OS Lab3 设计文档

516030910293　　姚子航

一、User Environments & Exception Handling

在本次的 lab 中，JOS 使用结构体 Env，即 Environment，来代替通常操作系统中的进程，而 Env 与 Process 不同的一点在于，在 JOS 中所有 Env 共用一个内核栈，而在 Linux 系统中，每个进程都有自己的内核栈。这样的设计是由于在同一时间只能有一个 Env 在内核态工作，所以使它们共用一个内核栈节省了空间，也简化了内存布局设计。

在完成 Exercise1 时，由于 envs 的设计与 pages 非常相似，所以仿照 pages 的代码就可以将 envs 的初始化代码补全，通过测试。

在完成 Exercise2 的时候，我们需要理解第一个用户程序 helloworld 是如何运行的，我们根据代码可以一步一步弄清楚流程：在 i386_init()函数中，调用 ENV_CREATE 宏创建 Env，进入这个宏，可以看到它调用了 env_create 函数，

```
#define ENV_CREATE(x, type)                             \
do {                                            \
        extern uint8_t ENV_PASTE3(_binary_obj_, x, _start)[]; \
        env_create(ENV_PASTE3(_binary_obj_, x, _start),         \
                type);                          \
} while (0)
```

这里有一个地方需要注意，在这个宏中，我们将第一个参数与_binary_obj_拼凑成用户程序的目标文件名称，在 lab 文档中已经说明了这些 symbol 如何创建以及用户程序是如何链接在内核中的，在此不再赘述，接着往下看，我们可以看到在 env_create 函数中，依次调用了 env_alloc 与 load_icode 函数，其中 env_alloc 从 env 的空闲列表中取了一个 env，并初始化它的各个属性以及虚存页表，在 load_icode 中，我们将用户程序的 elf 文件加载到新创建的 Env 的内存空间中，回到 i386_init 函数中，使用 env_run 函数运行该 env，完成了用户程序的运行初始化工作。我们只需要根据各个函数的注释 hint，补全代码即可，这里有几个问题需要注意：1.由于运行刚创建的 Env 也需要调用 env_pop_tf 函数,所以我们需要在 alloc 的时候就初始化好这个 Env 的 trapframe，使得它正确初始化初始状态. 2.在构建每个 env 的页表时，我选择了直接拷贝 kernel 页表中的 pde 目录项，这样公用不变的内核部分页表，节省了内存。

之后我们需要设置中断描述符表，我直接参考了 xv6 系统的中断设置，完成了中断表的设置，并且根据 TRAPHANDLER 与 TRAPHANDLER_NOEC 宏创建了中断的处理程序，这里对于每个中断是否会 push errorcode 我是通过查找 Intel 文档得到了答案。在编写_alltraps 函数的时候，需要注意一个问题，我们不可以直接通过 movw 来给 ds、es 寄存器赋值，所以我通过 push pop 这样一个 trick 来给这两个寄存器赋值。下面附上函数实现：

```
_alltraps:
        pushw $0
        pushw %ds
        pushw $0
        pushw %es
        pushal
        pushw $GD_KD
```

```
pushw $GD_KD
popw %ds
popw %es
pushl %esp
call trap
```

在设置中断描述符表 IDT 的时候，我们需要注意两个问题，第一个问题是 Gate 的种类，因为在 trap 函数中会先检查中断是否关闭，所以我们设置的 IDT 描述符都是 Interrupt Gate 而非 Trap Gate，这样在进入中断处理程序的时候，系统会自动帮我们关闭中断，而不需要我们手动调用 cli 指令关闭中断。第二个问题是 DPL，因为在我们的测试中有 breakpoint 测试需要使用 int3 指令触发 breakpoint 软中断，而要想实现这样的功能就需要我们将 interrupt gate 中的 DPL 位设置为 3，这样我们才可以使用 int 3 指令。这样就完成了 partA 的部分。

二、Page Faults, Breakpoints Exceptions, and System Calls

Exercise5 和 Exercise6 比较简单，就是对 trap_dispatch 函数做一些补充使得能够处理 page fault 和 breakpoint，我们根据传进来的 trap frame 的异常号就可以判断异常的类型。

Exercise7 是关于 system call，我梳理了一下用户是如何进行系统调用的：在 lib/syscall.c 中提供了一些系统调用函数给用户来调用，以 sys_getenvid 为例子：用户调用 getenvid 后，会调用 syscall 函数，该函数通过下面的代码出发了系统调用的软中断：

```
asm volatile("int %1\n"
             : "=a" (ret)
             : "i" (T_SYSCALL),
               "a" (num),
               "d" (a1),
               "c" (a2),
               "b" (a3),
               "D" (a4),
               "S" (a5)
             : "cc", "memory");
```

我们可以看到 int T_SYSCALL 命令的调用就进入了 trap 函数中，并且将系统调用编号存入 eax 寄存器，参数依次存入 edx, ecx, ebx 等寄存器。在 trap_dispatch 函数中编写调用 syscall 的函数，通过上面的内联汇编，我们可以通过 trap frame 的各个寄存器来访问 sysstem call 的类型及参数，并且将参数存放在 eax 寄存器中。注意为了能够使用 int 触发系统调用，我们也需要将系统调用的 interrupt gate DPL 位设置为 3.

Exercise8 中要求实现 Intel 引入的系统调用指令 sysenter 与 sysexit, 实现这两个指令需要阅读英特尔关于这两个指令的文档，通过阅读文档，我们得知在使用 sysenter 指令之前需要设置三个 Model specific register，分别是 IA32_SYSENTER_CS 用来存放 Ring0 的代码段选择符、IA32_SYSENTER_EIP 用来存放 sysenter_handler 的地址、IA32_SYSENTER_ESP 用来存放 Ring0 的栈基地址，我使用文档中给出的帮助函数设置 MSR，代码如下：

```
void msr_init(){
    wrmsr(0x174, GD_KT, 0);
    wrmsr(0x175, KSTACKTOP, 0);
    wrmsr(0x176, sysenter_handler, 0);
}
```

设置完 MSR 后，我们可以编写 sysenter 版本的 syscall 函数，内联代码如下：

```
asm volatile(
            "\tpushl %%esi\n"
            "\tpushl %%ebp\n"
            "\tmovl %%esp,%%ebp\n"
            "\tleal after_sysenter_label, %%esi\n"
            "\tsysenter \n"
            "after_sysenter_label:\n"
            "\tpopl %%ebp\n"
            "\tpopl %%esi\n"
          : "=a" (ret)
          : "i" (T_SYSCALL),
            "a" (num),
            "d" (a1),
            "c" (a2),
            "b" (a3),
            "D" (a4)
          : "cc", "memory");
```

可以看到，这段代码先保存了要使用的寄存器，接着按照文档上的提示设置好了 ebp 和 esi 两个寄存器，之后调用 sysenter 指令进入 syscall 执行，结束后恢复寄存器，就完成了整个系统调用。sysenter_handler 的代码如下：

```
.globl sysenter_handler;
.type sysenter_handler, @function;
.align 2;
sysenter_handler:
    pushl %ebp
    pushl %esi
    pushl $0
    pushl %edi
    pushl %ebx
    pushl %ecx
    pushl %edx
    pushl %eax
    call syscall
    addl $24, %esp
    popl %edx
    popl %ecx
    sysexit
```

首先为了使设置 MSR 的时候可以找到，我将这个 symbol 设置为 global，在 handler 处理的过程中有一个重要的点是从英特尔文档中的值，在 sysexit 指令调用时，CPU 会将 ecx 和 edx 的值赋值给 esp 和 eip，而在调用时我们将这两个值存储在 ebp 和 esi 中，所以我们通过两个 push&pop 完成这一要求。我将 syscall.c 中的系统调用都换成使用 sysenter 版本的函数，make grade 依然满分，说明我的程序是可用的。

Exercise9 相对简单，只需要在 libmain 中使用系统调用获取到当前的 Env 就可以通过测试。只需要一行代码：thisenv = &envs[ENVX(sys_getenvid())];

Exercise10 要求实现 sbrk 的系统调用，在实现这个功能时，我给 struct Env 添加了一个 env_break 属性，在 Env 初始化时将 break 初始化为 UTEXT，之后在 load_icode 函数中，我

在每次 load program 时更新一个全局变量，最终将堆的初始位置初始化为最终的变量值，之后每次调用 sbrk 的时候通过 page_insert 来分配堆内存。下面附上 sbrk 代码：

```
if(inc == 0) return curenv->env_break;
for(int i = 0; i < ROUNDUP(inc, PGSIZE)/PGSIZE; i++){
    struct PageInfo* p = page_alloc(1);
    if(p == NULL) return -E_NO_MEM;
    int r = page_insert(curenv->env_pgdir, p, (void*)(curenv->env_break + i*PGSIZE),
                        PTE_U | PTE_W);
    if(r < 0) return r;
}
curenv->env_break += ROUNDUP(inc, PGSIZE);
return curenv→env_break;
```

Exercise11 的实现是通过使用 user_mem_check 在 system call 中与 debug.c 中实现 check 的功能，这个 Exercise 最后提到的一个问题非常有趣，当我在我的 JOS 中运行 breakpoint 并调用 backtrace 的时候，它打出的 backtrace 分别是 monitor.c, trap.c, syscall.c, entry.S，与期望中的 backtrace 并不相同，我针对这个问题思考了一段时间，最终得出结论，monitor.c 与 trap.c 的 backtrace 并无问题，而在编译的时候 trap 函数将 trap_dispatch 函数内联进来，这就是为什么看不到 trap_dispatch 函数，而在 trap 的过程我们是通过_alltraps_中的 call trap 函数进入 trap，而此时在内核栈上的返回地址是 call trap 的下一条指令，通过查看汇编代码可以发现下一条指令就是 syscall 函数的开始，这也就是为什么我们的 backtrace 会回溯到 syscall 函数，而 syscall 函数之后是 entry.S, 这是因为在从用户态进入内核态直到调用 trap 函数的过程中，ebp 寄存器的值一直没有改变，因而把用户栈的 old ebp 的位置保存在了 trap 函数的栈上，使得我们可以从内核栈一路追溯到用户栈，直到用户程序的入口点 Entry.S 中的指令。最终产生 page fault 的原因是我的 backtrace 试图追溯函数的参数，不管是否存在该参数（因为它无法获知参数数量），它总会试图输出 5 个参数。当追溯到 libmain 函数时，由于 libmain 函数只有两个参数，再往前追溯则会越过用户的栈进入无效的虚拟地址区域，所以 backtrace 会引发 page fault。

最后一个 Exercise 难度较大，需要实现从 Ring 3 到 Ring 0 的转换，在这里我通过 call gate 实现这个优先级的切换。这里的设计分为几个步骤进行：1.使用 sgdt 指令将 GDTR 保存在内存中 2.使用 sys_map_kernel_page 系统调用将 GDT 映射到一个虚拟地址上 3.在 GDT 中通过宏设置一个 call gate，注意这里需要用一个全局变量保存被替换的段，之后结束调用后还需要将保存的段替换回去 4.通过 lcall 指令进入 Ring 0 权限，并根据 call gate 的设置执行 fun_ptr 函数 5.函数调用结束后，我们需要使用 leave 和 lret 指令从 call gate 返回到 Ring 3 用户态，再将原先的 GDT 恢复，就完成了整个的调用过程。