Lab3

徐天强 516030910391

exercise 1:

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

参照 pages 同样的初始化方法，通过调用 boot_alloc 来为 envs 分配相应的内存空间。

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

同时，使用 boot_map_region 函数，来将相应的物理页映射到虚拟地址的 UENVS 处，权限为 PTE_U 即 read-only.

exercise 2:

env_init()

```
void
env_init(void)
{
    // Set up envs array
    // LAB 3: Your code here.
    for(int i = 0; i < NENV - 1; i++){
        envs[i].env_id = 0;
        envs[i].env_link = &envs[i+1];
    }
    env_free_list = envs;
    // Per-CPU part of the initialization
    env_init_percpu();
}
```

负责初始化 envs 中所有的 env 结构体，并将修改它们的 env_link 成员变量，使得它们串成一个链表。

env_free_list = envs

将 env_free_list 指向 envs 的第一个 env struct，表示此时所有的 env 都是空闲的。

env_setup_vm()

```
e->env_pgdir[PDX(UVPT)] = PADDR(e->env_pgdir) | PTE_P | PTE_U;
```

为此 env 分配一个 page 作为 pagetable。并将此 page 映射到虚拟地址 UVPT 处。

region_alloc()

为一个 env 在虚地址 va 处，连续分配 len 长的内存空间。

函数主要使用两个函数，在 page_alloc 分配一个 page 之后，使用 page_insert 将此 page 映射到此 env 的 pagetable 中。具体实现如下：

```
int i = ROUNDDOWN((uint32_t)va, PGSIZE);
int end = ROUNDUP((uint32_t)va + len, PGSIZE);
for(; i < end; i+=PGSIZE){
    struct PageInfo * page = page_alloc(ALLOC_ZERO);
    if(!page)
        panic("there is no page\n");
    int ret = page_insert(e->env_pgdir, page, (void*)((uint32_t)i), PTE_U | PTE_W);
    if(ret)
        panic("there is error in insert");
}
```

load_icode()

此函数从 elf 文件头 binary 中，将文件的各个 segment 加载到对应的内存位置。此处的代码逻辑主要参考 boot/main.c 中的相关代码。

不过这个要特别注意，在进入此函数时，我们使用的还是 kernel 的页表，需要 lcr3 将当前的页表切换成对应 env 的页表，然后再加载 elf 中的各个 segment。在函数结尾处还需要将页表切换回来。

此处我们需要对 trapframe 做相应的修改，即将对应的 eip 设置为 elf 中 entry 位

置。具体如下：

```
region_alloc(e, (void*)(USTACKTOP-PGSIZE), PGSIZE);
lcr3(PADDR(kern_pgdir));
```

env_create()

调用 env_alloc 函数和 load_icode 函数来创建一个新的 env。

env_run()

env_run()将 env e 放到当前的 cpu 上执行，在这之前将 curenv 的 environment 的状态设置为 runnable。

然后切换新 env 的页表，并调用 env_pop_tf,因为我们之前在 load_icode 中设置了 eip，故可以在 env_pop_tf 后从正确的指令处开始执行。

exercise 4:

trapentry.S

```
TRAPHANDLER_NOEC(DIVIDE_HANDLER   , T_DIVIDE )
TRAPHANDLER_NOEC(DEBUG_HANDLER    , T_DEBUG  )
TRAPHANDLER_NOEC(NMI_HANDLER      , T_NMI    )
TRAPHANDLER_NOEC(BRKPT_HANDLER    , T_BRKPT  )
TRAPHANDLER_NOEC(OFLOW_HANDLER    , T_OFLOW  )
TRAPHANDLER_NOEC(BOUND_HANDLER    , T_BOUND  )
TRAPHANDLER_NOEC(ILLOP_HANDLER    , T_ILLOP  )
TRAPHANDLER_NOEC(DEVICE_HANDLER   , T_DEVICE )
TRAPHANDLER(     DBLFLT_HANDLER   , T_DBLFLT )
TRAPHANDLER(     TSS_HANDLER      , T_TSS    )
TRAPHANDLER(     SEGNP_HANDLER    , T_SEGNP  )
TRAPHANDLER(     STACK_HANDLER    , T_STACK  )
TRAPHANDLER(     GPFLT_HANDLER    , T_GPFLT  )
TRAPHANDLER(     PGFLT_HANDLER    , T_PGFLT  )
TRAPHANDLER_NOEC(FPERR_HANDLER    , T_FPERR  )
TRAPHANDLER(     ALIGN_HANDLER    , T_ALIGN  )
TRAPHANDLER_NOEC(MCHK_HANDLER     , T_MCHK   )
TRAPHANDLER_NOEC(SIMDERR_HANDLER  , T_SIMDERR)
TRAPHANDLER_NOEC(SYSCALL_HANDLER, T_SYSCALL) /*
```

首先，使用系统提供的宏，将对应的 trap number 与 handler 绑定在一起。其中 TRAPHANDLER_NOEC 表示 cpu 不会 push error num，因此需要用软件 push error num 来实现对齐。

```
.globl _alltraps
_alltraps:
    pushw $0
    pushw %ds
    pushw $0
    pushw %es
    pushal

    movl $(GD_KD), %eax
    movw %ax, %ds
    movw %ax, %es

    pushl %esp
    call trap
```

然后是 alltraps。将 trapframe 中除去 cpu 自动 push 到栈上的部分继续 push 到栈上。比如 ds，es，然后用 kernel 的 data segment 初始化 es ds 寄存器。当把 trapframe 所有的数据 push 到栈上后，此时 esp 指向的内存即为一个 trapframe，按照 x86 的 calling convention，将这个 esp 压栈，即作为 trap()函数的第一个参数。

```
extern void DIVIDE_HANDLER();
extern void DEBUG_HANDLER();
extern void NMI_HANDLER();
extern void BRKPT_HANDLER();
extern void OFLOW_HANDLER();
extern void BOUND_HANDLER();
extern void ILLOP_HANDLER();
extern void DEVICE_HANDLER();
extern void DBLFLT_HANDLER();
extern void TSS_HANDLER();
extern void SEGNP_HANDLER();
extern void STACK_HANDLER();
extern void GPFLT_HANDLER();
extern void PGFLT_HANDLER();
extern void FPERR_HANDLER();
extern void ALIGN_HANDLER();
extern void MCHK_HANDLER();
extern void SIMDERR_HANDLER();
extern void SYSCALL_HANDLER();
```

```
SETGATE(idt[T_DIVIDE] , 0, GD_KT, DIVIDE_HANDLER , 0);
SETGATE(idt[T_DEBUG]  , 0, GD_KT, DEBUG_HANDLER  , 0);
SETGATE(idt[T_NMI]    , 0, GD_KT, NMI_HANDLER    , 0);
SETGATE(idt[T_BRKPT]  , 0, GD_KT, BRKPT_HANDLER  , 3);
SETGATE(idt[T_OFLOW]  , 0, GD_KT, OFLOW_HANDLER  , 3);
SETGATE(idt[T_BOUND]  , 0, GD_KT, BOUND_HANDLER  , 3);
SETGATE(idt[T_ILLOP]  , 0, GD_KT, ILLOP_HANDLER  , 0);
SETGATE(idt[T_DEVICE] , 0, GD_KT, DEVICE_HANDLER , 0);
SETGATE(idt[T_DBLFLT] , 0, GD_KT, DBLFLT_HANDLER , 0);
SETGATE(idt[T_TSS]    , 0, GD_KT, TSS_HANDLER    , 0);
SETGATE(idt[T_SEGNP]  , 0, GD_KT, SEGNP_HANDLER  , 0);
SETGATE(idt[T_STACK]  , 0, GD_KT, STACK_HANDLER  , 0);
SETGATE(idt[T_GPFLT]  , 0, GD_KT, GPFLT_HANDLER  , 0);
SETGATE(idt[T_PGFLT]  , 0, GD_KT, PGFLT_HANDLER  , 0);
SETGATE(idt[T_FPERR]  , 0, GD_KT, FPERR_HANDLER  , 0);
SETGATE(idt[T_ALIGN]  , 0, GD_KT, ALIGN_HANDLER  , 0);
SETGATE(idt[T_MCHK]   , 0, GD_KT, MCHK_HANDLER   , 0);
SETGATE(idt[T_SIMDERR], 0, GD_KT, SIMDERR_HANDLER, 0);
```

最后需要修改的是 trap_init()函数。首先，使用 SETGATE 初始化 IDT 中不同 trapnumber 处的异常处理函数。关键是 istrap 和 dpl 两个参数。istrap 表示当前处理的是异常还是中断，所有 trapnumber 小于 32 的，被认为是异常，因此相应的 istrap 参数是 0，否则是终端，istrap 应为 1.dpl 全称是 Descriptor Privilege Level，描述当前的中断或者异常是否能做用户态被触发，如果是 3 则可以，是 0 则不行。

Exercise 5:

```
case T_PGFLT:
    page_fault_handler(tf);
    break;
```

修改 trap_dispatch()函数，根据 tf 的不同 trapnumber，来‘dispatch trap’，首先要处理的是 page fault，因此当判断当前的 trap number 是 page fault 时，调用 page_fault_handler.

Exercise 6:

```
case T_BRKPT:
    monitor(tf);
    break;
```

与 exercise 5 类似，当判断当前的 trap number 是 breakpoint 时，调用 monitor 函数。

Exercise 7:

需要添加 T_SYSCALL 的相应异常处理逻辑，与 exercise 4 相似，在 trapentry.S 中添加：

TRAPHANDLER_NOEC(SYSCALL_HANDLER, T_SYSCALL)

在 trap()函数中添加：

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

并在 trap_dispatch()函数中，将 T_SYSCALL 分支的处理逻辑设置为 syscall().由于保存 syscall 各个参数的寄存器都在注释中指出，我们可以直接使用 trapframe 对应的寄存器作为参数即可。又 syscall 函数的返回值需要放在 eax 中，即将其赋值给 trapframe 的 eax 即可。

接下来是实现 syscall()函数，根据 syscallno 来区分调用的具体 syscall 函数，使用 switch 分支语句完成相应逻辑即可。

Exercise 8:

由于用 exercise7 的方式实现 syscall 可能导致效率较低，我们可以使用 intel 提供的 sysenter 和 sysexit 指令来完成内核态、用户态的切换，以及调用 syscall 的逻辑。

```
"leal after_sysenter_label%=, %%esi\n\t"
"sysenter\n\t"
"after_sysenter_label%=:\n\t"
```

首先，修改用户可以调用的 lib 库中的 syscall 桩函数。将可能被破坏的寄存器 push 到栈上保存，使用 leal after_sysenter_label, %%esi. 指令来保存返回地址。通过 push pop 操作保存 ebp。然后调用 sysenter 指令，切换到内核态处理 syscall。

那么 cpu 如何知道当执行到 sysenter 时，要执行什么代码呢？

通过阅读 intel 文档，我们使用 wrmsr 进行初始化。

```
wrmsr(0x174, GD_KT , 0);//IA32_SYSENTER_CS
wrmsr(0x175, KSTACKTOP, 0);//IA32_SYSENTER_ESP
wrmsr(0x176, sysenter_handler, 0);//IA32_SYSENTER_EIP
```

在 trap_init()函数中，完成上述代码，即可告知 cpu，当执行到 sysenter 时，执行什么代码-“sysenter_handler”，使用什么栈-“KSTACKTOP”。

同时我们修改 trapentry.S 添加 sysenter_handler label：

```
.global sysenter_handler
sysenter_handler:
    pushl %esi
    pushl %edi
    pushl %ebx
    pushl %ecx
    pushl %edx
    pushl %eax
    call syscall
    movl %esi, %edx
    movl %ebp, %ecx
    sysexit
```

由于我们是直接通过 sysenter_handler 调用 syscall，因此需要先按照 calling convention 将参数 push 好。当 syscall 返回之后，需要通过 sysexit 切回用户态，而 sysexit 需要的参数通过 edx，ecx 寄存器传递。

Exercise 9:

```
thisenv = 0;
envid_t find = sys_getenvid();
for(int i = 0; i < NENV; i++){
    if(envs[i].env_id == find)
        thisenv = &envs[i];
}
```

修改 libmain 函数，将 thisenv 指针初始化指向当前的 env。那么我们如何找到当前的 env 呢？使用 sys_getenvid()可以找到对应的 envid，遍历 envs，直到找到匹配的 env，然后将 thisenv 指向它即可。

Exercise 10:

实现 sys_sbrk 函数。在此之前，需要先在 trapframe 数据结构中添加 sbrk 参数，并在 load_icode 中初始化。初始化成什么呢？sbrk 在初始化时应该指向 heap 的顶端，那么 heap 在哪呢？在我们 elf 加载进来的 segment 的顶端，也即 UTEXT + elf_load_sz 处。

然后我们判断 increment 之后的 sbrk 是否仍在当前的 page 中。如果是这样，则不需要新 allocate page，否则需要新 allocate page。

```
uint32_t mod = ((uint32_t)curenv->env_sbrk)%PGSIZE;
if(inc < PGSIZE){
    if((mod + inc) < PGSIZE){
        curenv->env_sbrk+=inc;
        return curenv->env_sbrk;
    }
}
```

其次，需要注意的是，此函数的返回值是修改后和 sbrk，与标准的 sbrk 函数的行为不同。

Exercise 11:

首先，我们需要修改 kern/trap.c，使得当在 kernel 态发生 page fault 时 panic。

我们怎么判断当前处于什么状态呢？tf_cs 的低 2 位来表示当前的特权级。其值代表当前运行在 ring 几。即 tf_cs & 0x3 == 3,则说明在 ring3，在用户态，如果 tf_cs & 0x3 == 0，则说明在 ring0，则在内核态。

在 page_fault_handler 中添加：

```
if((tf->tf_cs & 3) != 3){
    panic("panic at kernel page_fault\n");
}
```

使得如果 pagefault 在 kernel 态触发，则 panic。

接着，实现 user_mem_check，来实现权限检查。

```
int
user_mem_check(struct Env *env, const void *va, size_t len, int perm)
{
    perm |= PTE_P;
    // LAB 3: Your code here.
    uint32_t i = (uint32_t)va; //buggy lab3 buggy
    uint32_t end = (uint32_t)va + len;
    for(; i < end; i=ROUNDDOWN(i+PGSIZE, PGSIZE)){
        if((uint32_t)va > ULIM){
            user_mem_check_addr = (uintptr_t)i;
            return -E_FAULT;
        }
        pte_t *the_pte = pgdir_walk(env->env_pgdir, (void *)i, 0);
        if(!(*the_pte & perm)){
            user_mem_check_addr = (uintptr_t)i;
            return -E_FAULT;
        }
    }

    return 0;
}
```

我们使用 pgdir_walk 来找到 pte,并检测相应的权限。这里需要注意的是，要将 user_mem_check_addr 指向第一次出现 pagefault 的地址，因此应该是参数传入地址或者一个页的首地址，因此在 for 循环迭代时，需要注意比较 tricky 的 i=ROUNDDOWN(i+PGSIZE, PGSIZE)

之后，在 sys_cputs()函数中，调用 user_mem_assert 完成权限检查。

最后，在 debuginfo_eip 中，加入

```
if(user_mem_check(curenv, usd, sizeof(struct UserStabData), PTE_U | PTE_W) < 0){
    return -1;
}
```

```
if(user_mem_check(curenv, stabs, (stab_end - stabs) * sizeof(struct Stab), PTE_U | PTE_W) < 0){
    return -1;
}
if(user_mem_check(curenv, stabstr, (stabstr_end - stabstr) * sizeof(char), PTE_U | PTE_W) < 0){
    return -1;
}
```

完成对 usd，stabs，stabstr 的权限检查。

Exercise 13:

在 ring0 特权级下调用用户态函数。

按照 hint 提示一步步来即可。

进入 ring0 的指令是 lcall。

注意在映射内核页时的错误处理。

```
struct Pseudodesc r_gdt;
sgdt(&r_gdt);

int t = sys_map_kernel_page((void* )r_gdt.pd_base, (void* )vaddr);
if (t < 0) {
    cprintf("ring0_call: sys_map_kernel_page failed, %e\n", t);
}

uint32_t base = (uint32_t)(PGNUM(vaddr) << PTXSHIFT);
uint32_t index = GD_UD >> 3;
uint32_t offset = PGOFF(r_gdt.pd_base);

gdt = (struct Segdesc*)(base+offset);
entry = gdt + index;
old= *entry;

SETCALLGATE(*((struct Gatedesc*)entry), GD_KT, call_fun_ptr, 3);
asm volatile("lcall $0x20, $0");
```