lab3

Exercise 1: Allocate Env Array

- 修改pmap.c中的**mem_init()**，调用boot_alloc()为envs数组分配空间，大小为NENV个struct Env，并映射到虚拟地址UENVS处，权限设为user read-only使用户进程能够访问envs数组。

```
envs = (struct Env *)boot_alloc(NENV * sizeof(struct Env));
boot_map_region(kern_pgdir, UENVS, PTSIZE, PADDR(envs), PTE_U | PTE_P);
```

Exercise 2: Create and Run Environments

- **env_init():** 初始化envs数组内所有Env结构，将id设为0，status设为ENV_FREE，通过倒序执行循环将env**顺序**插入env_free_list，因为测试会顺序遍历执行所有env。

```
for(int i = NENV - 1; i >= 0; i--) {
    envs[i].env_id = 0;
    envs[i].env_status = ENV_FREE;
    envs[i].env_link = env_free_list;
    env_free_list = &envs[i];
}
```

- **env_setup_vm():** 申请并初始化env的页表目录。将e->env_pgdir指向已申请到的PageInfo *p, 并将该页的pp_ref加1。阅读注释可知，VA在UTOP以下的内容为空，在UTOP以上的内容可直接从kern_pgdir拷贝，所以调用memmove()拷贝相应size的条目。将UVPT的env自己的页表设为用户只读。

```
p->pp_ref++;
e->env_pgdir = page2kva(p);
memmove(e->env_pgdir+PDX(UTOP), kern_pgdir+PDX(UTOP), sizeof(pde_t) * (NPDENTRIES-
PDX(UTOP)));
e->env_pgdir[PDX(UVPT)] = PADDR(e->env_pgdir) | PTE_P | PTE_U;
```

- **region_alloc():** 为environment分配并映射物理地址。注意va和len可能未按页大小对齐，round va down并round (va+len) up。调用page_alloc(0)分配未初始化的物理页，并将权限设为可读可写，插入e->env_pgdir。

```
void *va_start = ROUNDDOWN(va, PGSIZE);
void *va_end = ROUNDUP(va+len, PGSIZE);
for (void *addr = va_start; addr < va_end; addr += PGSIZE) {
    struct PageInfo *pp = page_alloc(0);
    if (!pp)
        panic("region_alloc: can't alloc\n");
    else {
        if (page_insert(e->env_pgdir, pp, addr, PTE_U | PTE_W) != 0)
            panic("region_alloc: can't insert page\n");
    }
}
```

- **load_icode():** 加载kernel中内嵌的elf image到env的地址空间。
  - 参照bootmain函数中的代码和注释，将readseg()改为region_alloc()，仅加载ELF_PROG_LOAD类型的程序，调用memmove()拷贝整个file到ph->p_va处，调用memset()将filesz ~ memsz间的空间清零。
  - 观察bootmain加载完毕后直接调用`((void (*)(void)) (ELFHDR->e_entry))();`类似的，将env的tf_eip指向程序入口elf->e_entry。
  - 最后，给程序的初始栈(va = USTAKTOP-PGSIZE)映射一个物理页。

```
for (; ph < eph; ph++) {
    if (ph->p_type == ELF_PROG_LOAD) {  //only load this type
    if (ph->p_filesz > ph->p_memsz)
        panic("load icode: filesz > memsz\n");
    region_alloc(e, (void *)ph->p_va, ph->p_memsz);
    //copy [binary+offset,binary+offset+filesz] to va[p_va]
    memmove((void *)ph->p_va, binary+ph->p_offset, ph->p_filesz);
    //clear remaining memory to zero
    memset((void *)(ph->p_va + ph->p_filesz), 0, ph->p_memsz - ph->p_filesz);
    }
}
e->env_tf.tf_eip = elf->e_entry;
region_alloc(e, (void *)(USTACKTOP - PGSIZE), PGSIZE);
```

- **env_create():** 调用env_alloc()分配一个新的env，设置env type，调用load_icode()加载程序到env，**由于env与kernel在不同页表地址空间，需要在加载前后调用lcr3切换页表，否则会出现kernel page fault。**

```
struct Env *e;
if (env_alloc(&e, 0) < 0)
    panic("env_create: fail alloc env\n");
e->env_type = type;
lcr3(PADDR(e->env_pgdir));
load_icode(e, binary);
lcr3(PADDR(kern_pgdir));
```

- **env_run():** 需要切换到不同env时，先将当前env的status置为NOT_RUNNABLE，切换到新的env、改变status、增加env_runs，调用lcr3()切换地址空间。因为load_icode时设置过env_tf，调用env_pop_tf()后，会进入新的env。

```
if (curenv != e) {  //context switch
    if (curenv && curenv->env_status == ENV_RUNNING)
        curenv->env_status = ENV_NOT_RUNNABLE;
    curenv = e;
    curenv->env_status = ENV_RUNNING;
    curenv->env_runs++;
    lcr3(PADDR(curenv->env_pgdir)); //switch address space
}
env_pop_tf(&curenv->env_tf);
```

Exercise 4:

- **trapentry.S**

- 利用TRAPHANDLER和TRAPHANDLER_NOEC两个宏添加所有trap的入口，例如`TRAPHANDLER_NOEC(divide_handler, T_DIVIDE)`。第一个参数也是trap.c中定义的handler函数名，第二个参数是中断号。 参考intel i386手册9.10 Error Code Summary，以下几个中断有error code，使用宏TRAPHANDLER。

```
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)
```

- 增加_alltraps的定义:

```
.globl _alltraps;
_alltraps:
# 1. 构造Trap frame，push %ds和%es并利用pushal指令将需要的寄存器推到栈上，用来保存目前的状态。
pushl %ds
pushl %es
pushal
# 2. load GD_KD，参考了xv6源码，先movw $GD_KD到%ax，再movw给%ds和%es，从而切换到内核态。
movw $GD_KD, %ax
movw %ax, %ds
movw %ax, %es
# 3. 32位栈传参，pushl %esp作为trap()的参数，调用trap
pushl %esp
call trap
```

- **trap.c**:
  1. 声明所有trap handler函数，如`void divide_handler();`，函数NAME会传给TRAPHANDLER
  2. 在trap_init()中定义IDT，利用宏SETGATE，如`SETGATE(idt[T_DIVIDE], 1, GD_KT, divide_handler,0);`其中INT n, INTO, BOUND指令允许软中断，它们的dpl设为3。

```
SETGATE(idt[T_BRKPT], 1, GD_KT, brkpt_handler,3);
SETGATE(idt[T_OFLOW], 1, GD_KT, oflow_handler,3);
SETGATE(idt[T_BOUND], 1, GD_KT, bound_handler,3);
```

Exercise 5:

trap_dispatch()中，将T_PGFLT的中断分配给`page_fault_handler()`进行处理。

```
if (tf->tf_trapno == T_PGFLT)
    page_fault_handler(tf);
```

Exercise 6:

trap_dispatch()中，遇到breakpoint异常时调用`monitor()`，触发kernel monitor。

```
else if (tf->tf_trapno == T_BRKPT)
    monitor(tf);
```

Exercise 7:

syscall的过程: IDT -> syscall_handler -> _alltraps -> trap() -> trap_dispatch() -> syscall() -> sys_function()

- **trap_init()**：增加handler函数声明和IDT条目：`SETGATE(idt[T_SYSCALL], 0, GD_KT, syscall_handler,3);`
- **trapentry.S**：增加中断入口。
- **trap_dispatch()**：判断syscall中断时，调用syscall()，按注释要求的顺序传入寄存器，并将返回值存于%eax。

```
else if (tf->tf_trapno == 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);
}
```

- 实现kern/syscall.c中的syscall(): `switch (syscallno)`，根据syscallno调用已实现的kernel函数（按序传入需要的参数）并直接返回结果，遇到无效的syscallno返回-E_INVAL。例如：

```
switch (syscallno) {
    case SYS_cputs:
        sys_cputs((char *)a1, (size_t)a2);
        return 0;
    ...
```

Exercise 8:

- 利用`sysenter & sysexit`指令代替`int 0x30`，则现在要求syscall的过程：IDT -> sysenter_handler -> 手动push参数 -> call syscall -> sysenter -> sysexit

```
.globl sysenter_handler;
sysenter_handler:
    pushl %esi          # 栈传参，按照syscall参数逆序push寄存器
    pushl %edi
    pushl %ebx
    pushl %ecx
    pushl %edx
    pushl %eax
    call syscall
    movl %ebp, %ecx     # %ebp返回%esp-> %ecx
    movl %esi, %edx     # %esi返回pc -> %edx，用于执行sysexit
    sysexit
```

- 相应地，在**syscall()**内联汇编中增加sysenter指令和寄存器操作：push所有寄存器 + 执行sysenter + pop所有寄存器。其中sysenter需要手动将返回地址存到%esi、修改%ebp：

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

Exercise 9:

阅读env.h，可知envid_t包含几个部分，ENVX部分是env的index，可用以索引。

```
// An environment ID 'envid_t' has three parts:
// +1+---------------21-----------------+--------10--------+
// |0|          Uniqueifier             |   Environment    |
// | |                                  |      Index       |
// +------------------------------------+------------------+
//                                       \--- ENVX(eid) --/
```

所以用`ENVX(sys_getenvid)`索引到envs[]中的env，将**libmain.c**中的thisenv指向它，从而调用umain时访问thisenv->env_id不会出错。

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

Exercise 10:

- 给Env结构体增加成员变量uint32_t env_break, 用来保存当前program break地址。

```
// load_icode()中初始化env_break，在p_va + p_memsz上方
e->env_break = ROUNDUP(ph->p_va + ph->p_memsz, PGSIZE);
```

- 实现sys_sbrk():

```
// 调用region_alloc()从env_break分配page，向上增长地址，返回更新后的env_break。
static int sys_sbrk(uint32_t inc) {
    region_alloc(curenv, (void *)curenv->env_break, inc);
    curenv->env_break += inc;
    return curenv->env_break;
}
```

Exercise 11

- 在page_fault_handler()中检测是否是内核态，通过tf_cs低位是否置上来判断，是内核态则panic。

```
if (!(tf->tf_cs & 0x3)) {
    panic("kernel-mode page fault\n");
}
```

- 实现user_mem_check()，通过判断地址是否越界（>= ULIM）和页的权限是否为perm来检测，若出错需要将当前出错地址赋值给user_mem_check_addr，要注意该值是起始va时可不对齐，否则要与PGSIZE对齐。

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

- syscall.c：在sys_cputs()中增加assert语句。

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

- kdebug.c: debuginfo_eip()中调用user_mem_check检查usd stabs stabstr。

```
if (user_mem_check(curenv, 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

阅读evilhello2.c的注释，发现ring0_call()的完整代码已经给出，但测试仍然会报unhandled trap。于是寻找bug。发现call_fun_ptr()调用evil()后，`popl %ebp`然后直接`lret`，**没有将%ebp置给%esp，无法正确返回。**

所以应修改`popl %ebp`为：`leave`

```
void call_fun_ptr()
{
    evil();
    *entry = old;
    //movl %ebp, %esp
    //popl %ebp
    asm volatile("leave");
    asm volatile("lret");
}
```