Lab3

ChenZhikai 516370910008

Exercise 1

与lab2分配并且映射内存的方式完全相同

```
envs = boot_alloc(sizeof(struct Env) * NENV);
memset(envs, 0, sizeof(struct Env) * NENV);
boot_map_region(kern_pgdir, UENVS, ROUNDUP(NENV * sizeof(struct Env), PGSIZE),
PADDR(envs), PTE_U | PTE_P);
```

Exercise 2

这里实际上做的就是处理Process的相关代码。

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

这里做的与在pmap里做的有点类似，需要注意的就是序号是反的

```
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;

        // 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);
	p->pp_ref++;
	for(int p = PDX(UTOP); p < NPDENTRIES;p++){
		e->env_pgdir[p] = kern_pgdir[p];
	}
	// 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;
}
```

这里把env的page directory table建立起来，同时设定了一条映射

```
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!)
	uint32_t start_addr = (uint32_t) ROUNDDOWN(va, PGSIZE);
	uint32_t end_addr = (uint32_t)ROUNDUP(va + len, PGSIZE);
	struct PageInfo *page;
	for(uint32_t i = start_addr; i < end_addr;i+=PGSIZE){
		page = page_alloc(0);
		if (!page) {
			panic("Error: Can't allocate new physical pages\n");
		} else {
			if (page_insert(e->env_pgdir, page, (void *)i, PTE_P |
PTE_U | PTE_W)) {
				panic("Error: Can't append into the page
table\n");
			}
		}
```

```
        }
}
```

这里的代码是把va到va+len的部分都分配一个physical page,同时插入到page table里。

```
static void
load_icode(struct Env *e, uint8_t *binary)
{
        // Hints:
        //  Load each program segment into virtual memory
        //  at the address specified in the ELF segment header.
        //  You should only load segments with ph->p_type == ELF_PROG_LOAD.
        //  Each segment's virtual address can be found in ph->p_va
        //  and its size in memory can be found in ph->p_memsz.
        //  The ph->p_filesz bytes from the ELF binary, starting at
        //  'binary + ph->p_offset', should be copied to virtual address
        //  ph->p_va.  Any remaining memory bytes should be cleared to zero.
        //  (The ELF header should have ph->p_filesz <= ph->p_memsz.)
        //  Use functions from the previous lab to allocate and map pages.
        //
        //  All page protection bits should be user read/write for now.
        //  ELF segments are not necessarily page-aligned, but you can
        //  assume for this function that no two segments will touch
        //  the same virtual page.
        //
        //  You may find a function like region_alloc useful.
        //
        //  Loading the segments is much simpler if you can move data
        //  directly into the virtual addresses stored in the ELF binary.
        //  So which page directory should be in force during
        //  this function?
        //
        //  You must also do something with the program's entry point,
        //  to make sure that the environment starts executing there.
        //  What?  (See env_run() and env_pop_tf() below.)

        // LAB 3: Your code here.
        struct Proghdr *ph, *eph;

        struct Elf *elf = (struct Elf *)binary;

        ph = (struct Proghdr *)((uint8_t *)elf + elf->e_phoff);
        eph = ph + elf->e_phnum;
        if (elf->e_magic != ELF_MAGIC){
                panic("Error: The format of ELF is wrong\n");
        }
        lcr3(PADDR(e->env_pgdir));
        for(;ph < eph;ph++){
                if (ph->p_type == ELF_PROG_LOAD){
                        region_alloc(e, (void *)ph->p_va, ph->p_memsz);
```

```
                        memset((void *)ph->p_va, 0, ph->p_memsz);
                        memmove((void *)ph->p_va, (void *)(binary + ph->p_offset),
ph->p_filesz);
                }
        }
        lcr3(PADDR(kern_pgdir));
        e->env_tf.tf_eip = elf->e_entry;
        // Now map one page for the program's initial stack
        // at virtual address USTACKTOP - PGSIZE.
        region_alloc(e, (void *)(USTACKTOP - PGSIZE), PGSIZE);
        e->env_heap_top = (uint32_t)ROUNDDOWN(USTACKTOP - PGSIZE, PGSIZE);
        // LAB 3: Your code here.
}
```

这里是把以elf格式存的二进制文件读到address space里来，比较重要的两条lcr3切换了kernel与env之间的页表，总体上来说与boot loader里面读elf的代码是类似的。

```
void
env_create(uint8_t *binary, enum EnvType type)
{
        // LAB 3: Your code here.
        struct Elf *elf = (struct Elf *)binary;
        struct Env *e;
        if (env_alloc(&e, 0)){
                panic("Error: Not able to create the new env\n");
        }
        load_icode(e, binary);
        e->env_type = ENV_FREE;
}
```

这个代码是新建一个环境并且把elf二进制文件读进去。

```
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.

        // Hint: This function loads the new environment's state from
```

```
	//	e->env_tf.  Go back through the code you wrote above
	//	and make sure you have set the relevant parts of
	//	e->env_tf to sensible values.

	// LAB 3: Your code here.
	if (curenv != e){
		if (curenv){
			curenv->env_status = ENV_RUNNABLE;
		}
		curenv = e;
		curenv->env_status = ENV_RUNNING;
		curenv->env_runs++;
		lcr3(PADDR(curenv->env_pgdir));
	}
	env_pop_tf(&e->env_tf);
}
```

这里的代码是从一个env跳转到另一个env,同时修改状态，换页表

Exercise 4

代码4这里完成的工作在xv6里应该是由脚本做的，我们这里就是手动注册一下idt里面的进入点。

```
TRAPHANDLER_NOEC(ENTRY_DIVIDE, T_DIVIDE)
TRAPHANDLER_NOEC(ENTRY_DEBUG, T_DEBUG)
TRAPHANDLER_NOEC(ENTRY_NMI, T_NMI)
TRAPHANDLER_NOEC(ENTRY_BREAK, T_BRKPT)
TRAPHANDLER_NOEC(ENTRY_OVER, T_OFLOW)
TRAPHANDLER_NOEC(ENTRY_BOUND, T_BOUND)
TRAPHANDLER_NOEC(ENTRY_ILLOP, T_ILLOP)
TRAPHANDLER_NOEC(ENTRY_DEVICE, T_DEVICE)
TRAPHANDLER(ENTRY_DBLFLT, T_DBLFLT)
TRAPHANDLER(ENTRY_TSS, T_TSS)
TRAPHANDLER(ENTRY_SEGNP, T_SEGNP)
TRAPHANDLER(ENTRY_STACK, T_STACK)
TRAPHANDLER(ENTRY_GPFLT, T_GPFLT)
TRAPHANDLER(ENTRY_PGFLT, T_PGFLT)
TRAPHANDLER_NOEC(ENTRY_FPERR, T_FPERR)
TRAPHANDLER_NOEC(ENTRY_ALIGN, T_ALIGN)
TRAPHANDLER_NOEC(ENTRY_MCHK, T_MCHK)
TRAPHANDLER_NOEC(ENTRY_SIMDERR, T_SIMDERR)
TRAPHANDLER_NOEC(ENTRY_SYSCALL, T_SYSCALL)


/*
 * Lab 3: Your code here for _alltraps
 */

.globl _alltraps
_alltraps:
	pushw $0
```

```
        pushw %ds
        pushw $0
        pushw %es
        pushal

        /*Set up the data segment*/
        movl $GD_KD, %eax
        mov %ax, %ds
        mov %ax, %es

        pushl %esp
        call trap
```

这里上面是定义了各种trap的进入点，下面的alltraps与xv6里是一样的，所有的中断都会跳转到这里，保存建立trapframe,再通过trap完成态的跳转。

```
        extern struct Segdesc gdt[];

        // LAB 3: Your code here.
        extern void ENTRY_DIVIDE();
        extern void ENTRY_DEBUG();
        extern void ENTRY_NMI();
        extern void ENTRY_BREAK();
        extern void ENTRY_OVER();
        extern void ENTRY_BOUND();
        extern void ENTRY_ILLOP();
        extern void ENTRY_DEVICE();
        extern void ENTRY_DBLFLT();
        extern void ENTRY_TSS();
        extern void ENTRY_SEGNP();
        extern void ENTRY_STACK();
        extern void ENTRY_GPFLT();
        extern void ENTRY_PGFLT();
        extern void ENTRY_FPERR();
        extern void ENTRY_ALIGN();
        extern void ENTRY_MCHK();
        extern void ENTRY_SIMDERR();
        extern void ENTRY_SYSCALL();
        extern void sysenter_handler();

        SETGATE(idt[0], 0, GD_KT, ENTRY_DIVIDE, 0);
        SETGATE(idt[1], 0, GD_KT, ENTRY_DEBUG, 0);
        SETGATE(idt[2], 0, GD_KT, ENTRY_NMI, 0);
        SETGATE(idt[3], 0, GD_KT, ENTRY_BREAK, 3);
        SETGATE(idt[4], 0, GD_KT, ENTRY_OVER, 3);
        SETGATE(idt[5], 0, GD_KT, ENTRY_BOUND, 3);
        SETGATE(idt[6], 0, GD_KT, ENTRY_ILLOP, 0);
        SETGATE(idt[7], 0, GD_KT, ENTRY_DEVICE, 0);
        SETGATE(idt[8], 0, GD_KT, ENTRY_DBLFLT, 0);
        SETGATE(idt[10], 0, GD_KT, ENTRY_TSS, 0);
        SETGATE(idt[11], 0, GD_KT, ENTRY_SEGNP, 0);
```

```
        SETGATE(idt[12], 0, GD_KT, ENTRY_STACK, 0);
        SETGATE(idt[13], 0, GD_KT, ENTRY_GPFLT, 0);
        SETGATE(idt[14], 0, GD_KT, ENTRY_PGFLT, 0);
        SETGATE(idt[16], 0, GD_KT, ENTRY_FPERR, 0);
        SETGATE(idt[17], 0, GD_KT, ENTRY_ALIGN, 0);
        SETGATE(idt[18], 0, GD_KT, ENTRY_MCHK, 0);
        SETGATE(idt[19], 0, GD_KT, ENTRY_SIMDERR, 0);
        SETGATE(idt[48], 0, GD_KT, ENTRY_SYSCALL, 3);
```

这里是在trap_init()定义的跳转表，注意这里的DPL是0还是3很关键，判断的标准就是去看一下用户态应不应该引发这样的一个trap.

Question 1

对于每一种interrupt类型设定相应的处理函数可以使得我们知道具体是哪一个终端(我们在SETGATE设置的 时候提供了这样的信息)，如果只有一个interrupt handler的话，那我们就不知道来的是哪一个中断， 那就没有意义了,我们也无法调用对应的处理函数。

Question 2

因为这是一个用户态的程序，他想要去引发一个Page Fault, 我们认为这是一个不合理的越级行为，所以 所以触发了保护机制，返回13号中断，如果想要正确工作，可以把dpl改为3,但这样就引发了 安全问题，因为任何一个用户态的程序都有能力去引发一个虚假的Page Fault了。

Exercise 5

```
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;
                }
                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;
```

```
                }
                default:{
                        cprintf("trap no: %d\n", tf->tf_trapno);
                }


        }
        // Unexpected trap: The user process or the kernel has a bug.
        print_trapframe(tf);
        if (tf->tf_cs == GD_KT)
                panic("unhandled trap in kernel");
        else {
                env_destroy(curenv);
                return;
        }
}
```

这里就是写一个中断对应的跳转表了。注意在每一个case后面要直接return,因为下面那个if语句在语义上应该是并列的（只是因为它是原来就提供的，所以没有写在一起）

Exercise 6

就是上面的代码了，调用我们之前的monitor，然后传进去trapframe就可以了。

Question 3

这里其实还是Q2里面提到的DPL的问题，如果你DPL设成了0,那么无论怎么样都是返回Protection, 因为你没有权限；把DPL改为3之后，就是题目里想要的效果了

Question 4

就是Kernel与User态权限的保护。有些是用户态合理的权限，就应该设成3；但是像softint 这种通过用户态发动page fault不合理的权限就应该禁止。这样既保证了用户态的使用也保证了 整个OS的安全性。

Exercise 7

这里有一部分代码还是在上面那一部分已经写出来了。那里做的是interrupt层面的跳转表，这里现在的是system call根据system call number再做一个跳转表

```
// Dispatches to the correct kernel function, passing the arguments.
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((char *)a1, (size_t)a2);
```

```
                        return 0;
                        break;
                case SYS_cgetc:
                        return sys_cgetc();
                        break;
                case SYS_getenvid:
                        return sys_getenvid();
                        break;
                case SYS_env_destroy:
                        return sys_env_destroy((envid_t)a1);
                        break;
                case SYS_map_kernel_page:
                        return sys_map_kernel_page((void *)a1, (void *)a2);
                        break;
                case SYS_sbrk:
                        return sys_sbrk((uint32_t)a1);
                        break;
                case NSYSCALLS:
                default:
                        return -E_INVAL;
        }
}
```

大部分handler都已经提供了，sbrk()是之后的练习里要完成的。

Exercise 8

这是一个很困难的练习，同时grader也没有专门为这个练习做一项，同时这里sysenter与int 0x30两种system call的跳转方法只能二选一，所以我在原本的代码里留的是int 0x30的版本，这里把sysenter的解决方案放出来，但还不确定写的是不是对的。

```
#ifndef HELLO
// This is the code for exercise 8. Since the grader doesn't check this part, I keep the original version(interrupt).
// I'm not sure whether I implement this right or not hhhh
        asm volatile("pushl %%ecx\n"
                                "pushl %%edx\n"
                                "pushl %%ebx\n"
                                "pushl %%esp\n"
                                "pushl %%ebp\n"
                                "pushl %%esi\n"
                                "pushl %%edi\n"
                                "leal after_sysenter_label%=, %%esi\n"
                                "movl %%esp, %%ebp\n"
                                "sysenter\n"
                                "after_sysenter_label%=: \n"
                                "popl %%edi\n"
                                "popl %%esi\n"
                                "popl %%ebp\n"
                                "popl %%esp\n"
                                "popl %%ebx\n"
```

```
                                "popl %%edx\n"
                                "popl %%ecx\n"
                                :"=a" (ret)
                        :"a" (num),
                        "d" (a1),
                        "c" (a2),
                        "b" (a3),
                        "D" (a4)
                    : "cc", "memory");
#endif
```

这一段是用来替代原本的sycall

```
.globl sysenter_handler;
.type sysenter_handler, @function;
.align 2;
sysenter_handler:
        pushl %edi
        pushl %ebx
        pushl %ecx
        pushl %edx
        pushl %eax
        call syscall
        movl %ebp, %ecx
        movl %esi, %edx
        sysexit
```

这一段压栈的顺序是参照了syscall的几个参数

```
    wrmsr(0x174, GD_KT, 0);
        wrmsr(0x175, KSTACKTOP, 0);
        wrmsr(0x176, sysenter_handler, 0);
```

这里使用宏把手册上的几个MSR配置一下，具体的原理并没有搞懂

Exercise 9

```
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();
}
```

这里其实只要添加一行把thisenv赋值好就可以了

Exercise 10

```
static int
sys_sbrk(uint32_t inc)
{
    // LAB3: your code here.
        region_alloc(curenv, (void *)(curenv->env_heap_top - inc), inc);
        curenv->env_heap_top = (uint32_t)ROUNDDOWN(curenv->env_heap_top - inc,
PGSIZE);
    return curenv->env_heap_top;
}
```

这里的sbrk是通过region_alloc()实现的，因为注释里说的其实就是region_alloc的功能。然后这里env_heap_top需要在env的struct里添加一下并且初始化一下，这里原来写的是向上增长的，但是make grade不过，所以还是改成向下的了。

```
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
        uint32_t env_heap_top;

        // Address space
        pde_t *env_pgdir;               // Kernel virtual address of page dir
};
```

```
e->env_heap_top = (uint32_t)ROUNDDOWN(USTACKTOP - PGSIZE, PGSIZE);
```

这一行加在load_icode()里，第一次向用户态data写的地方。

Exercise 11

11、12两个练习其实是在一起的。我们这里要实现的就是保护内核，免受用户态Page Fault的干扰，用户态的Page Fault不能影响到Kernel本身。那么我们首先要判断它到底是哪个态的page fault

```
void
page_fault_handler(struct Trapframe *tf)
{
        uint32_t fault_va;

        // Read processor's CR2 register to find the faulting address
        fault_va = rcr2();

        // Handle kernel-mode page faults.

        // LAB 3: Your code here.

        // We've already handled kernel-mode exceptions, so if we get here,
        // the page fault happened in user mode.
        if ((tf->tf_cs & 0x3) == 0) {
                panic("Kernel page fault");
        }
        // Destroy the environment that caused the fault.
        cprintf("[%08x] user fault va %08x ip %08x\n",
                curenv->env_id, fault_va, tf->tf_eip);
        print_trapframe(tf);
        env_destroy(curenv);
}
```

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

```
                        return -E_FAULT;
                }
        }
        return 0;
}
```

接下来我们实现user_mem_check(),这个函数会检查一名用户对某块VA是否真的有访问权限。这里实现的细节是要做一个ROUNDDOWN,因为没有对齐。

```
static void
sys_cputs(const char *s, size_t len)
{
        // Check that the user has permission to read memory [s, s+len).
        // Destroy the environment if not.

        // LAB 3: Your code here.
        user_mem_assert(curenv, (void *)s, len, PTE_U);
        // Print the string supplied by the user.
        cprintf("%.*s", len, s);
}
```

然后在sys_cputs()里检查user memory的权限。 然后再去调用这个buggyhello,就会得到page fault了。报错的原因也不难理解，buggyhello想要向地址1去写数据，那么这么低的地址肯定不是你能写的，所以就报了page fault.

Exercise 13

最后一个部分不知道是不是助教同学忘了把答案删了，我看了半天发现怎么已经实现了？？？但是跑make grade跑不过，最后改了一行指令就能通过了，具体就是下面这个函数

```
void call_fun_ptr()
{
    ring_0_call();
    *entry = old;
    asm volatile("leave");
    asm volatile("lret");
}
```

第三行改成了`leave`就能过了。