操作系统上机实验指南

第二次作业说明

用户环境

单位：	南开大学机器智能研究所 Institute of Machine Intelligence. Nankai University
日期：	2016/11/01

目录

1. 概述

本次实验分为两部分：“用户环境与异常处理”、“缺页中断、断点异常、系统调用”。

作业概述：本次作业练习共 1 题，问题共 2 题，作业共 9 题。其中练习题为帮助你理解整个系统用，做自己练习，不需要提交。问题以文档的形式，写在你们最终的实验报告中。作业是一些代码工作，需要提交源码。

在本实验中，你将要实现基本的内核，需要获得一个保护的用户模式的运行环境。你需要在 JOS 内核建立数据结构以便和用户环境保持通信，创建一个单独的用户环境，加载程序镜像到这个用户环境并且启动。同时，你也需要让这个内核能够处理任何由用户环境发起的系统请求，并且处理它引起的所有异常。

注释：在本实验中，两个短语“environment（环境）”和“process（进程）”是可以互相代表的——他们有共同的含义。我们使用“environment（环境）”这个词是为了强调一点：JOS 环境并没有提供类似于 UNIX 进程一类同义的词语，尽管他们是类似的。

1.1.启动

使用 Git 获取最新版本的作业目录，然后创建一个本地的分支叫做 lab3，如下：

```
zhong@localhost:~/Desktop/lab$ git pull
Already up-to-date.
zhong@localhost:~/Desktop/lab$ git checkout -b lab3 origin/lab3
Branch lab3 set up to track remote branch refs/remotes/origin/lab3.
Switched to a new branch "lab3"
zhong@localhost:~/Desktop/lab$ git merge lab2
Merge made by recursive.
 kern/pmap.c |   42 ++++++++++++++++++++
 1 files changed, 42 insertions(+), 0 deletions(-)
```

Lab3 包含了一些新的源文件，目录如下：

```
inc/  env.h      Public definitions for user-mode environments
      trap.h     Public definitions for trap handling
      syscall.h  Public definitions for system calls from user environments to the kernel
      lib.h      Public definitions for the user-mode support library
kern/ env.h      Kernel-private definitions for user-mode environments
      env.c      Kernel code implementing user-mode environments
      trap.h     Kernel-private trap handling definitions
      trap.c     Trap handling code
      trapentry.S Assembly-language trap handler entry-points
      syscall.h  Kernel-private definitions for system call handling
      syscall.c  System call implementation code
lib/  Makefrag   Makefile fragment to build user-mode library, obj/lib/libuser.a
      entry.S    Assembly-language entry-point for user environments
      libmain.c  User-mode library setup code called from entry.S
      syscall.c  User-mode system call stub functions
                 User-mode implementations of
      console.c  putchar and getchar,
                 providing console I/O
      exit.c     User-mode implementation of exit
      panic.c    User-mode implementation of panic
user/ *          Various test programs to check kernel lab 3 code
```

另外，一些在 Lab2 中我们给出的源文件在 Lab3 中有所更改，想看这些更改，你可以输入：

```
$ git diff lab2 | more
```

你也可以看看 lab tools guide（http://pdos.csail.mit.edu/6.828/2011/labguide.html ），他包含了一些在本实验中相关的调试用户代码的信息。

1.2.内置汇编

在本实验中，你会发现 GCC 的内置汇编语言特性很有用，尽管也有可能不需要使用它就可以完成本次实验。最起码，你要理解提供给你们的代码中内置汇编语言的片段。在 refrence materials（http://pdos.csail.mit.edu/6.828/2011/reference.html ）页中你会发现 GCC 内置汇编语言的一些信息源。

2. 用户环境与异常处理

在文件 inc/env.h 中有 JOS 用户环境的基本定义，这部分可能会经常使用到。内核使用 Env 数据结构来保存每个用户环境的关键数据。在本次实验中，我们将只创建一个用户环境，但是我们必须使 JOS 内核可以支持多环境（multiple environments）的应用，在后续的实验中，我们将允许一个进程去 fork 其他的进程。在文件 kern/env.c 中，内核维护着如下三个重要的全局变量：

```
struct Env *envs = NULL;                // All environments
struct Env *curenv = NULL;              // The current env
static struct Env *env_free_list;       // Free environment list
```

一旦 JOS 启动并开始运行，envs 就会指向一个保存系统中所有用户环境的 Env 数组。在当前的设计中，JOS 内核将支持最多 NENV 个同时活动的（active）环境，当然，实际运行时的个数将比 NENV 少很多的（NENV 是在文件 inc/env.h 中使用#define 定义的常量）。一旦 envs 数组被分配之后，它就会为每一个存在的环境保存一份 Env 数据。

JOS 内核将所有未激活的（inactive）Env 结构保存在 env_free_list 中。这个设计使得系统可以非常快速的分配和释放用户环境，因为这个链表很少进行添加和删除操作。

内核使用变量 curenv 来保存当前运行的环境（currently executing environment）。在启动的过程中，在第一个环境运行之前，curenv 将被初始化为 NULL。

2.1.用户环境（User Environments）

2.1.1. 环境状态（Environment State）

在文件 inc/env.h 中 Env 是如下定义的：

```
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
};
```

✧ 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。

✧ env_link

指向 env_free_list 的 next Env 的指针，env_free_list 指向空闲列表的 the first free environment。

✧ env_id

该变量保存一个环境的 id，这个 id 唯一的标识了当前使用这个 Env 结构的用户环境。当一个用户环境终止之后，内核可能会将这个 Env 结构分配给新的用户环境，但是这时这个用户环境就会有一个不同的 env_id，即使新的用户环境和旧的使用的是 envs 数组中的同一个位置。

✧ env_parent_id

该变量保存着创建这个环境的环境的 env_id（parent），通过这种方式，内核可以生产一 个家谱（family tree），这在决定哪些环境可以对别的环境做出何种操作时十分有用。

✧ env_type

该变量用以区分环境的类型，对大部分环境来说，它是 ENV_TYPE_USER，空闲的环境是 ENV_TYPE_IDLE。我们会在后续试验介绍更多的类型。

✧ env_status

这个变量有如下值：

ENV_FREE：表明当前的 Env 是没有激活的(inactive)，也就是在 env_free_list 上的。

ENV_RUNNABLE：表明当前 Env 代表的环境正在等待处理器。

ENV_RUNNING:表示该环境正在运行。

ENV_NOT_RUNNABLE：表明这个 Env 是活动的（active）并且当前还没有准备好在处理器上运行，比如它正在其他环境的 IPC（进程通信）。

✧ env_pgdir

该变量保存这个环境页目录（page directory）的虚拟地址。

类似于 Unix 的进程（process），JOS 的环境将“线程（thread）”和“地址空间（address space）” 的概念相联结。线程由保存的寄存器值（env_tf）来定义，地址空间则由页目录和页表（env_pgdir）来决定。要运行一个用户环境，内核必须用保存的寄存器值和相应的地址空间来设置 CPU。

我们的 struct Enc 和 xv6 中的 struct proc 很类似。两种结构都使用 env_tf 来保存环境或者进程的用户态寄存器值。但是与 xv6 不同的是在 JOS 中，单个的环境没有自己的内核堆栈。JOS 中一个时刻只有一个环境运行，因此只需要一个内核栈（single kernel stack）。

2.1.2. 分配环境数组（Allocating the Environments Array）

在 lab1_2 中，我们在 mem_init()中使用 pages 数组分配内存，内核使用 pages 来记录哪些内存被使用哪些是 free 的。我们现在需要继续修改 mem_init()函数，以相似的方式来分配 envs。

作业 **1**

Modify mem_init() in kern/pmap.c to allocate and map the envs array. This array consists of exactly NENV instances of the Env structure allocated much like how you allocated the pages array. Also like the pages array， the memory backing envs should also be mapped user read-only at UENVS (defined in inc/memlayout.h) so user processes can read from this array.

You should run your code and make sure check_kern_pgdir() succeeds.

2.1.3. 创建并运行环境（Creating and Running Environments）

现在，我们将在文件 kern/env.c 中编写运行一个用户环境所需要的代码。由于我们当前 并没有文件系统，我们将设置内核来载入一个静态的嵌入到内核本身的二进制镜像。JOS 以 ELF 格式将这些二进制文件嵌入到内核。

Lab2 的 GNUmakefile 在目录 obj/user/下生成了一系列的二进制文件。如果你查看文件 kern/Makefrag，这些文件是被按照.o 文件的方法直接链接到内核中去的。在链接选项中的-b binary 的目的是将这些文件按照原始的(raw)二进制文件而不是普通的由编译器生成的.o 文 件进行链接。也可以在编译内核之后查看 obj/kern/kernel.sym 文件，你会发现链接程序生成 了一系列好玩并且有趣的符号， 例如:_binary_obj_user_hello_start 、_binary_obj_user_hello_end 和_binary_obj_user_hello_size。

在 kern/init.c 中的 i386_init()函数中，在一个环境中将会运行这些二进制镜像。但是创建用户环境的核心函数还没有完成，你需要完成它们。

作业 **2**

In the file env.c,finish coding the following functions:

env_init()：Initialize all of the Env structures in the envs array and add them to the env_free_list. Also calls env_init_percpu， which configures the segmentation hardware with separate segments for privilege level 0 (kernel) and privilege level 3 (user).

env_setup_vm()：Allocate a page directory for a new environment and initialize the kernel portion of the new environment's address space.

region_alloc()：Allocates and maps physical memory for an environment

load_icode()：You will need to parse an ELF binary image， much like the boot loader already does， and load its contents into the user address space of a new environment.

env_create()：Allocate an environment with env_alloc and call load_icode load an ELF binary into it.

env_run()：Start a given environment running in user mode.

As you write these functions， you might find the new cprintf verb %e useful -- it prints a description corresponding to an error code. For example,

```
r = -E_NO_MEM;
panic("env_alloc: %e", r);
```

will panic with the message "env_alloc: out of memory".

下面是运行一个用户环境的代码调用关系，确保你能理解每一步：

- start (kern/entry.S)
- i386_init (kern/init.c)
  - cons_init
  - mem_init
  - env_init
  - trap_init (still incomplete at this point)
  - env_create
  - env_run
    - env_pop_tf

一旦你完成了这部分的代码,就可以编译内核并在 qemu 中运行了。如果一切顺利，那系统进入用户空间，但是会在 hello 的 init 步骤调用系统时崩溃，因为此时 jos 还没有创建硬件以进行从用户空间到内核空间的转换。当 CPU 发现它无法处理这个系统调用中断，它会产生一个保护异常，发现它不能处理，产生双故障异常，又发现它不能处理，最后放弃给出了所谓的“三重故障”（triple fault）。通常，你会看到 CPU 复位并重新启动系统。虽然这对那些 legacy applications 非常重要，但是对内核开发是痛苦的，因此在我们当前版本的 qemu 上仅看到 register dump 和“triple fault”message.

我们后续会处理这个问题，现在先让我们使用 debugger 检查我们是否进入了用户态（user mode）。使用“make qemu-gdb”并在 env_pop_tf 处设置 GDB 断点，它是你进入用户态之前的最后一个函数。使用 si 命令做单步调试，在 irte 指令后进入用户态。之后,就会看到进入用户环境的第一条指令, 就是在 lib/entry.S 中 start 标签的 cmpl 指令。在 hello 的 sys_cputs() 的 int $0x30 处设置断点（使用 b *0x…命令，其用户态的地址可查找 obj/user/hello.asm），int 这条指令是打印一个字符到控制台的系统调用。如果你不能执行到 int，那你的地址空间设置或者程序代码载入部分可能有问题,在进入下一步之前,一定要将这个问题解决。

2.2.异常处理（Exception Handling）

2.2.1. 处理中断和异常（Handling Interrupts and Exceptions）

当前的代码中，用户程序中的第一个 int$0x30 系统调用就会使系统崩溃：一旦处理器进入用户模式,就无法再从中返回了。现在我们的目标就是实现基本的异常处理和系统调用，以便于内核能恢复对处理器的控制权。首先要做的是熟悉 x86 的中断与异常机制。

练习 **1**

Read Chapter 9

Exceptions and Interrupts (http://pdos.csail.mit.edu/6.828/2011/readings/i386/c09.htm) in the 80386 Programmer's Manual

(http://pdos.csail.mit.edu/6.828/2011/readings/i386/toc.htm) (or Chapter 5 of the IA-32 Developer's Manual http://pdos.csail.mit.edu/6.828/2011/readings/ia32/IA32-3A.pdf), if you haven't already.

本次实验中，我们将大体上遵循 Intel 的有关于中断、异常以及其他相关的术语。然 而，一定要明确的一点是类似于异常（Exception）、陷阱（Trap）、中断（Interrupt）、错误（Fault）以及中止（abort）这样的术语并没有一个标准化的定义，无论是在计算机体系结构中还是在操作系统中，经常忽视他们在不同的体系结构之间的具体区别而随意使用。如果你在本次实验之外的地方看到这些术语（你肯定会的），它们的含义可能有轻微的不同。

2.2.2. 控制权转移（Basics of Protected Control Transfer）

异常和中断都是控制权转移（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的机会去干预内核或者其他的进程。在 Intel 的术语中，中断（Interrupt）是一种保护下的控制权转移，通常是由外部的异步事件例如外部设备的 I/O 行为通知处理器所造成的;异常（Exception）则是由当前正在运行的代码的同步事件所造成的 控制权转移，比如除零错误或者非法的内存访问。

为了保证这些控制权转移是受保护的（protected），处理器的中断/异常机制被设计为在中断或者异常出现时，当前正在运行的代码不需要知道内核以何种方式在何处被载入。相反，处理器保证一定会在严格控制的条件下进入内核。在 x86 体系中，这种保护以如下两种方式体现：

1. 中断描述符表（The Interrupt Descriptor Table， IDT）。处理器保证中断或者异常只会使内核在少数特殊的、完善设计的、被内核自己所预先指定的地方载入，而不是中断或者异常出现时所正在运行的代码决定。

特别的说，在 x86 中，中断和异常最多被区分为 256 种类型，每一种都被一个特殊的中断号（interrupt number，有时候也称为异常号（exception number）或者陷阱号（trap number））所关联。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。IDT 是内核设立在内核的私有内存中的，类似于 GDT。在这个表的某项中，处理器可以获得：

- ✧ 要载入指令指针寄存器（EIP）的值，也就是用来处理这种类型异常的内核代码；
- ✧ 要载入代码段寄存器（CS）的的值，包括了将要执行的 exception handler 的权限等级（privilege），不过在 JOS 中，所有的中断都在内核中进行处理，所以它的权限

是 0。

2. 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。

2.2.3. 中断和异常的类型（Types of Exceptions and interrupts）

X86 处理器所能生成的所有同步异常在内部都是使用中断号 0 到 31 的，并且映射到 IDT 中的 0 到 31 号表项。例如，Intel 从硬件上将缺页中断处理（page fault handler）设置为 14。高于 31 的中断号只被用于处理被 INT 指令引起的软中断（software interrupt），或者是由外部设置在需要时所产生的异步硬中断（hardware interrupt）

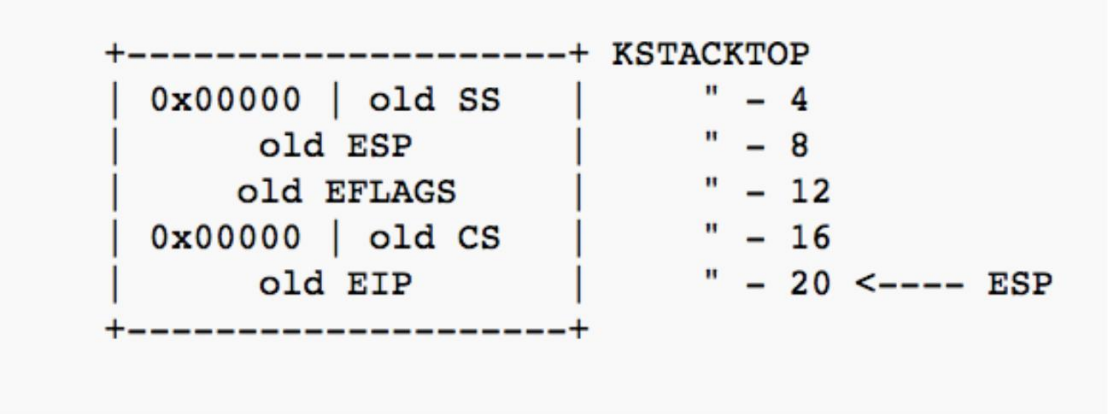
在本节中，我们将扩展 JOS 的中断处理功能，使其能够处理 0 到 31 号由 Intel 定义的中断。在下一节中，我们将扩展 JOS 能够处理软中断 0x30，JOS 使用该软中断进行系统调用的处理。在后续的实验中，我们将继续扩展 JOS，最终使其可以支持诸如时钟中断（clock interrupt）之类的硬中断。

2.2.4. An Example

假设处理器现在执行一个用户环境中的代码，执行到一个除法指令，其除数为 0。那么，将发生如下的事情：

1. 处理器转换到由 TSS 中的 SS0 和 ESP0 定义的堆栈，在 JOS 中，这两个区域分别保存着 GD_KD 和 KSTACKTOP；

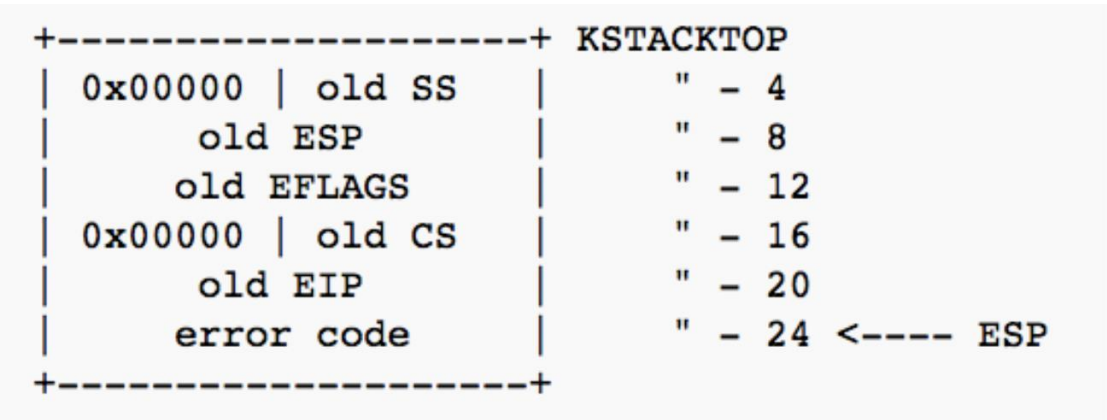
2. 处理器将异常参数压入到内核栈，起始地址为 KSTACKTOP：



3. 由于当前处理的是除法错误，在 x86 上其中断号为 0，处理器读取 IDT 的第 0 项并 将 CS:EIP 设置为指向此处理程序。

4. 处理程序获得控制权并处理异常，比如说结束该用户环境；

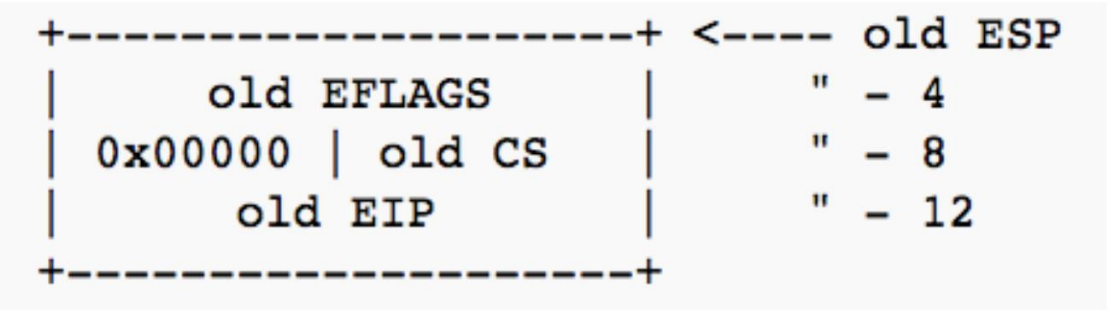
对于某些特定的 x86 异常，处理器会在上述的压栈过程中加入额外的数据：错误码（error code）(比如 page fault exception，number 14)。当要压入错误码时，处理器在从用户模式进入内核时其堆栈应该是如下情况：



2.2.5. 中断和异常的嵌套（Nested Exceptions and Interrupts）

处理器在用户态和内核态中都可以响应异常和中断，但是，只有从用户态转入内核态时，x86 处理器才会在将原有的寄存器值入栈前转换栈，并且从 IDT 中读取相应的异常处理函数入口。当中断或异常发生时，处理器已经处于内核（CS 的最低两个 bit 已经是 0 时），内核只需要将更多的值压入当前的内核栈。这样，内核就可以低代价的处理在内核中发生 的嵌套异常。这种特点在实现保护机制时是十分有效的。

如果处理器已经处于内核模式并产生一个嵌套的异常，由于它不需要转换栈，因此就 不再保存旧的 SS 或 ESP 寄存器的值。对于不需要压入错误码的异常，内核堆栈就是如下：



对于需要压入错误码的，处理器在旧的 EIP 之后压入一个 error code，和上面的一样。

关于嵌套异常还有一点重要的说明，如果处理器在内核模式下又处理了一个异常，并且由于诸如栈空间不足等原因无法将原有的处理器状态压入栈，那么处理器除了进行重置（reset）之外别无选择。也就是说，任何正常的内核都应该在设计时就保证这种情况永远都不会发生。

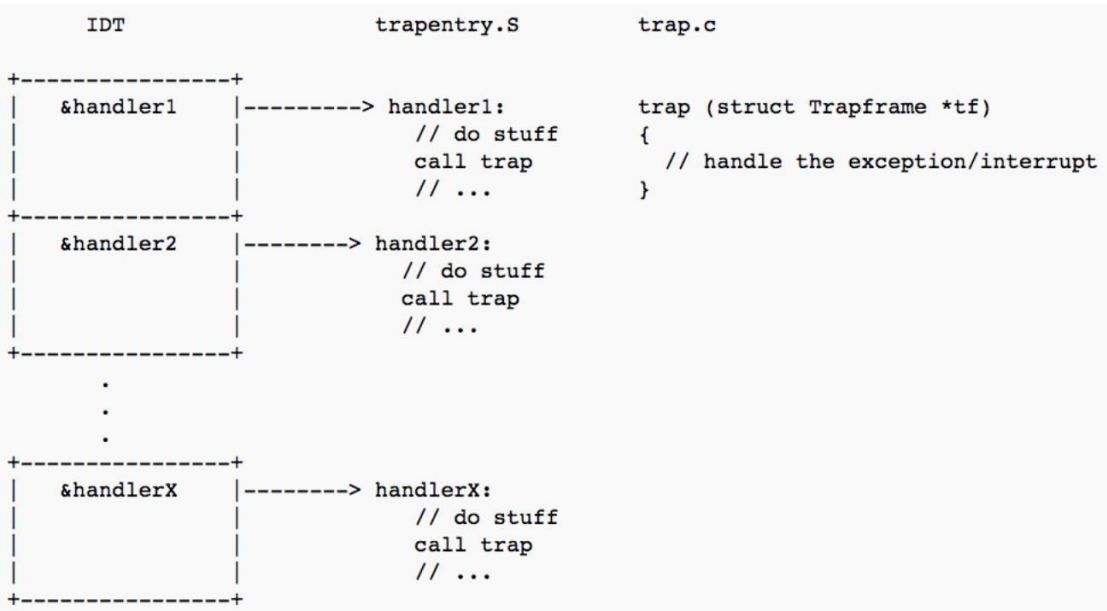
2.2.6. 设置 IDT（Setting Up the IDT）

现在，我们已经拥有足够的知识来设置 IDT 并且在 JOS 中处理异常。现在，我们将设置 IDT 来处理 0 到 31 号的处理器异常（the process exceptions）、32 到 47 的外部设备中断（the device IRQs），在后续将会继续添加。

文件 inc/trap.h 和 kern/trap.h 保存着与中断和异常有关的基本定义。文件 kern/trap.h 包 含的主要用于内核，而 inc/trap.h 则主要用于用户级别的程序和系统中所需要的库

注意：中断号 0 到 31 之间有一部分是被 Intel 定义为保留的，由于它们永远不会被处理 器所产生，所以你可以任意处理这一部分。

你所要实现的整个控制流程如下：



在文件 trapentry.S 中每一个异常或者中断都，有自己的处理程序（handler）。trap_init() 应当初始化 IDT 为这些处理程序的入口地址。每一个处理程序都要在栈上创建一个 结构 struct Trapframe（在 inc/trap.h 中）并且调用函数 trap()，将其参数设置为所指向对应的 Trapframe 的指针。trap()数就会处理这个异常或者将其分派给特定的处理函数。

作业 **3**

Edit trapentry.S and trap.c and implement the features described above. The macros TRAPHANDLER and TRAPHANDLER_NOEC in trapentry.S should help you, as well as the T_* defines in inc/trap.h. You will need to add an entry point in trapentry.S (using those macros) for each trap defined in inc/trap.h , and you'll have to provide _alltrapswhich the TRAPHANDLER macros refer to. You will also need to modify trap_init() to initialize the idt to point to each of these entry points defined in trapentry.S; the SETGATEmacro will be helpful here.

Your _alltraps should:

1. push values to make the stack look like a struct Trapframe
2. load GD_KD into %ds and %es
3. pushl %esp to pass a pointer to the Trapframe as an argument to trap()
4. call trap (can trap ever return?)

Consider using the pushal instruction; it fits nicely with the layout of the struct Trapframe.

Test your trap handling code using some of the test programs in the user directory that cause exceptions before making any system calls, such as user/divzero. You should be able to get make grade to succeed on the divzero , softint , and badsegment tests at this point.

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?)

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 14)?

3. 缺页中断、断点异常，系统调用

尽管你的内核已经有了一些基本的异常处理能力，但你需要细化它来提供基于异常处理的重要的操作系统机制。

3.1.缺页中断（Page Faults）

缺页中断，又称为页面失效，中断号为 14（T_PGFLT），是一个非常重要的中断，我们将在本次实验和下次实验中经常用到。当处理器遇到一个缺页中断时，它将引起错误的线性地址存储到一个特殊的寄存器 CR2 中。在 trap.c 中我们提供了函数 page_fault_handler()来处理缺页异常。

作业 **4**

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. Remember

that you can boot JOS into a particular user program using make run-x or make run-x-nox.

在实现了系统调用之后，你会进一步细化缺页中断。

3.2.断点异常（Breakpoint Exception）

断点异常，中断号为 3（T_BRKPT），通常是用来使调试器可以在程序的任意位置设置断点，调试器临时将该处的指令替换为一字节的 int3 软中断指令。在 JOS 中， 我们扩大它的使用范围，将其设置为原始的伪系统调用（primitive pseudo-system call），使得任何用户都可以调用它来引用 JOS 的内核监视器（JOS kernel monitor）。在用户模式下实现的 lib/panic.c 中的 panic()函数在输出信息之后就会使用 int3 中断。

作业 **5**

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.

问题 **2**

1、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?

2、What do you think is the point of these mechanisms， particularly in light of what the user/softint test program does?

3.3.系统调用（System Call）

用户进程通过使用系统调用来要求内核做一些事情。当用户进程调用系统调用时， 处理

器进入内核模式，处理器和内核协作保存用户进程的状态，内核执行适当的代码来启动系统

调用，然后恢复到用户进程。关键在于：在于用户进程如何获得内核的注意，以及根据系统的不同它如何设定哪一个调用是要执行的。

在 JOS 内核，我们将使用 int 指令，它可以引起处理器中断。也就是说，使用 int $0x30 作为系统调用的中断。代码中已经将 T_SYSCALL 定义为 0x30。之后，你必须设置中断描述符使用户进行可以引发中断。注意中断 0x30 不能从硬件产生，所以毫无疑问使用用户代码产生。

我们将在寄存器中传递系统调用的编号和参数。这样，就不需要在用户进程的堆栈或者

指令序列中搜寻了。系统调用的编号在 %eax 中，参数（最多 5 个）则依次保存在%edx、%ecx、 %ebx、%edi 和%esi 中。内核则将返回值保存在%eax 中。在 lib/syscall.c

中的 syscall()函数中，用户调用系统调用的汇编代码已经完成，请阅读该部分代码。

作业 **6**

在内核中为中断 T_SYSCALL 添加一个处理程序。

完成在 kern/trapentry.S 和 kern/trap.c 文件中的 trap_init()

修改 trap_dispathch()来处理系统调用中断，通过调用 syscall()（在文件 kern/syscall.c 中），使用合适的参数并将返回值写回%eax

完成 kern/syscall.c 文件中的 syscall()函数。如果系统调用号是无效的，syscall()函数返回 -E_INVAL。阅读并理解 lib/syscall.c 文件可以让你更加理解系统调用。

在你的内核运行 user/hello 程序（make run-hello）。它将在控制台输出“hello world”然后会引起一个用户模式下的缺页中断。

3.4.启动用户模式（User-mode Startup）

用户程序在 lib/entry.S 的顶部开始运行。进行一些设置之后，这部分代码会调用 lib/libmain.c 中的 libmain()函数。这个函数负责初始化全局的指向这个程序在 envs[]数组中的 Env 结构的 env 指针（注意 lib/entry.S 已经定义了 envs 来指向 UENVS）可查看 inc/env.h，使用 sys_getenvid。

之后，libmain()调用 umain，假设当前是运行在 user/hello.c 中的 hello 程序。注意在打印出“hello，world”之后，它会尝试访问 thisenv->env_id。这也是它为什么会在上面的实验中

出现缺页中断的原因。现在我们已经设置了 env，所以就不会出错了。如果仍然出错，可能是因为你没有映射 UENVS 区域为用户可读（user-readable）(在 pmap.c 中，这是我们第一次实际使用 UENVS 区域)。

作业 **7**

添加所需代码到用户字典中，然后启动你的内核。让它可以使 user/hello 打印出“hello，world”然后打印“i am environment 00001000”。然后 user/hello 会尝试调用 sys_env_destroy()退出（详见 lib/libmain.c 和 lib/exit.c）。由于内核当前只支持一个程序，所以当前程序退出后，内核就会显示当前唯一的程序已经退出并且陷入内核监视器。此时，make grade 就可以通过 hello 测试了。

3.5.缺页和存储保护（Page faults and memory protection）

存储保护是操作系统的一项至关重要的功能。 通过存储保护，操作系统可以保证一个错误的程序不会影响到其他的程序或者操作系统本身。

技术上讲，操作系统提供存储保护也依赖于硬件的支持。OS 保存着哪些虚拟地址可用而哪些不可用。 如果一个程序访问了不可用的虚拟地址，或者是它无权访问的地址，处理器就会停止执行该程序的这条指令并带着这些信息陷入内核。如果这个问题是可以被修复的，内核就修复该问题并继续运行原来的程序；如果是不可修复的，那么该程序就无法继续运行了，因为这条指令永远无法执行过去。

作为一个可修复错误的例子， 想象一下可自动扩展的栈。 在许多系统中，内核初始时只会分配一个单独的页， 如果程序错误的访问栈以下的页， 内核就会自动分配这些页并让程序继续。 这样，内核只需要尽可能的分配程序所需要的栈空间，而程序则可以假想拥有一个无限大的堆栈。

系统调用对存储保护也提出了一个有趣的问题。 大多数系统调用接口允许用户程序给内核传递指针。 这些指针指向需要被读写的用户内存区。 内核在执行系统调用时就会用到这些指针。这里，主要有如下两个问题：

在内核的缺页错误要比在用户程序中更为严重。 如果内核出现了缺页错误， 那么通常是内核的 bug，而且这些处理函数会停止内核(以及整个系统)。在系统调用中，当内核引用指向用户空间的指针时， 我们需要一种方法来确认这些引用所造成的缺页错误都是位于用户程序的

内核比用户程序有着更高的访问特权。 用户程序会请求内核读取或写入内核中某些用户程序不能读写的区域。 如果内核不够仔细的话， 一个错误的或者是恶意的程序就可以欺骗内核而进行各种非法的操作，甚至完全破坏内核的完整性。

基于以上这些原因，内核在引用用户程序所传递的指针时必须十分仔细。

现在，我们将着手解决这些问题。 JOS 将会仔细审核所有由用户程序传递到内核的指针，然后在内核中执行。也就是说，当指针被传递时， 内核就会检查该地址是否是用户程序可访问的， 以及用户程序的页表是否允许这些内存操作。

由此，内核就永远不会因为引用了一个用户提供的指针和出现缺页错误。如果出现缺页错误，它将会立刻停止。

作业 **8**

完成以下内容：

修改文件 kern/trap.c，使得在内核中出现缺页错误时，就终止(使用 panic)内核。要检验缺页中断是发生在用户模式还是内核模式，可以检查 tf_cs 的最低几位。

阅读文件 kern/pmap.c 中的 user_mem_assert()并实现同文件中的 user_mem_check()。

修改文件 kern/syscall.c 来检查传递给内核的参数

启动内核，运行 user/buggyhello。用户进程会被销毁而内核将会中止(panic)。将会出现如下输出：

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

最后，修改kern/kdebug.c 中的debuginfo_eip，在usd、stabs、stabstr 调用 user_mem_check。如果你运行 user/breakpoint，你就应该从内核监视器中可以运行 backtrace，并且可以在内核发生缺页错误之前看见 backtrace 贯穿 lib/libmain.c。这个缺页错误你不需要修改，但你需要知道为什么会发生这个错误。

注意你刚刚实现的机制与恶意的用户程序相同。

作业 **9**

启动你的内核，运行 user/evilhello。你的环境将会崩溃，内核将会 panic 停止，你将看见一下信息：

```
[00000000] new env 00001000
[00001000] user_mem_check assertion failure for va f0100020
[00001000] free env 00001000
```

至此，本次实验完成。确保你通过了所有的测试。

4. 作业要求

4.1.代码部分

✧ 修改后的源码

4.2.文档部分

需要在提交的作业中提供本次作业的文档，包括但不限于以下内容：
✧ 小组的组号，组中成员的姓名和学号；
✧ 小组中每一位成员在本次作业中的分工以及贡献比例；
✧ 本次作业的设计思路；
✧ 作业中的问题部分。

4.3.提交方式

✧ 当面提交，提交地点：计控学院楼 427　　宋佳慧　　曹先　　蒋 建 飞
✧ 提交要求：演示修改的源码，证明完成作业内容，讲解思路。

4.4.提交时间

本次作业于 2016 年 11 月 1 日发布， 请于 2016 年 11 月 2 7 日当天提交，当天为周日。过期提交者，将有可能在本次作业的评分上产生不利影响。小组如果确认完成无误，也可提前提交。