

MIT6.828 Lab3: User Environments

Zhuofan Zhang

Jan 2022

Contents

- 1 Physical Page Management 1**
 - 1.1 Environment State 1
 - 1.2 Allocating the Environments Array 2
 - 1.3 Creating and Running Environments 3
 - 1.4 Handling Interrupts and Exceptions 10

Chapter 1

User Environments and Exception Handling

本次 Lab 的第一部分主要处理 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 memlayout.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);

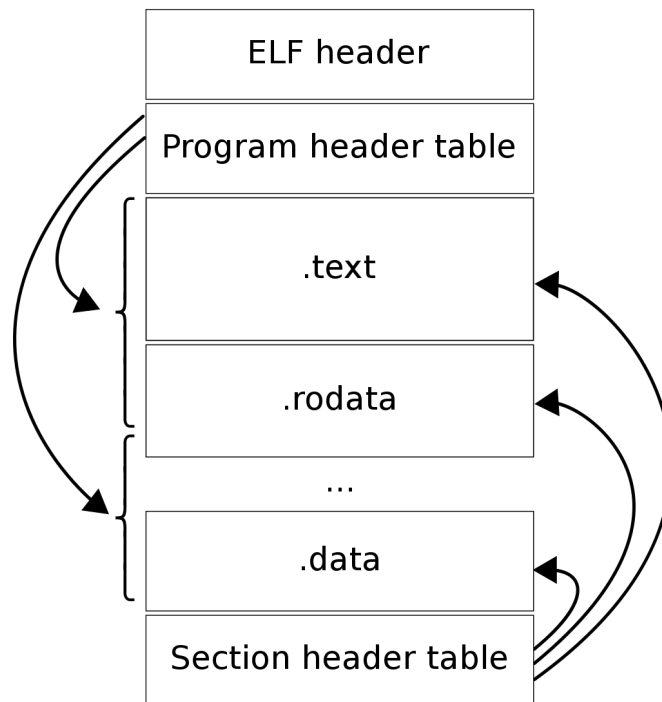
    if(!e)
        panic("region_alloc: env is NULL.\n");

    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_create: 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`。

```
[00000000] new env 00001000
EAX=00000000 EBX=00000000 ECX=0000000d EDX=eebfde78
ESI=00000000 EDI=00000000 EBP=eebfde20 ESP=eebfdde8
EIP=00800f06 EFL=00000016 [----AP-] CPL=3 II=0 A20=1 SMM=0 HLT=0
ES =0023 00000000 ffffffff 00cff300 DPL=3 DS [-WA]
CS =001b 00000000 ffffffff 00cffa00 DPL=3 CS32 [-R-]
SS =0023 00000000 ffffffff 00cff300 DPL=3 DS [-WA]
DS =0023 00000000 ffffffff 00cff300 DPL=3 DS [-WA]
FS =0023 00000000 ffffffff 00cff300 DPL=3 DS [-WA]
GS =0023 00000000 ffffffff 00cff300 DPL=3 DS [-WA]
LDT=0000 00000000 00000000 00008200 DPL=0 LDT
TR =0028 f0191b20 00000067 00408900 DPL=0 TSS32-avl
GDT= f011e500 0000002f
IDT= f0191320 000007ff
CR0=80050033 CR2=00000000 CR3=003bc000 CR4=00000000
DR0=00000000 DR1=00000000 DR2=00000000 DR3=00000000
DR6=ffff0ff0 DR7=00000400
EFER=0000000000000000
Triple fault. Halting for inspection via QEMU monitor.
```

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。

此外，在此处我们可以通过一个方法来确认是否真的已经切换到用户态：在 `env_run` 之前使用 `gdb` 的 `info registers` 命令调试，可以看到 `CS` 寄存器（可见部分）的值为 `0x8`，故 `CPL=0x7&0x3=0`；当运行至用户态代码时，查看 `info registers` 可得 `CS` 寄存器（可见部分）值为 `0x1b`，故 `CPL=3`，可以发现权限级别确实发生了变化。

注：`CS` 寄存器低位存储 `CPL` 细节见 *IA32-3A Chapter 4.5* 一节。

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。
- **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**）。

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。