MIT6.828 Lab3: User Environments

Zhuofan Zhang

Jan 2022

Contents

Chapter 1

User Environments and Exception Handling

本次 Lab 的第一部分主要要处理 JOS 对进程的抽象及异常处理两部分的内容。

1.1 Environment State

JOS 对进程（Process/Environment）的抽象位于 inc/env.h 及 kern/env.c 中，包括结构体 struct Env 及一系列接口。系统使用三个全局变量：envs, curenv, env_free_list 对所有用户进程及当前进程进行管理。

对于每一个用户进程，JOS 使用 struct Env 表示：

```
struct Env {
    struct Trapframe env_tf;  // Saved registers
    struct Env *env_link;     // Next free Env
    envid_t env_id;         // Unique environment identifier
    envid_t env_parent_id;    // env_id of this env's parent
    enum EnvType env_type;    // Indicates special system environments
    unsigned env_status;      // Status of the environment
    uint32_t env_runs;     // Number of times environment has run

    // Address space
    pde_t *env_pgdir;      // Kernel virtual address of page dir
};
```

其中，当空闲 env 被放入 env_free_list 中时依靠 env_link 构建链表。

JOS 的 Environment 组成与 *nix 系统相似，由 **Thread** 和 **Address Space** 两部分概念组成：前者由 env_tf 中的寄存器描述（即进程切换时需保存的现场），后者由 env_pgdir 指向的页表目录描述。

注：*JOS 与 xv6 设计存在差异。JOS 采用的是 Single Kernel Stack 的设计，一次只能有一个进程陷入内核；而 xv6 的每个进程都拥有独立的内核栈（xv6 的进程由 struct proc 描述）。*

在上述 Env 结构体中，需要注意用于记录进程当前状态的 env_status 变量。在 JOS 的设计中进程总共有 5 种可能的状态：

Status	Annotation
ENV_FREE	表示进程尚未被分配，此时应处于 env_free_list 中
ENV_RUNNABLE	表示进程已被分配且可在下次进程切换时被调度
ENV_RUNNING	表示为当前执行中的进程
ENV_NOT_RUNNABLE	表示进程活跃但不可调度：可能在等待 IPC 等状态
ENV_DYING	Zombie 进程。详细内容将在 Lab4 展开

1.2 Allocating the Environments Array

Exercise 1

Modify mem_init() in kern/pmap.c to allocate and map the envs array. This array consists of exactly NENV instances of the Env structure allocated much like how you allocated the pages array. Also like the pages array, the memory backing envs should also be mapped user read-only at UENVS (defined in inc/memlayout.h) so user processes can read from this array.

You should run your code and make sure check_kern_pgdir() succeeds.

第一个练习要求我们为 envs 数组分配物理空间，并将其映射至内核地址空间。分配方法与 Lab2 中分配与页管理数组 pages 方式接近，我们根据 memorylayout.h 文件的提示，在 mem_init() 中增加如下代码：

```
//////////////////////////////////////////////////////////////////////
// Make 'envs' point to an array of size 'NENV' of 'struct Env'.
// LAB 3: Your code here.
envs = (struct Env *) boot_alloc(NENV * sizeof(struct Env));

...
//////////////////////////////////////////////////////////////////////
// Map the 'envs' array read-only by the user at linear address UENVS
// (ie. perm = PTE_U | PTE_P).
// Permissions:
//    - the new image at UENVS  -- kernel R, user R
//    - envs itself -- kernel RW, user NONE
// LAB 3: Your code here.
boot_map_region(kern_pgdir, UENVS, PTSIZE, PADDR(envs), PTE_U);
```

1.3 Creating and Running Environments

本节的内容是构建进程管理的 API，包括实现进程创建分配、加载可运行镜像的功能。

Exercise 2

In the file env.c, finish coding the following functions:

env_init()

env_setup_vm()

region_alloc()

load_icode()

env_create()

env_run()

As you write these functions, you might find the new cprintf verb %e useful – it prints a description corresponding to an error code. For example,

r = -E_NO_MEM;

panic("env_alloc: %e", r);

will panic with the message "env_alloc: out of memory".

env_init

env_init 函数是实现对已在 mem_init 中分配的 envs 数据进行初始化，并将未分配进程放入空闲列表中。与页管理的 pages 数组及其空闲列表的管理模式相同。需要注意的是，实验对进程放入空闲列表的顺序有要求，根据实验提示完成即可：

```
void
env_init(void)
{
    assert(envs != NULL);  // Make sure the envs is allocated successfully
    assert(env_free_list == NULL);
    env_free_list = envs;
    envs[0].env_id = 0;
    envs[0].env_link = NULL; // Not necessary
    envs[0].env_status = ENV_FREE;
    for(int i = 1; i < NENV; ++i)
    {
        envs[i].env_id = 0;
        envs[i-1].env_link = &envs[i];
        envs[i-1].env_status = ENV_FREE;
    }
    envs[NENV-1].env_link = NULL;
    env_init_percpu();
}
```

env_setup_vm

env_setup_vm 函数是在分配新进程时，设置进程的内核部分地址空间的函数。所有进程高位处（即内核地址映射）均是相同的，根据提示，可以使用已经构建完整映射的内核页目录 kern_pgdir 作为模板构建（可直接使用 memcpy，注意地址转换）。此外，根据实验提示，需将本页目录自身映射到地址空间的 UVPT 处。实现如下：

```
static int
env_setup_vm(struct Env *e)
{
    struct PageInfo *p = NULL;

    // Allocate a page for the page directory
    if (!(p = page_alloc(ALLOC_ZERO)))
        return -E_NO_MEM;

    // Now, set e->env_pgdir and initialize the page directory.
    //
    // Hint:
    //    - The VA space of all envs is identical above UTOP
    //   (except at UVPT, which we've set below).
    //   See inc/memlayout.h for permissions and layout.
    //   Can you use kern_pgdir as a template?  Hint: Yes.
    //   (Make sure you got the permissions right in Lab 2.)
    //    - The initial VA below UTOP is empty.
    //    - You do not need to make any more calls to page_alloc.
    //    - Note: In general, pp_ref is not maintained for
    //   physical pages mapped only above UTOP, but env_pgdir
    //   is an exception -- you need to increment env_pgdir's
    //   pp_ref for env_free to work correctly.
    //    - The functions in kern/pmap.h are handy.

    // LAB 3: Your code here.
    e->env_pgdir = page2kva(p);
    memcpy(e->env_pgdir, kern_pgdir, PGSIZE);

    // UVPT maps the env's own page table read-only.
    // Permissions: kernel R, user R
    e->env_pgdir[PDX(UVPT)] = PADDR(e->env_pgdir) | PTE_P | PTE_U;

    // increment the pp_ref
    p->pp_ref++;

    return 0;
}
```

region_alloc

当进程申请物理空间并请求将其映射到要求的虚拟地址时调用该函数。注意请求地址及请求内存长度可以是非页表大小对齐的，因此实际分配前需完成对齐。

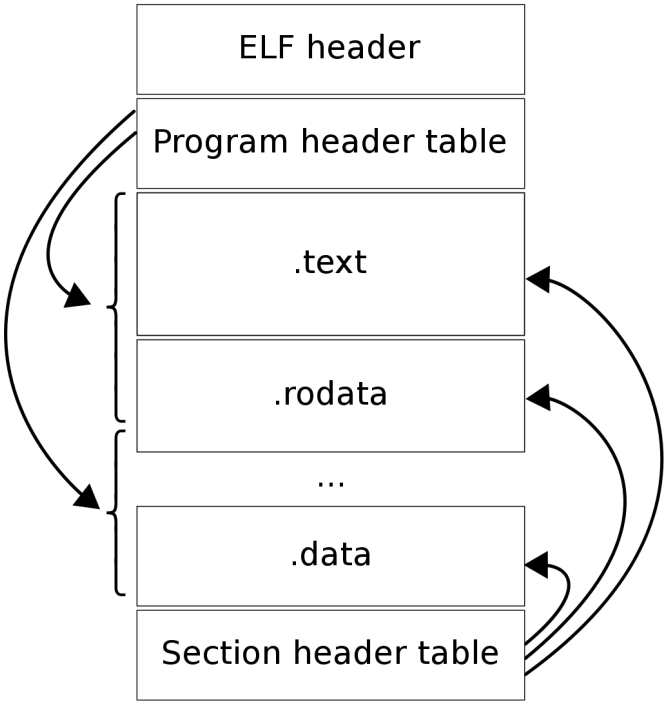
注：根据提示，该函数仅为 *load_icode* 映射 *ELF* 内容至进程虚拟空间调用

```
static void
region_alloc(struct Env *e, void *va, size_t len)
{
    // LAB 3: Your code here.
    // (But only if you need it for load_icode.)
    //
    // Hint: It is easier to use region_alloc if the caller can pass
    //   'va' and 'len' values that are not page-aligned.
    //   You should round va down, and round (va + len) up.
    //   (Watch out for corner-cases!)
    uintptr_t start = ROUNDDOWN((uintptr_t)(va), PGSIZE);
    uintptr_t end = ROUNDUP((uintptr_t)(va) + len, PGSIZE);

    if(!e)
        panic("region_alloc: env is NULL.\n");

    while(start < end)
    {
        struct PageInfo *p = page_alloc(0);
        if(!p)
            panic("region_alloc: page_alloc failed.\n");
        page_insert(e->env_pgdir, p, (void *)start, PTE_U | PTE_W);
        start += PGSIZE;
    }

}
```

load_icode

load_icode() 函数实现从 ELF 文件中加载信息，即实现 exec 的核心内容。

ELF 格式的可执行文件 的文件结构如下图所示，其中与文件内容到运行时虚拟内存空间映射相关的是 Program header table 部分：它提供了每个部分到虚拟内存空间的映射关系。Program header table 在 ELF 文件中位置由 e_phoff 给出，条目数目由 e_phnum 给出。



代码中还有几点需要注意：(1) 由于映射时需使用 memcpy/memset 函数，两个函数在地址翻译时使用的是当前 CR3 寄存器所指定的页表，因此需临时切换至当前进程的页目录，完成映射后再切回内核页目录；(2) 注意到 ELF 镜像通过一个指针传递，我们可以查看 load_icode 的调用链，发现其由 env_create 负责调用，而 env_create 调用时使用 env.h 中定义的宏 ENV_PASTE3，将调用的位置设置为 _binary_obj_[NAME]_start，其中 NAME 即为二进制程序的名字。关于这一系列变量的解释在 Lab 中也有给出：这是一种将二进制可执行文件嵌入内核代码的一种方式。

具体地，在 kern/Makeflag 我们可以看到内核链接部分的 flags 中有 *-b binary* 选项，这个选项意味着在链接过程中将文件按 raw-format （理论上可以是任意二进制格式）链接而不按.o 形式解析，被原封不动嵌入可执行文件中。此外，链接器还会为这些嵌入的二进制文件引入符号注明其位置，即上文提到的 _binary_obj_[NAME]_start 的形式：JOS 使用 nm（打印二进制文件中的符号表）命令将其输出到了 kernel.sym 文件中以便使用者查阅，我们也可以看到这些符号。因此，当内核被加载后，这些被嵌入的 raw-ELF 也被加载到内存中，并且其位置由上述链接器提供的全局符号标志，故 load_icode （以及后文出现的 env_create()）可以直接使用这些全局符号找到这些嵌入的 ELF。

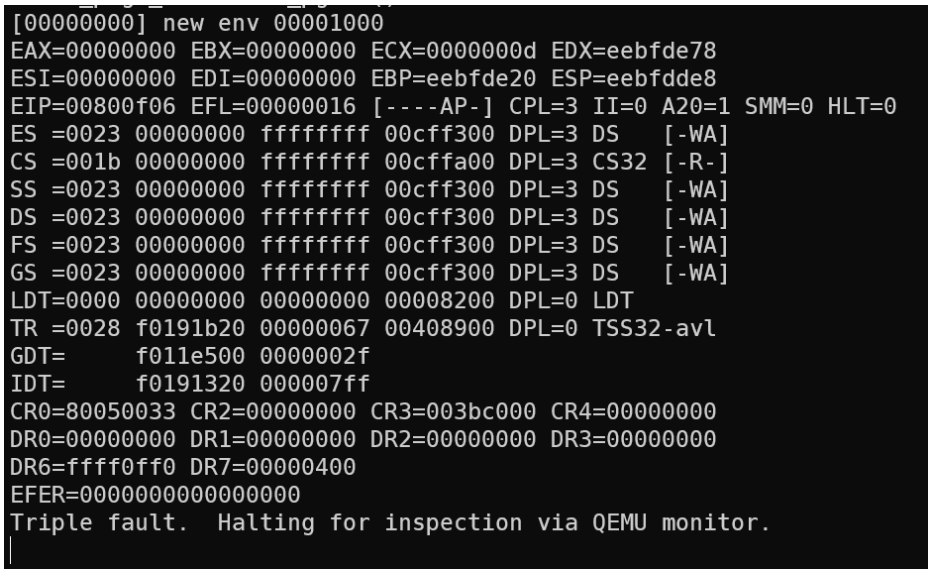
```
static void
load_icode(struct Env *e, uint8_t *binary)
{
    // LAB 3: Your code here.
    struct Elf *elf = (struct Elf *)binary;
    if(elf->e_magic != ELF_MAGIC)
        panic("load_icode: e_magic != ELF_MAGIC.\n");

    struct Proghdr *pht = (struct Proghdr *)(binary + elf->e_phoff);

    lcr3(PADDR(e->env_pgdir)); // for valid-use of memcpy/memset
    for(int i = 0; i < elf->e_phnum; ++i)
    {
        if(pht[i].p_type == ELF_PROG_LOAD)
        {
            // Allocate physical regions
            region_alloc(e, (void *)pht[i].p_va, pht[i].p_memsz);
            // Copy the contents from ELF
            memcpy(
                (void *)pht[i].p_va,
                (void *)(binary) + pht[i].p_offset,
                pht[i].p_filesz
            );
            // Set the rest to be zero
            if(pht[i].p_filesz < pht[i].p_memsz)
                memset(
                    (void *)pht[i].p_va + pht[i].p_filesz,
                    0,
                    pht[i].p_memsz - pht[i].p_filesz
                );

        }
    }
    lcr3(PADDR(kern_pgdir));

    // Setup the entry
    e->env_tf.tf_eip = elf->e_entry;

    // Now map one page for the program's initial stack
    // at virtual address USTACKTOP - PGSIZE.
    // LAB 3: Your code here.
    region_alloc(e, (void *)(USTACKTOP-PGSIZE), PGSIZE);
}
```

env_create & env_run

env_create 用于创建新进程并载入二进制程序，env_run 功能则为切换当前执行的进程。代码实现如下：

```
void
env_create(uint8_t *binary, enum EnvType type)
{
    // LAB 3: Your code here.
    struct Env *new_env = NULL;
    if(env_alloc(&new_env, 0) < 0)
        panic("env_crate: env_alloc failed.\n");
    new_env->env_type = type;
    load_icode(new_env, binary);
}

void
env_run(struct Env *e)
{
   // Step 1: If this is a context switch (a new environment is running):
   //    1. Set the current environment (if any) back to
   //       ENV_RUNNABLE if it is ENV_RUNNING (think about
   //       what other states it can be in),
   //    2. Set 'curenv' to the new environment,
   //    3. Set its status to ENV_RUNNING,
   //    4. Update its 'env_runs' counter,
   //    5. Use lcr3() to switch to its address space.
   // Step 2: Use env_pop_tf() to restore the environment's
   //    registers and drop into user mode in the
   //    environment.
    // ...
   // LAB 3: Your code here.
   if(curenv)
   {
      if(curenv->env_status == ENV_RUNNING)
         curenv->env_status = ENV_RUNNABLE;
   }
   curenv = e;
   curenv->env_status = ENV_RUNNING;
   curenv->env_runs++;
   // !Use physic addr of pgdir
   lcr3(PADDR(curenv->env_pgdir));
   env_pop_tf(&(curenv->env_tf));
}
```

在上述部分代码完成后，Lab 让我们实现一个调试：首先在 env_pop_tf 处设置断点后执行，通过查阅 kernel 源码我们知道当前内核启动后会创建第一个用户进程 hello 并使用 env_run 运行它，env_pop_tf 就发生在这次调用中。当 env_pop_tf 完成后，执行流进入用户进程的运行状态。此时 Lab 要求我们在 hello 进程中的 int $0x30 处打断点并执行，再单步执行后可发现内核发生 fault。



查看 hello.c 与 hello.asm 并结合内核打印的信息我们可以得知该 fault 在打印”hello world!”语句时发生；检查调用链可知，vcprintf 在调用 sys_cputs() 时产生了一个 syscall，该 syscall 调用出现问题的 int $0x30 指令。由此我们基本可以确认，这个 fault 是由于缺少处理 syscall 的中断处理程序而产生的。

此处我们还需厘清一个问题是：JOS 是在哪里完成对用户态/内核态的区分的？也即，我们分配的进程在哪里被设置为用户态程序？

我们知道在 x86 中，一段内容处于 Ring x 记录在当前 CS 寄存器中的 CPL 位，这一位通常与该段内容所处段的段选择子 DPL 位相等，因此设置一段内容的 Ring x 权限，必然涉及 GDT 的设置。我们使用 IDE 的搜索功能查询 GDT 相关关键字，很快就能找到相关代码：env_init_percpu()：

```
// Load GDT and segment descriptors.
void
env_init_percpu(void)
{
    lgdt(&gdt_pd);
    // The kernel never uses GS or FS, so we leave those set to
    // the user data segment.
    asm volatile("movw %%ax,%%gs" : : "a" (GD_UD|3));
    asm volatile("movw %%ax,%%fs" : : "a" (GD_UD|3));
    // The kernel does use ES, DS, and SS.  We'll change between
    // the kernel and user data segments as needed.
    asm volatile("movw %%ax,%%es" : : "a" (GD_KD));
    asm volatile("movw %%ax,%%ds" : : "a" (GD_KD));
    asm volatile("movw %%ax,%%ss" : : "a" (GD_KD));
    // Load the kernel text segment into CS.
    asm volatile("ljmp %0,$1f\n 1:\n" : : "i" (GD_KT));
    // For good measure, clear the local descriptor table (LDT),
    // since we don't use it.
    lldt(0);
}
```

这段代码使用 gdt_pd 变量的内容更新了当前的 GDT，同时设置了当前内核代码的 CS，ES，DS，SS 寄存器为 GD_KD 段选择子的位置。我们查看 gdt_pd 的内容：

```
struct Segdesc gdt[] =
{
    // 0x0 - unused (always faults -- for trapping NULL far pointers)
    SEG_NULL,
    // 0x8 - kernel code segment
    [GD_KT >> 3] = SEG(STA_X | STA_R, 0x0, 0xffffffff, 0),
    // 0x10 - kernel data segment
    [GD_KD >> 3] = SEG(STA_W, 0x0, 0xffffffff, 0),
    // 0x18 - user code segment
    [GD_UT >> 3] = SEG(STA_X | STA_R, 0x0, 0xffffffff, 3),
    // 0x20 - user data segment
    [GD_UD >> 3] = SEG(STA_W, 0x0, 0xffffffff, 3),
    // 0x28 - tss, initialized in trap_init_percpu()
    [GD_TSS0 >> 3] = SEG_NULL
};

struct Pseudodesc gdt_pd = {
    sizeof(gdt) - 1, (unsigned long) gdt
};
```

从上述代码我们就可以看到，新的 GDT 仍然采用 flat-pattern 的方式来忽略分段机制，但利用了段选择子的权限位设置实现内核态和用户态的权限区分（内核段为 Ring0，代码段为 Ring3）。所有用户态进程均有内核分配及初始化，因此内核的权限设置工作应当也由内核代码实现。同样进行搜索后发现，在分配进程的函数 env_alloc() 中就进行了相应设置，由此我们便可以理解操作系统如何提供运行在 Ring3 的用户态进程了：

```
int
env_alloc(struct Env **newenv_store, envid_t parent_id)
{

...
    // Set up appropriate initial values for the segment registers.
   // GD_UD is the user data segment selector in the GDT, and
   // GD_UT is the user text segment selector (see inc/memlayout.h).
   // The low 2 bits of each segment register contains the
   // Requestor Privilege Level (RPL); 3 means user mode.  When
   // we switch privilege levels, the hardware does various
   // checks involving the RPL and the Descriptor Privilege Level
   // (DPL) stored in the descriptors themselves.
   e->env_tf.tf_ds = GD_UD | 3;
   e->env_tf.tf_es = GD_UD | 3;
   e->env_tf.tf_ss = GD_UD | 3;
   e->env_tf.tf_esp = USTACKTOP;
   e->env_tf.tf_cs = GD_UT | 3;
...

}
```

1.4 Handling Interrupts and Exceptions

Exercise 3

Read Chapter 9, Exceptions and Interrupts in the 80386 Programmer's Manual (or Chapter 5 of the IA-32 Developer's Manual), if you haven't already.

Interrupts & Exceptions

通常情况下，**外部中断（External Interrupts）**可以被分为两类：中断（Interrupts）和异常（Exceptions）。其中两者可以被进一步细分。

- **中断（Interrupts）**：分为可屏蔽中断（Maskable Interrupts）与不可屏蔽中断（Nonmaskable Interrupts），前者由 CPU 的 INTR 引脚发出信号产生，后者由 NMI 引脚产生。一般来说不可屏蔽中断都由致命错误引发，通常的处理方式不是为其设置 Handler，而是直接发出错误警告终止程序。
  一般认为中断信号是异步信号，由 CPU 外设备产生。
- **异常（Exceptions）**：异常通常分为 (1) 由 CPU 自行检测到的异常，会被进一步分类为 trap/fault/abort，例如 divide zero 异常；(2) 还有部分信号是可编程的，例如 INT 0x3 指令触发，也被称为软中断（Software Interrupts）。
  与中断相对的，异常信号通常为同步信号，即由当前 CPU 运行中的代码产生的。

Enabling and Disabling Interrupts

中断与异常屏蔽方式有以下几种：

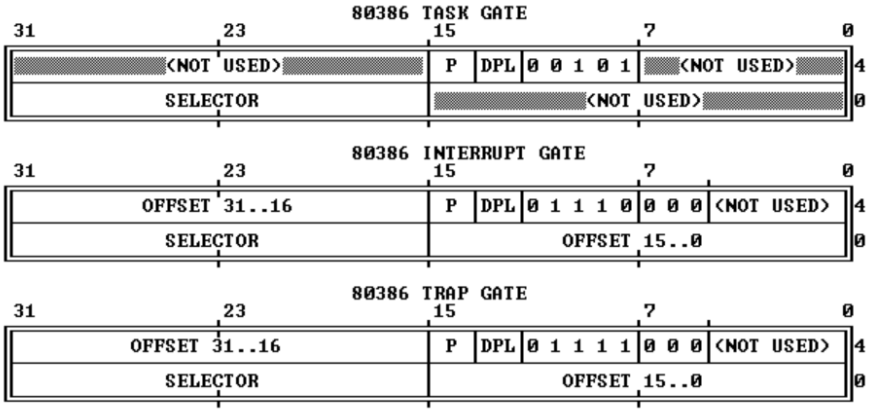
- **NMI Masks Further NMIs**：
  当一个 NMI 处理程序进行中时，其他的 NMI 会被暂时屏蔽，直到处理程序执行结束。
- **IF Masks INTR**：
  通过设置 IF(interrupt-enable flag) 可以开启/关闭可屏蔽中断。x86 提供 CLI/STI 指令设置该位（bootloader.S 中使用过该指令）。
- **RF Masks Debug Faults**：
  设置 EFLAGS 中的 RF 位可以开启/关闭 debug fault。
- **MOV or POP to SS Masks Some Interrupts and Exceptions**：
  这是一类应对不正常行为的中断屏蔽。如果在修改 SS 寄存器与 ESP 寄存器之间产生了中断，会产生 SS:ESP 的不一致性，可能影响中断处理程序的行为。因此，在修改 SS 寄存器时（MOV/POP）80386 架构会禁止 NMI、INTR 及异常中断，仅有 page fault 与 general protection fault 会发生。改为使用 LSS 指令可以避开这个问题处理。

1.5 Basics of Protected Control Transfer

为实现从用户态到内核态切换的同时保证对用户态可执行内容的限制，x86 提供了两种机制：**中断向量表（Interrupt Descriptor Table, IDT）**及**任务状态段（Task State Segment, TSS）**。

Interrupt Description Table (IDT)

IDT 的形式与 GDT 类似，地址存储在专门的寄存器 IDTR 中。IDT 提供一个 256 entries 的表记录处理不同类型中断/异常的函数入口，当异常发生时调用对应的函数（加载 handler 的 EIP/CS）。IDT 中包含三种不同的表项，如下图所示：



Task State Segment (TSS)

当调用中断处理程序时，需要我们保存当前执行流的现场以便完成异常处理后恢复执行。而且应当注意的是，执行流现场必须保存到高特权级的位置，否则可能遭到其他用户态带 bug 或恶意的代码的影响。

当发生中断/异常且 handler 向更高特权级转换时，x86 会切换当前使用的栈到高特权级栈（内核栈）。同时，x86 定义了 TSS 作为保存现场的数据结构，发生异常时将记录执行流相关信息的 TSS 压入内核栈中。

TSS 结构保存着大量信息，JOS 仅使用 ESP0 及 SS0 两个域，用于指定切换的内核栈位置。TSS 的地址被保存在 TR 寄存器中，可以通过 LTR/STR 指令进行操作。JOS 在 trap_init_percpu() 中执行了 TSS 段的初始化加载。

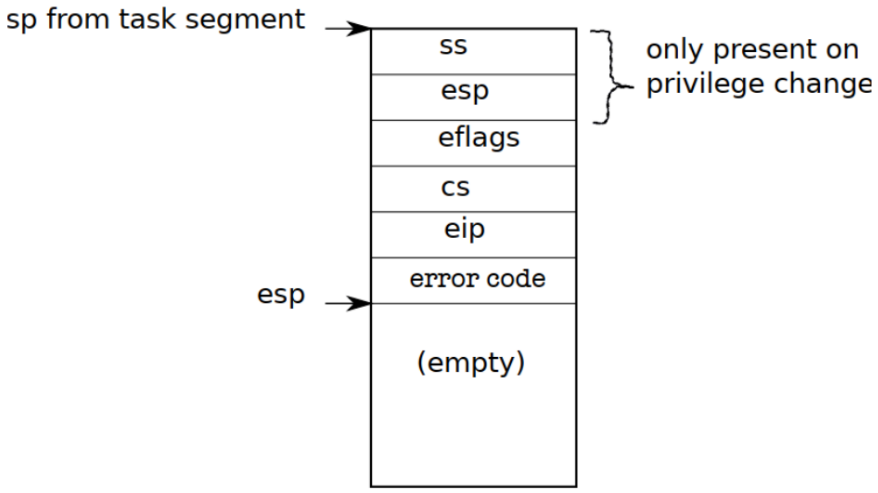
1.6 Types of Exceptions and Interrupts

在 x86 中，所有的（同步）异常被分配到 0-31 号中断向量上，即 IDT 的前 32 个 entries，例如 page-fault 就使用 14 号中断向量。

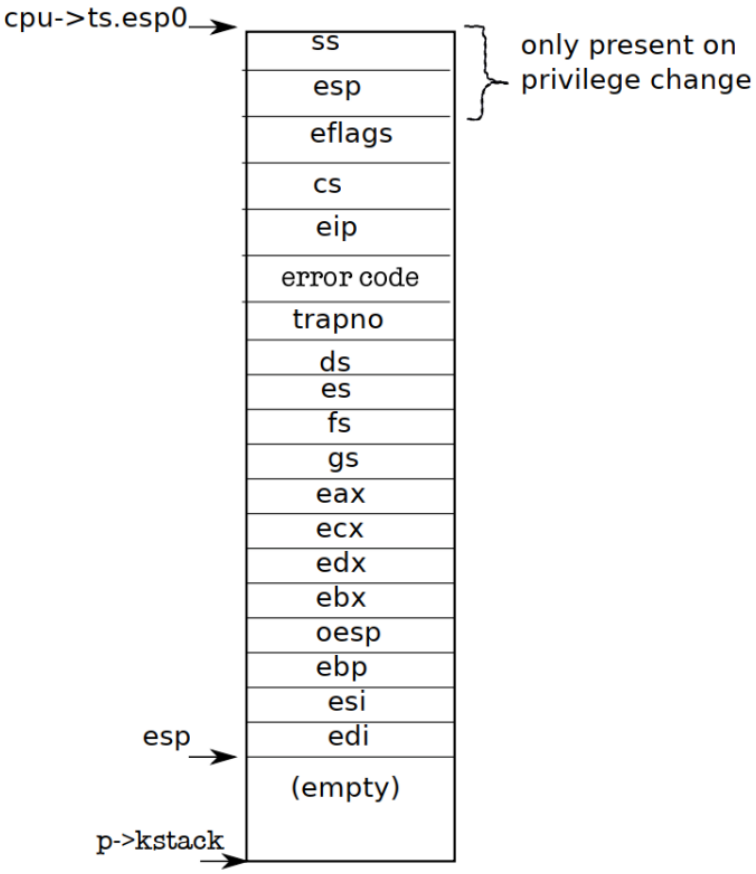
大于 31 号的中断向量被用于处理软件中断，可能的来源是 int 指令发送的中断及外部设备中断等异步中断。在 Lab3 中，我们会实现 JOS 对 0-31 号所有同步异常处理入口的设置以及系统调用中断的设置（JOS 使用了中断号 0x30，这个选择是可任意的）；在 Lab4 中会进一步拓展至可以处理时钟中断等外部中断。x86 中发射中断向量的指令是int n 指令，它完成以下任务：

- 根据索引 n 在 IDT 中找到处理函数的入口；
- 检查 CS 寄存器的权限位（CPL）；
- 将当前 ESP 寄存器及 SS 寄存器存放入 CPU 内部空闲寄存器；
- 从 TSS 中加载内核栈 的 ESP 寄存器及 SS 寄存器；
- 将原有的 ESP 寄存器、SS 寄存器（两者当前暂时存放于 CPU 其他寄存器中）等入内核栈；
- 将 EFLAGS 寄存器、当前进程的 CS 寄存器及 IP 寄存器入（内核栈）；
- 将 CPU 控制交给处理函数（处理函数负责调用 iret 命令从中断处理中返回）。

到这里，int 指令已经完成了填充内核栈信息的一部分工作，内核栈内容如下图所示：



为进一步填充完整信息（形成 TrapFrame），trap 入口函数提供进一步操作，如下图所示。JOS 的做法是在每个入口函数将 trapno 入栈后，统一进入 _alltrap 函数处理（Exercise 4 内容）。



1.7 Nested Exceptions and Interrupts

当已经处于内核态时，也可以继续嵌套处理中断和异常，此时会在内核栈中继续装填处理帧，唯一不同的是不会再将 SS 和 ESP 寄存器入栈。

1.8 Setting Up the IDT

Exercise 4

Edit trapentry.S and trap.c and implement the features described above. The macros **TRAPHANDLER** and **TRAPHANDLER_NOEC** in trapentry.S should help you, as well as the T_ defines in inc/trap.h. You will need to add an entry point in trapentry.S (using those macros) for each trap defined in inc/trap.h, and you'll have to provide _alltraps which the TRAPHANDLER macros refer to. You will also need to modify trap_init() to initialize the idt to point to each of these entry points defined in trapentry.S; the **SETGATE** macro will be helpful here.

Test your trap handling code using some of the test programs in the user directory that cause exceptions before making any system calls, such as user/divzero. You should be able to get make grade to succeed on the divzero, softint, and badsegment tests at this point.

Part A 最后一部分要求修改 trapentry.S 及 trap.c 文件，对 IDT 进行初始化。trapentry.S 提供了两个宏：TRAPHANDLER 与 TRAPHANDLER_NOEC 用于生成处理函数入口（与 xv6 中的 vectors.pl 功能相近）。设置代码如下：

```
/*
* Lab 3: Your code here for generating entry points for the different traps.
*/
TRAPHANDLER_NOEC(DIVIDE, T_DIVIDE)
TRAPHANDLER_NOEC(DEBUG, T_DEBUG)
TRAPHANDLER_NOEC(NMI, T_NMI)
TRAPHANDLER_NOEC(BRKPT, T_BRKPT)
TRAPHANDLER_NOEC(OFLOW, T_OFLOW)
TRAPHANDLER_NOEC(BOUND, T_BOUND)
TRAPHANDLER_NOEC(ILLOP, T_ILLOP)
TRAPHANDLER_NOEC(DEVICE, T_DEVICE)
TRAPHANDLER(DBLFLT, T_DBLFLT)
TRAPHANDLER(TSS, T_TSS)
TRAPHANDLER(SEGNP, T_SEGNP)
TRAPHANDLER(STACK, T_STACK)
TRAPHANDLER(GPFLT, T_GPFLT)
TRAPHANDLER(PGFLT, T_PGFLT)
TRAPHANDLER_NOEC(FPERR, T_FPERR)
TRAPHANDLER(ALIGN, T_ALIGN)
TRAPHANDLER_NOEC(MCHK, T_MCHK)
TRAPHANDLER_NOEC(SIMDERR, T_SIMDERR)
TRAPHANDLER_NOEC(SYSCALL, T_SYSCALL)
TRAPHANDLER_NOEC(DEFAULT, T_DEFAULT)
```

观察两个宏的定义，发现设置后的入口均会跳转至 __alltraps 入口进行通用处理。这是所有处理函数的通用入口，它填充内核栈使其具有完整的 Trapframe 结构，并根据 Lab 提示加载 DS、ES 寄存器，并将 ESP 寄存器入栈。参考 xv6 源码，对 __alltraps 补充如下：

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

观察可知，这段代码最终跳转至 trap.c/trap()，由该函数进行分发与中断处理。参考 xv6，完成 trap.c/trap_init() 函数：

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

    // LAB 3: Your code here.
    extern void DIVIDE();
    extern void DEBUG();
    ...
    extern void SYSCALL();
    extern void DEFAULT();

    SETGATE(idt[T_DIVIDE], 0, GD_KT, DIVIDE, 0);
    SETGATE(idt[T_DEBUG], 0, GD_KT, DEBUG, 0);
    SETGATE(idt[T_NMI], 0, GD_KT, NMI, 0);
    SETGATE(idt[T_BRKPT], 0, GD_KT, BRKPT, 0);
    SETGATE(idt[T_OFLOW], 0, GD_KT, OFLOW, 0);
    SETGATE(idt[T_BOUND], 0, GD_KT, BOUND, 0);
    SETGATE(idt[T_ILLOP], 0, GD_KT, ILLOP, 0);
    SETGATE(idt[T_DEVICE], 0, GD_KT, DEVICE, 0);
    SETGATE(idt[T_DBLFLT], 0, GD_KT, DBLFLT, 0);
    SETGATE(idt[T_TSS], 0, GD_KT, TSS, 0);
    SETGATE(idt[T_SEGNP], 0, GD_KT, SEGNP, 0);
    SETGATE(idt[T_STACK], 0, GD_KT, STACK, 0);
    SETGATE(idt[T_GPFLT], 0, GD_KT, GPFLT, 0);
    SETGATE(idt[T_PGFLT], 0, GD_KT, PGFLT, 0);
    SETGATE(idt[T_FPERR], 0, GD_KT, FPERR, 0);
    SETGATE(idt[T_ALIGN], 0, GD_KT, ALIGN, 0);
    SETGATE(idt[T_MCHK], 0, GD_KT, MCHK, 0);
    SETGATE(idt[T_SIMDERR], 0, GD_KT, SIMDERR, 0);
    SETGATE(idt[T_SYSCALL], 1, GD_KT, SYSCALL, 3);
    SETGATE(idt[T_DEFAULT], 0, GD_KT, DEFAULT, 0);
    // Per-CPU setup
    trap_init_percpu();
}
```