MIT6.828 Lab3: User Environment

Zhuofan Zhang

June 2020

Contents

Chapter 1

User Environment and Exceptions Handling

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，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）。*

1.2 Allocating the Environments Array

JOS 管理 Environments 的方式与管理 Pages 相同，维护一个数组 struct Env *envs 。

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.

本题要求完成对 envs 的内存映射（修改 pmap.c/mem_init() ）。具体地，与 Lab2 中对 Pages 的处理相同，为该数组分配物理空间，并将其映射到内核 VAS 中：根据 memlayout.h 中对 ENVS 段的虚拟地址及其宏定义（UENVS），得到结果如下。应注意到，为 envs 分配物理空间发生于完成内核 VAS 映射前。

```
// LAB 3: Your code here.
extern struct Env* envs;   // In env.c
envs = (struct Env*) boot_alloc(NENV*sizeof(struct Env));
memset(envs, 0, NENV*sizeof(struct Env));

...

// LAB 3: Your code here.
boot_map_region(kern_pgdir, UENVS,
                ROUNDUP(NENV * sizeof(struct Env), PGSIZE),
                PADDR(envs), PTE_U);
```

1.3 Creating and Running Environments

映射完成后，需要对 Environments 进行初始化使其可以运行。在当前实验阶段 JOS 还没有文件系统，因此实验**从镜像中读取进程状态**。这些镜像被嵌在 JOS 内核中，以 **ELF** 形式保存。根据实验提示，观察 kern/Makeflag：

```
# Binary program images to embed within the kernel.
# Binary files for LAB3
KERN_BINFILES :=  user/hello \
                  user/buggyhello \
                  user/buggyhello2 \
                  user/evilhello \
                  user/testbss \
                  user/divzero \
                  user/breakpoint \
                  user/softint \
                  user/badsegment \
                  user/faultread \
```

```
                user/faultreadkernel \
                user/faultwrite \
                user/faultwritekernel
```

这些二进制文件像.o 文件一样被直接“链接”到内核中。再观察 obj/kern/kernel.sym （kernel 的符号表），可以发现链接器会赋予这些二进制文件一些相对复杂的符号供内核使用：

```
# Binary program images to embed within the kernel.
# Binary files for LAB3
0000e844 A _binary_obj_user_hello_size
0000e846 A _binary_obj_user_softint_size
0000e848 A _binary_obj_user_evilhello_size
...
```

Exercise 2

Finish coding the env_init(), env_setup_vm(), region_alloc(), load_icode(), env_create(), env_run().

env_init() 函数在 init.c 中被调用，将前述分配的 envs 数组进行 0 初始化并放入空闲链表 env_free_list 中（与 Lab2 中的 mem_init() 相似）并初始化每个 CPU 的配置。

```
void env_init(void)
{
  // Set up envs array
  // LAB 3: Your code here.
  int i = 0;
  for(; i < NENV - 1; ++i)
  {
    envs[i].env_link = &(envs[i+1]);
    envs[i].env_status = ENV_FREE;
    envs[i].env_id = 0;
  }
  envs[i].env_link = NULL;
  envs[i].env_status = ENV_FREE;
  envs[i].env_id = 0;
  env_free_list = envs;
  // Per-CPU part of the initialization
    // Configures the segmentation hardware with
    // separate segments for privilege level 0(kernel)
    // and privilege level 3(user)
  env_init_percpu();
}
```

env_setup_vm() 在创建新进程时被调用，用于为新进程构建地址空间（进程页表目录）并初始化。由于每个进程内核段的映射与内核 VAS 相同，可以直接复制内核部分映射。除此之外，还需映射用于保存页表目录自身地址的 UVPT 段（JOS 使用这段映射来访问页表及页目录，具体见 Lab4 Exercise 12）。

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

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

  // LAB 3: Your code here.
  p->pp_ref++;
  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;

  return 0;
}
```

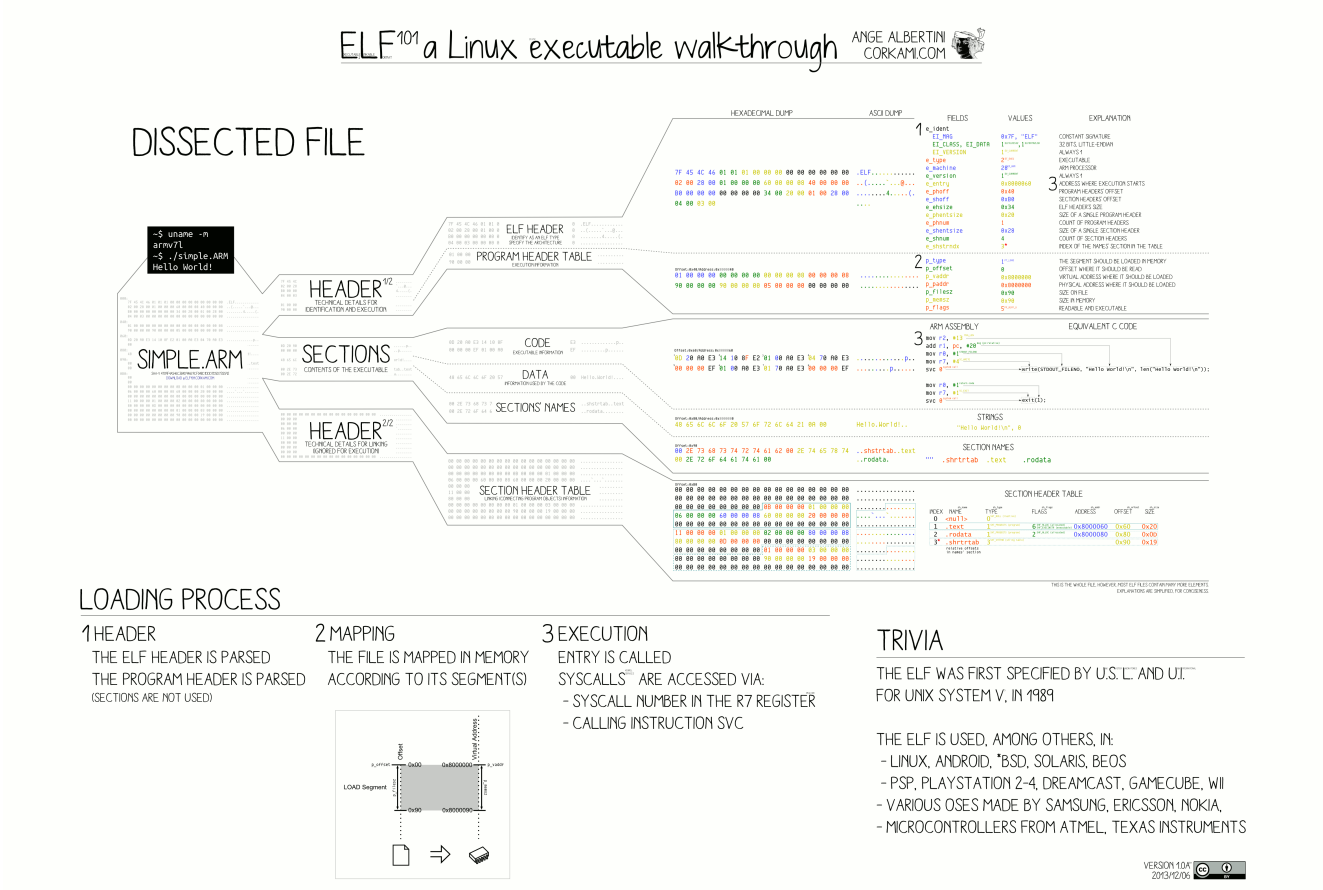
region_alloc() 通过调用页表层的抽象结构，实现申请并映射新物理空间的功能。

```
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!)
  void* start = ROUNDDOWN(va, PGSIZE);
  void* last = (void *)ROUNDUP(va+len, PGSIZE);
  struct PageInfo* pp;

  if(e == NULL)
    panic("The struct e could not be null.");
```

```
  while(start < last)
  {
    if(!(pp = page_alloc(1)))
      panic("region_alloc: page alloc failed.");
    if(page_insert(e->env_pgdir, pp, start, PTE_U | PTE_W))
      panic("region_alloc: page mapped failed.");
    start += PGSIZE;
  }
}
```

load_icode() 函数用于从 ELF 形式的镜像中读取内容，加载到对应的进程中。当创建进程时（根据 env.c，创建进程由 env_alloc() 完成，其调用前面完成的 env_setup_vm()，因此进程内核映射已经设置完成），该函数将相应的内容加载映射到 user 部分。我们参考 boot/main.c/bootmain() 读取 ELF 文件的方法。



从 load_icode() 的接口设计，可认为 ELF 文件已被读取到内存。函数从 ELF 文件的 *Programming Header* 中获取信息，找到对应内容，并利用 region_alloc() 完成内存分配，将内容复制进新分配的地址。需要注意的是，如果使用 memset/memcpy 进行复制，需手动将当前页表目录切换为处理进程的页表目录，并在完成读取后恢复。

```
static void
load_icode(struct Env *e, uint8_t *binary)
{
  // LAB 3: Your code here.
  struct Proghdr* ph, *eph;
  struct Elf* elf;
  elf = (struct Elf*) binary;
  if(elf->e_magic != ELF_MAGIC)
    panic("load_icode error: ELF file not valid.");

  ph = (struct Proghdr *) ((uint8_t *) elf + elf->e_phoff);
  eph = ph + elf->e_phnum;

  lcr3(PADDR(e->env_pgdir));

  uint32_t va;
  for(; ph < eph; ph++)
  {
    if(ph->p_type != ELF_PROG_LOAD)
      continue;
    va = (uint32_t)binary + ph->p_offset;
    region_alloc(e, (void*)ph->p_va, ph->p_memsz);
    memset((void*)ph->p_va, 0, ph->p_memsz);
    memcpy((void*)ph->p_va, (void*)va, ph->p_filesz);

  }

  // 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);
  lcr3(PADDR(kern_pgdir));
  e->env_tf.tf_eip = elf->e_entry;
}
```

最后需为进程 VAS 的 [USTACKTOP - PGSIZE, USTACKTOP) 段分配空间（根据 memlayout.h，这一段地址为 **User Exception Stack**），并设置 PC 为 ELF 中的程序入口地址。

env_create() 用于创建进程。根据接口设计，它调用底层的 load_icode() 。

```
void
env_create(uint8_t *binary, enum EnvType type)
{
  // LAB 3: Your code here.
  struct Env* e;
  if(env_alloc(&e, 0) < 0)
```

```
    panic("Env create error: env_alloc failed.");
  load_icode(e, binary);
  e->env_type = type;
}
```

env_run() 用于切换当前运行中的进程。env_pop_tf() 函数用于将 trapframe 中的保存的寄存器写回，同时返回至 *user mode* （说明该函数应该为陷入 *kernel mode* 时运行）。

```
void
env_run(struct Env *e)
{
  // LAB 3: Your code here.
    // Noted that without the first condition
    // (the curenv is not NULL) you can't pass
    // 'make grade' test.
  if(curenv && curenv->env_status == ENV_RUNNING)
    curenv->env_status = ENV_RUNNABLE;
  curenv = e;
  curenv->env_status = ENV_RUNNING;
  curenv->env_runs++;
  lcr3(PADDR(curenv->env_pgdir));

  env_pop_tf(&(curenv->env_tf));
  // panic("env_run not yet implemented");
}
```

1.4 Handling Interrupts and Exceptions

Part A 后半部分的内容涉及中断和异常的处理。按照作业提示，在 hello.asm 中找到引发程序停止的位置 int $0x30 ，这个错误是由于 int 指令发送中断向量后没有对应的处理程序 (**handler**) 所致。

（注：在该 *Lab* 中，*interrupt*，*trap*，*exception* 和 *fault* 的细微差别不被考虑，视作等价概念）

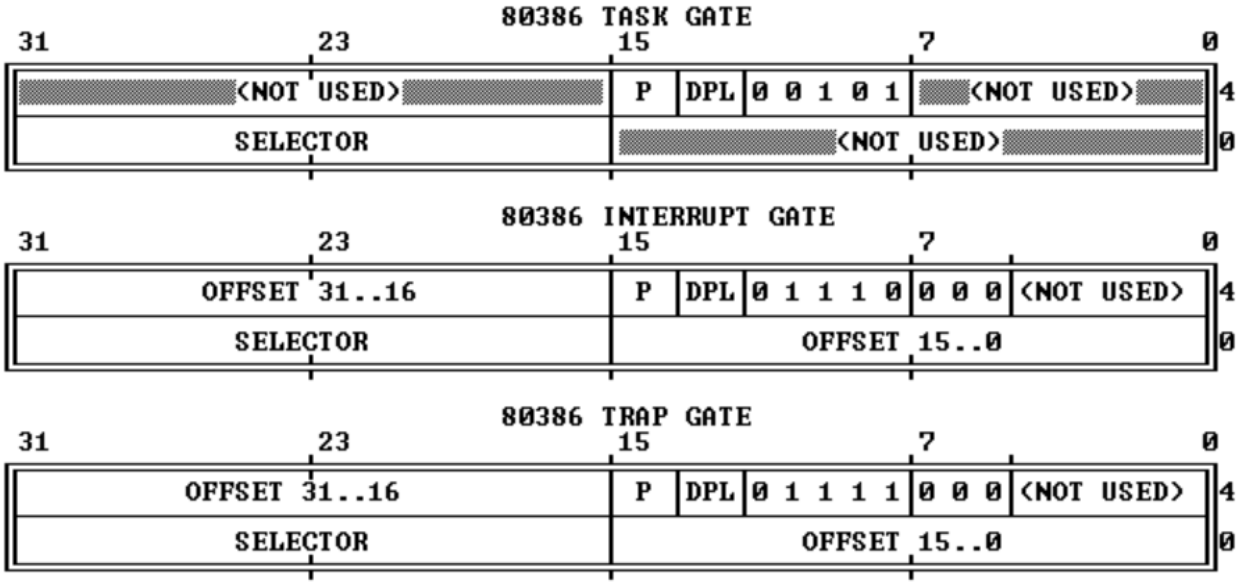
Exercise 3

阅读 *Manual Chapter 9*，详细笔记见 *1.5* 节与 *1.6* 节。

1.5 Basics of Protected Control Transfer

异常（Exception）和中断（Interrupt）都属于保护控制转移（Protect Control Transfer），根据 Intel 的定义，前者是指由当前运行的进程引起的同步（synchronous）信号，也称为**内部中断**（一般被设计为不可屏蔽的）；而后者指外部设备引发的异步（Asynchronous）信号，例如硬盘等 I/O 设备发送的信号，也称**外部中断**。

x86 提供两种机制提供这种保护：**中断向量表（Interrupt Descriptor Table, IDT）**及**任务状态段（Task State Segment, TSS）**。IDT 设置对应不同中断向量（整形值）的处理函数入口，Entry 如下图中 (2) 所示。



x86 最多接受 256 个中断/异常。CPU 以 IDT 为索引找到对应的处理函数，将处理函数程序加载到 CS 寄存器（程序段寄存器）及 EIP 寄存器（即 PC 寄存器），并将特权级加载至 SS 寄存器。JOS 通过 *mmu.h* 中的 SETGATE 对 IDT 插入表项。

x86 提供了一个 TSS 结构，一般它向 OS 提供 CPU 寄存器状态、IO 端口权限，且通常也保存 Inner-Level 的栈指针。一般而言，每个 CPU 维护自己的 TSS。它可以被存放在内存的任意位置，由 GDT 保存其入口，而 x86 提供 TR 寄存器保存该 GDT 项的段选择子（在初始化 GDT 项后，使用 *ltr* 将段选择子加载到 TR 中）。

JOS 只使用 ESP0 和 SS0 段（在 trap.c/trap_init_percpu() 中初始化），用于切换到内核栈。

1.6 Types of Exceptions and Interrupts

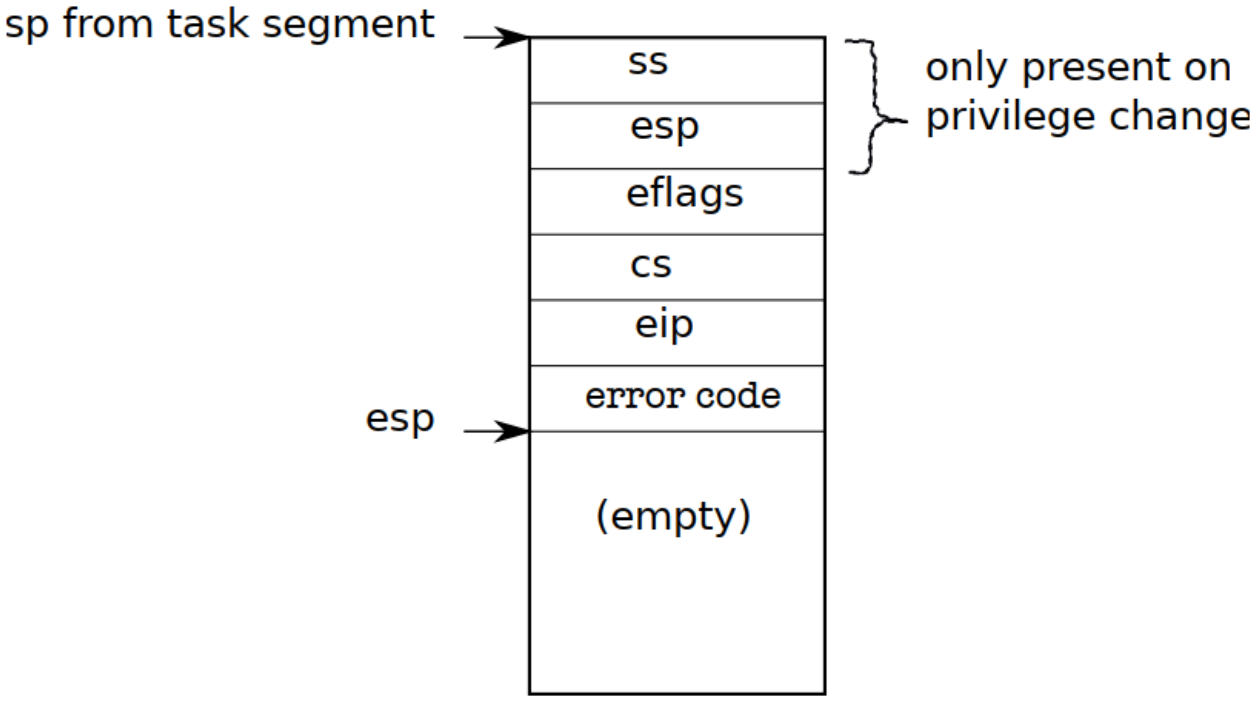
x86 的 0-31 中断向量用于映射已定义的硬件中断，大于 31 的中断向量只能被软件中断或异步中断使用。Part A 的最后一个任务是使 JOS 正常处理 0-31 号中断，在 Part B 会尝试处理软件中断。x86 中发射中断向量的指令是 int n 指令，它完成以下任务：

根据索引 n 在 IDT 中找到处理函数的入口；

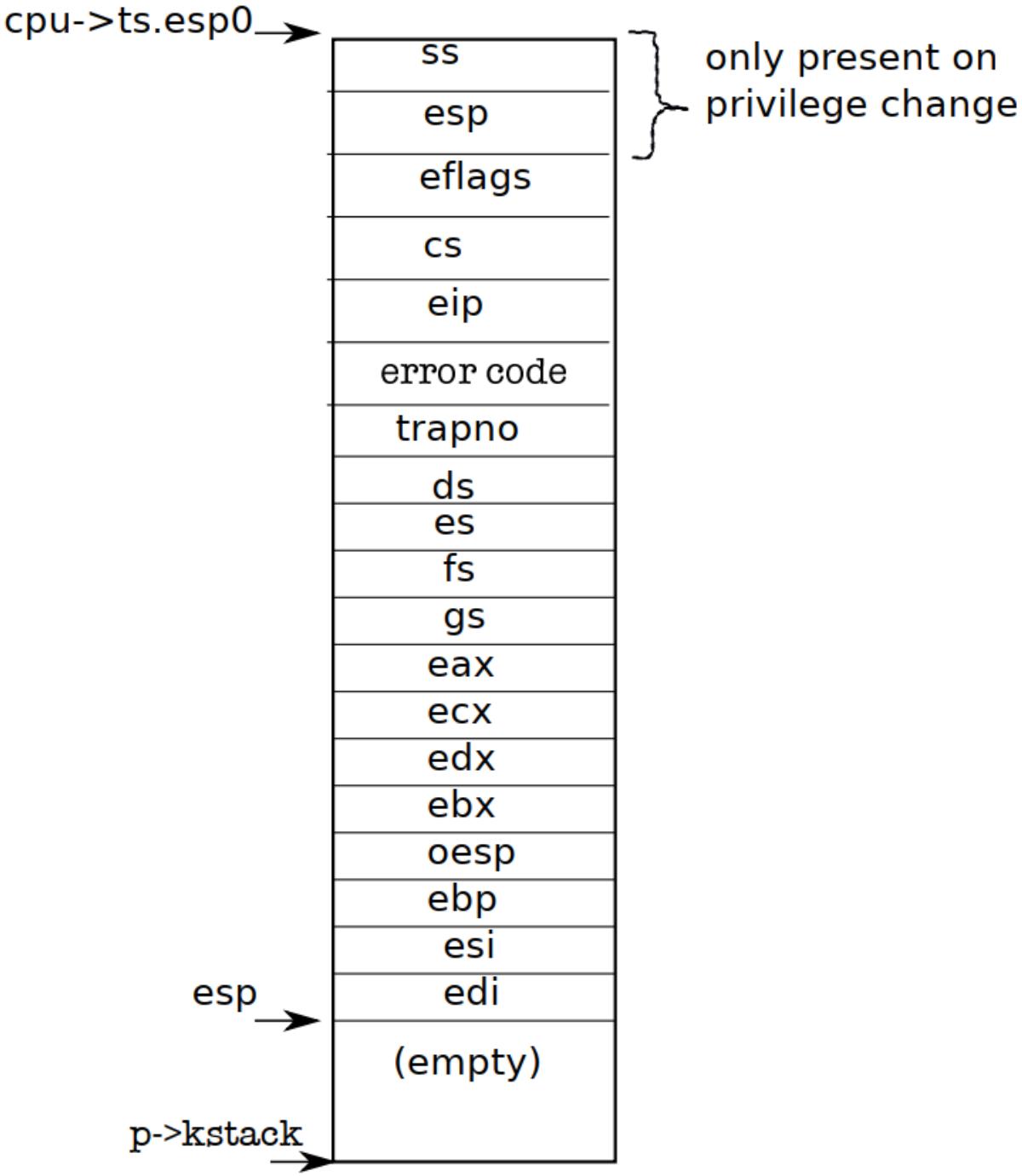
检查 CS 寄存器的权限位（CPL）；

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。为填充完整信息，trap 入口函数提供进一步操作，最终在内核栈中构成完整的 TrapFrame 结构，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.

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：

```
/*
 * Lab 3: Your code here for _alltraps
 */
 .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.
  // Declare the functions from trapentry.S
  void DIVIDE();
  void DEBUG();
  void NMI();
  void BRKPT();
  void OFLOW();
  void BOUND();
  void ILLOP();
  void DEVICE();
  void DBLFLT();
  void TSS();
  void SEGNP();
  void STACK();
  void GPFLT();
  void PGFLT();
  void FPERR();
  void ALIGN();
  void MCHK();
  void SIMDERR();
  void SYSCALL();
  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();
}
```

Question

1. What is the purpose of having an individual handler function for each exception/interrupt? (i.e., if all exceptions/interrupts were delivered to the same handler, what feature that exists in the current implementation could not be provided?)

2. Did you have to do anything to make the user/softint program behave correctly? The grade script expects it to produce a general protection fault (trap 13), but softint's code says int $14. Why should this produce interrupt vector 13? What happens if the kernel actually allows softint's int $14 instruction to invoke the kernel's page fault handler (which is interrupt vector 14)?

1. 因为不同的中断有不同的处理方式及不同的权限，因此不能统一从一个函数接口进入再进行分发。

2. 除了系统调用，其余 GATE 都被设计为只允许内核特权级通过，用户态发送特权级向量将触发 general protection fault，这是操作系统对内核的保护。

Chapter 2

Page Faults, Breakpoints Exceptions, and System Calls

2.1 Handling Page Faults

Part B 的内容是构建简单的异常处理函数构建。例如这一节处理的 Page Fault 异常。我们已经知道，所有异常最终将进入 trap() 函数，而 trap() 通过 trap_dispatch() 函数将内容分发给指定的处理函数。在 x86 中，页错误被映射为 14 号中断向量。

Exercise 5

Modify trap_dispatch() to dispatch page fault exceptions to page_fault_handler().

```
static void
trap_dispatch(struct Trapframe *tf)
{
  // Handle processor exceptions.
  // LAB 3: Your code here.
  switch(tf->tf_trapno)
  {
    case T_PGFLT:
      page_fault_handler(tf);
  }
  ...
}
```

2.2 The Breakpoint Exception

断点异常常用于对程序的调试，其工作原理是将设置断点处的命令前插入命令 int 3 。

Exercise 6

Modify trap_dispatch() to make breakpoint exceptions invoke the kernel monitor.

本节的基础内容与上一小节类似，修改 trap_dispatch() 完成对 BreakPoint 异常的分发。实际上 monitor 需在提高部分完成后才具备调试功能。

```
static void
trap_dispatch(struct Trapframe *tf)
{
  // Handle processor exceptions.
  // LAB 3: Your code here.
  switch(tf->tf_trapno)
  {
    case T_PGFLT:
      page_fault_handler(tf);
      return;
    case T_BRKPT:
      monitor(tf);
      return;
  }
  ...
}
```

Question

The break point test case will either generate a break point exception or a general protection fault depending on how you initialized the break point entry in the IDT (i.e., your call to SETGATE from trap_init). Why? How do you need to set it up in order to get the breakpoint exception to work as specified above and what incorrect setup would cause it to trigger a general protection fault?

这与 SETGATE 宏的最后一个参数有关：它设置了入口的 DPL(Descriptor of Privilege Level)。只有当当前程序的优先级小于等于 DPL 时才会触发正常的 BreakPoint Exception。若处于用户态，当调用内核态的指令时就会触发 general protection fault。

What do you think is the point of these mechanisms, particularly in light of what the user/softint test program does?

上述机制使得用户进程不能随意访问内核态的代码和内存，保证内核安全。

2.3 System calls

根据操作系统的设计原理可知，用户态进程通过系统调用（System Call）使用内核功能。当切换到内核模式时，CPU 与内核代码共同参与对当前用户状态的保存，随后内核调用对应系统调用的实现，并恢复原有用户态进程。整个设计的关键在于：用户态进程如何通知内核进行系统调用，以及如何确定确切的系统调用。

在 JOS 的设计中，系统调用的中断向量为 48(0x30)，根据前述，这是一个软件中断，不会引起硬件产生相同中断的二义性。系统调用号、参数及返回值均通过寄存器传递。

Exercise 7

Add a handler in the kernel for interrupt vector T_SYSCALL. Handle all the system calls listed in inc/syscall.h by invoking the corresponding kernel function for each call.

根据 Lab 的说明，调用时系统调用号放入 EAX 寄存器，参数依次放入 EDX、ECX、EBX、EDI 及 ESI 寄存器，调用返回后将返回值放入 EAX 寄存器。kern/syscall.c 提供了 syscall() 接口（未完成）供调用。修改 trap_dispatch() 函数如下：

```
static void
trap_dispatch(struct Trapframe *tf)
{
  // LAB 3: Your code here.
  switch(tf->tf_trapno)
   {
      case T_PGFLT:
         page_fault_handler(tf);
         return;
      case T_BRKPT:
         monitor(tf);
         return;
      case T_SYSCALL:
         tf->tf_regs.reg_eax = syscall(
                               tf->tf_regs.reg_eax,
                               tf->tf_regs.reg_edx,
                               tf->tf_regs.reg_ecx,
                               tf->tf_regs.reg_ebx,
                               tf->tf_regs.reg_edi,
                               tf->tf_regs.reg_esi
                              );
         return;
   }
  ...
}
```

第二步是实现 kern/syscall.c/syscall() 。实验要求完成所有 inc/syscall.h 列表中的系统调用，这些调用都已在 kern/syscall.c 中实现，只需根据系统调用号将任务分发即可。

```
int32_t
syscall(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t a3, uint32_t a4, uint32_t
    a5)
{
  // Call the function corresponding to the 'syscallno' parameter.
  // Return any appropriate return value.
  // LAB 3: Your code here.

  switch (syscallno) {

  case SYS_cputs:
    sys_cputs((const char *)a1, a2);
    return 0;
  case SYS_cgetc:
    return sys_cgetc();
  case SYS_getenvid:
    return sys_getenvid();
  case SYS_env_destroy:
    return sys_env_destroy(a1);

  default:
    return -E_INVAL;
  }
}
```

2.4 User-mode startup

用户程序的启动入口是 lib/entry.S（区别于 kern/entry.S），经过一系列设置后调用 lib/libmain.c/libmain()，这个函数启动用户程序（通过统一命名的 umain() 接口）。

Exercise 8

Add the required code to the user library, then boot your kernel. You should see user/hello print "hello, world" and then print "i am environment 00001000". user/hello then attempts to "exit" by calling sys_env_destroy() (see lib/libmain.c and lib/exit.c).

根据实验提示完善代码：

```
void
libmain(int argc, char **argv)
{
  // set thisenv to point at our Env structure in envs[].
  // LAB 3: Your code here.
  thisenv = envs + ENVX(sys_getenvid());

  // save the name of the program so that panic() can use it
  if (argc > 0)
    binaryname = argv[0];

  // call user main routine
  umain(argc, argv);

  // exit gracefully
  exit();
}
```

2.5 Page faults and memory protection

这部分内容是完成一个简单的 Page Fault 的处理。操作系统依赖硬件对内存保护提供的支持，当硬件检测到访问无效地址、越级访问等问题时，硬件发送中断向量，操作系统进入 trap 处理。这样的设计在系统调用时会产生这样的情景：用户态程序需要向内核传递指针，这些指针应当指向用户地址区。这里产生两个关键的问题：

1. 内核需要知道指针的传递者是内核还是用户，拦截用户态对内核段的非法访问；
2. 内核态的 Page Fault 较用户态严重，因为它将引起内核停止并陷入异常处理中。

因此，JOS 在处理 Page Fault 导致的中断时，对其来源于用户态/内核态进行了检查和区分处理。

Exercise 9+10

Change kern/trap.c to panic if a page fault happens in kernel mode.

Read user_mem_assert in kern/pmap.c and implement user_mem_check in that same file.

Change kern/syscall.c to sanity check arguments to system calls.

Finally, change debuginfo_eip in kern/kdebug.c to call user_mem_check on usd, stabs, and stabstr.

基于操作系统代码段引发的 Page Fault 可能引起严重的问题，JOS 的决策是停止内核的运行。根据提示，首先修改 trap.c/page_fault_handler()，使内核态的 Page Fault 直接 panic 并退出运行。

```
  ...
  // LAB 3: Your code here.
   if(tf->tf_cs == GD_KT)
    panic("page_fault_handler: kernel page fault");
  ...
```

进一步根据实验提示，完成 pmap.c/user_mem_check() 函数。这个函数的作用是检查某一进程（此处的检查仅针对用户进程）访问某一地址是否存在缺页、越界访问或缺少访问权限的问题。user_mem_assert() 调用该函数，当检查用户进程对内存存在访问问题时，销毁该进程。

```
int
user_mem_check(struct Env *env, const void *va, size_t len, int perm)
{
  // LAB 3: Your code here.
  uint32_t start = (uint32_t)ROUNDDOWN(va, PGSIZE);
  uint32_t end = (uint32_t)ROUNDUP(va + len, PGSIZE);
  pte_t *page;
  for(; start < end; start += PGSIZE)
  {
    page = pgdir_walk(env->env_pgdir, (void *)start, 0);
    if(!page || start > ULIM || ((uint32_t)(*page) & perm) != perm)
    {
      if(start <= (uint32_t)va)
        user_mem_check_addr = (uintptr_t)va;
      else
        user_mem_check_addr = (uintptr_t)start;
      return -E_FAULT;
    }
  }

  return 0;
}
```

当拥有检查用户进程权限的功能后，实验要求我们修改 kern/syscall.c 中系统调用函数，增加检查权限的内容。具体地，当前实现的系统调用中仅有 sys_cputs() 需向内核传递指针。因此，修改该函数如下：

```
static void
sys_cputs(const char *s, size_t len)
{
  user_mem_assert(curenv, s, len, 0);
  // Print the string supplied by the user.
  cprintf("%.*s", len, s);
}
```

另一部分内容是修改 debuginfo_eip() 函数，为该调试函数增加访问权限检查的功能：

```
...
// Make sure this memory is valid.
// Return -1 if it is not.  Hint: Call user_mem_check.
// LAB 3: Your code here.
if(user_mem_check(curenv, usd, sizeof(struct UserStabData), PTE_U))
  return -1;

stabs = usd->stabs;
stab_end = usd->stab_end;
stabstr = usd->stabstr;
stabstr_end = usd->stabstr_end;

// Make sure the STABS and string table memory is valid.
// LAB 3: Your code here.
if(user_mem_check(curenv, stabs, (stab_end - stabs) * sizeof(struct Stab), PTE_U | PTE_P)
    != 0)
  return -1;
if(user_mem_check(curenv, stabstr, stabstr_end - stabstr, PTE_U | PTE_P) != 0)
  return -1;
...
```

Chapter 3

Appendix: X86-Protection

这里补充说明 x86 提供的特权级保护机制。

x86 提供了 4 个特权级：level0　level3，level0 表示最高权限，并引入当前特权级 *(Current Privilege Level, CPL)* 与描述符特权级 *(Descriptor Privilege Level, DPL)* 的概念。CPL 位保存在代码段寄存器 CS 中，它记录当前运行代码所在的特权级；DPL 位保存在段描述符（即 GDT 表项）或 IDT 项中，它记录访问此段所需的特权级。当且仅当 CPL <= DPL 成立时，访问才被允许。