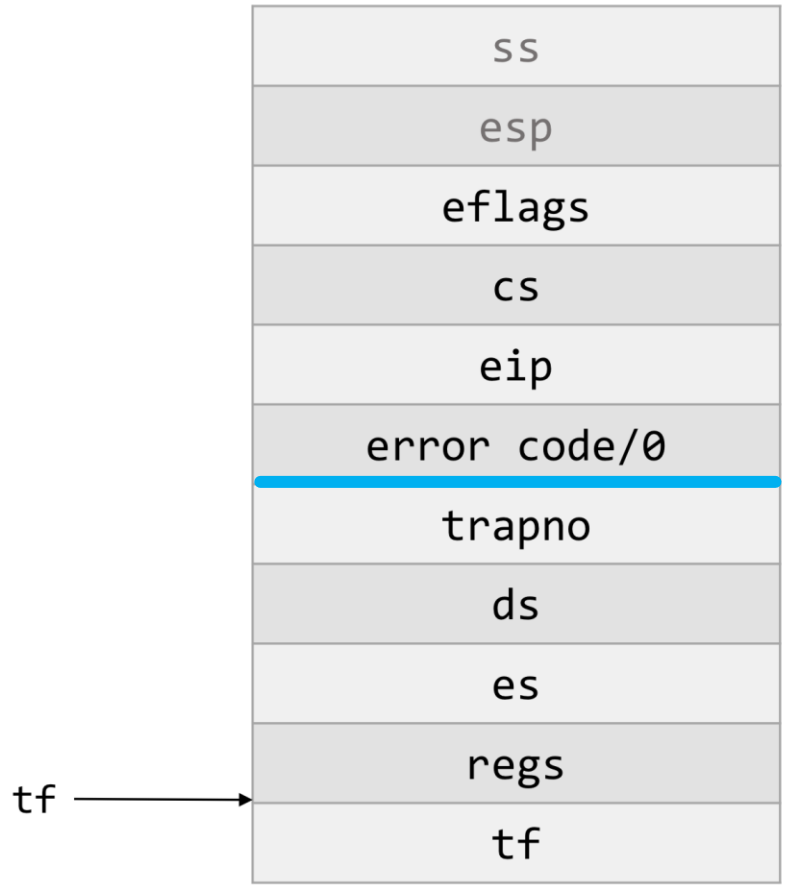
IF=0不允许中断

- 硬件部分：保存现场
- 特权级检查，同时检查是否发生特权级变化
- 压栈EFLAGS，如果是中断门，则将IF设为0.
- 压栈旧的cs和eip，并设置新的cs和eip为程序入口地址
- 压栈error code(如果需要)

在中断处理过程中，软件和硬件分别做了什么？

- 软件部分：数据结构初始化，构建trapframe，保存现场，切换数据段
- 数据结构初始化
  - env_init()初始化GDT
  - trap_init()初始化IDT和TSS
- 构建trapframe，保存现场，切换数据段

```
_alltraps:
    pushl   %ds
    pushl   %es
    pushal
    movw    $GD_KD, %ax
    movw    %ax, %ds
    movw    %ax, %es
    pushl   %esp
    call    trap
```



TRAPHANDLE和TRAPHANDLE_NOEC有什么区别？

- 他们的功能都是接受一个函数名和中断向量编号，把二者绑定。
- 二者的不同是CPU是否会把该中断的错误代码压到栈中，比如除零中断就不会放，用户使用int调用的也不会放。那么在不放的情况下TRAPHANDLE_NOEC就要手动补齐这个空间，手动push 0。

系统调用

- syscall是一种特殊的软中断
- JOS中如何发起syscall?
  - int $0x30
- JOS中syscall的调用规范是什么?
  - 调用号如何存放?
  - 调用参数如何存放?
  - 调用返回值如何存放?

```
case T_SYSCALL:
    regs = &tf->tf_regs;
    regs->reg_eax = syscall(regs->reg_eax,
                            regs->reg_edx,
                            regs->reg_ecx,
                            regs->reg_ebx,
                            regs->reg_edi,
                            regs->reg_esi);
    break;
```

```
envid_t
  sys_testarg(void)
{
  return syscall(42, 0, 101, 102, 103, 104, 105);
}
```

```
008001a9 <sys_testarg>:
  8001a9:       f3 0f 1e fb             endbr32
  8001ad:       55                      push   %ebp
  8001ae:       89 e5                   mov    %esp,%ebp
  8001b0:       57                      push   %edi
  8001b1:       56                      push   %esi
  8001b2:       53                      push   %ebx
  8001b3:       b8 2a 00 00 00          mov    $0x2a,%eax
  8001b8:       ba 65 00 00 00          mov    $0x65,%edx
  8001bd:       b9 66 00 00 00          mov    $0x66,%ecx
  8001c2:       bb 67 00 00 00          mov    $0x67,%ebx
  8001c7:       bf 68 00 00 00          mov    $0x68,%edi
  8001cc:       be 69 00 00 00          mov    $0x69,%esi
  8001d1:       cd 30                   int    $0x30
  8001d3:       5b                      pop    %ebx
  8001d4:       5e                      pop    %esi
  8001d5:       5f                      pop    %edi
  8001d6:       5d                      pop    %ebp
  8001d7:       c3                      ret
```

调用号+返回值

调用参数1-5

发起系统调用

Questions：user/softint为什么会产生13号中断?

- grade script显示它会产生general protection fault（陷阱 13），但 softint 的代码显示 int $14
- int $14 是页面错误，JOS 将 DPL 设置为 0。 应用程序处于用户模式，DPL 为 3，不允许调用 int $14，因此得到了General Protection Fault。
- 将T_PGFLT的 DPL 设置为 3，可以看到调用了 int $14。

Thanks