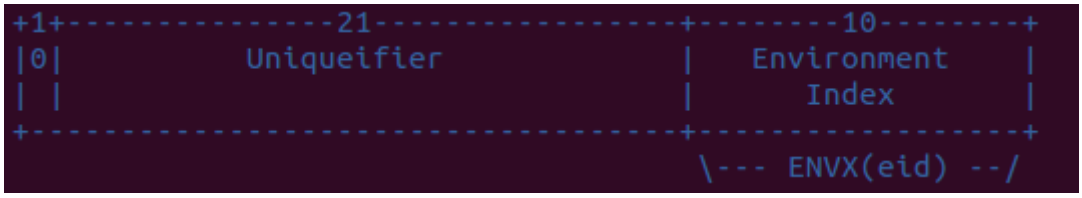
Lab3

一.分析 inc/env.h 文件

根据 inc/env.h 中的注释提示，可以知道每一个进程都有一个进程号。一共 32 位，分为三个部分:最后 10 位标识该进程在进程数组 envs[]中的位置，由此可见所有的进程都被保存的 envs[]数组中统一管理,并且只能支持 NENV(1024)个进程;中间 21 位 Uniqueifier(Unique Identifier)表示不同时间创建的进程（在一个进程结束后，一个新的进程可以占用之前进程所占用的 env 数据结构，但是 Uniqueifier 不能相同，即所有进程具有唯一标识符）;最高一位 sign 标志位，sign 为 0 时标识当前确定的进程;
值得注意的是 ENVX（eid）就是该进程在 envs 数组中的 index(以后会用到`)。



下图为 env 的结构：

```
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

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

注释已经非常详细。以下为补充
1.其中“enum EnvType env_type”:在该文件中也有定义。一共两种状态：ENV_TYPE_USER 和 ENV_TYPE_IDEL（空闲）.

2.其中“unsigned env_status”只用四种取值：
ENV_FREE,ENV_RUNNABLE,ENV_RUNNING,ENV_NOT_RUNNABLE

3.env_tf
其定义在 inc/trap.h,当内核或者其他的用户环境在运行时,该结构保存着当这个环境
不运行时寄存器的值,内核当由用户态转向内核态时保存这些值,以便之后还可以 resume
该环境。

作业 1：
要求：在 kern/pmap.c 的函数 mem_init()中为 envs 数组分配空间，并将其映射到虚拟地址 ENVS 上。
答：这与 lab2 中为 pages 数组分配空间和将其映射到 UPAGES 上类似。都是使用 boot_alloc()来分配空间，然后使用 boot_map_region()来映射地址。
其中 boot_alloc()的为 envs 分配指定大小的空间。boot_map_region()将指定物理地址映射到指定 va。
以下为代码截图：

```
// LAB 3: Your code here.
envs=(struct Env*)boot_alloc(NENV*sizeof(struct Env));
```

```
    boot_map_region(kern_pgdir,UENVS,ROUNDUP(NENV*sizeof(struct Env),PGSIZE)
,PADDR((uintptr_t*)envs),PTE_U);
```

作业 2：

在 kern/env.c 中编写运行一个用户线程所需要的代码。下面依次解释各个函数

1.env_init():

该函数主要作用是初始化 envs 数组，将 envs 数组所有项的 env_id 和 env_status 标志为空 闲状态，即此时没有线程使用该结构。同时将所有结构加入到 env_free_list 链表中，该链表保存所有空闲的 env 结构，主要用处是方便为新线程分配 env 结构以及会后结束的线程的 env 结构。

以下为代码截图：

```
void
env_init(void)
{
    // Set up envs array
    // LAB 3: Your code here.
    for(int 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();
}
```

2.env_setup_vm():

该函数为每一个线程分配一个页目录。

原因：首先回忆一下 lab2 中页目录和页表的构成，每个页目录项对应一个页表(1024 个页表项)，所以页目录项值的前 20 位即可确定页表的位置，剩下的 12 位是各种权限标志位。由于每一个线程拥有不同的地址空间，如果所有线程共用一个页目录，而权限标志位有限，所以就将无法标识各个线程的权限，最终导致各个线程的内存越界。

函数设计思路：

由于所有程序（内核和用户程序）共用同样的页表，所以函数只需要 copy 一份 kern_pgdir 到 env_pgdir 即可。这样的话线程通过访问 env_pgdir 即可访问到目的页面，而且只有在该线程的 env_pgdir 中标记为可访问的页面，才能被访问。如一来，便解决了线程互相越界的隐患。

代码截图：

```
e->env_pgdir=page2kva(p);
for(int i=0;i<1024;i++){
   e->env_pgdir[i]=kern_pgdir[i];//kern_pgdir一共1024个目录项
}
p->pp_ref++;
```

3.region_alloc():

问题：Allocates and maps physical memory for an environment

该函数为一个线程分配指定大小的物理页，并且将物理页映射到指定 va 上。

代码设计思路：

通过 lab2 知道使用 page_alloc 来分配一个页，page_insert 将一个物理页映射到指定 va 上。由于分配的页面物理上不一定连续，所以不能使用 boot_map_region().

接下来要解决的就是如何知道分配多少页。根据 region_alloc()的第二个参数(起始 va)和第三个参数(空间大小)，我们可以使用 for 循环来不断分配物理页。由于 page_alloc 要求地址与 4K 对齐，同时为了保证足够的页面，所以 va 要下对齐，va+len 要上对齐。

代码截图：

```
    //使用page_alloc来分配页
    void* lower=ROUNDDOWN(va,PGSIZE);
    void* upper=ROUNDUP(va+len,PGSIZE);
    struct Page*p;//用于保存分配的页面
    for(va=lower;va<upper;va+=PGSIZE){
       p=page_alloc(1);
       //因为page_alloc是根据page_free_list来分配空闲页的，而boot_map_region
是将连续的pa映射到va
       //所以使用page_insert来将物理页映射到虚拟地址;
       page_insert(e->env_pgdir,p,va,PTE_U|PTE_W);
```

4.load_icode():

该函数在第一个进程开始前为进程加载可执行文件。因为到目前还没有建立文件系统，所以我们不能正常加载进程的可执行文件，JOS 便提前在内核文件中以 Elf 格式嵌入进程的二进制镜像。这样在 lab2 中加载内核文件时，第一个进程的可执行代码就被加载到内存中了。这时我们只需要根据 inc/elf.h 中 Proghdr 结构中 p_type(当为 ELF_PROG_LOAD 时，即为可执行程序的段)的值来将可执行文件加载线程的虚拟空间中。其中 ELF_PROG_LOAD=1 也在 inc/elf.h 中有定义。

代码设计分析：

（1）在 lab2 的页式管理内存时，我们注意到在内核文件 kern/entry.S 中，将 CR0 的 PG 标志位置为 1,表示此后将启用页管理内存。

根据 wikipedia 中对 CR0 和 CR3 的介绍(下图)同时我们应该知道，**当启用 paging 时，CR3 将作为系统页目录。**

所以在 kern/pmap.c 中，JOS 在检查完 kern_pgdir(check_kern_pgdir())之后，就将 kern_pgdir 加载到了 cr3,方便今后内核对虚拟内存的访问。因为进程根据自己的 env_pgdir 进行内存的访问，所以为了在 env_pgdir 中方便的使用虚存地址，我们在 load_icode()刚开始就把 env_pgdir 加载到 cr3 中，但是**在 load_icode 结束时要还原 cr3**.

CR0 的 PG 标志位：

31	PG	Paging	If 1, enable paging and use the CR3 register, else disable paging

CR3 的作用：

CR3 [edit]

Used when virtual addressing is enabled, hence when the PG bit is set in CR0. CR3 enables the processor to translate linear addresses into physical addresses by locating the page directory and page tables for the current task. Typically, the upper 20 bits of CR3 become the *page directory base register* (PDBR), which stores the physical address of the first page directory entry.

由介绍可知，CR3 将虚拟的线性地址转换为物理地址，因此首先把 env_pgdir 加载到 CR3 中。

为了我们方便地将 env_pgdir 加载 CR3 中，JOS 在 inc/x86.h 中为了提供了一个 lcr3()的函数，该函数直接调用汇编语句将指定地址加载到 CR3，其参数页目录的物理地址，因此我们可以直接调用 lcr3。

即 lcr3(e->env_pgdir);

（2）再一次大致分析一下 inc/Elf 文件格式。（只介绍将要用到的）

Elf 文件包含三个结构体。struct Elf,struct Proghdr,struct Secthdr(暂时不知道)

struct Elf：其中的 e_magic:标志是否为 Elf 文件
e_phoff:proghdr(段目录)的偏移量
e_phnum:段的数目

struct Proghdr： p_type:标志是否为可加载(第一个线程的可执行二进制)项
p_offset:具体数据段到 Elf 首部的偏移量
p_va:应该加载到的 va
p_filesz:二进制文件的大小

（3）根据注释的提示，我们在加载完二进制可执行文件后，需要为新的线程分配一个初始的栈供其使用。

而我们之前完成的 region_alloc()便是为进程分配空间，并映射到指定 va 上。所以直接调用 region_alloc 来创建栈。
同时根据注释提示，使用 region_alloc 来为 code 分配空间。
代码截图：

```
// LAB 3: Your code here.
//complian:Elf结构中的e_phoff为proghdr距Elf起始位置的距离
//proghdr中的p_offset为该结构对应的具体段距离Elf的距离
lcr3(PADDR(e->env_pgdir));//load env_pgdir to cr3
struct Elf* ELFHDR=(struct Elf*)binary;
struct Proghdr* ph,*eph;
//首先检测该二进制是否为Elf文件格式(可执行文件被以Elf格式加载到内存)
if(ELFHDR->e_magic!=ELF_MAGIC)
    panic("binary file error");
ph=(struct Proghdr*)((uint8_t*)ELFHDR+ELFHDR->e_phoff);
eph=ph+ELFHDR->e_phnum;
int i;
for(;ph<eph;ph++){
    if(ph->p_type==ELF_PROG_LOAD){
        //using region_alloc to alloc enough mem to store exectable file
        region_alloc(e,(void*)ph->p_va,ph->p_memsz);
        //copy excutable file to p_va(指明可执行文件应在的va)
        memset((void*)ph->p_va,0,ph->p_memsz);
        memmove((void*)ph->p_va,binary+ph->p_offset,ph->p_filesz);
        //char*va=(char*)ph->p_va;
        //for(i=0;i<ph->p_filesz;i++)
            //va[i]=binary[ph->p_offset+i];
    }
}
//还原cr3
lcr3(PADDR(kern_pgdir));

e->env_tf.tf_eip=ELFHDR->e_entry;
// Now map one page for the program's initial stack
// at virtual address USTACKTOP - PGSIZE.

// LAB 3: Your code here.
region_alloc(e,(void *)(USTACKTOP-PGSIZE),PGSIZE);
//struct Page*p=page_alloc(1);
////if(!p)
 // panic("no enough page");
//page_insert(e->env_pgdir,p,(void*)(USTACKTOP-PGSIZE),PTE_W|PTE_U);
```

注：其中 memeset 和 memmove 函数的声明在 inc/string.h，实现在 lin/string.c 中

5.env_create()
该函数用来创建一个进程。
代码设计思路：
该函数主要完成：
a.为新的进程分配一个 envs 数组项并初始化
b.为该 env 进程数组项分配虚拟页目录。
(a.b 有 env_alloc 函数完成)
c.加载可执行文件到指定内存
（c 由 load_icode 完成）
注：关于 env_alloc()函数也在 env.c 中，不过 JOS 已经完成了它。该函数通过检测 env_free_list 是否为空，确定是否为新进程分配进程数组，并初始化 tf 的值（指定空间和权限），该函数返回 0。
代码截图：

```
void
env_create(uint8_t *binary, size_t size, enum EnvType type)
{
    // LAB 3: Your code here.
    struct Env *newenv_store;
    if(env_alloc(&newenv_store,0)==0){
    (newenv_store)->env_type=type;
    load_icode(newenv_store,binary,size);
    }
}
```

6.env_run(struct Env *e)

该函数为新进程 e 抢占 cpu。将当前进程从 ENV_RUNNING 状态转换为 ENV_RUNNABLE，而将新进程 e 的状态转换为 ENV_RUNNING。

代码思路：（根据代码注释的提示）

首先通过 curenv(指向当前进程的数组项)检查当前进程是否存在，如果存在就将其 env_status 从 ENV_RUNNING 转换为 ENV_RUNNABLE。

然后将 e 设置为当前运行进程（curenv=e），然后将进程页目录加载到 cr3 中（前文有解释）。

**在注释提示的“Step 2”中提示使用 env_pop_tf()来恢复进程的寄存器，然后进入用户态。

现在来分析一下 env_pop_tf()函数，该函数在 kern/env.c 中被定义。

函数截图如下：

```
// Restores the register values in the Trapframe with the 'iret' instruction
.
// This exits the kernel and starts executing some environment's code.
//
// This function does not return.
//
void
env_pop_tf(struct Trapframe *tf)
{
    __asm __volatile("movl %0,%%esp\n"
        "\tpopal\n"
        "\tpopl %%es\n"
        "\tpopl %%ds\n"
        "\taddl $0x8,%%esp\n" /* skip tf_trapno and tf_errcode */
        "\tiret"
        : : "g" (tf) : "memory");
    panic("iret failed");  /* mostly to placate the compiler */
}
```

由注释可以知道，该函数使用 iret 指令来恢复参数 tf 结构中各项寄存器的值。

iret 指令：Interrupt Return，中断返回。该指令将指向指令的指针返回到 EIP 寄存器，将代码段选择子返回到 CS 寄存器，将 EFLAGS（程序状态）返回到 EFLAGS（标志）寄存器。

由此可见通过 iret 指令可以根据 tf 的值使程序返回到原来中断的地方。在我们创建进程时，调用了 load_icode 函数，该函数将程序入口地址保存在 tf 的 eip 中，所以通过 iret 指令便可以进入程序代码。

```
// 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++;
lcr3(PADDR(curenv->env_pgdir));
env_pop_tf(&(curenv->env_tf));
panic("env_run not yet implemented");
```

验证代码：

到目前位置作业 2 基本就做完了。但是根据作业说明的提示，此时由于 JOS 此时还没有创建硬件以进行从用户空间到内核空间的转换（进入用户态后不能再次回到内核态，即不能完成系统调用），所以此时使用 make qemu 时会出现 Triple fault。（make qemu 时默认执行 user/hello.c 函数，该进程在 kern/init.c 中被创建和运行。）根据提示下面使用 gdb 进行验证作业 2 的代码是否正确。

首先在 env_pop_tf 设置断点，然后单步执行，如图所示：

```
(gdb) b env_pop_tf
Breakpoint 1 at 0xf0103505: file kern/env.c, line 472.
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0xf0103505 <env_pop_tf>:     push   %ebp

Breakpoint 1, env_pop_tf (tf=0xf01a0000) at kern/env.c:472
472     {
(gdb) si
=> 0xf0103506 <env_pop_tf+1>:   mov    %esp,%ebp
0xf0103506      472     {
(gdb) si
=> 0xf0103508 <env_pop_tf+3>:   sub    $0x18,%esp
0xf0103508      472     {
(gdb) si
=> 0xf010350b <env_pop_tf+6>:   mov    0x8(%ebp),%esp
473             __asm __volatile("movl %0,%%esp\n"
```

然后一直单步执行到“iret”指令处，该指令进入用户态，再一次单步执行，如果成功执行用户进程代码（lib/entry.S 的 cmp），只需最后一步验证了

```
(gdb) si
=> 0xf0103514 <env_pop_tf+15>:  iret
0xf0103514      473             __asm __volatile("movl %0,%%esp\n"
(gdb) si
=> 0x800020:    cmp    $0xeebfe000,%esp
0x00800020 in ?? ()
(gdb) si
```

查看 lib/entry.S 可以发现，代码最后调用了“call libmain”，而 libmain 函数在 obj/user/hello.asm 中，所以程序从 entry.S 进入 hello.asm。然后根据作业提示在“int $0x30”语句处设置断点（int $0x30 在 hello.asm 中可以找到，该地址为 0x800c23）.

```
(gdb) b *0x800c23
Breakpoint 2 at 0x800c23
(gdb) c
Continuing.
=> 0x800c23:    int    $0x30
```

至此验证完毕。此时再单步 si 就会出现 triple fault,因为 hello.c 需要调用系统进行打印工作，而现在还不能系统调用。

控制权的转移：异常和中断

异常（Exception）：程序自身在代码执行过程在进行条件的检测而发生的控制权转移。如 IO 操作

中断（Interrupt）：由进程外部的异步事件所造成的。如除 0 错误或非法访问内存。

异常和中断都是控制器的转移(Protected Control Transfer),它将使得处理器从用户模式转移到内核模式而不给用户代码任何机会去干涉内核或者其他进程。
为了保证这些控制是受保护的，处理器的中断/异常机制被设计成在中断或者异常出现时，处理程序在处理器一定严格控制下进入内核，而且内核以何种方式将中断处理程序载入何处与用户进程无关。
在 x86 中，这种保护以如下方式体现：
a.IDT(Interrupt Descriptor Table)：被设立在内核的私有内存中。该表中事先定义了各种中断类型，并且记录了中断处理程序的入口地址（后续我们会初始化 IDT 表）。一旦处理器确认一个特定的中断或者异常发生了,它就将中断号作为在 IDT 中的索引。
b.TSS(The Task State Segment)：
任务状态段(The Task State Segment,TSS) 。为了保证在内核中保证中断处理程序
能够拥有一个完善的载入点(entry-point),在处理器执行中断处理程序之前,也需要一个
地方来存储旧的处理器状态,比如 EIP 和 CS 的值,这样就可以在执行完中断处理程序之
后,处理器还可以继续执行被中断的程序。当然,这部分区域必须保证不能被未授权的代码
访问,否则,有错误的程序或者恶意代码就可以简单的危害到内核(现在使用这种方式进行
攻击的依然不在少数)
因此,当 x86 处理器响应一个中断或陷阱时,会引起从用户态到内核态权限的改变,也
会切换到内核的栈。TTS 指定了新栈的 the segment selector and address。处理器将 SS, ESP, EFLAGS, CS, EIP, and an optional error code 压入新栈,然后从 IDT 加载 CS 和 EIP,设置
SS 与 ESP 为新栈。
尽管 TTS 是一个非常庞大且支持多种用途的结构,在 JOS 中,它将只被用于定义当 处
理器从用户态转移到核心态时的内核栈。由于 JOS 中的内核模式在 x86 的特权等级体系中
是 0,当进入内核模式时处理器使用 TTS 中的 ESP0 和 SS0 两部分来定义内核栈;TS 的其
他部分在 JOS 中并不被使用。

作业 3.
目的：设置 IDT。
从作业的提示中，可以知道在用户程序发生中断时（系统调用），处理器将跳转到由 TTS 中 SS0 和 ESP0 定义的内核栈;然后将异常参数压入进内核栈。但是在将参数压入进栈时有两种情况，第一种是在压完 SS，ESP，EFLAGS，CS 后需要将错误参数压入栈;第二种情况是不需要将错误参数压入栈，JOS 中在第二种情况时将 0 压入栈。对应 trapentry.S 中的两个宏定义。
情况 1：

```
+--------------------+ KSTACKTOP
| 0x00000 | old SS   |     " - 4
|      old ESP       |     " - 8
|     old EFLAGS     |     " - 12
| 0x00000 | old CS   |     " - 16
|      old EIP       |     " - 20 <---- ESP
+--------------------+
```

情况 2：

```
+--------------------+ KSTACKTOP
| 0x00000 | old SS   |      " - 4
|      old ESP       |      " - 8
|     old EFLAGS     |      " - 12
| 0x00000 | old CS   |      " - 16
|      old EIP       |      " - 20
|     error code     |      " - 24 <---- ESP
+--------------------+
```

1.在 trapentry.S 中一共有两个宏定义。
TRAPHANDLER()和 TRAPHANLER_NOEC()
TRAPHANDLER():定义一个全局可见的函数来处理陷入，其有两个参数，第一个为中断处理程序的名字，第二个为中断号。
TRAPHANDLER_NOEC:该用于处理那些不用压入错误参数的中断处理程序。
2.现在的任务就是找到哪些中断没有错误码而哪些有。
在 MIT 网站上可以找到 x86_idt 的相关描述表格（如下图），就是作业提示中让我们看的 chapter 9
网址：https://pdos.csail.mit.edu/6.828/2011/readings/i386/s09_10.htm

Description	Interrupt Number	Error Code
Divide error	0	No
Debug exceptions	1	No
Breakpoint	3	No
Overflow	4	No
Bounds check	5	No
Invalid opcode	6	No
Coprocessor not available	7	No
System error	8	Yes (always 0)
Coprocessor Segment Overrun	9	No
Invalid TSS	10	Yes
Segment not present	11	Yes
Stack exception	12	Yes
General protection fault	13	Yes
Page fault	14	Yes
Coprocessor error	16	No
Two-byte SW interrupt	0-255	No

3.因此我们根据上图来确定使用哪个宏定义来为中断定义处理函数。凡是上图中 Error code 为 NO 的都不需要压入错误码，使用宏定义 TRAPHANDLER_NOEC();余下的使用 TRAPHANDLER()。
代码如下：

```
/*
 * Lab 3: Your code here for generating entry points for the different traps
.
 */
    TRAPHANDLER_NOEC(trap_handler0,0)
    TRAPHANDLER_NOEC(trap_handler1,1)
    TRAPHANDLER_NOEC(trap_handler2,2)
    TRAPHANDLER_NOEC(trap_handler3,3)
    TRAPHANDLER_NOEC(trap_handler4,4)
    TRAPHANDLER_NOEC(trap_handler5,5)
    TRAPHANDLER_NOEC(trap_handler6,6)
    TRAPHANDLER_NOEC(trap_handler7,7)
    TRAPHANDLER(trap_handler8,8)
    TRAPHANDLER_NOEC(trap_handler9,9)
    TRAPHANDLER(trap_handler10,10)
    TRAPHANDLER(trap_handler11,11)
    TRAPHANDLER(trap_handler12,12)
    TRAPHANDLER(trap_handler,13)
    TRAPHANDLER(trap_handler14,14)
    TRAPHANDLER_NOEC(trap_handler16,16)
    TRAPHANDLER(trap_handler17,17)
    TRAPHANDLER_NOEC(trap_handler18,18)
    TRAPHANDLER_NOEC(trap_handler19,19)


/*
```

此时还没有定义系统调用中断（后续会有作业加入 T_SYSCALL）。

4.然后完成_alltraps.

Trapentry.S 为汇编文件，而且 JOS 是 32 位的，所以 x86 汇编指令。

简单介绍一下 x86 中的寄存器：

EAX,EBX,ECX,EDX:类似 8086 的 AX，BX，CX，DX

EBP:用来获得栈中的数据。

ESI，EDI：基址寄存器

SS,DS,ES,FS,GS:段寄存器

EIP：程序计数器（指令指针），一般跟 CS（代码段寄存器）一起使用

ESP：栈指针，指向栈顶

EFLAGS:标志寄存器。

根据作业提示，我们需要使用 pushal

查看 x86 汇编指令知道 pushal:push EDI,ESI,EBP,ESP,EBP,EBX,EDX,ECX,EAX。在查看 inc/trap.h 时，可以知道这恰好是结构 PushRegs 中的各个寄存器，而该结构又是结构 Trapframe 的第一个变量类型，所以一条 pushal 就将 Trapframe 中的 tf_regs 的各个寄存器值压入栈中。结构如下：

```
struct PushRegs {
    /* registers as pushed by pusha */
    uint32_t reg_edi;
    uint32_t reg_esi;
    uint32_t reg_ebp;
    uint32_t reg_oesp;      /* Useless */
    uint32_t reg_ebx;
    uint32_t reg_edx;
    uint32_t reg_ecx;
    uint32_t reg_eax;
} __attribute__((packed));
```

```
struct Trapframe {
    struct PushRegs tf_regs;
    uint16_t tf_es;
    uint16_t tf_padding1;
    uint16_t tf_ds;
    uint16_t tf_padding2;
    uint32_t tf_trapno;
    /* below here defined by x86 hardware */
    uint32_t tf_err;
    uintptr_t tf_eip;
    uint16_t tf_cs;
    uint16_t tf_padding3;
    uint32_t tf_eflags;
    /* below here only when crossing rings, such as from user to kernel */
    uintptr_t tf_esp;
    uint16_t tf_ss;
    uint16_t tf_padding4;
} __attribute__((packed));
```

由上图可以看到 tf_regs 下面是 tf_es 和 tf_ds。而根据_alltrap 的要求，我们需要将 GD_KD 的值传递给%ds 和%es。但是根据汇编知道 es 和 ds 是不能直接通过变量赋值的，所以可以使用 EAX 来作为中间媒介来传值，当然也可以使用 pushl 和 popl 来实现。根据提示：

a.将%ds 和%es 原有的值压入栈，然后把使用 pushal 把 tf_es 的值压入栈。至于 Trapframe 的其他值根据注释提示一部分是由 x86 定义，另一部分的 esp 和 ss 上文介绍过，其在从用户态转换为内核态时定义内核栈。

b.将$GD_KD 传递给%ds 和%es：首先将$GD_KD 压如栈，然后使用 popl %ds,将$GD_KD 的值传给%ds,改变%es 的值类似。(其中 GD_KD 在 inc/memlayout.h 中定义，GD：为 Global Descriptot，KD 表示内核数据空间)

c.将%esp 的值压入栈中

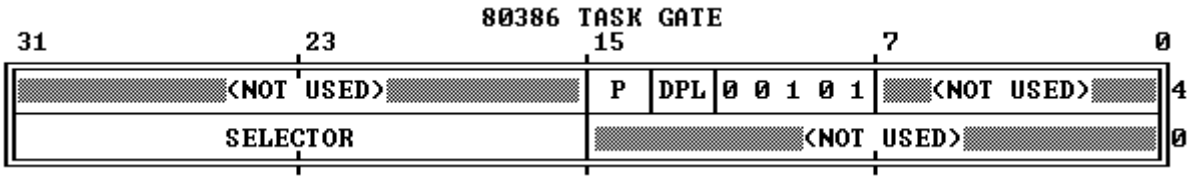
d.调用 trap

```
/*
 * Lab 3: Your code here for _alltraps
 */
_alltraps:
    pushl %ds
    pushl %es
    pushal
    pushl $GD_KD
    popl %ds
    pushl $GD_KD
    popl %es
    pushl %esp
    call trap
```

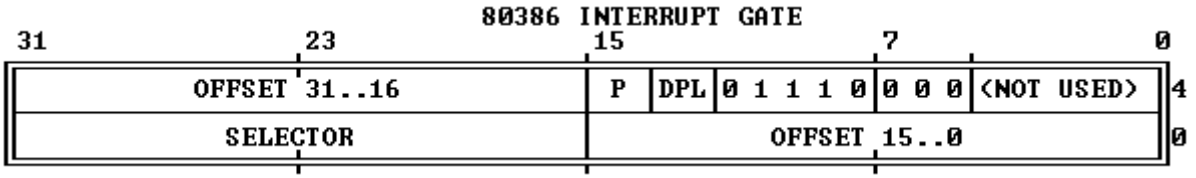
注：这里强调一下 TRAPHANDLER 和_alltraps 压栈的目的，这个在 trap 函数中非常重要。注意在_alltrap 中我们将 esp 的值压入栈中再调用 trap，其实是将栈顶地址作为参数传递给 trap，然后根据上述的对 tf 结构的分析和压栈顺寻可以知道，这时栈顶向下的一部分俨然是一个 tf 结构的一部分（tf_regs 到 trapno），其中 trapno 就是在 TRAPHANDLER 中压入的 NUM。这样在 trap 中就能用 tf_tf_**来访问内存。

5.在上面"call trap"后进入 kern/trap.c 中的 trap 函数，但是在真正运行处理中断处理程序

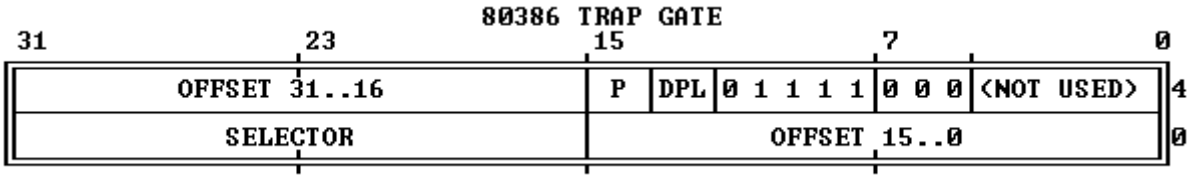
之前，我们需要完成 kern/trap.c 中的 trap_init().
根据提示需要使用 SETGATE 的宏定义，该宏定义在 inc/mmu.h 中。其作用是建立一个中断或者异常描述符。
#define SETGATE(gate, istrap, sel, off, dpl)
第一个参数 gate：表示该中断/异常描述符在 IDT 的位置表项如 idt[0]
第二个参数 istrap：区分是 trap gate 还是 interrupt gate;1 代表 trap(exception)gate，0 代表 interrupt gate。
Trap gate 和 interrupt gate 的区别：
1.在 https://pdos.csail.mit.edu/6.828/2011/readings/i386/s09_10.htm 中 9.5 的 IDT Descriptors 中介绍了三种 gate：Task gate;Interrupt gate;Trap gate
Task gate：



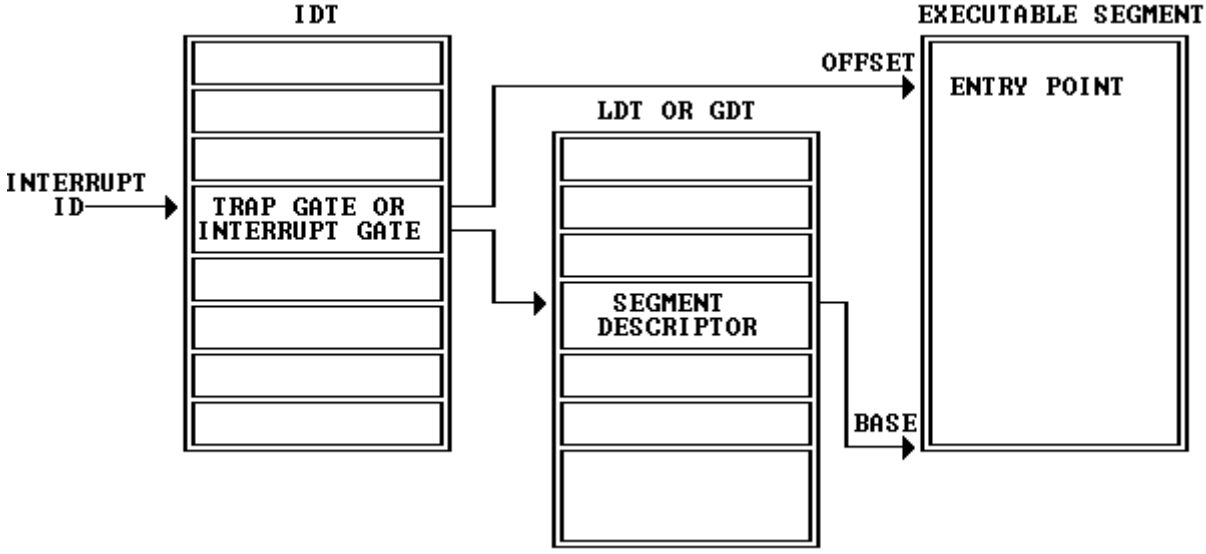
Interrupt gate：



Trap gate：



这里分析一下 trap gate 和 interrupt gate。interrupt 和 exception 可以使用 call 指令来调用相应的处理程序。当一个进程需要响应 interrupt 或者 exception 时，进程使用 interrupt 和 exception 的标识符如 traphandler0（在 trapentry.S 中定义，在 trap.c 中将表述符与 idt 表项连接）在 idt 中检索，如果找到则进入相应的程序段。

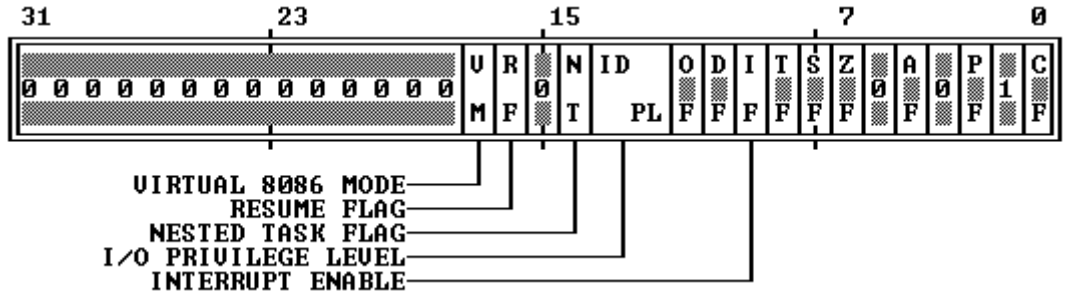


区别：
The difference between an interrupt gate and a trap gate is in the effect on IF (the interrupt-enable flag). An interrupt that vectors through an interrupt gate resets IF, thereby preventing other

interrupts from interfering with the current interrupt handler. A subsequent IRET instruction restores IF to the value in the EFLAGS image on the stack. An interrupt through a trap gate does not change IF.

上述这段话在前文网址的 9.6.1.3.

EFLAGS：



简单的说，就是 interrupt 会将 EFLAGS 的 IF 标志位置 0，由上图可知 IF 是（Interrupt Enable）标志位，如果为 0 则不再响应其他 interrupt。也就是说在一个 interrupt 没有处理完之前，其他 interrupt 不能得到响应，而 trap 则不会，即后续的 trap 可能会打断前面正在执行的 trap handler。

由前文介绍（https://pdos.csail.mit.edu/6.828/2011/readings/i386/s09_10.htm Error code 的图）可知我们目前写的都是 interrupt 处理程序，所以 istrap 为 0.

第三个参数：sel：Code Segment selector for interrupt/trap handler。前文说过所有 interrupt/trap 处理函数都是由系统内核加载和处理，与当前执行的用户程序无关，所以第三个参数设为 GD_KT

第四个参数：offset in code segment for interrupt/trap handler.即代码段各个函数的首地址，使用函数名来标记。

第五个参数 dpl(Descriptor Privilege Level)：这里定义的 interrupt 都是在运行是产生(比如发现除数为 0)，并不存在用户的调用，所以权限等级设为 0,只有内核能调用。

至此 trap_init()分析完毕，代码截图如下：

```
// LAB 3: Your code here.
extern void trap_handler0();
extern void trap_handler1();
extern void trap_handler2();
extern void trap_handler3();
extern void trap_handler4();
extern void trap_handler5();
extern void trap_handler6();
extern void trap_handler7();
extern void trap_handler8();
extern void trap_handler10();
extern void trap_handler11();
extern void trap_handler12();
extern void trap_handler13();
extern void trap_handler14();
extern void trap_handler16();
extern void trap_handler17();
extern void trap_handler18();
extern void trap_handler19();
extern void syscall_handler();
```

```
SETGATE(idt[0],1,GD_KT,trap_handler0,0);
SETGATE(idt[1],0,GD_KT,trap_handler1,0);
SETGATE(idt[2],0,GD_KT,trap_handler2,0);
SETGATE(idt[3],0,GD_KT,trap_handler3,3);
SETGATE(idt[4],0,GD_KT,trap_handler4,0);
SETGATE(idt[5],0,GD_KT,trap_handler5,0);
SETGATE(idt[6],0,GD_KT,trap_handler6,0);
SETGATE(idt[7],0,GD_KT,trap_handler7,0);
SETGATE(idt[8],0,GD_KT,trap_handler8,0);
SETGATE(idt[10],0,GD_KT,trap_handler10,0);
SETGATE(idt[11],0,GD_KT,trap_handler11,0);
SETGATE(idt[12],0,GD_KT,trap_handler12,0);
SETGATE(idt[13],0,GD_KT,trap_handler13,0);
SETGATE(idt[14],0,GD_KT,trap_handler14,0);
SETGATE(idt[16],0,GD_KT,trap_handler16,0);
SETGATE(idt[17],0,GD_KT,trap_handler17,0);
SETGATE(idt[18],0,GD_KT,trap_handler18,0);
SETGATE(idt[19],0,GD_KT,trap_handler19,0);
SETGATE(idt[T_SYSCALL],0,GD_KT,syscall_handler,3);
 //Per-CPU setup
```

以上两张图都在 `trap_init()`中

至此作业 3 完成。

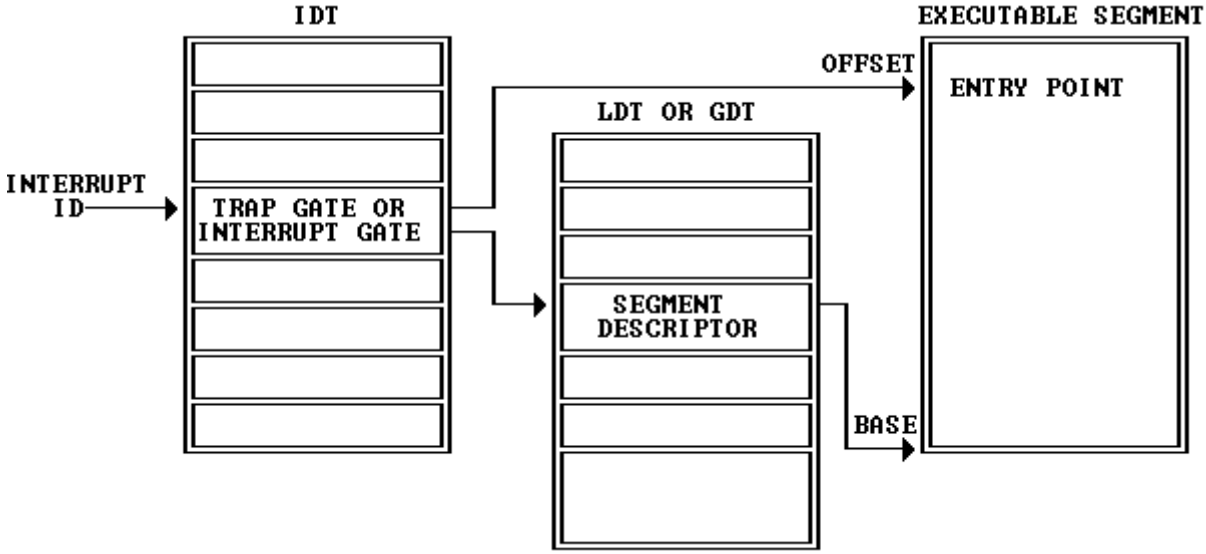
到这里我们可以根据提示使用 make geade 来检查我们的代码：

```
divzero: OK (2.3s)
softint: OK (1.6s)
badsegment: OK (1.6s)
Part A score: 30/30
```

至此就 Part A 完成了。

问题 1：

1 . What is the purpose of having an individual handler function for each exception/interrupt? (i.e., if all exceptions/interrupts were delivered to the same handler, what feature that exists in the current implementation could not be provided?)



答：上图描述了 Interrupt 执行的过程，首先程序根据中断标识符在 IDT 中检索，找到之后通过 GDT 的信息进入处理程序。但是如果所有的 Interrupt 都跳转到同一个处理程序，那么在该处理程序中很难分辨是那种 interupt，从而很难确定执行那一分支语句。

因为从 trapentry.S 中“#define TRAPHANDLER(name, num)”可以看到处理函数只执行了两条指令。即 push 一个常数值(trapno)，再跳转到_alltrap。所以如果所有中断号都映射到一个 handler 时，即没法在 handler 中区分 trapno。

2.Did you have to do anything to make the user/softint program behave correctly? The grade script expects it to produce a general protection fault (trap 13), but softint's code says int $ 14. Why should this produce interrupt vector 13? What happens if the kernel actually allows softint's int $14 instruction to invoke the kernel's page fault handler (which is interrupt vector 1 4)?

答：因为我们在 kern/trap.c 的 trap_init()函数中，将 Interrupt Handler 的第五个参数(中断描述符权限)都设置为 0,只能由内核调用。而 int 14 中断是用户产生的中断，所以会显示 general protection。

注：`softint`（`user` 中的 `softint.c`，调用 `int 14` 缺页中断）由 `grade script` 调用来检测是否会产生 `general protection` 中断。

如下图：（运行 mkae run-softint）

```
Incoming TRAP frame at 0xefbfffbc
TRAP frame at 0xf01a0000
  edi  0x00000000
  esi  0x00000000
  ebp  0xeebfde60
  oesp 0xefbfffdc
  ebx  0x00000000
  edx  0xeebfde88
  ecx  0x0000000d
  eax  0x00000000
  es   0x----0023
  ds   0x----0023
  trap 0x0000000d General Protection
  err  0x00000182
  eip  0x00800c23
  cs   0x----001b
  flag 0x00000092
  esp  0xeebfde54
  ss   0x----0023
[00001000] free env 00001000
```

如果内核允许产生 int 14 中断，即将 trap_init()中 trap_handler14 的第五个参数设置为 3（3 表示允许用户调用）
“SETGATE(idt[14],0,GD_KT,trap_handler14,3);”
此时会产生 page fault。（记得测试完改回，因为 make grade 测试 softint 时要检测是否产生 general protection fault,如果没有则该项算没过。）

缺页中断，断点异常，系统调用

作业 4.

问题：Modify trap_dispatch() to dispatch page fault exceptions to page_fault_handler(). You should now be able to get make grade to succeed on the faultread, faultreadkernel, faultwrite, and faultwritekernel tests. If any of them don't work, figure out why and fix them. Rememberthat you can boot JOS into a particular user program using make run-x or make run-x-nox.

答：在_alltraps 的最后一句我们调用了 trap，该函数在 kern/trap.c 中。在 trap()函数中，会调用 trap_dispatch()函数，该函数根据中断号来调用不同的中断处理程序。而且 page_fault_handler()函数 JOS 已经给出。因此只需在 trap_dispatch()中加入一条条件跳转语句即可。
代码截图：

```
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);
```

作业 5.

问题：Modify trap_dispatch() to make breakpoint exceptions invoke the kernel monitor. You should now be able to get make grade to succeed on the breakpoint test.

答：和作业四类似。breakpoint 中断号为 3 号。为了使用户能够调用该中断，在 trap_init()中，将 3 号的 dpl 改为 3.然后在 trap_dispatch()中添加一条条件跳转语句，跳转到 monitor()。monitor()是在

kern/monitor.c 中定义的函数。
代码截图：

```
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);
    if(tf->tf_trapno==T_BRKPT)
       monitor(tf);
```

问题 2：
1.题目：The break point test case will either generate a break point exception or a general protection fault depending on how you initialized the break point entry in the IDT (i.e., your call to SETGATE from trap_init). Why? How do you need to set it up in order to get the breakpoint exception to work as specified above and what incorrect setup would cause it to trigger a general protection fault?
答：因为在 trap_init()中 SETGATE 时，将最后一位权限全部设置为 0,也就是说这些中断只能由系统调用。所以当用户试图调用时，将会提示 general protection fault.这个与问题一的第二问类似。
如果我们希望正常的处理 breakpoint 中断，可以把 trap_init()中的权限为改为 3.
未改变权限之前：

```
breakpoint: missing '  trap 0x00000003 Breakpoint'
```

改变权限之后：

```
breakpoint: OK (1.6s)
```

2.题目：What do you think is the point of these mechanisms, particularly in light of(根据)what the user/softint test program does
答：这些机制的重点是权限的设置。通过 SETGATE 权限位的设置，可以使一些中断处理程序只能由内核调用。只将一部分特定的中断提供给用户，保护了内核的中断调用程序。user/softint 函数中调用系统断页中断(14 号)，所以首先在 IDT 中检索 14 号中断的处理函数 trap_handler14,IDT 也会提供 DPL 的值，而在 trap.c 中的 SETAGTE 时，14 号缺页中断的 dpl 为 0,即只能由内核调用，所以会出现 general protection fault。
系统调用：
1.用户进程通过系统调用来要求内核完成一些工作，比如读入/打印字符，然后处理器进入内核模式，执行中断处理程序，完成系统调用后，恢复用户进程。
JOS 实验在 inc/trap.h 中定义系统掉用的中断号为 48.
2.由于系统调用由不同的功能，所以以编号来区分用户进程需要使用哪种系统功能。编号放在在%eax 寄存器中。同时我们还可以在%edx,%ecx,%edi,%esi 中放入相关的参数，内核将返回的结果 放入%eax 中。

作业 6.
问题:为 T_SYSCALL 添加处理程序
答：（1）在 kern/trapentry.S 中使用 TRAPHANDLER_NOCE 为 T_SYSCALL 添加全局函数

```
TRAPHANDLER_NOEC(syscall_handler,T_SYSCALL);
```

（2）在 kern/trap.c 的 trapinit 中使用 SETGATE 将 syscall_handler 与 IDT 关联。并将其 dpl 设置为 3（用户可调用）。

```
SETGATE(idt[T_SYSCALL],0,GD_KT,syscall_handler,3);
```

（3）a.根据之前的分析知道，syscall_handler 执行完_alltraps 之后跳转到 kern/trap.c 中的 trap()函数中，该函数会调用 trap_dispatch()来根据不同 trapno 调用不同处理程序。所以需要在 trap_dispatch()中根据 trapno 来调用不同的处理函数。

b.下图中 syscall 函数 JOS 已经在 kern/syscall.c 中给出，该函数原型：

“int32_t syscall(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t a3, uint32_t a4, uint32_t a5)”

由原型可知该函数接受五个参数。具体代码参见 kern/syscall.c。

```
    int flag;
    if(tf->tf_trapno==T_SYSCALL){
    cprintf("System call\n");
       flag=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);
    if(flag<0)
      panic("syscall error.");
    tf->tf_regs.reg_eax=flag;
```

此时系统调用的处理函数就完成了。根据作业要求运行 make qemu 会打印 hello,之后会出发用户缺页中断，page_fault_handler 还未完成。

至于为什么会触发缺页中断，可以在 user/hello.c 中看到该函数除了打印”hello world”,还有如下语句

cprintf("i am environment %08x\n", thisenv->env_id);

在把该语句注释掉后就没有 page_fault 了。所以可以断定该语句会产生一个缺页中断。原因：因为 thisenv 变量在 lib/libmain.c 中，libmain()的作用是初始化 env 数组再调用 umian()，而此时我们还未在 libmian()初始化 thisenv。所以此时 thisenv 并不存在与内存中，所以会发生缺页中断。

也可以参见 obj/user/hello.asm 中的汇编，该汇编包括 lib/entry.S 和 lib/libmian 等。

效果图：

```
[00000000] new env 00001000
Incoming TRAP frame at 0xefbfffbc
hello, world
Incoming TRAP frame at 0xefbfffbc
[00001000] user fault va 00000048 ip 0080004a
TRAP frame at 0xf01a0000
  edi  0x00000000
  esi  0x00000000
  ebp  0xeebfdfd0
  oesp 0xefbfffdc
  ebx  0x00000000
  edx  0xeebfde88
  ecx  0x0000000d
  eax  0x00000000
  es   0x----0023
  ds   0x----0023
  trap 0x0000000e Page Fault
  cr2  0x00000048
  err  0x00000004 [user, read, not-present]
  eip  0x0080004a
  cs   0x----001b
  flag 0x00000092
  esp  0xeebfdfb8
  ss   0x----0023
[00001000] free env 00001000
```

3.4 启动用户模式

这里先分析一下代码执行的过程：

1. 在 lab2 中我们知道，引导加载程序 boot/main.c 加载完内核后，进入 kern/entry.S 中开启虚拟地址和页表管理;之后进入 kern/init.c 的函数 i386_init()中进行内存初始化，进程数组初始化，中断初始化;之后使用宏定义 ENV_CREATE 来创建进程(JOS 创建的是 user_hello 进程)。
2.在使用 ENV_CREATE 创建进程时，使用的 kern/env_create 函数，该函数中调用 env_alloc 为进程分配内存，load_icode 加载 hello 的二进制文件;注意在 load_icode 中，我们将 ELFHDR->entry 赋值给 tf.eip。
3.在创建完进程后，kern/init.c 调用 env_run 函数，env_run 函数调用 env_pop_tf，env_pop_tf 恢复进程环境状态后使用 iret 从 kern 态进入用户态，执行用户代码，即进入 lib/entry.S，然后调用 libmain 函数进入 lib/libmain.c 中;该函数调用 umain,进入 user/hello.c 中(默认情况下).

作业 7.
问题：在 lib/libmain.c 中添加代码，将指针 thisenv 指向当前进程(curenv).
分析：在 3.4 的第一部分已经分析了代码的执行流程。在作业 6 中由于在使用未初始化的 thisenv 所以出现 page_fault（缺页中断，即 thisenv 执行的 env 不在内存中）。在这里根据作业提示我们使用 sys_getenvid()函数（在 inc/libmain.h 中引用了 kern/syscall.c 中的函数）来获得 curenv 的 id。
注：使用宏 ENVX(inc/env.h)来将 envid 转换为 envs 数组中的 index.（在 lab 最开始介绍过）
答案截图：

```
// LAB 3: Your code here.
//thisenv = 0;
thisenv=&envs[ENVX(sys_getenvid())];
```

作业 8.
1.问题：修改 kern/trap.c，使得在内核出现缺页时使用 panic()中断内核。
分析：根据提示，可以通过检查 tf_cs 的最低几位来判断缺页中断是发生在用户模式还是内核模式。在 inc/trap.h 的 Trapframe 结构体中，tf_cs 由 x86 硬件定义。我们可以在 kern/trap.c 的 trap()函数中看到当 tf_cs 的最低几位为 3 时为用户模式，在介绍宏 SETGATE 时知道内核的 dpl 为 0.
答案截图：

```
// Handle kernel-mode page faults.
if((tf->tf_cs&3)==0)
  panic("kernl page fault.");
// LAB 3: Your code here.
```

2.问题：修该 kern/pmap.c 的 user_mem_check()函数，用来检查指定区域是否能被指定进程访问
分析：由之前的实验知道，每个进程都有一个独立的页目录，该页目录管理该进程的所以可访问页问页面。所以我么只需要检测指定[va,va+len)对应的页表项是否存在，如果存在再检查指定进程是否由足够的权限访问页面。
在 lab2 中，我们使用 pgdir_walk()函数来找指定 va 对应的页表项，此时我们也使用该函数。页表项的后十二位为权限位，用权限位跟 PTE_U|PTE_P|perm 进行与操作。
其中 user_mem_check_addr 是越界访问时的页面地址。make grade 会用到。
答案截图：

```
int
user_mem_check(struct Env *env, const void *va, size_t len, int perm)
{
    // LAB 3: Your code here.
    void*buttom=ROUNDDOWN((void*)va,PGSIZE);
    void*top=ROUNDUP((void*)(va+len),PGSIZE);
    uintptr_t check_addr=(uintptr_t)va;
    if((uintptr_t)top>=ULIM){
      user_mem_check_addr=(uintptr_t)(va+len);
      return -E_FAULT;}
    perm=perm|PTE_U|PTE_P;
    pte_t*pte;
    user_mem_check_addr=(uintptr_t)check_addr;
    //使用pgdir_walk()获得va对应的页表项。
    for(buttom;buttom<top;buttom+=PGSIZE){
       pte=pgdir_walk(env->env_pgdir,buttom,0);
       if(pte==NULL||(*pte&perm)!=perm)
          return -E_FAULT;
       user_mem_check_addr=(uintptr_t)buttom;
       //check_addr+=PGSIZE;
    }
    return 0;
}
```

3.问题：修改 kern/syscall.c 中的 sys_cputs 函数，让其在打印字符串之前先检查一下当前进程是否有足够的权限访问该字符串。

分析：在问题 2 中我们完成了 kern/pmap.c 的 user_mem_check 函数，该函数的功能就是检查指定进程对指定区域是否由足够权限。

答案截图：

```
// LAB 3: Your code here.
user_mem_assert(curenv,s,len,PTE_U);
```

此时启动内核，运行 user/buggyhello 程序(make run-buggyhello):

该函数使用一个未映射的指针来产生权限不足的 failure 供 grade script 使用。

```
[00001000] user_mem_check assertion failure for va 00000001
[00001000] free env 00001000
Destroyed the only environment - nothing more to do!
```

4.修改 kern/kdebug.c 的 debuginfo_eip()函数。

分析：其中的 STABS 全称为：Symbol Table Entries。是一种将程序描述给 debugger 的信息格式。该数据结构 struct stab 在 inc/stab.h 中定义。

具体介绍参见：https://sourceware.org/gdb/onlinedocs/stabs.html

这里我们只需要使用 user_mem_check()来检测该区间是否可访问。如果没有找到（即不可以访问就返回-1）

然后根据提示检查 stabs 到 stabs_end 是否可以访问;stabstr 到 stabstr_end 是否可以访问。

答案截图：

```
        // Make sure this memory is valid.
        // Return -1 if it is not.  Hint: Call user_mem_check.
        // LAB 3: Your code here.
        if(user_mem_check(curenv,usd,sizeof(struct UserStabData),PTE_U)!=0)
          return -1;
        stabs = usd->stabs;
        stab_end = usd->stab_end;
        stabstr = usd->stabstr;
        stabstr_end = usd->stabstr_end;

        // Make sure the STABS and string table memory is valid.
        // LAB 3: Your code here.
        if(user_mem_check(curenv,stabs,stab_end-stabs,PTE_U)!=0||user_mem_ch
eck(curenv,stabstr,stabstr_end-stabstr,PTE_U)!=0)
          return -1;
```

5.问题：运行 user/breakpoint，然后使用 mointor.c 的 backtrace 观察函数运行。

分析：a.本实验的 monitor.c 中没有 bakctrace 指令，需要我们自己添加。

b.在 lab2 中我们修改了 kern/monitor.c 的 mon_backtrace 函数，使其打印函数调用时栈的变化。这里我们依然使用同样的方法来观察栈的变化。

c.在问题 4 中，我们完善了 debuginfo_eip()函数，该函数返回指定指令的 info，该信息的格式为 Eipdebuginfo 结构。所以我们可以在 mon_backtrace()中使用 debuginfo_eip()来得到当前 eip 的信息。

代码如下：

```
int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
    // Your code here.
    uint32_t* ebp = (uint32_t*) read_ebp();
    cprintf("Stack backtrace:\n");
    while (ebp) {
        uint32_t eip = ebp[1];
        cprintf("ebp %x  eip %x  args", ebp, eip);
        int i;
        for (i = 2; i <= 6; ++i)
            cprintf(" %08.x", ebp[i]);
        cprintf("\n");
        struct Eipdebuginfo info;
        debuginfo_eip(eip, &info);
        cprintf("\t%s:%d: %.*s+%d\n",
            info.eip_file, info.eip_line,
            info.eip_fn_namelen, info.eip_fn_name,
            eip-info.eip_fn_addr);
//         kern/monitor.c:143: monitor+106
        ebp = (uint32_t*) *ebp;
    }
    return 0;
}
```

此时在运行 user/breakpoint,然后使用 backtrace 命令。

```
K> backtrace
Stack backtrace:
ebp efbfff10  eip f0100948  args 00000001 efbfff28 f01a1000 00000000 f017ea80
            kern/monitor.c:146: monitor+285
ebp efbfff80  eip f0103e9a  args f01a1000 efbfffbc 00000000 00000000 00000000
            kern/trap.c:186: trap+185
ebp efbfffb0  eip f0103fe0  args efbfffbc 00000000 00000000 eebfdfd0 efbfffdc
            kern/syscall.c:68: syscall+0
ebp eebfdfd0  eip 800078  args 00000000 00000000 00000000 00000000 00000000
            lib/libmain.c:26: libmain+63
ebp eebfdff0  eip 800031  args 00000000 00000000Incoming TRAP frame at 0xefbffe7
c
kernel panic at kern/trap.c:257: kernl page fault.
Welcome to the JOS kernel monitor!
```

分析上图：a.首先我们知道在进入 entry.S 后，会调用 libmain.c 函数;b.然后该函数调用 sys_getenvid()函数(属于系统调用);c.然后进入 breakpoint 的 umain 函数中 d.umain 函数调用 int 3 中断进入 trap.c 中，由 trap.c 的 trap_dispatch()进入 kern/monitor.c 的 monitor 函数。刚好符合上图。

作业 9.

运行 user/evilhello

```
[00000000] new env 00001000
Incoming TRAP frame at 0xefbfffbc
Incoming TRAP frame at 0xefbfffbc
[00001000] user_mem_check assertion failure for va f0100070
[00001000] free env 00001000
Destroyed the only environment - nothing more to do!
Welcome to the JOS kernel monitor!
```

最后 make grade：

```
divzero: OK (1.6s)
softint: OK (1.6s)
badsegment: OK (1.6s)
Part A score: 30/30

faultread: OK (1.6s)
faultreadkernel: OK (1.6s)
faultwrite: OK (1.6s)
faultwritekernel: OK (1.6s)
breakpoint: OK (1.6s)
testbss: OK (1.6s)
hello: OK (1.6s)
buggyhello: OK (1.6s)
buggyhello2: OK (1.6s)
evilhello: OK (1.6s)
Part B score: 50/50

Score: 80/80
```