Exercise 1

类似于`npages`的创建,调用`boot_alloc()`来给`envs`分配`NENV`个`Env`的空间,并将数据置零

```
envs = (struct Env *) boot_alloc(NENV * sizeof(struct Env));
memset(envs, 0, NENV * sizeof(struct Env));
```

接着调用`boot_map_region()`将`envs`映射到`UENVS`开始的,大小为`PTSIZE`的内存区域,将权限标记为read-only

```
boot_map_region(kern_pgdir, UENVS, PTSIZE, PADDR(envs), PTE_U | PTE_P);
```

Exercise 2

`env_init()`

定义`tmp`为`env_free_list`的最后一个节点, 每次新加节点时先插入在`tmp`的`env_link`中, 然后将`tmp`往后指一个节点. 这样可以保证`env_free_list`中的节点按照序号排列

```
while (cur != NULL) {
        cur->env_status = ENV_FREE;
        cur->env_id = 0;
        if (tmp != NULL) {
                tmp->env_link = cur;
                tmp = tmp->env_link;
        }
        else {
                env_free_list = cur;
                tmp = env_free_list;
        }
        cur = cur->env_link;
}
```

`env_setup_vm()`

由于给出的代码中已经申请了物理页, 所以先将它映射到`e`的`env_pgdir`, 并将`pp_ref`自增
接着按照给出的提示将`PDX(UTOP)`到`NPDENTRIES`之间除了`PDX(UVPT)`之外的页都从`kern_pgdir`复制一份到`env_pgdir`

```
e->env_pgdir = page2kva(p);
p->pp_ref++;

for (int i = PDX(UTOP); i < NPDENTRIES; i++) {
        if (i == PDX(UVPT)) continue;
```

```
        e->env_pgdir[i] = kern_pgdir[i];
}
```

region_alloc()

首先调用ROUNDDOWN和ROUNDUP函数将开始地址以及结束地址按照PGSIZE对齐，然后对这之间的每个页都调用page_insert插入进env_pgdir中

```
uintptr_t start, end;
start = (uintptr_t)ROUNDDOWN(va, PGSIZE);
end = (uintptr_t)ROUNDUP(va + len, PGSIZE);

for (uintptr_t i = start; i < end; i += PGSIZE) {
        struct PageInfo *pp = page_alloc(0);
        page_insert(e->env_pgdir, pp, (void*)i, PTE_U | PTE_W);
}
```

load_icode()

模仿bootmain()中的写法，不过不需要去硬盘中读取文件，而是直接将参数中的binary转换类型成需要的elf结构. 接着为文件中每一个不是ELF_PROG_LOAD类型的段在环境e中分配内存页，并将文件段中的内容复制进e中对应的位置
由于需要在环境e中对内存页进行修改，所以要先用lcr3()将pgdir切换为e->env_pgdir，执行结束后再切换回kern_pgdir
同时也需要将该环境的tf_eip设为elf的入口，方便切换至该环境时能够从正确的地方开始执行
最后调用region_alloc给该环境的栈分配了一个大小为PGSIZE的页

```
static void
load_icode(struct Env *e, uint8_t *binary)
{
        struct Proghdr *ph, *eph;
        struct Elf *elf = (struct Elf *)binary;

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

        lcr3(PADDR(e->env_pgdir));
        for (; ph < eph; ph++) {
                if (ph->p_type != ELF_PROG_LOAD) continue;
                region_alloc(e, (void*)ph->p_va, ph->p_memsz);
                memset((void*)ph->p_va, 0, ph->p_memsz);
                memmove((void*)ph->p_va, binary + ph->p_offset, ph->p_filesz);
        }
        lcr3(PADDR(kern_pgdir));

        e->env_tf.tf_eip = elf->e_entry;
```

```
        region_alloc(e, (void*)USTACKTOP - PGSIZE, PGSIZE);
}
```

env_create()

环境的创建首先需要调用env_alloc创建一个空的环境，然后调用load_icode将参数中的二进制文件加载进该环境，最后将环境的类型设为给定参数中的类型

```
void
env_create(uint8_t *binary, enum EnvType type)
{
        struct Env *env;
        env_alloc(&env, 0);
        load_icode(env, binary);
        env->env_type = type;
}
```

env_run()

将当前环境切换至目标环境，首先需要将当前环境的env_status置为ENV_RUNNABLE，然后将curenv设成目标环境e并启用目标环境的env_pgdir. 最后调用env_pop_tf()还原环境的参数

```
void
env_run(struct Env *e)
{
        if (curenv != e) {
                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);
}
```

Exercise 4

这里以第3号中断breakpoint为例，首先在trap.c中定义处理函数H_BRKPT，然后调用SETGATE绑定H_BRKPT到idt[3]，这里dpl设置为3是为了能够让用户程序直接调用，如果设置为0则用户程序不能触发该中断而是会触发13号general protection fault中断
对于_alltraps，首先将原来的寄存器都push到栈上，然后%ds，es的值都设为GD_KD，代表kernel data段. 最后pushl %esp并调用trap

```
# kern/trap.c

void H_BRKPT();

SETGATE(idt[3], 1, GD_KT, H_BRKPT, 3);

# kern/trapentry.S

TRAPHANDLER_NOEC(H_BRKPT, T_BRKPT)

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

Exercise 5

在`trap_dispatch`中判断`tf->tf_trapno`, 如果等于`T_PGFLT`则调用`page_fault_handler`

```
static void
trap_dispatch(struct Trapframe *tf)
{
        int r = 0;
        switch(tf->tf_trapno) {
                case T_PGFLT:
                        page_fault_handler(tf);
                        break;
                default: {
                        print_trapframe(tf);
                        if (tf->tf_cs == GD_KT)
                                panic("unhandled trap in kernel");
                        else {
                                env_destroy(curenv);
                                return;
                        }
                }
        }
}
```

Exercise 6

在`trap_dispatch`中增加一个case判断条件，如果`tf->tf_trapno`等于`T_BRKPT`，首先应该打印`trapframe`然后不销毁环境直接进入到`monitor`中

```
case T_BRKPT:
        print_trapframe(tf);
        while (1)
                monitor(NULL);
        break;
```

Exercise 7

创建处理函数`H_SYSCALL`并调用`SETGATE`绑定至`idt[T_SYSCALL]`，由于系统调用需要能被用户程序使用，所以将`dpl`设成3.
在`trap_dispatch`中增加一个case判断条件，如果`tf->tf_trapno`等于`T_SYSCALL`，则调用函数`syscall`参数分别为`%eax, %edx, %ecx, %ebx, %edi, %esi`，最后将返回值传给`%eax`.

```
void H_SYSCALL();

SETGATE(idt[T_SYSCALL], 1, GD_KT, H_SYSCALL,3);

TRAPHANDLER_NOEC(H_SYSCALL, T_SYSCALL)

case T_SYSCALL:
        r = 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);
        tf->tf_regs.reg_eax = r;
        break;
```

`syscall`函数的实现与`trap_dispatch`类似，根据传入的`syscallno`处理不同种类的系统调用

```
int32_t
syscall(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t a3, uint32_t a4,
uint32_t a5)
{
        switch (syscallno) {
                case SYS_cputs:
                        sys_cputs((const char *)a1, (size_t)a2);
                        return 0;
                case SYS_cgetc:
                        return sys_cgetc();
                case SYS_getenvid:
                        return sys_getenvid();
                case SYS_env_destroy:
```

```
                        return sys_env_destroy((envid_t)a1);
                case SYS_map_kernel_page:
                        return sys_map_kernel_page((void *)a1, (void *)a2);
                case SYS_sbrk:
                        return sys_sbrk(a1);
                default:
                        return -E_INVAL;
        }
}
```

Exercise 9

根据提示调用`sys_getenvid()`获取当前环境的id, 宏`ENVX`可以获得该id二进制的后9位(0 ~ 1023).

```
thisenv = &envs[ENVX(sys_getenvid())];
```

Exercise 10

根据提示修改`struct Env`新增属性`env_break`记录当前program的break信息, 并将其初始化为`UTEXT`, 代表堆从这里开始.
函数实现和`region_alloc`的实现基本一样, 但由于无法直接调用`region_alloc`所以直接复制了一段代码= =
最后将`env_break`加上增加的大小后返回

```
static int
sys_sbrk(uint32_t inc)
{
        if (inc == 0) return (int)curenv->env_break;
        if (curenv->env_break + inc > UTOP) return -1;

        uintptr_t start, end;
        start = (uintptr_t)ROUNDDOWN(curenv->env_break, PGSIZE);
        end = (uintptr_t)ROUNDUP(curenv->env_break + inc, PGSIZE);

        for (uintptr_t i = start; i < end; i += PGSIZE) {
                pte_t *pte = pgdir_walk(curenv->env_pgdir, (void*)i, 0);
                struct PageInfo *p = page_alloc(0);
                if ((pte = pgdir_walk(curenv->env_pgdir, (void*)i, 0)) == NULL ||
!(*pte & PTE_P)){
                        if ((p = page_alloc(0)) == NULL)
                                return -1;
                        if (page_insert(curenv->env_pgdir, p, (void*)i, PTE_U |
PTE_W) < 0)
                                return -1;
                }
        }

        curenv->env_break += inc;
```

```
        return (int)curenv->env_break;
}
```

Exercise 11

user_mem_check的思想和前面region_alloc之类的思想都很相似，给定了开始地址和长度，只需要遍历这之间的page，对每个页都走一遍pgdir_walk根据结果判断是否符合要求即可
需要注意的是测试环境buggyhello中访问了地址0x00000001，这里应该将user_mem_check_addr置为0x00000001而不是0x00000000

```
int
user_mem_check(struct Env *env, const void *va, size_t len, int perm)
{
        uintptr_t start, end;
        start = (uintptr_t)ROUNDDOWN(va, PGSIZE);
        end = (uintptr_t)ROUNDUP(va + len, PGSIZE);
        for (uintptr_t i = start; i < end; i += PGSIZE) {
                pte_t* pte = pgdir_walk(curenv->env_pgdir, (void*)i, 0);
                if (pte == NULL || i >= ULIM || !((perm | PTE_P) & *pte)) {
                        // buggyhello: 0x00000001
                        user_mem_check_addr = i < (uintptr_t)va ? (uintptr_t)va :
i;
                        return -E_FAULT;
                }
        }
        return 0;
}
```

接着需要在sys_cputs中加上对访问地址的判断

```
user_mem_assert(curenv, s, len, PTE_U);
```

最后是kdebug.c中对usd, stabs以及stabstr增加判断

```
if (user_mem_check(curenv, (void*)usd, sizeof(struct UserStabData), PTE_U) < 0)
return -1;

if (user_mem_check(curenv, stabs, stab_end - stabs, PTE_U) < 0) return -1;

if (user_mem_check(curenv, stabstr, stabstr_end - stabstr, PTE_U) < 0) return -1;
```

Exercise 13

看了一眼代码发现该写的都写好了，然后根据提示加了个wrapper，虽然不知道加了这个有什么区别 = =
但是加了后通不过测试，看了很久把一行asm volatile("popl %ebp")改成了asm volatile("leave")就好

了... 觉得应该是`leave`包含了`movl %ebp, %esp`和`popl %ebp`

```
static void (*wrapper)(void) = NULL;

wrapper = fun_ptr;

void call_fun_ptr()
{
    wrapper();
    *entry = old;
    asm volatile("leave");
    asm volatile("lret");
}
```