JOS-Lab-3 实验报告

熊伟伦
5120379076
azardf4yy@gmail.com

2014年10月29日-11月3日

Contents

1 前言

该报告描述了我在lab3实验的过程中遇到的问题与解决的方法，介绍了lab3的整体结构。指导中问题的解答参考上传的压缩包中的answers-lab3.txt文件。

2 User Environments and Exception Handling

这一部分首先实现User Environments相关的空间分配，地址映射。这一部分主要和kern/env.c文件有关。

2.1 Environment State

这一部分主要是讲解说明，lab的材料说明的十分详细。唯一比较奇怪的是为何下方会有env_cr3的说明出现，源代码中明明没有这个变量。不过根据我的推测，可能是env_pgdir和env_cr3的作用重复了，就删掉了这个变量。

2.2 Allocating the Environments Array

这一部分包含了exercise-1，需要在物理内存中分配一块给env链表使用，并且映射到相应的虚拟内存空间。 首先是物理内存分配，调用boot_alloc函数。

kern/pmap.c

```
1 envs = boot_alloc(NENV * sizeof(struct Env));
```

代码比较简单，分配一块NENV(1024)个的大小为Env的空间，内存空间头赋给envs。boot_alloc函数是在lab2中实现的，会自动按PGSIZE对齐。

kern/pmap.c

```
1 boot_map_region(kern_pgdir, UENVS,
2                 ROUNDUP(NENV*sizeof(struct Env), PGSIZE),
3                 PADDR(envs), PTE_U | PTE_P);
```

然后需要映射到虚拟内存中的UENVS段，权限位按照memlayout.h以及注释中说明的设置，用户能够读取这一部分的内容。

这样这一部分的exercise应该算完成了，运行make qemu，显示check_kern_pgdir()和check_page_installed_pgdir()成功。该部分完成，成功分配一块内存用于env。

2.3 Creating and Running Environments

首先资料说明了由于JOS目前还没有文件系统，所以用户环境需要直接读入ELF二进制文件。接下来又介绍了一系列整个lab如何组织编译读取ELF文件的。因为这一部分的

代码与读取ELF文件有关。接下来是exercise-2。

这一部分比较长，是整个组建用户环境的代码，也比较复杂。

首先是env_init函数

kern/env.c

```
void
env_init(void)
{
    // Set up envs array
    // LAB 3: Your code here.
    int32_t i;
    env_free_list = NULL;
    for (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();
}
```

参照注释，env_free_list是整个env的链表头，并且注释要求env_alloc返回envs数组的第0个元素，即envs[0]。因此为了符合这个规律习惯，我将链表按照地址大小从小到大链起来。

因此从后面开始初始化，分别设置env_status，env_id，并且将env_link指向地址更大的一个Env结构，再将env_free_list指向该块。最终初始化完毕。

然后又调用了env_init_percpu()，重新设置了段寄存器的权限使用属于kernel还是user，因为这一部分不需要我们实现，就不详细讨论了。

接下来是env_setup_vm函数，全称是setup kernel virtual memory layout for env。

kern/env.c

```
static int
env_setup_vm(struct Env *e)
{
    int i;
    struct Page *p = NULL;

    if (!(p = page_alloc(ALLOC_ZERO)))
        return -E_NO_MEM;

    e->env_pgdir = page2kva(p);
    p->pp_ref++;
    for (i = PDX(UTOP); i < NPDENTRIES; i++){
```

```
13          e->env_pgdir[i] = kern_pgdir[i];
14      }
15
16      e->env_pgdir[PDX(UVPT)] = PADDR(e->env_pgdir)
17                                | PTE_P | PTE_U;
18
19      return 0;
20  }
```

按照注释要求，首先分配一块页作为一个Env的env_pgdir使用，该页被引用次数加1。再将UTOP以上的位置从kern_pgdir中复制到env_pgdir中，以便env能访问这些位置(部分虽然没有权限)。

我测试输出得到PDX(UTOP)为955，也就是i的范围是955到1023。

然后又单独设置，将该环境本身的env_pgdir赋给env_pgdir的PDX(UVPT)位置，这4MB的虚拟空间刚好对应env_pgdir。

需要明确注意的是，e->env_pgdir本身的值是指向内核虚拟地址中的一个页，也就是物理内存地址加上0xF0000000。而整个env的pgdir的4kb大小保存的是指向这个环境的虚拟地址的值的pte_t。他们的domain不一样，这是我的理解。虽然他们在UTOP之上的值是完全一样的。

接下来是region_alloc函数。

kern/env.c

```
1   static void
2   region_alloc(struct Env *e, void *va, size_t len)
3   {
4       // LAB 3: Your code here.
5       // (But only if you need it for load_icode.)
6       //
7       // Hint: It is easier to use region_alloc if the caller can pass
8       //   'va' and 'len' values that are not page-aligned.
9       //   You should round va down, and round (va + len) up.
10      //   (Watch out for corner-cases!)
11
12      uint32_t va_start = (uint32_t)ROUNDDOWN(va, PGSIZE);
13      uint32_t va_end = (uint32_t)ROUNDUP(va+len, PGSIZE);
14      struct Page *cur_page;
15
16      uint32_t i;
17      for (i = va_start; i < va_end; i += PGSIZE) {
18          cur_page = page_alloc(0);
19          if (!cur_page) {
20              panic("env alloc page but out of memory\n");
21          } else {
22              if (page_insert(e->env_pgdir, cur_page,
23                  (void*)i, PTE_U | PTE_W))
24                  panic("insert page failed\n");
25          }
```

```
26      }
27  }
```

虽然资料后面说panic可以使用 %e来表示错误值，但我更喜欢用自己的语言表达，文档这里由于显示问题我将panic的内容删减了，源代码中稍有不同。

这个函数的作用是在用户环境下，实际分配一块内存。首先对齐得到用户环境下的虚拟地址的范围，然后分配物理页，注意，这里调用了page_alloc是真实分配物理内存，物理内存会真的减少。然后调用page_insert函数修改env_pgdir的信息，更新刚刚分配的新的页的信息。这里的panic的判断我用了比较省行数的写法。

下一个函数是load_icode，比较麻烦，主要是读取ELF文件到用户环境下。

kern/env.c

```
1  static void
2  load_icode(struct Env *e, uint8_t *binary, size_t size)
3  {
4      // LAB 3: Your code here.
5      struct Proghdr *ph, *eph;
6      struct Elf *elf = (struct Elf*)binary;
7
8      if (elf->e_magic != ELF_MAGIC)
9          panic("load_icode failed: ELF format error");
10
11     ph = (struct Proghdr*)((uint8_t*)elf + elf->e_phoff);
12     eph = ph + elf->e_phnum;
13
14     lcr3(PADDR(e->env_pgdir));
15     for(; ph < eph; ph++) {
16         if (ph->p_type == ELF_PROG_LOAD) {
17             region_alloc(e, (void*)ph->p_va, ph->p_memsz);
18             memmove((void*)ph->p_va,
19                     (void*)(binary+ph->p_offset),
20                     ph->p_filesz);
21             memset((void*)(ph->p_va+ph->p_filesz),
22                    0, (ph->p_memsz-ph->p_filesz));
23         }
24     }
25     lcr3(PADDR(kern_pgdir));
26
27     e->env_tf.tf_eip = elf->e_entry;
28
29     // LAB 3: Your code here.
30     region_alloc(e, (void*)(USTACKTOP-PGSIZE), PGSIZE);
31     return;
32 }
```

一开始我是写不来的，很不好入手，毕竟ELF是个很麻烦的东西，但根据注释参照boot_main读取ELF的方式，我就照抄。

最值得注意的一点是调用了两次lcr3函数，在上面一个函数我说到了用户环境的虚

拟地址和内核的虚拟地址的区别。这个lcr3的调用就是因为这个区别。

因为调用这个函数肯定是从内核态开始读取ELF文件，相当于一个内核打开一个新进程的过程，使用的是kern_pgdir。但是在读取ELF文件的数据的时候，应该是在用户态进行的，因为还要调用region_alloc函数，所以之前调用lcr3读取env_pgdir是切换到用户环境的虚拟内存地址中。

在读取完毕后又切回到kernel的虚拟内存地址中。lcr3的参数是物理地址，因此需要用PADDR转换一下。

根据注释，还需要设置入口elf->e_entry。

最后还需要分配一个实际的页用于用户进程的栈，因为栈是向下长的，所以第一个PGSIZE就是从 USTACKTOP-PGSIZE开始的，有一点奇怪的是如果每次load_icode都需要分配这个内核态的虚拟页给Env，多个进程同时需要创建会如何，这个应该在后面会出现解决方案，暂时先不考虑。

接下来是env_create，我依旧没使用panic的%e。

kern/env.c

```
1  void
2  env_create(uint8_t *binary, size_t size, enum EnvType type)
3  {
4      // LAB 3: Your code here.
5      struct Env *e;
6      int t = env_alloc(&e, 0);
7
8      if (t == -E_NO_MEM) {
9          panic("env_alloc out of memory\n");
10         return;
11     }
12     if (t == -E_NO_FREE_ENV) {
13         panic("env_alloc no more env to use\n");
14         return;
15     }
16     load_icode(e, binary, size);
17     e->env_type = type;
18     return;
19 }
```

这个函数就是先调用env_alloc在内存中组织好一个Env的空间结构，然后调用load_icode读取ELF文件。

其中env_alloc函数已经帮我们写好了，主要是调用开始实现的env_setup_vm函数，并且再设置传入的Env的各个参数，维护好env_free_list，设置保存的寄存器等。最后会cprintf一下创建新env成功，也就是资料中说的cprintt这句话就说明这一个exercise完成了。总而言之就是为还没载入ELF的新Env组织好Env链表中的信息。

最后是env_run。

kern/env.c

```
1  env_run(struct Env *e)
2  {
3      // LAB 3: Your code here.
4
5      //panic("env_run not yet implemented");
6
7      if (curenv != e) {
8          if (curenv && curenv->env_status == ENV_RUNNING)
9              curenv->env_status = ENV_RUNNABLE;
10         curenv = e;
11         curenv->env_status = ENV_RUNNING;
12         curenv->env_runs++;
13         lcr3(PADDR(curenv->env_pgdir));
14     }
15     env_pop_tf(&curenv->env_tf);
16 }
```

这个函数就是切换进程的函数。按照注释一步一步来。

首先判断是否是切换进程，如果是并且当前进程是ENV_RUNNING，则变成ENV_RUNNABLE，然后将curenv换成e，改变curenv的状态为ENV_RUNNING，env_runs加1，切换到该进程的虚拟地址空间。最后调用env_pop_tfduqu读取该进程之前保存的寄存器的信息并且跳转到新的进程。

env_pop_tf函数是汇编内联，并且调用了popal，比较复杂，虽然明白这个函数的作用，但要完全说清楚很难。大致是先设置esp，之后popal将寄存器数据从栈中读出，最后iret跳转到用户进程的入口。

这个时候exercise-2的代码应该算是完成了，运行make qemu，出现了env_alloc中cprintf的信息。显示[00000000] new env 00001000。并且无限重启，根据资料应该会死triple fault导致的重启。Move on!

接下来资料告诉我triple fault的来由，并且让我使用gdb测试是否进入syscall。

按照提示，我根据生成的.asm文件，分别在env_pop_tf的入口0xF0103940和syscall的入口0x00800D09设置break point，成功进入0x00800D09，说明这个syscall调用成功了，中断成功调用。

根据分析，这个syscall是hello.c中调用cprintf出现的，至于为什么没能继续执行，是因为我的handle interrupt的功能还没写。Move on!

2.4 Handling Interrupts and Exceptions

这里给出的第一个链接很简单明了的介绍了中断和异常，夏老师在上课时也花了很长时间进行了介绍。

简单的说，exception，又叫trap gate通常用于用户控制的syscall，比如debug调试的时候，而interrupt通常是CPU调度进程切换的时候使用的。而且trap gate不会屏蔽其他中断，在执行trap gate调用的代码的过程中会被其他中断抢占，而interrupt会屏蔽其他中断，就像lab1中boot的时候一样进行的寄存器位设置进行控制这个特性。

2.5 Basics of Protected Control Transfer

这里介绍了夏老师上课说的一些内容。exception属于程序本身的保护机制，比如除0，进行syscall，shi是同步的。interrupt属于系统要求程序中断，例如外界来了一个IO信号，是异步的。

The Interrupt Descriptor Table 简称IDT，是内核本身存在的一个表，用例表示中断的信号是属于哪一种情况，从0到255。又说明了JOS中所有的interrupte的handle都是内核进行的。

The Task State Segment 简称TSS，用于在中断的时候保存CPU寄存器的值，主要当前状态的权限有关。

2.6 Types of Exceptions and Interrupts

资料又介绍了IDT的分配，从0 31是同步的exception，31以上用于软中断和异步中断。以及当前section需要实现0 31的功能，后面的section需要实现JOS中的48号syscall功能(对应于x86应该是0x80)。Lab4需要实现外部硬件中断例如时钟中断。

2.7 Setting Up the IDT

这里需要实现前面空缺的Interrupt Handle的功能，部分代码已经帮我们实现好了。在kern/trapentry.S中。

kern/trapentry.c

```
/*
 * Lab 3: Your code here for generating
   entry points for the different traps.
 */

    TRAPHANDLER_NOEC(entry0, T_DIVIDE);
    TRAPHANDLER_NOEC(entry1, T_DEBUG);
    TRAPHANDLER_NOEC(entry2, T_NMI);
    TRAPHANDLER_NOEC(entry3, T_BRKPT);
```

```
10      TRAPHANDLER_NOEC(entry4, T_OFLOW);
11      TRAPHANDLER_NOEC(entry5, T_BOUND);
12      TRAPHANDLER_NOEC(entry6, T_ILLOP);
13      TRAPHANDLER_NOEC(entry7, T_DEVICE);
14      TRAPHANDLER(entry8, T_DBLFLT);
15      TRAPHANDLER(entry10, T_TSS);
16      TRAPHANDLER(entry11, T_SEGNP);
17      TRAPHANDLER(entry12, T_STACK);
18      TRAPHANDLER(entry13, T_GPFLT);
19      TRAPHANDLER(entry14, T_PGFLT);
20      TRAPHANDLER_NOEC(entry16, T_FPERR);
21      TRAPHANDLER(entry17, T_ALIGN);
22      TRAPHANDLER_NOEC(entry18, T_MCHK);
23      TRAPHANDLER_NOEC(entry19, T_SIMDERR );
```

这里主要是注册下函数入口应对每个IDT的项，TRAPHANDLER比TRAPHANDLER_NONEC多了一个压入error code，后者则只压入0，如何使用参考的Intel的手册。

kern/trapentry.c

```
1   #see as inc/trap.h
2   _alltraps:
3       pushw $0    #uint16_t padding
4       pushw %ds
5       pushw $0    #uint16_t padding
6       pushw %es
7       pushal
8
9       movl $GD_KD, %eax
10      movw %ax, %ds
11      movw %ax, %es
12
13      pushl %esp
14
15      call trap
```

这一段参考了inc/trap.h的Trapframe的结构，然后按照注释的要求进行填写，主要作用是组织中断的堆栈结构，然后call trap函数执行中断。

之后还需要在trap_init()函数中调用SETGATE宏，这一段代码比较长我就不贴了，该函数在kern/trap.c中，主要是注册中断的trap函数和IDT表之间的映射关系，并且设置端和权限位。SETGATE宏在inc/mmu.h中定义。之后还调用了已经写好的trap_init_percpu。参照注释中的说明作用是读取内核态的TSS完成寄存器状态的切换。

随后make grade就会显示Part A通过。两个Question参考answers-lab3.txt文件。

对于Question2，我使用make run-softint-nox，确实显示出General Protection，与grade文件一样。

3 Page Faults, Breakpoints Exceptions, and System Calls

3.1 Handling Page Faults

这里我就在trap_dispatch函数中添加了两行代码，判断下是否是T_PGFLT，如果是就进图page_fault_handler函数并传入Trapframe。

kern/trap.c

```
static void
trap_dispatch(struct Trapframe *tf)
{
    // Handle processor exceptions.
    // LAB 3: Your code here.

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

make grade确实part B的前4个都能通过。到目前为止的实现应该都没有问题。

3.2 The Breakpoint Exception

这一步的grade-breakpoint.sh我还需要安装tcl expect等工具，lab提供的虚拟机竟然没装！而且monitor.c中还缺少了必须的头文件kern/env.h，导致我env_run都没找到，实在无语。

这个exercise-6实在是太坑啦！我进入了debug后程序永远会死在一个panic里面然后无限int3，经过我漫长的单步调试发现是breakpoint在exit的时候调用了syscall然后sysexit又没有做，这个练习在后面，导致我这里panic程序无法结束，这太坑了！所以我是做完了后面一个练习再回来debug这个练习的，好在相比于后面一个练习，这个练习简直容易，因为后面一个练习更坑！好了，不说废话了。

kern/trap.c

```
switch(tf->tf_trapno) {
        case T_PGFLT:
            page_fault_handler(tf);
            break;
        case T_DEBUG:
        case T_BRKPT:
            monitor(tf);
            break;
    }
```

首先修改trap_dispatch，这个比较简单，我把DEBUG跟BRKPT转到monitor去处理就好了。这里我发现单步调试每次是trap DEBUG，而不是BRKPT。

之后在monitor.h里面先声明下函数

kern/monitor.c

```
int
mon_debug_continue(int argc, char **argv, struct Trapframe *tf)
{
    uint32_t eflags;
    if (tf == NULL) {
        cprintf("No trapped environment\n");
        return 1;
    }
    eflags = tf->tf_eflags;
    eflags &= ~FL_TF;
    tf->tf_eflags = eflags;
    env_run(curenv);
    return 0;
}
```

continue按照资料写本身很简单，将单步调试的FL_TF位变成0即可。

但是我在这里卡了好久，完成了step后单步调试，手动int3很久慢慢跟踪才发现是breakpoint调用syscall然后panic了，panic竟然会无限重复jmp。所以等我完成了后面的sysenter跟sysexit后这个练习就能够按照预期进行下去，最后经过一些输出格式的修改跟grade脚本一样后就得分了。

kern/monitor.c

```
int
mon_debug_step(int argc, char **argv, struct Trapframe *tf)
{
    uint32_t eflags;
    if (tf == NULL) {
        cprintf("No trapped environment\n");
        return 1;
    }
    eflags = tf->tf_eflags;
    eflags |= FL_TF;
    tf->tf_eflags = eflags;

    cprintf("tf_eip=0x%x\n", tf->tf_eip);
    env_run(curenv);
    return 0;
}
```

单步调试和continue差不多，就是将FL_TF位变成1。

kern/monitor.c

```
int
mon_debug_display(int argc, char **argv, struct Trapframe *tf)
{
    if (argc != 2) {
```

```
5           cprintf("please␣enter␣x␣addr");
6       }
7       uint32_t get_addr;
8       get_addr = strtol(argv[1], NULL, 16);
9
10      uint32_t get_val;
11      __asm __volatile("movl␣(%0),␣%0"
12      : "=r" (get_val) : "r" (get_addr));
13
14      cprintf("%d\n", get_val);
15      return 0;
16  }
```

display的话我查了各个函数文件很久，模仿写内嵌汇编和strtol函数的调用，好在这些都不是很困难。x 后面输入0x或者不输入0x都默认是16进制，这个和strtol的实现有关，使用0开头就认为是8进制。这个操作后面的地址随意输入有可能会造成page-fault。

经过一段时间的调试通过了grade-breakpoint.sh。

Question3和4参见anwers-lab3.txt。

3.3 System calls

这里比较麻烦的是搞清楚整个syscall的调用流程，我的理解是这样的。用户进行syscall的时候，调用的是lib/syscall.c中的syscall，然后进入kern/trap.c的路由根据中断号，进入kern/syscall.c中的syscall函数，然后根据对应syscallno参数再路由一次调用对应的操作。操作结束后在trap的调用中返回curenv。

这个exercise-7在我还没有理解或者说写一遍正常的syscall的时候直接做我觉得是非常不合理的。

这个练习实现的是跳过中间的路由进入trap的阶段，直接根据syscallno调用需要的操作。虽然知道目的和意思，但是实际写起来太难了，很多东西需要自己查wiki查手册，而且还查不到。根据课程网站的资料，首先修改kern/trapentry.S中的代码。

kern/trapentry.c

```
1       pushl $GD_UD|3
2       pushl %ebp
3       pushfl
4       pushl $GD_UT|3
5       pushl %esi
6       pushl $0
7       pushl $0
8       pushl %ds
9       pushl %es
10      pushal
11      movw $GD_KD, %ax
```

```
12      movw %ax, %ds
13      movw %ax, %es
14      pushl %esp
15      call my_syscall
16      popl %esp
17      popal
18      popl %es
19      popl %ds
20      movl %ebp, %ecx
21      movl %esi, %edx
22      sysexit
```

这里完成了sysenter函数的入口，组织好了压栈的参数后call了my_syscall函数，参数的组织结构参考Trapframe。my_syscall函数定义在kern/syscall.c中，根据这里组织的参数直接调用kern/syscall.c中的syscall函数。my_syscall就是简单的将最后1个参数补成0了，减少了我在汇编中组织参数的麻烦工作。

当然首先lib/syscall需要支持sysenter，这一步很难写，好在网站上有参考用跳转来判断是否需要进入sysenter。

我有一个疑问就是直接使用网站上的代码会提示我标签重复申明，但我去掉标签又提示找不到，最后我查看stackoverflow上的解决方法是使用local标签，分别使用1表示标签，1f表示跳转解决了这个问题。我也不知道为什么会说我重复标签申明。

lib/syscall.c

```
1  //Lab 3: Your code here
2                  "movl␣%%esp,%%ebp\n\t"
3                   "leal␣1f,␣%%esi\n\t"
4                   "sysenter\n\t"
5                   "1:\n\t"
6
7                   "popl␣%%edi\n\t"
8                   "popl␣%%esi\n\t"
9                   "popl␣%%ebp\n\t"
10                  "popl␣%%esp\n\t"
11                  "popl␣%%ebx\n\t"
12                  "popl␣%%edx\n\t"
13                  "popl␣%%ecx\n\t"
14
15                  : "=a" (ret)
16                  : "a" (num),
17                    "d" (a1),
18                    "c" (a2),
19                    "b" (a3),
20                    "D" (a4)
21                  : "cc", "memory");
```

在trap.init中也要申明下sysenter特例。这一步的wrmsr没按照要求写在inc/x86.h中，在这里用就在这个文件里写宏吧。

kern/trap.c

```
#define wrmsr(msr,val1,val2) \
    __asm__ __volatile__("wrmsr" \
    : \
    : "c" (msr), "a" (val1), "d" (val2))
```

kern/trap.c

```
void
trap_init(void)
{
    .....
    // Per-CPU setup
    extern void sysenter_handler();
    wrmsr(0x174, GD_KT, 0);
    wrmsr(0x175, KSTACKTOP, 0);
    wrmsr(0x176, sysenter_handler, 0);


    // Per-CPU setup
    trap_init_percpu();
}
```

在进入了kern/syscall.c后，这里还需要判断syscallno，因此要写个路由。

kern/syscall.c

```
int32_t syscall(uint32_t syscallno, uint32_t a1,
uint32_t a2, uint32_t a3, uint32_t a4, uint32_t a5)
{

    switch (syscallno){
        case SYS_getenvid:
            return sys_getenvid();
        case SYS_cputs:
            sys_cputs((const char*) a1,a2);
            return 0;
        case SYS_cgetc:
            return sys_cgetc();
        case SYS_env_destroy:
            return sys_env_destroy(a1);
        case SYS_map_kernel_page:
            return sys_map_kernel_page
            ((void *)a1, (void *)a2);
        case SYS_sbrk:
            return sys_sbrk(a1);
        default:
            return -E_INVAL;
}
```

这里的SYS_sbrk是后面的练习，因为我这部分的文档是在lab代码写完后写的，所以就把sbrk也当做写完了吧。

最后这个通过了testbss，实在是艰难，这两个练习我搞了一整天，而且不写完这个前面的debug练习是无法exit的。

3.4 User-mode startup

exercise-8这个按照提示非常简单，这个sys_getenvid的就在kern/syscall.c中，是一个syscall。

在lib/libmain.c中添加一行代码修改下全局的thisenv即可。

lib/libmain.c

```
1  thisenv = envs + ENVX(sys_getenvid());
```

exercise-9这个sbrk开始由于忘记搞syscall的路由导致一直返回-3，好在很快解决了。根据课程网站的提示，完成起来比较容易，比syscall跟debug简单多了。

首先我在每个Env中加了一个变量用来保存当前分配到的虚拟空间栈的地址的底。

lib/libmain.c

```
1
2  struct Env {
3      ......
4
5      uint32_t env_va_bottom;
6
7      ......
8  }
```

然后sys_sbrk就根据保存的break开始分配inc个byte的栈空间即可，函数region_alloc都写好了。我还修改了region_alloc删掉了开头的static，让我能在其他文件中调用它。轻松通过测试。

kern/syscall.c

```
1  static int
2  sys_sbrk(uint32_t inc)
3  {
4      // LAB3: your code sbrk here...
5      region_alloc(curenv, (void*)(curenv->env_va_bottom - inc),
6                   inc);
7      return curenv->env_va_bottom;
8  }
```

3.5 Page faults and memory protection

首先资料介绍说明并且强调了user的page-fault和kernel的page-fault应该使用不同的处理方式，前者可以继续执行，认为只是user的process的错误，不会影响ker-

nel，后者则认为是kernel错误，后果比较严重。

首先在kern/trap.c中的page_fault_handler函数中，判断是否是kernel产生的page-fault，如果是，则panic。按照资料写，一句话判断。

kern/trap.c

```
void
page_fault_handler(struct Trapframe *tf)
{
    ......
    if (!(tf->tf_cs & 0x3)) {
        panic("kernel page fault");
    }
    ......
}
```

接下来进行user的page-fault判定，下面按层次介绍。

首先是sys_cputs函数，在kern/syscall中，在发生syscall中断后进入该函数判定user这个页操作是否允许，添加user_mem_assert的调用。

kern/syscall.c

```
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 | PTE_P);

    // Print the string supplied by the user.
    cprintf("%.*s", len, s);
}
```

user_mem_assert函数在熟悉的kern/pmap.c中，调用user_mem_check判断是否是user process的page-fault，如果是，直接env_destroy这个env。

kern/pmap.c

```
void
user_mem_assert(struct Env *env, const void *va, size_t len, int perm)
{
    if (user_mem_check(env, va, len, perm | PTE_U) < 0) {
        cprintf("[%08x] user_mem_check assertion failure for "
            "va %08x\n", env->env_id, user_mem_check_addr);
        env_destroy(env);   // may not return
    }
}
```

user_mem_check函数同样在kern/pmap.c中，判断下所调用的va是否越界，如果没有越界，再用pgdir_walk函数判定页表中的情况，这样一层一层的设计十分巧妙但是复杂。最后，如果越界了，直接抛出-E_FAULT，在上面的sys_mem_assert捕获到这个负的返回值后就直接消灭这个env了。

kern/pmap.c

```
1  int
2  user_mem_check(struct Env *env, const void *va, size_t len, int perm)
3  {
4      // LAB 3: Your code here.
5      uintptr_t lva = (uintptr_t) va;
6      uintptr_t rva = (uintptr_t) va + len - 1;
7      uintptr_t idx;
8      pte_t *pte;
9      perm |= PTE_U | PTE_P;
10
11     for (idx = lva; idx <= rva; idx = ROUNDDOWN(idx+PGSIZE, PGSIZE)) {
12         if (idx >= ULIM) {
13             user_mem_check_addr = idx;
14             return -E_FAULT;
15         }
16         pte = pgdir_walk (env->env_pgdir, (void*)idx, 0);
17         if (pte == NULL || (*pte & perm) != perm) {
18             user_mem_check_addr = idx;
19             return -E_FAULT;
20         }
21     }
22     return 0;
23 }
```

最后还要再kern/kdebug.c的debuginfo_eip函数中调用下user_mem_check函数在debug调用该函数的时候同样判断下是否user越界。

随后在breakpoint进程中使用lab1写的backtrace操作确实产生了user态的page-fault。evilhello进程使用了sys_cputs调用了该进程不能访问的kernel中的部分，同样产生了user态的page-fault，env被杀死。

exercise-10和11 pass。

exercise-12又要做hacker了，这个练习分别在0和3的权限下调用了接触kernel内存部分的代码，当然在0下会执行在3下会user page-fault，需要我们实现如何进入0权限下后调用evil函数。知道了目的，要做的就是如何hack kernel了，hacking to the gate!

根据提示，先使用汇编的sgdt命令读取gdt的内容，通过sys_map_kernel_page将这块映射到user态创建的一块内存空间vaddr中。如下ring0_call函数的第3到8行。然后根据创建的vaddr，得到entry的位置。在改gdt之前先保存下以便恢复，然后第18行很hack的修改gdt，19行调用进入0权限并且进入call_fun_ptr包装函数，此时处于0权限

下，调用evil函数，kernel态当然能直接搞0xF0000000以上的空间。最后恢复保存的gdt并且返回。

user/evilhello2.c

```
1  void ring0_call(void (*fun_ptr)(void)) {
2      struct Pseudodesc r_gdt;
3      get_gdt(&r_gdt);
4
5      int t = sys_map_kernel_page((void* )r_gdt.pd_base, (void* )vaddr);
6      if (t < 0) {
7          cprintf("ring0_call: sys_map_kernel_page failed, %e\n", t);
8      }
9
10     uint32_t base = (uint32_t)(PGNUM(vaddr) << PTXSHIFT);
11     uint32_t index = GD_UD >> 3;
12     uint32_t offset = PGOFF(r_gdt.pd_base);
13
14     gdt = (struct Segdesc*)(base+offset);
15     entry = gdt + index;
16     oldold = *entry;
17
18     SETCALLGATE(*((struct Gatedesc*)entry), GD_KT, call_fun_ptr, 3);
19     asm volatile("lcall $0x20, $0");
20 }
```

user/evilhello2.c

```
1  void call_fun_ptr()
2  {
3      evil();
4      *entry = old;
5      asm volatile("popl %ebp");
6      asm volatile("lret");
7  }
```

说句题外话，如果kernel要解决这种漏洞，我觉得关键是在user态不能随意调用sys_map_kernel_page操作，应该增加user态的判断是否超出user能够使用的范围。这样能够接触kernel态的东西并且随便映射和直接接触kernel态的才能使用的空间没有任何区别。当然我不能修复这个漏洞，不然我lab怎么pass得分。

4 总结

这个lab的难度明显比lab1和lab2难，我从星期四晚上写到第二个星期的星期一上午才写完，包括文档和answer估测25到30小时，挖坑无数。因为还要准备GRE，如果后面的lab5,lab6也是这个难度我可能要权衡下时间问题了T_T。