Jos Lab 3: User Environments

Li Xinyu 515030910292

Part A: User Environments and Exception Handling

Allocating the Environments Array

> **Exercise 1**
> 修改 `mem_init()` 的代码，调用 `boot_alloc` 分配大小为 `sizeof(struct Env) * NENV` 用来映射 `envs` 数组。然后调用 `boot_map_region(kern_pgdir, UENVS, env_size, PADDR(envs), PTE_U);` 将 `envs` 映射在起始地址为 `UENVS` 的内存区域，并且设置权限为用户只读。

Creating and Running Environments

> **Exercise 2**
> `env_int()` 从最后一个元素开始遍历数组 `envs`，每次循环时将其中的元素 `env_id` 设置为0，`env_status` 设置为 `ENV_FREE`，并将 `env_link` 指向 `env_free_list`，更新 `env_free_list` 指向当前元素。这样在循环结束时 `env_free_list` 就指向 `envs` 的头一个元素(为了保证init.c中的 `env_run(&envs[0])` 能够正常运行)。
> `env_setup_vm()` 中将 `p->pp_ref` 加一，然后将 `env_pgdir` 设置为 `page2kva(p)` ,然后将[PDX(UTOP), NPDENTRIES)范围内的每一个 `e->env_pgdir[i]` 设置为 `kern_pgdir[i]`，最后设置 `e->env_pgdir[PDX(UVPT)] = PADDR(e->env_pgdir) | PTE_P | PTE_U` 将 `e->env_pgdir` 本身映射到页目录中条目 `PDX(UVPT)`，且设置权限为只读。
> `region_alloc()` 首先调用 `ROUNDDOWN(va, PGSIZE)` 得到起始地址，`ROUNDUP(va+len, PGSIZE)` 得到结束地址。然后遍历这个区间，每次调用 `page_alloc(0)` 分配一个页，如果分配失败则调用 `panic`，否则调用 `page_insert()` 将分配的页插到页表中，设置权限为 `PTE_W|PTE_U`，如果插入失败则调用 `panic`。
> `load_icode()` 参考 `boot_main` 的代码，首先将 `binary` 转换为 `struct Elf*`，然后判断 `elf->e_magic` 是否为 `ELF_MAGIC` ,否则 `panic`，是则加载 `elf` 中的每个 `p_type` 为 `ELF_PROG_LOAD` 的段，在加载前调用 `lcr3(PADDR(e->env_pgdir))` 将内容加载到用户空间。调用 `region_alloc` 分配从 `ph->p_va` 开始的大小为 `ph->p_memsz` 的空间，调用 `memset` 将其初始化为0，然后调用 `memmove` 将从 `binary+ph->p_offset` 开始大小为 `ph->p_filesz` 的内容加载到之间分配的空间内。加载完成后将 `e->env_tf.tf_eip` 设置为 `elf->e_entry`，然后调用 `region_alloc` 为栈分配大小为 `PGSIZE` 的初始空间，起始地址为 `USTACKTOP - PGSIZE`，最后调用 `lcr3(PADDR(kern_pgdir))` 将 `CR3` 重新设置为 `kern_pgdir` 的物理地址。

`env_create()` 首先调用 `env_alloc` 分配一个 `struct Env`，如果失败则 `panic`，否则调用 `load_icode` 加载ELF文件，最后将 `env_type` 设置为参数 `type`。
`env_run()` 判断 `curenv` 是否于参数 `e` 相等，否则判断 `curenv` 是否为空且其 `env_status` 是否为 `ENV_RUNNING`，是则将状态设置为 `ENV_RUNNABLE`，然后将 `e` 的状态设置为 `ENV_RUNNING`， `env_runs` 加一，并将 `curenv` 设置为 `e`，然后调用 `lcr3(PADDR(curenv->env_pgdir))` 加载页表。最后无论 `curenv` 是否于参数 `e` 相等，都调用 `env_pop_tf(&curenv->env_tf)` 开始执行用户程序。

Setting Up the IDT

Exercise 4
在文件 `trapentry.S` 中根据 80386 Programmer's Manual中的Table 9-7. Error-Code Summary使用TRAPHANDLER和TRAPHANDLER_NOEC定义各个trap的trap_handler。根据定义函数 `_alltraps`：根据 `struct Trapframe` 的定义，需要在栈上压入 `%ds`、`%es`、和 `struct PushRegs` 中的寄存器值，以及 `tf_padding1` 和 `tf_padding2`。然后使用 `movw` 将 `GD_KD` 加载到 `%ds` 和 `%es` 中，然后调用 `pushl %esp` 传入 `struct Trapframe*`，最后 `call trap` 调用 `trap` 函数。
在函数 `trap_init` 中使用 `extern void th_ ## name` 和 `SETGATE` 来初始化IDT，注册每个trap的entrypoint。

Challenge：

使用指示符 `.text` 和 `.data` 将 `TRAPHANDLER` 修改为：

```
#define TRAPHANDLER(name, num)                  \
.data;\
  .long name;\
.text;\
  .globl name;    /* define global symbol for 'name' */ \
  .type name, @function;  /* symbol type is function */   \
  .align 2;   /* align function definition */   \
  name:     /* function starts here */    \
  pushl $(num);             \
  jmp _alltraps
```

`TRAPHANDLER_NOEC` 也做出相应的修改，
并新增一个宏定义 `TRAPHANDLER_RESERVED`：

```
#define TRAPHANDLER_RESERVED()
.data;\
  .long 0;
```

用于在table中占位(因为trapnum 9 和 15作为保留不被使用)。在使用宏定义 `TRAPHANDLER` 和 `TRAPHANDLER_NOEC` 生成 `entrypoint` 之前，添加指示符如下

```
.data
.globl vectors;
vectors:
```

来生成vector table。
同时修改 `trap_init`，首先获取global的vector table– `extern int vectors[]`，然后使用循环遍历 `vectors` 调用 `SETGATE(idt[i], 1, GD_KT, vectors[i], 0)` 注册entrypoint。

Part B: Page Faults, Breakpoints Exceptions, and System Calls

Handling Page Faults

Exercise 5
修改 `trap_dispatch()`，添加 `switch` 语句，如果 `tf->tf_trapno` 等于 `T_PGFLT`，调用 `page_fault_handler(tf)` 处理page fault。

System calls

Exercise 6
在文件 `trapentry.S` 中添加 `sysenter_handler` 的定义，根据提示首先在栈上压入0用作第五个参数，然后依次压入 `%edi`，`%ebx`，`%ecx`，`%edx`，之后使用 `call` 指令调用 `syscall`。在 `syscall` 返回后，使用 `movl` 指令将 `%ebp` 中存储的 `return esp` 传给 `%ecx`，将 `%esi` 中存储的 `return PC` 传给 `%edx`，然后调用 `sysexit` 退出。
在文件 `kern/init.c` 中设置MSRs如下：

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

其中 `wrmsr` 位于 `inc/x86.h` 中，定义如下：

```
static __inline void
wrmsr(uint32_t msr, uint32_t low, uint32_t high)
{
  asm volatile("wrmsr\n"
       : : "c" (msr), "a"(low), "d" (high));
}
```

在文件 `lib/syscall.c` 中添加代码如下：

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

在文件 `kern/syscall.c` 的函数 `syscall` 中添加 `switch` 语句，根据参数 `syscallno` 的值调用对应的sys_*函数，默认返回 `-E_INVAL`。

User-mode startup

> **Exercise 7**
> 在函数`libmain`中将`thisenv`设置为`envs + ENVX(sys_getenvid())`，使其指向当前的env。

-

> **Exercise 8**
> 在文件`inc/env.h`中，为`struct Env`添加变量`uint32_t env_break`用于记录当前的program break位置。然后在`load_icode`的代码中添加关于`env_break`的代码如下：

```
if (ph->p_va + ph->p_memsz > e->env_break) {
  e->env_break = ROUNDUP(ph->p_va + ph->p_memsz, PGSIZE);
}
```

> 在函数`sys_sbrk`中，首先调用`ROUNDDOWN(va, PGSIZE)`得到起始地址，`ROUNDUP(va+inc, PGSIZE)`得到结束地址。然后遍历这个区间，每次调用`page_alloc(0)`分配
> 一个页，如果分配失败则调用`panic`，否则调用`page_insert()`将分配的页插到页表中，设置权限为`PTE_W|PTE_U`，如果插入失败则调用`panic`。最后将`curenv->env_break`设置为结束地址，并将其返回。

The Breakpoint Exception

> **Exercise 9**
> 在`trap_dispatch()`的`switch`语句中添加`case T_BRKPT: case T_DEBUG:`，如果`tf->tf_trapno`的取值是这两种情况，则调用`monitor(tf)`进入控制台。
> 在文件`monitor.c`的`commands[]`添加三条命令：`c`、`si`和`x`，同时添加对应的三个函数：`mon_continue`、`mon_stepinto`和`mon_display`。
> 根据80386 Programmer's Manual中的System Flags, TF(Trap Flag)被置为1时会使处理器进入单步执行的调试模式。
> `mon_continue`将`tf->tf_eflags`的TF置为0，然后调用`env_run(curenv)`继续运行程序。
> `mon_stepinto`将`tf->tf_eflags`的TF置为1，然后使用`debuginfo_eip`打印出当前的程序运行信息，最后调用`env_run(curenv)`继续运行程序。
> `mon_display`将传入的虚拟地址转换为(uint32_t*)，然后解引用即取出该地址处的字节内容，并将其输出到控制台。

Page faults and memory protection

> **Exercise 10**
> 在函数`page_fault_handler`中添加语句判断`tf->tf_cs & 0x3`是否为0，是则说明内核中发生了page fault，于是调用`panic`。
> 在函数`user_mem_check`中遍历[va, va+len)的区间，如果`cur_addr`不低于`ULIM`，则

将 `user_mem_check_addr` 设置为 `cur_addr` 并返回 `-E_FAULT`；否则调用 `pgdir_walk` 找到当前地址的PTE，若PTE为空或权限不够(即 `(*pte & perm) != perm`)则将 `user_mem_check_addr` 设置为 `cur_addr` 并返回 `-E_FAULT`。

在函数 `sys_cputs` 调用 `user_mem_assert(curenv, s, len, PTE_U | PTE_P)` 检查权限。

在函数 `debuginfo_eip` 中调用 `user_mem_check` 检查权限：

```
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 12

在文件 `evilhello.c` 中申明几个全局变量如下：

```
char gdtpd[PGSIZE];
struct Segdesc oldent;
struct Segdesc* entry;
void (*gfun_ptr)(void);
```

在函数 `ring0_call` 中，首先通过 `sgdt(&gdtd)` 获取gdt的内容，然后调用 `sys_map_kernel_page` 将gdt的内容映射到用户态地址空间 `gptpd` 中，然后获取gdt的起始位置为 `struct Segdesc* gdt = (struct Segdesc*)(PGNUM(gdtpd)<<PTXSHIFT) + PGOFF(gdtd.pd_base)` ，然后得到 `entry` 的位置为 `gdt + (GD_UT >> 3)` ，并在修改前将 `entry` 中的内容保存到 `oldent` 中以便恢复，同时将 `gfun_ptr` 设置为 `fun_ptr` 。之后调用 `SETCALLGATE(*(struct Gatedesc*)entry, GD_KT, call_fun_ptr, 3)` 修改 `entry` 中的内容，然后使用 `lcall` 指令– `asm volatile("lcall $0x18, $0")` 进入包装函数 `gfun_ptr` 如下：

```
void call_fun_ptr()
{
  gfun_ptr();
  *entry = oldent;
  asm volatile("leave\n\t"
      "lret\n\t");
}
```

函数 `call_fun_ptr` 首先通过全局函数指针 `gfun_ptr` 进行函数调用，然后通过指令 `leave` 和 `lret` 退出返回。