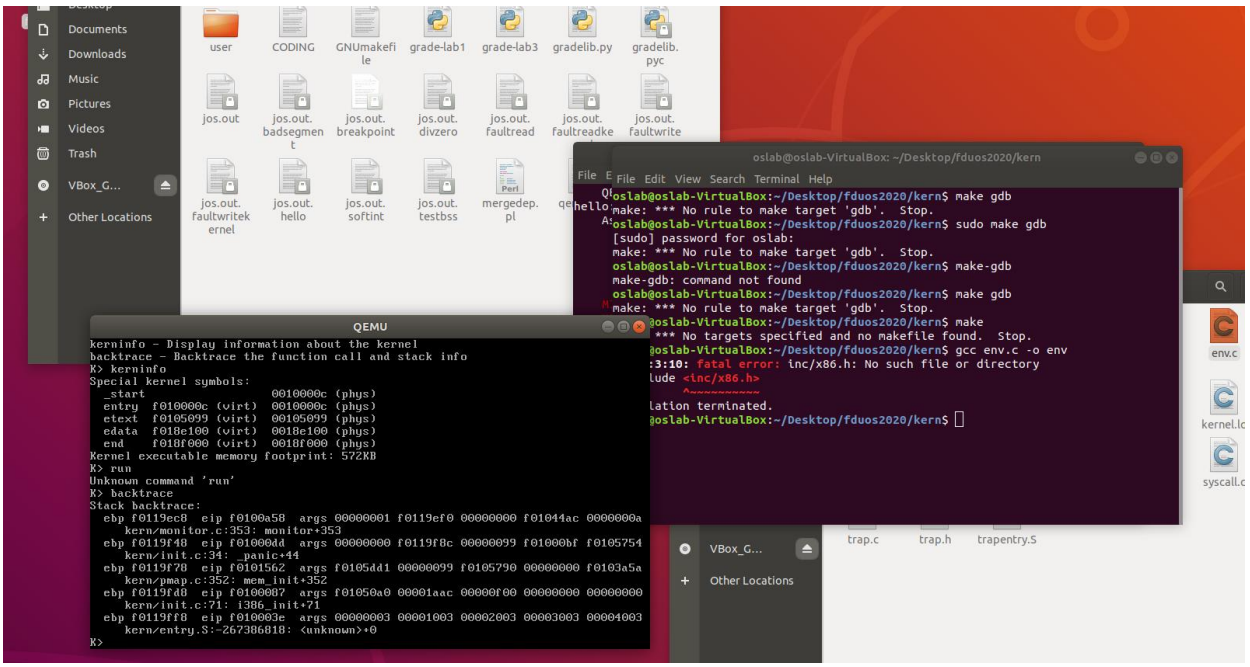
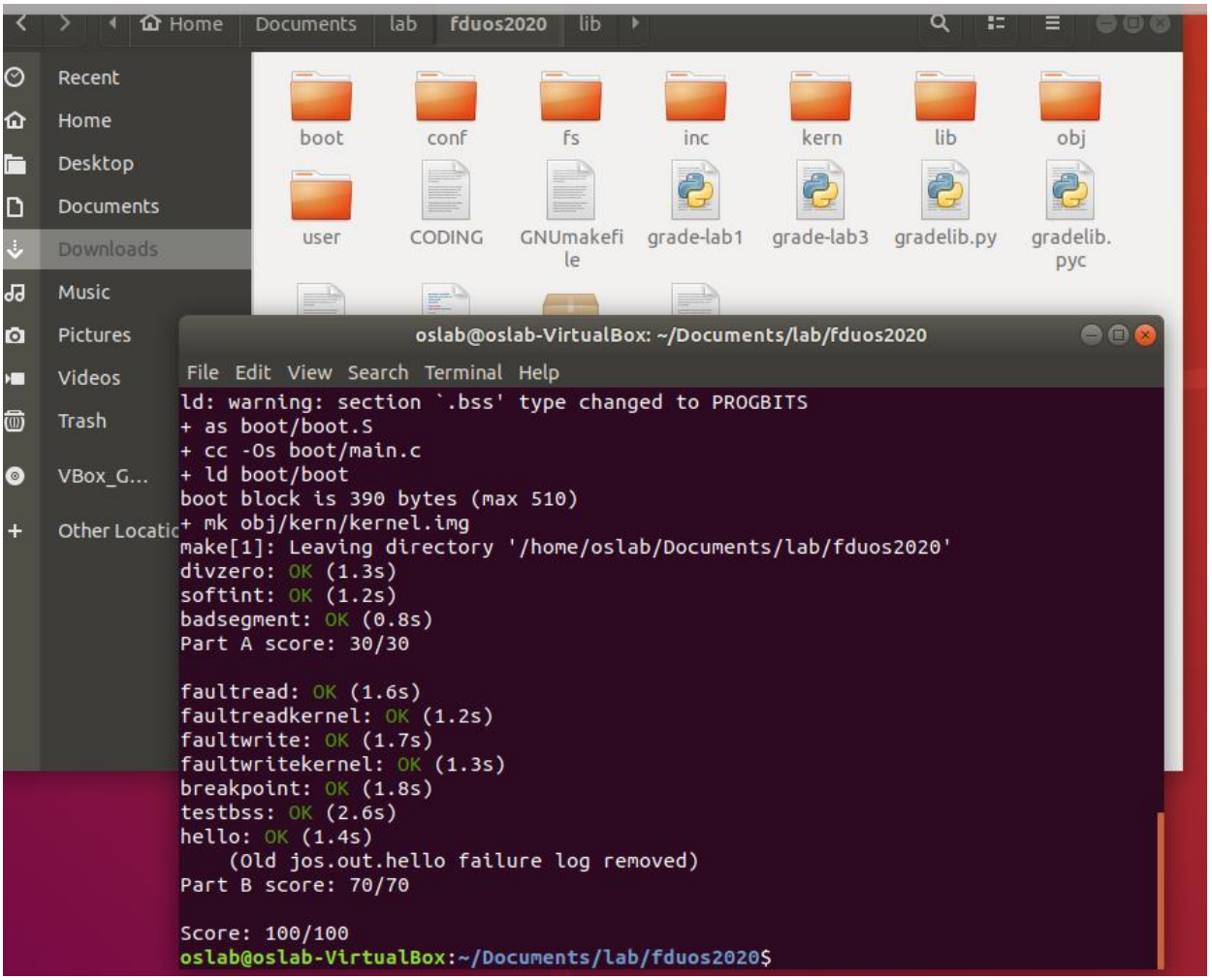




由于没给hello分配空间，所以缺页了。

Exercise2

env_init函数遍历 envs 数组中的所有 Env 结构体，把每一个结构体的 env_id 字段置0，因为要求所有的 Env 在 env_free_list 中的顺序，要和它在 envs 中的顺序一致，所以需要采用头插法。

```
void
env_init(void)
{
 // Set up envs array
 // LAB 3: Your code here.
 int i;
 env_free_list = NULL;
 for (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];
  }
 // Per-CPU part of the initialization
 env_init_percpu();
}
```

env_setup_vm

函数主要是初始化新的用户环境的页目录表，不过只设置页目录表中和操作系统内核跟内核相关的页目录项，用户环境的页目录项不要设置，因为所有用户环境的页目录表中和操作系统相关的页目录项都是一样的（除了虚拟地址UVPT，这个也会单独进行设置），所以我们可以参照 kern_pgdir 中的内容来设置 env_pgdir 中的内容。

env_run()

函数功能：运行一个给定的环境

- 如果是环境切换（即有环境正在运行）：

  a. 如果当前环境curenv的env_status为ENV_RUNNING，设置它为ENV_RUNNABLE

  b. 设置curenv为新的环境

c. 设置新的环境的env_status为ENV_RUNNING

d. 更新env_runs计数

e. 利用lcr3()切换地址空间

f. 利用env_pop_tf()恢复环境的重要寄存器并进入用户模式

```
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 != NULL && curenv->env_status == ENV_RUNNING) {
  curenv->env_status = ENV_RUNNABLE;
  }
  curenv = e;
  curenv->env_status = ENV_RUNNING;
  curenv->env_runs++;

  // 设置cr3中的页表目录地址：地址为物理地址
  lcr3(PADDR(curenv->env_pgdir));
  env_pop_tf(&curenv->env_tf);

```

```
33   // panic("env_run not yet implemented");
34  }
```

Exercise4

需要编辑trapentry.S和trap.c实现上述功能。在trapentry.S中有两个宏用于处理中断和异常：

TRAPHANDLER：将中断号压栈，然后跳转到_alltraps。用来处理有错误码的异常（CPU自动压入错误码）

TRAPHANDLER_NOEC：用来处理没有错误码的异常（压入0代替错误码）

这两个宏都接受两个参数(name, num)，其中name是中断处理程序的函数名，num是相应中断号

在这个实验中，我们需要：

利用这两个宏在trapentry.S中为定义在inc/trap.h中的每个异常编写入口点

为这两个宏编写_alltraps

利用SETGATE宏修改trap_init()为这些入口点初始化中断向量表

IDT_alltraps需要能够：

1. 以Trapframe结构体的格式把值压栈（err, trapno由宏已压入，SS, ESP, EFLAGS, CS, EIP压到内核栈中，之后从内核栈恢复，这里不用再压一遍）
2. 把GD_KD加载到ds和es
3. 将esp压入，为trap()传参
4. 调用trap()

```
1  /*
2   * Lab 3: Your code here for generating entry points for the different tr
aps.
3   */
4
5  TRAPHANDLER_NOEC(t_divide, T_DIVIDE)
6  TRAPHANDLER_NOEC(t_debug, T_DEBUG)
7  TRAPHANDLER_NOEC(t_nmi, T_NMI)
8  TRAPHANDLER_NOEC(t_brkpt, T_BRKPT)
9  TRAPHANDLER_NOEC(t_oflow, T_OFLOW)
10  TRAPHANDLER_NOEC(t_bound, T_BOUND)
11  TRAPHANDLER_NOEC(t_illop, T_ILLOP)
12  TRAPHANDLER_NOEC(t_device, T_DEVICE)
```

```
13 TRAPHANDLER(t_dblflt, T_DBLFLT)
14 TRAPHANDLER(t_tss, T_TSS)
15 TRAPHANDLER(t_segnp, T_SEGNP)
16 TRAPHANDLER(t_stack, T_STACK)
17 TRAPHANDLER(t_gpflt, T_GPFLT)
18 TRAPHANDLER(t_pgflt, T_PGFLT)
19 TRAPHANDLER_NOEC(t_fperr, T_FPERR)
20 TRAPHANDLER(t_align, T_ALIGN)
21 TRAPHANDLER_NOEC(t_mchk, T_MCHK)
22 TRAPHANDLER_NOEC(t_simderr, T_SIMDERR)
23
24
25 /*
26  * Lab 3: Your code here for _alltraps
27  */
28
29 _alltraps:
30  pushl %ds
31  pushl %es
32  pushal
33  movw $GD_KD, %ax
34  movw %ax, %ds
35  movw %ax, %es
36  pushl %esp
37  call trap
```

trap.c

```
1 void
2 trap_init(void)
3 {
4  extern struct Segdesc gdt[];
5  void t_divide();
6  void t_debug();
7  void t_nmi();
8  void t_brkpt();
9  void t_oflow();
10  void t_bound();
11  void t_illop();
12  void t_device();
13  void t_dblflt();
14  void t_tss();
```

```
15  void t_segnp();
16  void t_stack();
17  void t_gpflt();
18  void t_pgflt();
19  void t_fperr();
20  void t_align();
21  void t_mchk();
22  void t_simderr();
23
24  // LAB 3: Your code here.
25  SETGATE(idt[T_DIVIDE], 0, GD_KT, t_divide, 0);
26  SETGATE(idt[T_DEBUG], 0, GD_KT, t_debug, 0);
27  SETGATE(idt[T_NMI], 0, GD_KT, t_nmi, 0);
28  SETGATE(idt[T_BRKPT], 1, GD_KT, t_brkpt, 0);
29  SETGATE(idt[T_OFLOW], 1, GD_KT, t_oflow, 0);
30  SETGATE(idt[T_BOUND], 0, GD_KT, t_bound, 0);
31  SETGATE(idt[T_ILLOP], 0, GD_KT, t_illop, 0);
32  SETGATE(idt[T_DEVICE], 0, GD_KT, t_device, 0);
33  SETGATE(idt[T_DBLFLT], 0, GD_KT, t_dblflt, 0);
34  SETGATE(idt[T_TSS], 0, GD_KT, t_tss, 0);
35  SETGATE(idt[T_SEGNP], 0, GD_KT, t_segnp, 0);
36  SETGATE(idt[T_STACK], 0, GD_KT, t_stack, 0);
37  SETGATE(idt[T_GPFLT], 0, GD_KT, t_gpflt, 0);
38  SETGATE(idt[T_PGFLT], 0, GD_KT, t_pgflt, 0);
39  SETGATE(idt[T_FPERR], 0, GD_KT, t_fperr, 0);
40  SETGATE(idt[T_ALIGN], 0, GD_KT, t_align, 0);
41  SETGATE(idt[T_MCHK], 0, GD_KT, t_mchk, 0);
42  SETGATE(idt[T_SIMDERR], 0, GD_KT, t_simderr, 0);
43
44  // Per-CPU setup
45  trap_init_percpu();
46 }
```

Exercise 5

根据 trapentry.S 文件中的 TRAPHANDLER 函数可知，这个函数会把当前中断的中断码压入堆栈中，再根据 inc/trap.h 文件中的 Trapframe 结构体我们可以知道，Trapframe 中的 tf_trapno 成员代表这个中断的中断码。所以在 trap_dispatch 函数中我们需要根据输入的 Trapframe 指针 tf 中的 tf_trapno 成员来判断到来的中断是什么中断，这里我们需要判断是否是缺页中断，如果是则执行 page_fault_handler 函数，所以我们可以这么修改代码

```
1 static void
2 trap_dispatch(struct Trapframe *tf)
```

```
{
 int32_t ret_code;
 // Handle processor exceptions.
 // LAB 3: Your code here.
 switch(tf->tf_trapno) {
 case (T_PGFLT):
 page_fault_handler(tf);
  break;
  default:
  // 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;
  }
  }
}
```

、Exercise 6

处理断点中断 (T_BRKPT)，kernel monitor 就是定义在 kern/monitor.c 文件中的 monitor 函数

```
static void
trap_dispatch(struct Trapframe *tf)
{
 int32_t ret_code;
 // Handle processor exceptions.
 // LAB 3: Your code here.
 switch(tf->tf_trapno) {
 case (T_PGFLT):
 page_fault_handler(tf);
  break;
  case (T_BRKPT):
  monitor(tf);
  break;
  default:
  // Unexpected trap: The user process or the kernel has a bug.
  print_trapframe(tf);
  if (tf->tf_cs == GD_KT)
```

```
18   panic("unhandled trap in kernel");
19   else {
20   env_destroy(curenv);
21   return;
22   }
23   }
24  }
```

、Exercise7

当用户程序中要调用系统调用时，比如 sys_cputs，从它的汇编代码中我们会发现，它会执行一个 int $0x30 指令，这个指令就是软件中断指令，这个中断的中断号就是 0x30，即 T_SYSCALL，所以题目中让我们首先为这个中断号编写一个中断处理函数，我们首先就要在 kern/trapentry.S 文件中为它声明它的中断处理函数，即TRAPHANDLER_NOEC.

```
1  kern/trapentry.S
2
3  .....
4  TRAPHANDLER_NOEC(t_fperr, T_FPERR)
5  TRAPHANDLER(t_align, T_ALIGN)
6  TRAPHANDLER_NOEC(t_mchk, T_MCHK)
7  TRAPHANDLER_NOEC(t_simderr, T_SIMDERR)
8
9  TRAPHANDLER_NOEC(t_syscall, T_SYSCALL)
10
11  _alltraps
12  ....
```

后在trap.c 文件中声明 t_syscall() 函数。并且在 trap_init() 函数中为它注册， kern/syscall.c 中的所有函数居然和 lib/syscall.c 中的所有函数都是一样的,在 lib/syscall.c 中，是直接调用 syscall 的,cprintf函数其实就是通过调用 lib/syscall.c 中的 sys_cputs 来实现的，由于此时系统已经处于内核态了，所以这个 sys_cputs 可以被执行

、Exercise 8

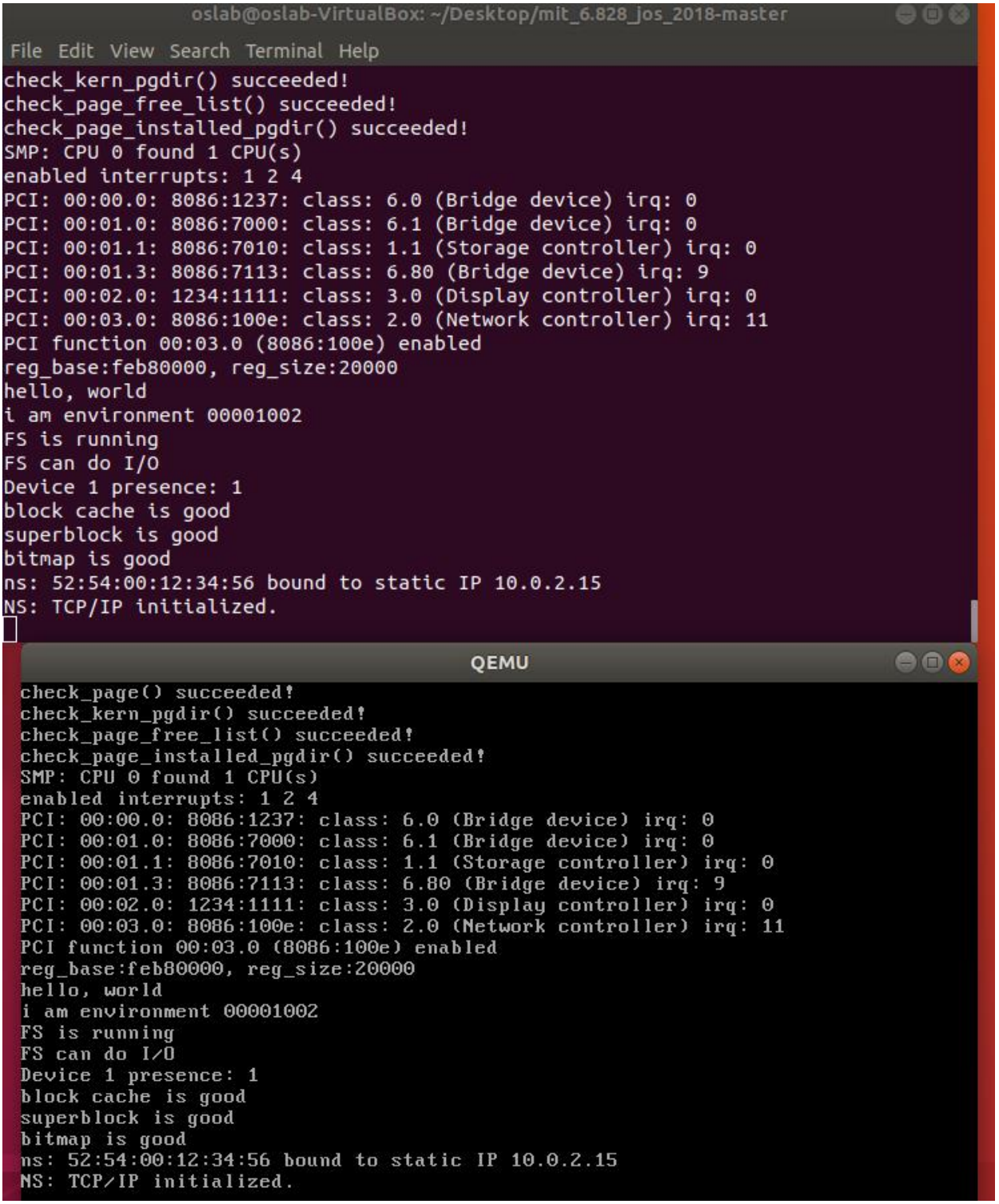
通过程序获得当前正在运行的用户环境的 env_id , 以及这个用户环境所对应的 Env 结构体的指针。 env_id 我们可以通过调用 sys_getenvid() 这个函数来获得。

```
1  void
2  libmain(int argc, char **argv)
3  {
4   // set thisenv to point at our Env structure in envs[].
5   // LAB 3: Your code here.
6   thisenv = &envs[ENVX(sys_getenvid())];
7
8   // save the name of the program so that panic() can use it
```

```
9  if (argc > 0)
10  binaryname = argv[0];
11
12  // call user main routine
13  umain(argc, argv);
14
15  // exit gracefully
16  exit();
17 }
```

lib/env.h 文件我们知道，env_id的值包含三部分，第31位被固定为0；第10~30这21位是标识符，标示这个用户环境；第0~9位代表这个用户环境所采用的 Env 结构体，在envs数组中的索引。所以我们只需知道 env_id 的低 0~9 位，我们就可以获得这个用户环境对应的 Env 结构体了。

SYS_ show_environments 系统变量相关代码的实现思路



在lib/syscall.c中添加sys_show_environments

```
static void
sys_show_environments(void){

 int i=0;
 while(i < NENV){ //1024
 cprintf("env_id:%08x\n",envs[i].env_id);
 cprintf("registers\n");
 struct PushRegs regs =envs[i].env_tf.tf_regs;
```

```
 cprintf("edi: %08x\n",regs.reg_edi);
 cprintf("esi: %08x\n",regs.reg_esi);
 cprintf("ebp: %08x\n",regs.reg_ebp);
 cprintf("ebx: %08x\n",regs.reg_ebx);
 cprintf("edx: %08x\n",regs.reg_edx);
 cprintf("ecx: %08x\n",regs.reg_ecx);
 cprintf("eax: %08x\n",regs.reg_eax);
 cprintf("edi:%08x\n",envs[i].env_id);
 cprintf("env_parent_id: %08x\n",envs[i].env_parent_id);
 cprintf("env status: %08x\n",envs[i].env_status);
 i++;
 }
}
```

在ihelloworld中添加打印寄存器sys_show_environments();

```
// hello, world
#include <inc/lib.h>
#include <lib/syscall.c>
void
umain(int argc, char **argv)
{
 cprintf("hello, world\n");
 cprintf("i am environment %08x\n", thisenv->env_id);
 sys_show_environments();

}
```

执行make run-hello

```
PCI function 00:03.0 (8086:100e) enabled
reg_base:feb80000, reg_size:20000
hello, world
FS is running
FS can do I/O
Device 1 presence: 1
block cache is good
i am environment 00001002
env_id:00001000
registers
edi: 00000000
esi: 00000000
ebp: eebfdc20
superblock is good
bitmap is good
ebx: 00000000
edx: eebfdc48
ecx: 00000014
eax: 00000000
edi:00001000
```

```
ecx: 00000014
eax: 00000000
edi:00001000
env_parent_id: 00000000
env status: 00000004
env_id:00001001
registers
edi: 00973000
esi: 00000805
ebp: eebfdf70
ebx: 00001003
edx: 00000000
ecx: 00973000
eax: 00000000
edi:00001001
env_parent_id: 00000000
env status: 00000002
env_id:00001002
registers
edi: 00000000
esi: 00000000
ebp: eebfde10
ebx: 00000000
edx: eebfde38
```

```
registers
edi: 00000000
esi: 00000000
ebp: eebfde10
ebx: 00000000
edx: eebfde38
ecx: 0000000a
eax: 00000000
edi:00001002
env_parent_id: 00000000
env status: 00000003
env_id:00001003
registers
edi: 00000000
esi: 00000000
ebp: eebfdfc0
ebx: 00001001
edx: 00000000
ecx: 0080fdab
eax: 00000000
edi:00001003
env_parent_id: 00001001
env status: 00000002
env_id:00001004
```

```
ecx: 0080fdab
eax: 00000000
edi:00001003
env_parent_id: 00001001
env status: 00000002
env_id:00001004
registers
edi: 00000000
esi: 00000000
ebp: eebfdfc0
ebx: 00001001
edx: 00001003
ecx: 00000002
eax: 00000000
edi:00001004
env_parent_id: 00001001
env status: 00000004
env_id:00000000
registers
edi: 00000000
esi: 00000000
ebp: 00000000
ebx: 00000000
edx: 00000000
```

(1) JOS 中, 内核态页表基地址定义在哪个变量中？ 页表基地址存在哪个寄存器中？ JOS 中如何切换页表基地址空间?

syscall.h | entrypgdir.c | pmap.c | Makefrag | syscall.c

kern > entrypgdir.c

boot Aa Abl .* ? of 1

```c
#include <inc/mmu.h>
#include <inc/memlayout.h>

pte_t entry_pgtable[NPTENTRIES];

// The entry.S page directory maps the first 4MB of physical memory
// starting at virtual address KERNBASE (that is, it maps virtual
// addresses [KERNBASE, KERNBASE+4MB) to physical addresses [0, 4MB)).
// We choose 4MB because that's how much we can map with one page
// table and it's enough to get us through early boot.  We also map
// virtual addresses [0, 4MB) to physical addresses [0, 4MB); this
// region is critical for a few instructions in entry.S and then we
// never use it again.
//
// Page directories (and page tables), must start on a page boundary,
// hence the "__aligned__" attribute.  Also, because of restrictions
// related to linking and static initializers, we use "x + PTE_P"
// here, rather than the more standard "x | PTE_P".  Everywhere else
// you should use "|" to combine flags.
__attribute__((__aligned__(PGSIZE)))
pde_t entry_pgdir[NPDENTRIES] = {
    // Map VA's [0, 4MB) to PA's [0, 4MB)
    [0]
        = ((uintptr_t)entry_pgtable - KERNBASE) + PTE_P,
    // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
    [KERNBASE>>PDXSHIFT]
        = ((uintptr_t)entry_pgtable - KERNBASE) + PTE_P + PTE_W
};

// Entry 0 of the page table maps to physical page 0, entry 1 to
// physical page 1, etc.
__attribute__((__aligned__(PGSIZE)))
pte_t entry_pgtable[NPTENTRIES] = {
    0x000000 | PTE_P | PTE_W,
    0x001000 | PTE_P | PTE_W,
    0x002000 | PTE_P | PTE_W,
```

定义在kern_pgdir中，寄存器cr3保存了页表基地址，

```
// Global descriptor table.
//
// Set up global descriptor table (GDT) with separate segments for
// kernel mode and user mode.  Segments serve many purposes on the x86.
// We don't use any of their memory-mapping capabilities, but we need
// them to switch privilege levels.
//
// The kernel and user segments are identical except for the DPL.
// To load the SS register, the CPL must equal the DPL.  Thus,
// we must duplicate the segments for the user and the kernel.
//
// In particular, the last argument to the SEG macro used in the
// definition of gdt specifies the Descriptor Privilege Level (DPL)
// of that descriptor: 0 for kernel and 3 for user.
//
struct Segdesc gdt[] =
{
    // 0x0 - unused (always faults -- for trapping NULL far pointers)
    SEG_NULL,

    // 0x8 - kernel code segment
    [GD_KT >> 3] = SEG(STA_X | STA_R, 0x0, 0xffffffff, 0),

    // 0x10 - kernel data segment
    [GD_KD >> 3] = SEG(STA_W, 0x0, 0xffffffff, 0),

    // 0x18 - user code segment
    [GD_UT >> 3] = SEG(STA_X | STA_R, 0x0, 0xffffffff, 3),

    // 0x20 - user data segment
    [GD_UD >> 3] = SEG(STA_W, 0x0, 0xffffffff, 3),

    // 0x28 - tss, initialized in trap_init_percpu()
    [GD_TSS0 >> 3] = SEG_NULL
};
```

对于用户级与内核态的切换，JOS是使用kern_pgdir到env_pgdir来对于页表基地址的切换。

(2) iret 指令的功能和作用是什么？ kernel stack 的栈顶在在哪里定义？ Exception 陷入 kernel 的时候，esp, eip, eax, ebx 分别是谁(processer，jos kernel)保存的?

在用IRET指令返回到相同保护级别的任务时，IRET会从堆栈弹出代码段选择子及指令指针分别到CS与IP寄存器，并弹出标志寄存器内容到EFLAGS寄存器。当使用IRET指令返回到一个不同的保护级别时，IRET不仅会从堆栈弹出以上内容，还会弹出堆栈段选择子及堆栈指针分别到SS与SP寄存器。

esp, eip, eax, ebx 分别是谁(processer，jos kernel)保存的?

ESP和EIP保存在`struct Trapframe`内核态的结构体中 由processer保存，EAX 和EBX保存在`struct PushRegs`由jos kernel保存。

(3) IDT 和 GDT 存储的信息分别是什么?

IDT是**中断描述符表**，GDT是全局描述表，分别是中断描述符和段描述符的索引。