Lab 3: 用户环境

Lab 3 练习思路

本次将会实现保护用户模式环境（即进程）所必须的基本内核功能。主要改进如下：

- 改进JOS内核，设置一些数据结构来：追踪用户环境、创建单用户环境、加载程序映像、运行内核
- 使JOS内核能够处理所有用户环境提出的系统调用以及其他一些用户环境导致的异常

我个人认为主要就是两个改进方面，增加数据结构和处理异常。

实验说明中提到，本次实验中，“环境”和“进程”是可以互换的，都是指用户运行程序。之所以介绍环境的概念而不是传统的进程概念，是为了强调JOS环境和UNIX进程提供的是不同的接口、不同的语义。不过因为在学校操作系统课程上多强调进程，个人对进程相对更熟悉一些，所以很多地方还是从进程的角度去理解的。

第一部分 用户环境和异常处理

新文件 `inc/env.h` 包含了JOS用户环境的基本定义。内核使用 `Env` 数据结构来跟踪每个用户环境。

以 `Env` 数据结构为基础， `kern/env.c` 维护了三个相关的全局变量：

```
struct Env *envs = NULL; // Env数组
struct Env *curenv = NULL; // 当前正在运行的Env
static struct Env *env_free_list; //空闲Env链表
```

JOS内核最多支持 `NENV` （ `inc/env.h` 中定义）个同时处于活动状态的环境。数组 `envs` 将为每一个活跃环境维护一个 `Env` 结构。

所有不活跃的 `Env` 保存在 `env_free_list` 上。这种设计方便于分配和释放环境。

`curenv` 指向正在执行的 `Env` ，最初设置为 `NULL` 。

1. 环境状态

结构 `Env` 在 `inc/env.h` 中定义如下：

```
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
};
```

详解：

- `env_tf`：`inc/trap.h`中定义，存放环境暂停运行时寄存器的值。从用户模式切换到内核模式时，内核将保存这些内容，以便后期恢复。
- `env_link`：指向`env_free_list`中下一个空闲`Env`。学习笔记中说：“前提是这个结构体还没有被 分配给任意一个用户环境时，该域才有用”。
- `env_id`：唯一标识当前使用这个`Env`的环境。在某一用户环境终止之后，内核可能会将相同的`Env`重新分配给另一环境，但新环境具有不同的`env_id`。
- `env_parent_id`：创建此环境的父环境的`env_id`。
- `env_type`：一般是`ENV_TYPE_USER`。
- `env_status`：可能是以下值：
    - `ENV_FREE`：不活跃，在`env_free_list`中
    - `ENV_RUNNABLE`：就绪，等待分配处理机
    - `ENV_RUNNING`：正在运行
    - `ENV_NOT_RUNNABLE`：活跃但不可运行，因为正在等待其他环境传递消息给它。
    - `ENV_DYING`：终止。下一次陷入内核时，将被释放。这里不大清楚什么叫“陷入内核”，我个人认为就是到了内核模式的时候。
- `env_pgdir`：保存环境的页目录的虚拟地址。

像Unix进程一样，JOS环境将“线程”和“地址空间”的概念相结合。线程主要由保存的寄存器定义，地址空间由`env_pgdir`指向的页目录和页表定义。要运行一个环境，内核必须设置合适的寄存器和适当的地址空间。

2. 分配环境数组

> **练习一：**
>
> 在`kern/pmap.c`中修改`mem_init()`分配`envs`数组。 该数组恰好由`NENV`个`Env`结构实例组成，非常类似于分配页数组。也像`pages`数组一样，`envs`的内存也应该映射到UENVS（在`inc/memlayout.h`中定义），因此用户进程可以从这个数组中读取。
>
> 我的思路

3. 创建、运行环境

现在编写 `kern/env.c` 来运行一个用户环境。目前没有文件系统，所以必须让内核能够加载静态二进制程序映像文件。

Lab 3里的 `GNUmakefile` 文件在 `obj/user/` 目录下生成了一系列二进制映像文件。阅读 `kern/Makefrag` 文件，发现一些地方把二进制文件直接链接到内核可执行文件中，只要这些文件是 `.o` 文件。其中在链接器命令行中的 `-b binary` 选项会使这些文件被当做二进制执行文件链接到内核之后。

`i386_init()` 函数中可以看到运行上述二进制文件的代码，但是我们需要完成能够设置这些代码的运行用户环境的功能。

> **练习二：**
>
> 在 `env.c` 中完成下列函数： `env_init()` `env_setup_ym()` `region_alloc()` `load_icode()` `env_create()` `env_run()`
>
> 我的思路

4. 中断和异常

到目前为止，程序运行到 `int $0x30` 系统调用就会终止，原因是处理器进入用户模式之后无法跳出。所以现在需要实现一个基本的处理异常和系统调用的机制，让内核可以从用户模式中恢复对处理器的控制。首先要对异常和中断有一个基本的了解。

> **练习三：**
>
> 阅读 80386 Programmer's Manual 的第九章异常和中断。
>
> 我的思路

5. 控制转移

异常和中断都可以使处理器从用户态转为内核态。中断指的是由外部异步事件引起的处理器控制权转移，比如外部IO设备发送来的中断信号；异常则是由于当前正在运行的指令所带来的同步的处理器控制权的转移，比如除零溢出异常。

为了确保这些控制的转移受到保护，处理器的中断异常机制通常被设计为：

- 用户态的代码无权选择内核中的代码从哪里开始执行
- 处理器可以确保只有在某些条件下，才能进入内核态

x86有两种机制提供保护：

1. 中断向量表：

   处理器保证中断和异常只能够引起内核进入到一些特定的，被事先定义好的程序入口点。x86允许多达256个不同的中断和异常，每一个都配备一个独一无二的中断向量。一个中断向量的值是根据中断源来决定的：不同设备，错误条件，以及对内核的请求都会产生出不同的中断和中断向量。CPU将使用这个向量作为这个中断在中断向量表中的索引。这个表是由内核设置的，放在内核空间中，通过这个表中的任意一个表项，处理器可以知道：

   - 需要加载到EIP寄存器中的值，指向中断处理程序的位置
   - 需要加载到CS寄存器中的值，包含中断处理程序的运行特权级（即该程序是在用户态还是内核态下运行）。

2. 任务状态段

   处理器还需要一个地方来存放处理器的状态，如EIP和CS寄存器的值。从而中断处理程序之后可以恢复到原程序中。这段内存自然也要保护起来，不能被用户态的程序所篡改。

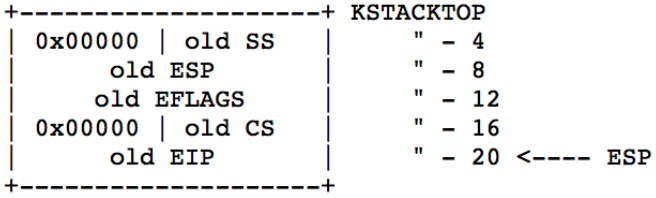
   因此，当一个x86处理器要处理中断异常时，也会把堆栈切换到内核空间中。任务状态段TSS这一数据结构将会详细记录这个堆栈所在的段的段描述符和地址。处理器会把SS、ESP、EFLAGS、CS、EIP以及错误码（可选）等值压入到这个堆栈上。然后加载中断处理程序的CS，EIP值，并且设置ESP，SS寄存器指向新的堆栈。

   TSS非常大，并且有很多其他的功能，但是JOS仅用它来定义处理器从用户态转向内核态所采用的内核堆栈。由于JOS中的内核态指的就是特权级0，所以处理器用TSS中的ESP0、SS0字段来指明这个内核堆栈的位置、大小。

6. 举例

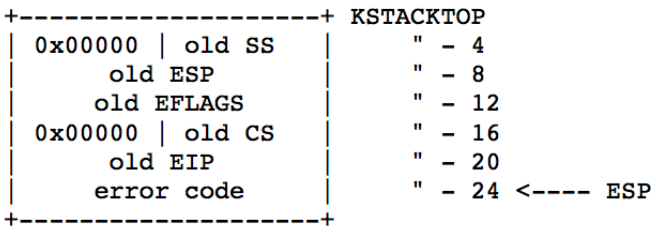
假设处理器正在用户态下运行代码，但是遇到了除数为零：

1. 处理器会首先切换自己的堆栈，切换到由TSS的SS0，ESP0字段所指定的内核堆栈区，这两个字段分别存放着 `GD_KD` 和 `KSTACKTOP` 的值
2. 处理器把异常参数压入到内核堆栈中，起始地址 `KSTACKTOP`



3. 除零异常的中断向量是0，处理器会读取IDT表中的0号表项，并且把CS: EIP的值设置为0号中断处理函数的地址值
4. 中断处理函数开始执行

某些异常除了前边中要保存的五个值之外，还要再压入一个值，错误码。比如页表出错，就是其中一个实例：

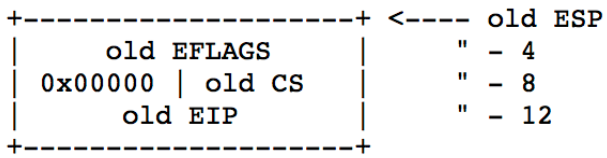


以上几步都是由硬件自动完成的。

7. 嵌套异常和中断

处理器在用户态下和内核态下都可以处理异常或中断。但只有当处理器从用户态切换到内核态时，才会自动切换堆栈，把一些寄存器中原来的值压入到堆栈上，并且触发相应的中断处理函数。如果处理器处在内核态时又出现异常或中断，此时CPU只会向内核堆栈压入更多的值。通过这种方式，内核就可处理嵌套中断。

如果处理器已经在内核态下并且遇到嵌套中断，因为它不需要切换堆栈，所以不需要存储SS、ESP寄存器的值。此时内核堆栈如下图：



8. 设置中断向量表

现在应该设置IDT表，在JOS下处理异常，目前只处理0～31号的内部异常。

`kern/trap.h` 文件中包含了仅内核可见的一些中断异常相关的定义， `inc/trap.h` 中包含了用户态也可见的一些定义。

每一个中断或异常都有自己的中断处理函数，定义在 `trapentry.s` 中。 `trap_init()` 初始化IDT表。每一个处理函数都应该在堆栈上构建一个 `Trapframe` ，并调用 `trap()` 函数指向这个 `Trapframe` ， `trap()` 然后处理异常中断。

根据学习笔记的提示，中断异常处理的步骤如下：

1. 初始化IDT
2. 捕捉中断，查表
3. 保存被中断程序上下文
4. 调用执行中断处理函数
5. 恢复原程序上下文

练习四：

编辑文件 `trapentry.s` 和 `trap.c` 实现上述功能。宏定义 `TRAPHANDLER` 和 `TRAPHANDLER_NOEC` 和 `inc/trap.h` 中的 `T_*` 定义会有帮助。

- 在 `trapentry.s` 文件中为 `inc/trap.h` 中的每一个trap添加一个入口
- 修改 `trap_init()` 函数来初始化 `idt` ，使表中每一项都指向定义在 `trapentry.s` 中入口指针， `SETGATE` 宏定义在这里用得上
- 提供 `_alttraps` 的值， `_alttraps` 应该：
    - 把值压入堆栈使堆栈看起来像一个 `Trapframe`
    - 加载 `GD_KD` 的值到 `%ds` `%es` 寄存器中
    - 把 `%esp` 的值压入，并且传递一个指向 `Trapframe` 的指针到 `trap()` 函数中
    - 调用 `trap`

考虑使用 `pushal` 指令，它会很好得和结构体 `Trapframe` 的布局配合。

我的思路

第二部分 缺页中断、断点异常、系统调用

1. 处理缺页中断

缺页中断是一个非常重要的中断，因为后续的实验非常依赖于能够处理缺页中断的能力。当缺页中断发生时，系统会把引起中断的线性地址存放到控制寄存器CR2中。在 `trap.c` 中，已经提供了一个能够处理这种缺页异常的函数 `page_fault_handler()` 。

练习五：

修改 `trap_dispatch()` 函数，使系统能够把缺页异常引导到 `page_fault_handler()` 上执行。在修改完成后，执行 `make grade` ，出现的结果应该是你修改后的JOS可以成功运行 `faultread` `faultreadkernel` `faultwrite` `faultwritekernel` 测试程序。

我的思路

4. 断点异常

断点异常通常用于允许调试器通过int3软中断指令临时替换相关程序指令来在程序代码中插入断点。JOS将通过将其转换为原始的所有用户可用的伪系统调用。

练习六：

修改 `trap_dispatch()` 使断点异常发生时，能够触发kernel monitor。修改完成后运行 `make grade` ，修改后应当能够正确运行 `breakpoint` 测试程序。

我的思路

5. 系统调用

用户程序会要求内核帮助它完成系统调用。当用户程序触发系统调用，系统进入内核态。处理器和操作系统将保存该用户程序当前的上下文状态，然后由内核将执行正确的代码完成系统调用，然后回到用户程序继续执行。而用户程序到底是如何得到操作系统的注意，以及它如何说明它希望操作系统做什么事情的方法是有很多不同的实现方式的。

在JOS中，我们会采用 `int` 指令，这个指令会触发一个处理器的中断。特别的，我们用 `int $0x30` 来代表系统调用中断。注意，中断 `0x30` 不是通过硬件产生的。

应用程序会把系统调用号以及系统调用的参数放到寄存器中。通过这种方法，内核就不需要去查询用户程序的堆栈了。系统调用号存放到 `%eax` 中，参数则存放在 `%edx` `%ecx` `%ebx` `%edi` 和 `%esi` 中。内核会把返回值送到 `%eax` 中。在 `lib/syscall.c` 中已经写好了触发一个系统调用的代码。

练习七：

给中断向量 `T_SYSCALL` 编写一个中断处理函数。编辑 `kern/trapentry.s` 和 `kern/trap.c` 中的 `trap_init()` 函数。你也需要修改 `trap_dispatch()` 函数，使其能够通过调用在 `kern/syscall.c` 中定义的 `syscall()` 函数去处理系统调用中断。最终你需要去实现 `kern/syscall.c` 中的 `syscall()` 函数。确保这个函数会在系统调用号为非法值时返回 `-E_INVAL` 。你应该充分理解 `lib/syscall.c` 文件。我们要处理在 `inc/syscall.h` 文件中定义的所有系统调用。

通过 `make run-hello` 指令来运行 `user/hello` 程序，它应该在控制台上输出 `hello, world` 然后发出一个页中断。

我的思路

6. 用户态启动

用户程序真正开始运行的地方是在 `lib/entry.s` 文件中。该文件中，首先会进行一些设置，然后就会调用 `lib/libmain.c` 文件中的 `libmain()` 函数。首先修改 `libmain()` 函数，初始化全局指针 `thisenv` ，让它指向当前用户环境的 `Env` 。

然后 `libmain()` 函数就会调用 `umain` ，恰好是 `user/hello.c` 中被调用的函数。

练习八：

补全上述代码，然后重启内核，此时你应该看到 `hello, world` 和 `i am environment 00001000` 然后 `user/hello.c` 就会尝试通过调用 `sys_env_destroy()` 退出。由于内核目前仅仅支持一个用户运行环境，所以应该会提示 “已经销毁用户环境”，然后退回 kernel monitor。

我的思路

7. 页中断和内存保护

内存保护是操作系统的非常重要的一项功能，它可以防止由于用户程序崩溃对操作系统带来的破坏与影响。

操作系统通常依赖于硬件的支持来实现内存保护。操作系统可以让硬件能够始终知晓哪些虚拟地址是有效的，哪些是无效的。当程序尝试去访问一个无效地址，或者尝试去访问一个超出它访问权限的地址时，处理器会在这个指令处终止，并且触发异常，陷入内核态，与此同时把错误的信息报告给内核。如果这个异常是可以被修复的，那么内核会修复这个异常，然后程序继续运行。如果异常无法被修复，则程序永远不会继续运行。

作为一个可修复异常的例子，让我们考虑一下可自动扩展的堆栈。在许多系统中，内核在初始情况下只会分配一个内核堆栈页，如果程序想要访问这个内核堆栈页之外的堆栈空间的话，就会触发异常，此时内核会自动再分配一些页给这个程序，程序就可以继续运行了。

系统调用也为内存保护带来了问题。大部分系统调用接口让用户程序传递一个指针参数给内核。这些指针指向的是用户缓冲区。通过这种方式，系统调用在执行时就可以解引用这些指针。但是这里有两个问题：

1. 在内核中的缺页要比在用户程序中的缺页更严重。如果内核在操作自己的数据结构时出现 缺页错误，这是一个内核的错误，而且异常处理程序会中断整个内核。但是当内核在解引用由用户程序传递来的指针时，它需要一种方法去记录此时出现的任何page faults都是由用户程序带来的。
2. 内核通常比用户程序有着更高的内存访问权限。用户程序很有可能要传递一个指针给系统调用，这个指针指向的内存区域是内核可以进行读写的，但是用户程序不能。此时内核必须小心不要去解析这个指针，否则的话内核的重要信息很有可能被泄露。

现在你需要通过仔细检查所有由用户传递来指针所指向的空间来解决上述两个问题。当一个程序传递给内核一个指针时，内核会检查这个地址是在整个地址空间的用户地址空间部分，而且页表也运行进行内存的操作。

练习九：

修改 `kern/trap.c` 文件，使其能够在内核模式下发现页错时发出警告。

提示：为了能够判断这个错误是出现在内核模式下还是用户模式下，我们应该检查 `tf_cs` 的低几位。

阅读 `kern/pmap.c` 的 `user_mem_assert`，实现 `user_mem_check`。

修改 `kern/syscall.c`，检查输入参数。

启动内核后，运行 `user/buggyhello` 程序，用户环境会被销毁，但内核不会发出警告，应该看到：

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

最后，更改 `kern/kdebug.c` 中的 `debuginfo_eip`，在 `usd`， `stabs` 和 `stabstr` 上调用 `user_mem_check`。现在运行 `user/breakpoint`，则应该能够从内核监视器运行 backtrace，并在内核遇到页面错误之前看到 `backtrace` 遍历到 `lib/libmain.c`。是什么原因导致页面错误？暂不需要修复它，但应该明白为什么会这样。

我的思路

注意到上一练习中实现的代码同样适用于一些古怪的程序，比如 `user/evilhello.c`（虽然我也不知道到底哪里不怀好意了）。

练习十：

启动内核，运行 `user/evilhello.c`。用户环境将会被销毁，但是内核不会发出警告，应该会看到：

```
[00000000] new env 00001000
...
[00001000] user_mem_check assertion failure for va f010000c
[00001000] free env 00001000
```

我的思路

练习思路

> **练习一：**

分配 `env` 数组的方法和分配 `pages` 的方法几乎一模一样，以下为两部分代码比对：

```
152     //////////////////////////////////////////////////////////////////////
153     // Allocate an array of npages 'struct PageInfo's and store it in 'pages'.
154     // The kernel uses this array to keep track of physical pages: for
155     // each physical page, there is a corresponding struct PageInfo in this
156     // array.  'npages' is the number of physical pages in memory.  Use memset
157     // to initialize all fields of each struct PageInfo to 0.
158     // Your code goes here:
159     pages = (struct PageInfo *) boot_alloc(npages * sizeof(struct PageInfo *));
160     memset(pages, 0, npages * sizeof(struct PageInfo *));

162     //////////////////////////////////////////////////////////////////////
163     // Make 'envs' point to an array of size 'NENV' of 'struct Env'.
164     // LAB 3: Your code here.
165     envs = (struct Env *) boot_alloc(NENV * sizeof(struct Env));
166     memset(envs, 0, NENV * sizeof(struct Env));
```

然后根据文件中注释的提示和函数 `boot_map_region()` 功能，设置用户只读，并将其映射至 `UENV` 处，代码如下：

```
192     //////////////////////////////////////////////////////////////////////
193     // Map the 'envs' array read-only by the user at linear address UENVS
194     // (ie. perm = PTE_U | PTE_P).
195     // Permissions:
196     //    - the new image at UENVS  -- kernel R, user R
197     //    - envs itself -- kernel RW, user NONE
198     // LAB 3: Your code here.
199     boot_map_region(kern_pgdir, UENVS, PTSIZE, PADDR(envs), PTE_U | PTE_P);
```

该练习相对简单，按照注释说明完成即可。

练习二：

`env_init()` 是初始化 `envs` 数组的函数，一个比较简单的遍历，代码如下：

```
121 void
122 env_init(void)
123 {
124     // Set up envs array
125     // LAB 3: Your code here.
126     for (int i = NENV - 1; i >= 0; i--) {
127         envs[i].env_status = ENV_FREE;
128         envs[i].env_id = 0;
129
130         if (i == NENV - 1)
131             envs[i].env_link = NULL;
132         else
133             envs[i].env_link = &envs[i + 1];
134     }
135     env_free_list = &envs[0];
136
137     // Per-CPU part of the initialization
138     env_init_percpu();
139 }
```

练习三：

阅读 80386 Programmer's Manual 的第九章异常和中断。

练习四：

这一部分刚开始说实话没什么头绪，并不知道是要干点什么，以及题目老说什么函数有用、什么宏定义有用但又不告知这个定义在哪、什么意思、有什么功能。不过熬过一段时间以后发现还是自己太心急，对题意、源代码的理解还是不够透彻。

所以首先我一个字一个字重读了一遍题目和题目前边的介绍，然后花了大概几个小时翻译了一下练习三的网址给出的有关x86下中断异常的知识，一方面因为是英文的，另一方面网址上给的比较全面，所以我只是粗略地过了一遍，大概理解了一点皮毛。中途网上查阅到一片类似的中文博客 x86关于中断和异常的总结，仔细研读了一下加深了印象。

然后我没有直接下手，先对照着题目中各种晦涩的定义研究了一下 `kern/trap.h` `kern/trap.c` `kern/trapentry.s` `inc/trap.h` 和 `inc/mmu.h` （宏定义 `SETGATE` 在这里）里的现有代码，终于在 `kern/trapentry.s` 中找到了入手点，并且和学习笔记里的分析找到了对应，以下是完成过程：

首先仔细研究 `trapentry.s` 文件，其中有两个最重要的宏定义：

- `TRAPHANDLER`

```
/* TRAPHANDLER defines a globally-visible function for handling a trap.
 * It pushes a trap number onto the stack, then jumps to _alltraps.
 * Use TRAPHANDLER for traps where the CPU automatically pushes an error code.
 *
 * You shouldn't call a TRAPHANDLER function from C, but you may
 * need to _declare_ one in C (for instance, to get a function pointer
 * during IDT setup).  You can declare the function with
 *   void NAME();
 * where NAME is the argument passed to TRAPHANDLER.
 */
#define TRAPHANDLER(name, num)                                          \
        .globl name;            /* define global symbol for 'name' */   \
        .type name, @function;  /* symbol type is function */           \
        .align 2;               /* align function definition */         \
        name:                   /* function starts here */              \
        pushl $(num);                                                   \
        jmp _alltraps
```

- `TRAPHANDLER_NOEC`

```
/* Use TRAPHANDLER_NOEC for traps where the CPU doesn't push an error code.
 * It pushes a 0 in place of the error code, so the trap frame has the same
 * format in either case.
 */
#define TRAPHANDLER_NOEC(name, num)                                     \
        .globl name;                                                    \
        .type name, @function;                                          \
        .align 2;                                                       \
        name:                                                           \
        pushl $0;                                                       \
        pushl $(num);                                                   \
        jmp _alltraps
```

二者的差异在于该异常是否向堆栈中压入错误代码，通过查阅资料可知是否需要压入错误代码，然后对应选

取一个合适的即可，如下：

```
48 /*
49  * Lab 3: Your code here for generating entry points for the different traps.
50  */
51 TRAPHANDLER_NOEC(t_divide, T_DIVIDE);
52 TRAPHANDLER_NOEC(t_dibug, T_DEBUG);
53 TRAPHANDLER_NOEC(t_nmi, T_NMI);
54 TRAPHANDLER_NOEC(t_brkpt, T_BRKPT);
55 TRAPHANDLER_NOEC(t_oflow, T_OFLOW);
56 TRAPHANDLER_NOEC(t_bound, T_BOUND);
57 TRAPHANDLER_NOEC(t_illop, T_ILLOP);
58 TRAPHANDLER_NOEC(t_device, T_DEVICE);
59 TRAPHANDLER(t_dbflt, T_DBLFLT);
60 TRAPHANDLER(t_tss, T_TSS);
61 TRAPHANDLER(t_segnp, T_SEGNP);
62 TRAPHANDLER(t_stack, T_STACK);
63 TRAPHANDLER(t_gpflt, T_GPFLT);
64 TRAPHANDLER(t_pgflt, T_PGFLT);
65 TRAPHANDLER_NOEC(t_fperr, T_FPERR);
66 TRAPHANDLER(t_align, T_ALIGN);
67 TRAPHANDLER_NOEC(t_mchk, T_MCHK);
68 TRAPHANDLER_NOEC(t_simderr, T_SIMDERR);
69 TRAPHANDLER_NOEC(t_syscall, T_SYSCALL)
```

上述宏定义中 `name` 为入口函数的名称，对应 `trap.c` 中的函数定义，故在 `trap.c` 中添加代码：

```
14 void t_divide();
15 void t_debug();
16 void t_nmi();
17 void t_brkpt();
18 void t_oflow();
19 void t_bound();
20 void t_illop();
21 void t_device();
22 void t_dblflt();
23 void t_tss();
24 void t_segnp();
25 void t_stack();
26 void t_gpflt();
27 void t_pgflt();
28 void t_fperr();
29 void t_align();
30 void t_mchk();
31 void t_simderr();
32 void t_syscall();
```

最后在 `trap.c` 中修改函数 `trap_init()` ，利用 `SETGATE` 宏初始化 `idt` ：

```
 85 void
 86 trap_init(void)
 87 {
 88     extern struct Segdesc gdt[];
 89
 90     // LAB 3: Your code here.
 91     SETGATE(idt[T_DIVIDE],  0, GD_KT, divide,   0);
 92     SETGATE(idt[T_DEBUG],   0, GD_KT, debug,    0);
 93     SETGATE(idt[T_NMI],     0, GD_KT, nmi,      0);
 94     SETGATE(idt[T_BRKPT],   0, GD_KT, brkpt,    3);
 95     SETGATE(idt[T_OFLOW],   0, GD_KT, oflow,    0);
 96     SETGATE(idt[T_BOUND],   0, GD_KT, bound,    0);
 97     SETGATE(idt[T_ILLOP],   0, GD_KT, illop,    0);
 98     SETGATE(idt[T_DEVICE],  0, GD_KT, device,   0);
 99     SETGATE(idt[T_DBLFLT],  0, GD_KT, dblflt,   0);
100     SETGATE(idt[T_TSS],     0, GD_KT, tss,      0);
101     SETGATE(idt[T_SEGNP],   0, GD_KT, segnp,    0);
102     SETGATE(idt[T_STACK],   0, GD_KT, stack,    0);
103     SETGATE(idt[T_GPFLT],   0, GD_KT, gpflt,    0);
104     SETGATE(idt[T_PGFLT],   0, GD_KT, pgflt,    0);
105     SETGATE(idt[T_FPERR],   0, GD_KT, fperr,    0);
106     SETGATE(idt[T_ALIGN],   0, GD_KT, align,    0);
107     SETGATE(idt[T_MCHK],    0, GD_KT, mchk,     0);
108     SETGATE(idt[T_SIMDERR], 0, GD_KT, simderr,  0);
109     SETGATE(idt[T_SYSCALL], 0, GD_KT, syscall,  3);
110
111     // Per-CPU setup
112     trap_init_percpu();
113 }
```

`inc/mmu.h` 中 `SETGATE` 宏定义和功能如下：

```
268 // Set up a normal interrupt/trap gate descriptor.
269 // - istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate.
270     //   see section 9.6.1.3 of the i386 reference: "The difference between
271     //   an interrupt gate and a trap gate is in the effect on IF (the
272     //   interrupt-enable flag). An interrupt that vectors through an
273     //   interrupt gate resets IF, thereby preventing other interrupts from
274     //   interfering with the current interrupt handler. A subsequent IRET
275     //   instruction restores IF to the value in the EFLAGS image on the
276     //   stack. An interrupt through a trap gate does not change IF."
277 // - sel: Code segment selector for interrupt/trap handler
278 // - off: Offset in code segment for interrupt/trap handler
279 // - dpl: Descriptor Privilege Level -
280 //        the privilege level required for software to invoke
281 //        this interrupt/trap gate explicitly using an int instruction.
282 #define SETGATE(gate, istrap, sel, off, dpl)                    \
```

其中最后一个参数表示权限等级，不同的中断有不同的权限等级，但大多数都是0或3级（不知道和张老师课上讲的环0和环3是不是一个概念），可以通过查阅资料得到权限等级。

最后实现 `_alltrap`，根据题目的要求：

- 把值压入堆栈使堆栈看起来像一个 `Trapframe`
- 加载 `GD_KD` 的值到 `%ds` `%es` 寄存器中
- 把 `%esp` 的值压入，并且传递一个指向 `Trapframe` 的指针到 `trap()` 函数中
- 调用 `trap`

代码如下：

```
/*
 * Lab 3: Your code here for _alltraps
 */
.text
_alltraps:
                        // **LIRONGJIA**
    pushl %ds           // Push ds and es. Use pushal instruction
    pushl %es           // to fit into 'Trapframe' layout.
    pushal              //
                        //
    movl $GD_KD, %eax   // Load GD_KD into ds and es with the help
    movl %eax, %ds      // of eax.
    movl %eax, %es      //
                        //
    push %esp           // Push esp.
    call trap           // Call trap.
```

练习五：

这个题目比较简单，只要判断一下 `trap_dispatch()` 的参数 `tf` 是否为缺页中断所对应的 `Trapframe` 即可，由 `inc/trap.h` 中结构体 `Trapframe` 的定义很容易看出 `tf_trapno` 就表示中断类型。代码如下：

```
static void
trap_dispatch(struct Trapframe *tf)
{
    // 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;
            }
    }
}
```

最开始我用的是 `if` 判断，因为后边有很多类似的判断，所以后来选用了 `switch-case` 语句。

练习六：

这个题目和练习五基本一样，只要能找到触发 kernel monitor 的是 `kern/monitor.c` 中的函数 `monitor()` 就行了。修改过的代码如下：

```
183 static void
184 trap_dispatch(struct Trapframe *tf)
185 {
186     // Handle processor exceptions.
187     // LAB 3: Your code here.
188     switch (tf->tf_trapno) {
189         case T_PGFLT:
190             page_fault_handler(tf);
191             break;
192         case T_BRKPT:
193             monitor(tf);
194             break;
195         default:
196             // Unexpected trap: The user process or the kernel has a bug.
197             print_trapframe(tf);
198             if (tf->tf_cs == GD_KT)
199                 panic("unhandled trap in kernel");
200             else {
201                 env_destroy(curenv);
202                 return;
203             }
204     }
205 }
```

练习七：

首先要为系统调用中断编写中断处理函数、提供接口，和其他中断的道理一样，对以下文件进行修改：

`kern/trapentry.s`：

```
70 TRAPHANDLER_NOEC(t_syscall, T_SYSCALL)
```

`kern/trap.c`：

```
31 void t_syscall();
```

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

然后修改`trap_dispatch()`函数，遇到系统调用中断时调用函数`syscall()`。根据头文件引用，这里的`syscall()`的声明位于`kern/syscall.h`，如下：

```
 9 int32_t syscall(uint32_t num, uint32_t a1, uint32_t a2, uint32_t a3, uint32_t a4, uint32_t
   a5);
```

Lab中已经做过提示：应用程序会把系统调用号以及系统调用的参数放到寄存器中。系统调用号存放到`%eax`中，参数则存放在`%edx` `%ecx` `%ebx` `%edi`和`%esi`中。内核会把返回值送到`%eax`中。所以只需关注`Trapframe`中的成员`tf_regs`即可，代码如下：

```
195         case T_SYSCALL:
196             tf->tf_regs.reg_eax = syscall(tf->tf_regs.reg_eax,
197                                           tf->tf_regs.reg_edx,
198                                           tf->tf_regs.reg_ecx,
199                                           tf->tf_regs.reg_ebx,
200                                           tf->tf_regs.reg_edi,
201                                           tf->tf_regs.reg_esi);
202             break;
```

最后对 `kern/syscall.c` 进行修改，这个部分的作用就是遇到系统调用中断并且已经调用 `syscall()` 函数之后，通过对 `syscallno` 的判断，分别由内核调用相应的系统函数，比如 `sys_cgetc` `sys_getenvid` 等等。具体每一个函数的作用在源代码注视中已经说明清楚，现只需对 `syscallno` 进行分类、对应的参数从前到后往进填即可代码如下：

```
65 // Dispatches to the correct kernel function, passing the arguments.
66 int32_t
67 syscall(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t a3, uint32_t a4, uint32_t a5)
68 {
69     // Call the function corresponding to the 'syscallno' parameter.
70     // Return any appropriate return value.
71     // LAB 3: Your code here.
72
73     //  panic("syscall not implemented");
74
75     switch (syscallno) {
76         case (SYS_cputs):
77             sys_cputs((const char *)a1, a2);
78             return 0;
79         case (SYS_cgetc):
80             return sys_cgetc();
81         case (SYS_getenvid):
82             return sys_getenvid();
83         case (SYS_env_destroy):
84             return sys_env_destroy(a1);
85         default:
86             return -E_INVAL;
87     }
88 }
```

不过在 `inc/syscall.h` 中的枚举里还有一个是 `NSYSCALLS` 不太确定是什么，我大概估计意思是没有定义的系统调用，参考了其他代码也没有考虑这个情况，所以大概就是对应 `default` 了。

练习八：

只需要在 `lib/libmain.c` 里加一句就可以了：

```
11 void
12 libmain(int argc, char **argv)
13 {
14     // set thisenv to point at our Env structure in envs[].
15     // LAB 3: Your code here.
16     thisenv = &envs[ENVX(sys_getenvid())];
17     // save the name of the program so that panic() can use it
18     if (argc > 0)
19         binaryname = argv[0];
20
21     // call user main routine
22     umain(argc, argv);
23
24     // exit gracefully
25     exit();
26 }
```

练习九：

之前都是在用户模式下警告页错误，这一练习处理内核态下的页错误。只需要在 `page_fault_handler` 里判断内核态就可以了。根据提示，应该和 `tf_cs` 的低几位有关系，查阅资料看到段寄存器的低两位决定当前特权级CPL，00为内核级，11为用户级，应该就是环0和环3的意思。据此就可以直接判断当前特权级了，代码如下：

```
268     // LAB 3: Your code here.
269     if(tf->tf_cs && 0x01 == 0) {
270         panic("page_fault in kernel mode, fault address %d\n", fault_va);
271     }
```

然后实现 `user_mem_check` 函数，这个函数的实现我是先看的学习笔记里的代码，然后去理解笔者的想法的。代码如下：

```
616 user_mem_check(struct Env *env, const void *va, size_t len, int perm)
617 {
618     // LAB 3: Your code here.
619     char * end = ROUNDUP((char *)(va + len), PGSIZE);
620     char * start = ROUNDDOWN((char *)va, PGSIZE);
621     pte_t *cur = NULL;
622 
623     for(; start < end; start += PGSIZE) {
624         cur = pgdir_walk(env->env_pgdir, (void *)start, 0);
625         if((int)start > ULIM || cur == NULL || ((uint32_t)(*cur) & perm) != perm) {
626             if(start == ROUNDDOWN((char *)va, PGSIZE))
627                 user_mem_check_addr = (uintptr_t)va;
628             else
629                 user_mem_check_addr = (uintptr_t)start;
630             return -E_FAULT;
631         }
632     }
633     return 0;
634 }
```

对以上代码理解如下：首先要把虚拟地址的范围 [`va` , `va + len`) 转换成整页地址，用之前用过的 `ROUNDUP` 和 `ROUNDDOWN` 。然后在页表中一页一页查看其权限位是不是和 `perm` 一致，这是 `for` 循环的作用。紧接着的 `if` 判断表示，如果页表入口地址在ULIM以上（这一部分内存仅内核可读写）、虚拟地址不存在对应页表入口或者当前页的权限位与 `perm` 不一致，则用户对其没有访问权限，所以将 `user_mem_check_addr` 置为有问题的地址，然后返回错误代码 `-E_FAULT` ，不过要判断一下这个 `start` 是不是第一项，如果是的话实际上的地址应该是 `va` 本身。

最后在 `kern/syscall.c` 的 `sysy_cputs()` 里调用一下就可以了：

```
17 static void
18 sys_cputs(const char *s, size_t len)
19 {
20     // Check that the user has permission to read memory [s, s+len).
21     // Destroy the environment if not:.
22 
23     // LAB 3: Your code here.
24     user_mem_assert(curenv, s, len, 0);
25     // Print the string supplied by the user.
26     cprintf("%.*s", len, s);
27 }
```

练习十：

练习九完成以后，练习十就可以直接过了。