fudos2020: environents

exercise2

env_init()

由于在mem_init中已经为envs数组分配了内存，env_init只需要初始化envs数组中的每一个 struct env 的 env_id、env_type、env_status、env_tf 等字段即可。同时要将 envs 数组通过 env_link 字段按顺序构造成一个单链表，且让 env_free_list 的值为链表的第一个表元的地址。代码如下

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

    if(!envs){
        panic("envs not set yet!");
    }
    for(i=0;i<NENV-1;i++){
        envs[i].env_type = ENV_TYPE_USER;
        envs[i].env_parent_id = 0;
        memset(&(envs[i].env_tf),0,sizeof(struct Trapframe));
        envs[i].env_id = 0;
        envs[i].env_link = &envs[i+1];
        envs[i].env_status = ENV_FREE;
        envs[i].env_pgdir = NULL;
        envs[i].env_runs = 0;
    }
    envs[NENV-1].env_id = 0;
    envs[NENV-1].env_link = NULL;
    envs[NENV-1].env_status = ENV_FREE;
    envs[NENV-1].env_parent_id = 0;
    memset(&(envs[i].env_tf),0,sizeof(struct Trapframe));
    envs[i].env_status = ENV_FREE;
    env_free_list = envs;

    //panic("env_init not yet implemented");
    // Per-CPU part of the initialization
    env_init_percpu();
}
```

env_create

利用env_alloc为要装载的 environment 申请一个 env 结构，若env_alloc失败，输出异常信息并 return。

env_alloc 中会调用env_set_vm 设置给定 env 的env_pgdir，即对应 environment 所使用的二级页表的页表目录，并计算 env_id 同时设置 env_tf 中的段寄存器为相应的段选择子(GD_UD、GD_UT 以及 选择子中的标志位)，并设置 env_tf 中的 esp 。

```
    struct Env *new_env_ptr=0;
    int res;
    res = env_alloc(&new_env_ptr,0);
    if(res<0){
        panic("exception in env_create: %e", res);
        return;
    }
```

设置 env_type 为给定的参数 type，通过 load_icode 将 binary 文件装载进内存。load_icode 中将指定将 elf 文件中设置好的各段地址作为各段的虚拟地址，通过 region_alloc 将其装载到内存中并设置好 env 对应页表中的内存映射。

```
    new_env_ptr->env_type = type;
    load_icode(new_env_ptr,binary);
```

env_run

将 curenv 的 env_status 设置为 ENV_RUNNABLE （若其原本为 ENV_RUNNING）

```
    if(curenv && (curenv->env_status == ENV_RUNNING)){
        curenv->env_status = ENV_RUNNABLE;
    }
```

将 curenv 指向新 env 的地址，并设置 env_status 以及 env_runs

```
    curenv = e;
    e->env_status = ENV_RUNNING;
    e->env_runs++;
```

通过 lcr3 将 env 中的 page_dir 放入 cr3 寄存器，即更改虚拟内存映射关系为新 env 页表指定的映射关系。

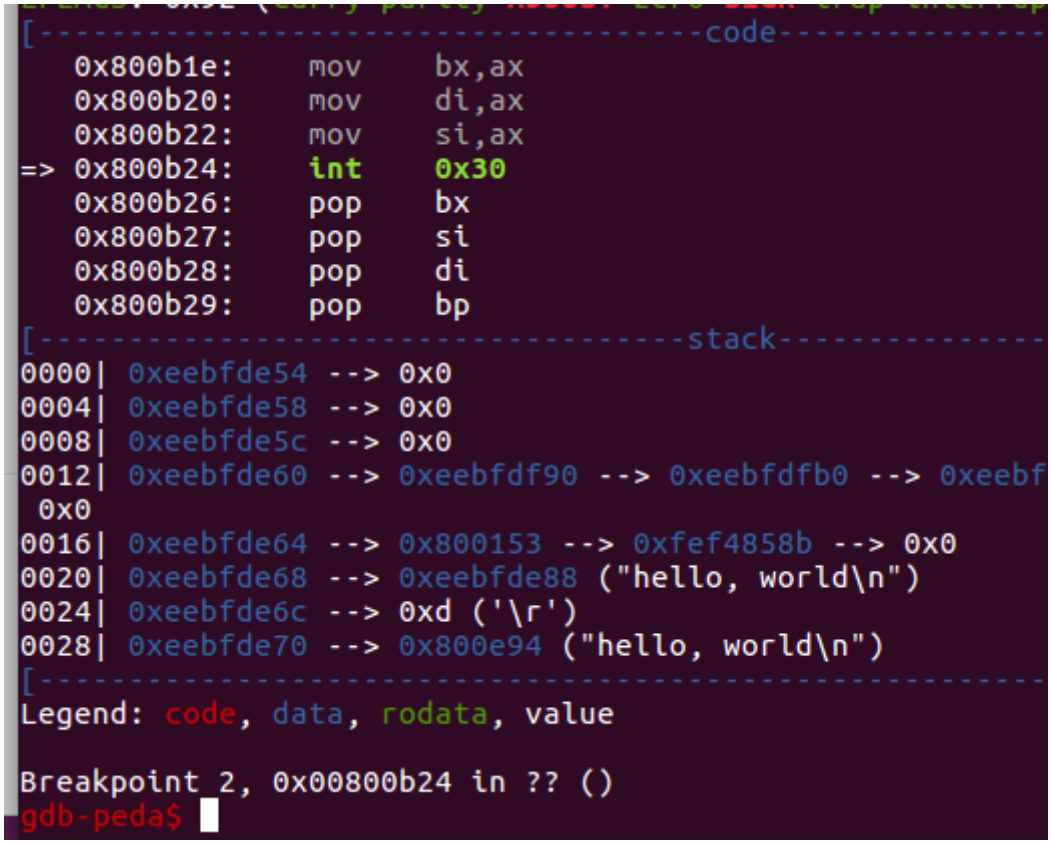
再通过 env_pop_tf 将 env_tf 中的 context 放入对应的寄存器中，通过 iret 将控制流交给用户程序 (即更改 eip 为用户 elf 的entry)

```
    lcr3(PADDR(e->env_pgdir));
    env_pop_tf(&(e->env_tf));
```

exercise2完成之后，运行 jos 后，内核会装载 hello 作为对应的 env，可以一直运行到 hello 中的第一个 trap 处（hello中是int 0x30），此时由于 idt 表还未设置， 处理 trap 时就会遇到一个空指针，于是 jos 就会反复重启。

此时可以通过跟踪 hello 对应 env 的运行过程来确定是否在执行hello 中的指令。当 env_pop_tf 成功执行，虚拟内存映射方式改变后，可以将断点设置到 hello 中的 int 0x30 处，并continue。若能运行到该处，说明 hello 运行正常。

截图如下



可以看到运行到了0x800b24处的 int 0x30

exercise4

设置每个 trap 的 entry

使用已经给的宏，为序号为0-19的每个需要的 trap/interrupt 设置一个 entry，宏的参数 name 为 entry 的符号的名字，宏的参数 num 为 interrupt 的序号

这里哪些trap需要errorcode、哪些不需要errorcode，需要参考exercise3中阅读文档的9.10（error code summary）中的表

```
    TRAPHANDLER_NOEC(DIVIDE_ENTRY,T_DIVIDE)     //0
    TRAPHANDLER_NOEC(DEBUG_ENTRY,T_DEBUG)       //1
    TRAPHANDLER_NOEC(NMI_ENTRY,T_NMI)       //2
    TRAPHANDLER_NOEC(BRKPT_ENTRY,T_BRKPT)       //3
    TRAPHANDLER_NOEC(OFLOW_ENTRY,T_OFLOW)       //4
    TRAPHANDLER_NOEC(BOUND_ENTRY,T_BOUND)       //5
    TRAPHANDLER_NOEC(ILLOP_ENTRY,T_ILLOP)       //6
    TRAPHANDLER_NOEC(DEVICE_ENTRY,T_DEVICE)     //7

    TRAPHANDLER(DBLFLT_ENTRY,T_DBLFLT)          //8

    TRAPHANDLER_NOEC(COPROC_ENTRY,T_COPROC) //9

    TRAPHANDLER(TSS_ENTRY,T_TSS)                //10
    TRAPHANDLER(SEGNP_ENTRY,T_SEGNP)            //11
    TRAPHANDLER(STACK_ENTRY,T_STACK)            //12
    TRAPHANDLER(GPFLT_ENTRY,T_GPFLT)            //13
    TRAPHANDLER(PGFLT_ENTRY,T_PGFLT)            //14
    TRAPHANDLER(RES_ENTRY,T_RES)        //15
```

```
    TRAPHANDLER_NOEC(FPERR_ENTRY,T_FPERR)       //16

    TRAPHANDLER_NOEC(ALIGN_ENTRY,T_ALIGN)       //17
    TRAPHANDLER_NOEC(MCHK_ENTRY,T_MCHK)         //18
    TRAPHANDLER_NOEC(SIMDERR_ENTRY,T_SIMDERR)   //19

    TRAPHANDLER_NOEC(SYSCALL_ENTRY,T_SYSCALL)       //48
```

除了在 trapentry.S 中定义各个 entry 之外，还需要在 kern/trap.c 中对每个 entry 进行 extern 函数声明， 这样 trap.c 才可以找到位于 trapentry.S 中的符号

```
extern void DIVIDE_ENTRY();
extern void DEBUG_ENTRY();
extern void NMI_ENTRY();
extern void BRKPT_ENTRY();
extern void OFLOW_ENTRY();
extern void BOUND_ENTRY();
extern void ILLOP_ENTRY();
extern void DEVICE_ENTRY();
extern void DBLFLT_ENTRY();
extern void COPROC_ENTRY();
extern void TSS_ENTRY();
extern void SEGNP_ENTRY();
extern void STACK_ENTRY();
extern void GPFLT_ENTRY();
extern void PGFLT_ENTRY();
extern void RES_ENTRY();
extern void FPERR_ENTRY();
extern void ALIGN_ENTRY();
extern void MCHK_ENTRY();
extern void SIMDERR_ENTRY();
extern void SYSCALL_ENTRY();
```

_alltrap

各个 entry 以及 _alltrap 的功能，就是内核栈上构造出一个 trapframe 结构

在 inc/trap.h 中定义的trapframe结构中：

```
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));
```

tf_error 以及其往下的字段都被处理器设置好了，tf_trapno由前面定义的entry设置好了，因此 _alltrap 需要的就是设置剩余部分

可以使用 pushal 设置 tf_regs，pushl 构造tf_es、tf_ds（它们是16位的，因此还需要加上一个16位的padding才能构成32位）

同时还要将当前的数据段选择子设置为 GD_KD（内核数据段，因为当前已经是在内核中执行），即将 GD_KD 放入 ds 和 es 寄存器中

最后调用 trap 函数

```
    .globl _alltraps;
    .align 2;
    _alltraps:
    pushl %ds
    pushl %es
    pushal
    pushl $GD_KD
    popl %ds
    pushl $GD_KD
    popl %es
    pushl %esp
    call trap
```

trap_init

trap_init 负责设置idt表

设置idt表只需要通过使用宏 SETGATE 即可，需要正确理解 SETGATE 的参数的含义

```
#define SETGATE(gate, istrap, sel, off, dpl)
```

- gate

  即 idt 的表项

- istrap

  是否为trap，trap = 1, interrupt = 0

- sel

  handler代码(即之前设置的entry)所在代码段的选择子， 即GD_KT (内核代码段选择子)

- off

  handler代码在段中的offset，即各个 entry 的地址

- dpl

  Descriptor Privilege Level，该项不能比产生中断的一方具有更高的权限，即，对于 int xx 这样的软件调用中断而言，应 dpl = user = 3，对于处理器产生的中断， dpl = 0

  要设置的中断idt表项中，breakpoint 和 syscall 均为软中断，应设置 dpl = 3，否则会出现 generation fault，直接跳往idt [13]

```
void
trap_init(void)
{
```

```
    extern struct Segdesc gdt[];
    // LAB 3: Your code here.
    SETGATE(idt[T_DIVIDE],0,GD_KT,DIVIDE_ENTRY,0)
    SETGATE(idt[T_DEBUG],0,GD_KT,DEBUG_ENTRY,0)
    SETGATE(idt[T_BRKPT],0,GD_KT,BRKPT_ENTRY,3)
    SETGATE(idt[T_OFLOW],0,GD_KT,OFLOW_ENTRY,0)
    SETGATE(idt[T_BOUND],0,GD_KT,BOUND_ENTRY,0)
    SETGATE(idt[T_ILLOP],0,GD_KT,ILLOP_ENTRY,0)
    SETGATE(idt[T_DEVICE],0,GD_KT,DEVICE_ENTRY,0)
    SETGATE(idt[T_DBLFLT],0,GD_KT,T_DBLFLT,0)
    SETGATE(idt[T_COPROC],0,GD_KT,COPROC_ENTRY,0)
    SETGATE(idt[T_TSS],0,GD_KT,TSS_ENTRY,0)
    SETGATE(idt[T_SEGNP],0,GD_KT,SEGNP_ENTRY,0)
    SETGATE(idt[T_STACK],0,GD_KT,STACK_ENTRY,0)
    SETGATE(idt[T_GPFLT],0,GD_KT,GPFLT_ENTRY,0)
    SETGATE(idt[T_PGFLT],0,GD_KT,PGFLT_ENTRY,0)
    SETGATE(idt[T_NMI],0,GD_KT,NMI_ENTRY,0)
    SETGATE(idt[T_FPERR],0,GD_KT,FPERR_ENTRY,0)
    SETGATE(idt[T_SYSCALL],1,GD_KT,SYSCALL_ENTRY,3)
    //panic("trap_init not yet implemented");
    // Per-CPU setup
    trap_init_percpu();
}
```

exercise5

trap 的处理流程为：

产生trap/interrupt -----> 通过idt表跳转到entry -------> _alltrap -------> trap --------> trap_dispatch ------> 对应handler -------> -----> trap ------> 返回用户程序

exercise5 要求在trap_dispatch中将 page_fault 交给 page_fault_handler 函数，因此只需要加上一个 if 判断traono，并进行跳转即可

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

exercise6

同exercise5，添加一个if判断是否为breakpoint

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

exercise7

首先在 trap_dispatch 中将 trapno == T_SYSCALL 的 trap 交给syscall()函数处理

```
if(tf->tf_trapno == T_SYSCALL){
        tf->tf_regs.reg_eax = 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);
        return;
    }
```

首先，产生syscall是因为用户程序显示调用，此前用户会将系统调用号以及参数分别放在 eax edx ecx ebx edi esi 寄存器中，由于中断，这些寄存器的值会被保存在 tf ，因此就将 tf 中对应的各寄存器的字段传递给 syscall ，且 syscall 的返回值应保存在 tf 中的eax字段处，以便返回用户程序时，用户程序能从eax中得到返回值。

还要写好 syscall 函数，由于每个syscall的具体调用都已经定义好了，所以syscall中只要根据syscallno调用对应的函数即可

```
int32_t
syscall(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t a3, uint32_t a4, uint32_t a5)
{
    // Call the function corresponding to the 'syscallno' parameter.
    // Return any appropriate return value.
    // LAB 3: Your code here.
    //panic("syscall not implemented");

    switch (syscallno) {
    case SYS_cputs:
        sys_cputs((char*)a1,(size_t)a2);
        return 0;
    case SYS_cgetc:
        return sys_cgetc();
    case SYS_getenvid:
        return sys_getenvid();
    case SYS_env_destroy:
        return sys_env_destroy(a1);
    case SYS_show_environments:
        sys_show_environments();
        return 0;
    default:
        return -E_INVAL;
    }
}
```

exercise8

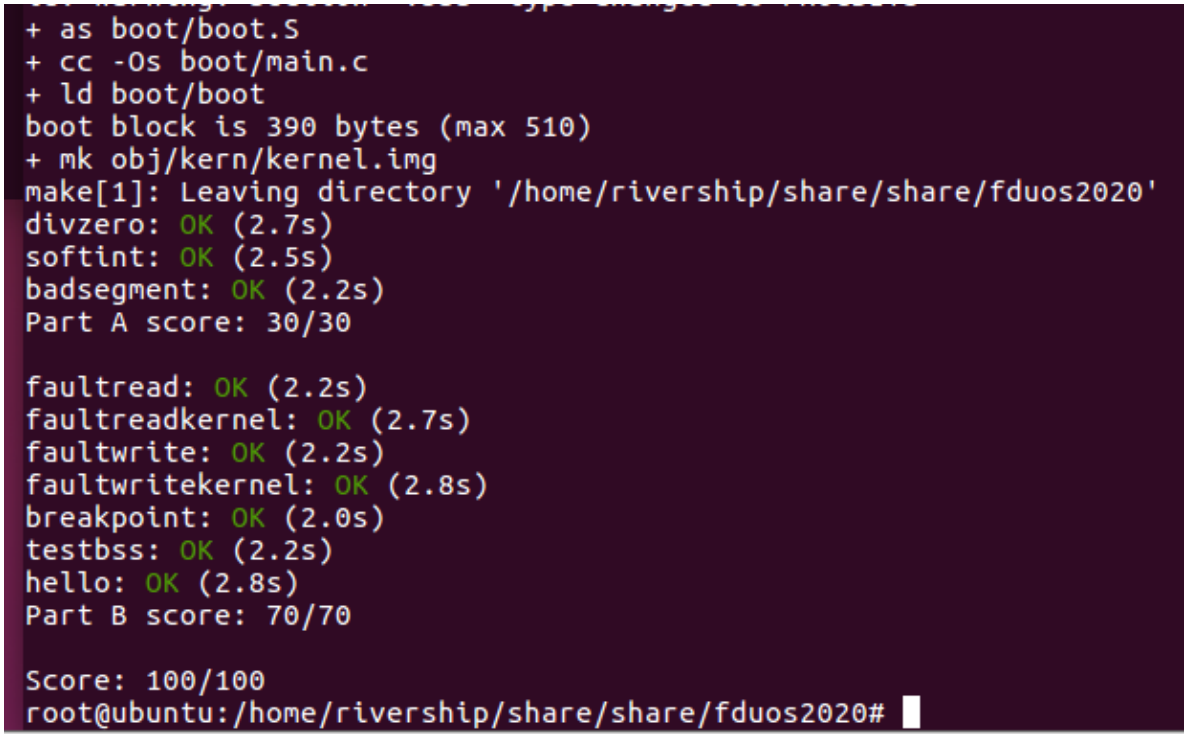
要求在kern/libmain.c 的 libmain 函数中设置好 thisenv 指针

由于宏ENVX可以利用envid计算出相应env的下标，所以很容易可以得到thisenv，如下

```
    thisenv = 0;
    envid_t envid = sys_getenvid();
    thisenv = &envs[ENVX(envid)];
```

make grade

截图如下：



sys_show_environments

在 inc/syscall.h 中添加SYS_show_environments

```
enum {
    SYS_cputs = 0,
    SYS_cgetc,
    SYS_getenvid,
    SYS_env_destroy,
    SYS_show_environments,          //add this
    NSYSCALLS
};
```

在 lib/syscall.c 中添加 sys_show_environments 的定义（即用户调用的 sys_show_environments）

```
void sys_show_environments(void){
    syscall(SYS_show_environments, 0, 0, 0, 0, 0, 0);
}
```

在 kern/syscall.c 中添加 sys_show_environments 的定义 （内核调用的 sys_show_environments）

```
static
void sys_show_environments(){
    int i;
    for(i=0;i<NENV;i++){
        if(envs[i].env_status!=ENV_FREE){
            print_trapframe(&envs[i].env_tf);
            cprintf("env_link: 0x%08x\n", envs[i].env_link);
            cprintf("env_id  : 0x%08x\n", envs[i].env_id);
            cprintf("envir_parent_id:  0x%08x\n", envs[i].env_parent_id);
            cprintf("env_runs: %d\n", envs[i].env_runs);
        }
    }
}
```

这里需要声明为 static ，只能在 kern/syscall.c 中调用，且不会与lib/syscall.c 中的定义重复

再在 syscall 函数中添加：

```
case SYS_show_environments:
    sys_show_environments();
    return 0;
```

make run-hello后，结果如下

```
root@ubuntu:/home/rivership/share/share/fduos2020# make run-hello
make[1]: Entering directory '/home/rivership/share/share/fduos2020'
make[1]: Leaving directory '/home/rivership/share/share/fduos2020'
qemu-system-i386 -drive file=obj/kern/kernel.img,index=0,media=disk,format=raw -
serial mon:stdio -gdb tcp::26000 -D qemu.log
6828 decimal is 15254 octal!
Physical memory: 131072K available, base = 640K, extended = 130432K
check_page_free_list() succeeded!
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_free_list() succeeded!
check_page_installed_pgdir() succeeded!
[00000000] new env 00001000
Incoming TRAP frame at 0xeffffffc
Incoming TRAP frame at 0xeffffffc
hello, world
Incoming TRAP frame at 0xeffffffc
i am environment 00001000
Incoming TRAP frame at 0xeffffffc
TRAP frame at 0xf01bf000
  edi  0x00000000
  esi  0x00000000
  ebp  0xeebfdfb0
  oesp 0xefffffdc
  ebx  0x00000000
  edx  0x00000000
  ecx  0x00000000
  eax  0x00000004
  es   0x----0023
  ds   0x----0023
  trap 0x00000030 System call
  err  0x00000000
  eip  0x00800bc4
  cs   0x----001b
  flag 0x00000092
  esp  0xeebfdfa4
  ss   0x----0023
next environment is at 0xf01bf060
current environment id is 0x00001000
environment parent id is 0x00000000
Number of times environment has run is 4
Incoming TRAP frame at 0xeffffffc
[00001000] exiting gracefully
[00001000] free env 00001000
Destroyed the only environment - nothing more to do!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

可以看到在hello之后又打印除了env结构中的信息，包括 env_tf env_link env_runs env_id

问题回答

(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?

内核态页表基地址位于变量 kern_pgdir 中，在kern/pmap.c定义

```
pde_t *kern_pgdir;      // Kernel's initial page directory
```

页表基地址存在 **cr3** 寄存器中，JOS 通过 lcr3 函数将页表基地址 load 进 cr3 寄存器，从而实现页表基地址空间的切换

(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?

iret 指令为 interrupt return，即从中断中返回，用于从内核态返回用户态，和ret指令类似，iret会将用户代码的继续运行地址pop回eip，除此之外，iret还会将用户的代码段选择子(GD_UT) pop 回 cs 寄存器，并 pop 回 EFLAGS 寄存器的内容。

kernel stack用宏 KSTACKTOP 表示，定义在 inc/memlayout.h 中。

esp eip 由 processer 保存（在通过idt表跳转前就已经push到了内核栈上），eax ebx 由 jos kernel 保存（通过_alltrap保存）

(3) IDT 和 GDT 存储的信息分别是什么?

IDT 存储的是 interrupt descriptor 。即每个 trap/interrupt 对应的 handler 的起始地址的后16位(定义在 trapentry.S 中)，同时 IDT 表项还保存了这些 handler 所在代码段的段选择子，还设置了一些标志位，表明该 trap/interrupt 的 privilege level 以及是 trap 还是 interrupt 等信息。

GDT 存储的是 Global descriptor，保存了各个段的段描述符

```
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
};
```