[TOC]

练习1

mem_init

与lab2初始化pages数组和映射类似

练习2

env_init

初始化全部 envs 数组中的 Env 结构体，并将它们加入到 env_free_list 中。还要调用 env_init_percpu ，这个函数会通过配置段硬件，将其分隔为特权等级 0 (内核) 和特权等级 3（用户）两个不同的段。 通过注释可以得知，第一次调用env_alloc()应该返回envs[0]，所以链表应该倒序存储。

envs

env.h中定义

```
extern struct Env *envs;		// All environments
```

env_setup_vm

> identical 相同的

为新的进程分配一个页目录，并初始化新进程的地址空间对应的内核部分。 所有envs结构的虚拟地址都是在UTOP上的。 UTOP之上env_pgdir可以拷贝kern_pgdir, 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，但是UVPT处的env_pgdir需要单独设置成用户只读。

> 每个用户进程都需要共享内核空间，所以对于用户进程而言，在UTOP以上的部分，和系统内核的空间是完全一样的。

memcpy

string.h中定义

```
void *	memcpy(void *dst, const void *src, size_t len);
```

memcpy函数的功能是从源src所指的内存地址的起始位置开始拷贝n个字节到目标dest所指的内存地址的起始位置中。

region_alloc

> corner-cases 极端情况

为进程分配和映射物理内存。 利用lab2中的 page_alloc() 完成内存页的分配， page_insert() 完成虚拟地址到物理页的映射。 注意把虚拟地址va，va+len以4k为单位取整，方便后面以页为单位计算。 还需要检查页表是否申请成功，是否映射到对应的物理地址。

load_icode

为每一个用户进程设置它的初始代码区，堆栈以及处理器标识位。因为用户程序是ELF文件，所以要解析ELF文件。

> 函数只在内核初始化且第一个用户进程未运行时被调用，从ELF文件头部指明的虚拟地址开始加载需要加载的字段到用户内存

该段代码需要参考boot/main.c文件来写。 注释提示通过参考env_run和env_pop_tf，修改程序的入口，来确保进程正确开始执行。根据env_run的注释，需要回来修改代码给e->env_tf附上正确的值。

Proghdr

通过查询得知，p_type,p_va,p_memsz是结构Proghdr中的参数。 elf.h中定义

```
struct Proghdr {
        uint32_t p_type;
        uint32_t p_offset;
        uint32_t p_va;
        uint32_t p_pa;
        uint32_t p_filesz;
        uint32_t p_memsz;
        uint32_t p_flags;
        uint32_t p_align;
};
```

通过上网查询可知Proghdr中各个数据类型的意义 uint32_t p_type; // 段类型，说明载入的是代码还是数据，用于动态链接 uint32_t p_offset; // 段相对文件头的偏移值 uint32_t p_va; // 段的第一个字节将被放到内存中的虚拟地址 uint32_t p_pa; // 物理地址，没有使用 uint32_t p_filesz; // 文件中段的长度 uint memsz; // 段在内存中占用的空间(每个程序的大小) uint32_t p_flags; // 读/写/执行位 uint32_t p_align; // 需求队列，总是物理页大小

Elf

elf.h中定义

```
struct Elf {
        uint32_t e_magic;       // must equal ELF_MAGIC
        uint8_t e_elf[12];
        uint16_t e_type;
        uint16_t e_machine;
        uint32_t e_version;
        uint32_t e_entry;
        uint32_t e_phoff;
```

```
        uint32_t e_shoff;
        uint32_t e_flags;
        uint16_t e_ehsize;
        uint16_t e_phentsize;
        uint16_t e_phnum;
        uint16_t e_shentsize;
        uint16_t e_shnum;
        uint16_t e_shstrndx;
};
```

Elf结构具体每个类型的意义可参考下图：

```
/* The ELF file header.  This appears at the start of every ELF file.  */

#define EI_NIDENT (16)

typedef struct
{
  unsigned char e_ident[EI_NIDENT];     /* Magic number and other info */
  Elf32_Half    e_type;                 /* Object file type */
  Elf32_Half    e_machine;              /* Architecture */
  Elf32_Word    e_version;              /* Object file version */
  Elf32_Addr    e_entry;                /* Entry point virtual address */
  Elf32_Off     e_phoff;                /* Program header table file offset */
  Elf32_Off     e_shoff;                /* Section header table file offset */
  Elf32_Word    e_flags;                /* Processor-specific flags */
  Elf32_Half    e_ehsize;               /* ELF header size in bytes */
  Elf32_Half    e_phentsize;            /* Program header table entry size */
  Elf32_Half    e_phnum;                /* Program header table entry count */
  Elf32_Half    e_shentsize;            /* Section header table entry size */
  Elf32_Half    e_shnum;                /* Section header table entry count */
  Elf32_Half    e_shstrndx;             /* Section header string table index */
} Elf32_Ehdr;
```

以及博客： https://www.cnblogs.com/dengxiaojun/p/4279407.html

具体ELF文件格式可参考下面这篇博客： https://blog.csdn.net/fang92/article/details/48092165

Trapframe

trap.h中定义

```
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));
```

env_create

调用env_alloc，从env_free_list中取出一个env结构体，再通过 env_setup_vm为其初始化，申请新的页目录;然后执行load_icode,这个函数加载elf文件(二进制文件)，它会调用region_alloc为其分配页，并将虚拟地址和物理地址作出映射，load_icon之后分配进程栈，以及，将env->env_tf.tf_eip指向将执行进程函数的入口(等待env_pop_tf的调用)

env_run

启动进程，curenv结构体指向当前运行的进程env,改变curenv结构体中运行状态等信息,通过env_pop_tf函数，将env结构体中保存的寄存器中的信息加在到真正的寄存器中。

curenv

env.h中定义

```
extern struct Env *curenv;              // Current environment
```

env_pop_tf

使用'iret'指令复原Trapframe中的寄存器值，退出内核，开始运行一些进程的代码。 ENV结构中提到Trapframe结构的寄存器 `struct Trapframe env_tf; // 保存的寄存器` env.h中定义

```
void    env_pop_tf(struct Trapframe *tf) __attribute__((noreturn));
```

env.c中实现

```
// 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 */
}
```

lcr3

x86.h中定义

```
static __inline void
lcr3(uint32_t val)
{
        __asm __volatile("movl %0,%%cr3" : : "r" (val));
}
```

汇编代码，将地址装入cr3寄存器，而cr3中装的都是页目录的起始地址。

debug

memmove error

运行gdb的时候发现报了memmove错误 然后按照错误提示去lib/string.c中查找memmove，发现自己原本写的memcpy在这里的string.c中竟然直接调用memmove,就顺手查了关于这两个函数的区别

```
void *
memmove(void *dst, const void *src, size_t n)
{
        const char *s;
        char *d;

        s = src;
        d = dst;
        if (s < d && s + n > d) {
                s += n;
                d += n;
                while (n-- > 0)
                        *--d = *--s;
        } else
                while (n-- > 0)
                        *d++ = *s++;

        return dst;
}
```

```
void *
memcpy(void *dst, const void *src, size_t n)
{
        return memmove(dst, src, n);
}
```

具体这两个函数的区分可以参考下面的博客，解析的十分详细
https://blog.csdn.net/li_ning_/article/details/51418400

发现报错的memmove是发生在调用汇编代码的memmove时

```
void *
memmove(void *dst, const void *src, size_t n)
{
        const char *s;
        char *d;

        s = src;
        d = dst;
        if (s < d && s + n > d) {
                s += n;
                d += n;
                if ((int)s%4 == 0 && (int)d%4 == 0 && n%4 == 0)
                        asm volatile("std; rep movsl\n"
                                :: "D" (d-4), "S" (s-4), "c" (n/4) : "cc",
"memory");
                else
                        asm volatile("std; rep movsb\n"
                                :: "D" (d-1), "S" (s-1), "c" (n) : "cc",
"memory");
                // Some versions of GCC rely on DF being clear
                asm volatile("cld" ::: "cc");
        } else {
                if ((int)s%4 == 0 && (int)d%4 == 0 && n%4 == 0)
                        asm volatile("cld; rep movsl\n"
                                :: "D" (d), "S" (s), "c" (n/4) : "cc",
"memory");
                else
                        asm volatile("cld; rep movsb\n"
                                :: "D" (d), "S" (s), "c" (n) : "cc",
"memory");
        }
        return dst;
}
```

后面发现是前面region_alloc中对页表的分配出错了，导致后面内存异常。

PADDR called with invalid kva 00000000

改完上面的错误，然后出现了如下图的错误 找到半天panic出处最后发现是PADDR出问题了，看PADDR源码可知是因为转换的地址小于KERNBASE，也就是说不是在内核所映射的空间中。

```
#define PADDR(kva) _paddr(__FILE__, __LINE__, kva)

static inline physaddr_t
_paddr(const char *file, int line, void *kva)
{
        if ((uint32_t)kva < KERNBASE)
                _panic(file, line, "PADDR called with invalid kva %08lx",
kva);
        return (physaddr_t)kva - KERNBASE;
}
```

= =最后发现是env_init出错了。。

some

start (kern/entry.S) 由实模式进入保护模式 i386_init (kern/init.c) 初始化内核 cons_init(); 初始化显示屏 mem_init(); 内存管理初始化 env_init(); 进程初始化 trap_init(); 中断初始化

gbd调试

通过观察指令地址发现指令iret后进入了实模式 在obj/user/hello.asm中查找int $0x30的地址 在int $0x30的地址0x800a1c处设置断点，成功运行到该代码处说明前面写的代码没有出错

练习4

trapentry.S中汇编代码

.global/.globl:用来定义一个全局的符号 使用.global/.globl将函数声明为全局函数以后就可以被其他文件调用。 .type:用来指定一个符号的类型是函数类型或者是对象类型,对象类型一般是数据 .align:用来指定内存对齐方式

在CPU返回error code的时候调用TRAPHANDLER，没有则调用TRAPHANDLER_NOEC

first

关于CPU何时返回error code，可参考Intel手册
https://pdos.csail.mit.edu/6.828/2014/readings/i386/s09_10.htm

> 手册中没有说明是否返回error code的中断暂时没加上

second

根据提示可知就是将Trapframe结构中的所有数据压入栈中，文档告诉我们 tf_ss,tf_esp,tf_eflags,tf_cs,tf_eip,tf_err在中断发生时由CPU压栈，所以只需要将剩下的寄存器tf_es,tf_ds压入

栈中即可。因为需要将栈模仿成Trapframe结构，而且CPU已经将其余寄存器的值倒序压入，所以需要注意先压入ds，后压入es。

pushal

pushal指令会按顺序将eax到edi压入栈中,具体参考下面网站
http://www.fermimn.gov.it/linux/quarta/x86/pusha.htm

GD_KD

memlayout.h中定义 `#define GD_KD 0x10 // kernel data` 这里需要注意不能直接用立即数给段寄存器赋值，至于为什么还是不太懂，可以参考下面的讨论 https://bbs.csdn.net/topics/340215235

所以应该先给通用寄存器赋值，或者入栈出栈来实现赋值。

trap_init

根据trapentery.S中的提示，声明函数