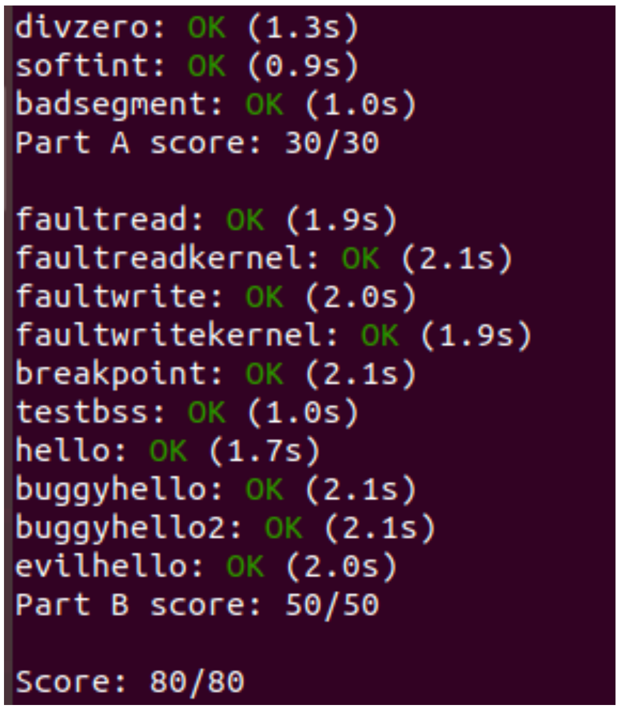
Lab3 User Environments

By 江晓湖 软件工程学院 同济大学

完成度

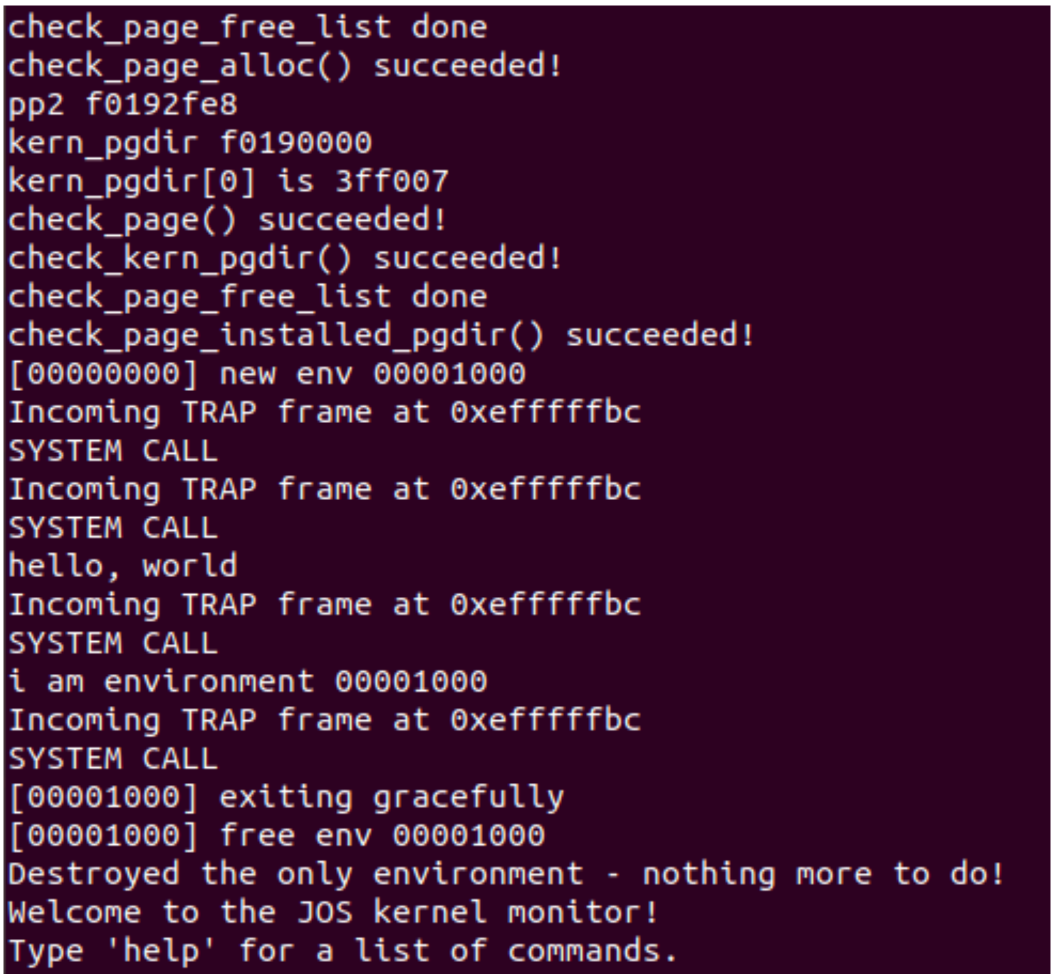
- 完成了所有的exericse
- 完成了一个challenge
- 完成了所有的question
- 代码地址：https://github.com/PtNan/OSCD/tree/lab3
- 结果示例：



```
divzero: OK (1.3s)
softint: OK (0.9s)
badsegment: OK (1.0s)
Part A score: 30/30

faultread: OK (1.9s)
faultreadkernel: OK (2.1s)
faultwrite: OK (2.0s)
faultwritekernel: OK (1.9s)
breakpoint: OK (2.1s)
testbss: OK (1.0s)
hello: OK (1.7s)
buggyhello: OK (2.1s)
buggyhello2: OK (2.1s)
evilhello: OK (2.0s)
Part B score: 50/50

Score: 80/80
```


```
check_page_free_list done
check_page_alloc() succeeded!
pp2 f0192fe8
kern_pgdir f0190000
kern_pgdir[0] is 3ff007
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_free_list done
check_page_installed_pgdir() succeeded!
[00000000] new env 00001000
Incoming TRAP frame at 0xefffffbc
SYSTEM CALL
Incoming TRAP frame at 0xefffffbc
SYSTEM CALL
hello, world
Incoming TRAP frame at 0xefffffbc
SYSTEM CALL
i am environment 00001000
Incoming TRAP frame at 0xefffffbc
SYSTEM CALL
[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.
```

总览

Lab3介绍

提交`make clean && make handin`

Lab3的任务是提供一个用户保护的运行环境。

Lab3实现目标

- 需要增强JOS内核,建立一个数据结构来跟踪记录用户环境。
- 创建单用户的用户环境,把运行程序镜像载入,并运行
- 你的内核还要能够处理相应用户环境里程序的系统调用以及各种异常

Lab3可分为两个部分

- ***PartA***

  User Environments and Exception Handling 用户环境和异常处理

- ***PartB***

  Page Faults, Breakingpoints Exceptions, and System Call 页面错误处理、断点处理、系统调用处理

Lab3新增文件

folder	file	description
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
	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/libjos.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
	console.c	User-mode implementations of 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

Part A: User Environments and Exception Handling

在这部分，我们需要设计JOS内核来创建和支持用户环境。

阅读`inc/env.h`,可以看到Env结构体,内核要用该结构体来跟踪用户环境。

接下来看到`kern/env.c`

```
struct Env *envs = NULL;          // 所有的 environments
struct Env *curenv = NULL;        // 当前的 current env
static struct Env *env_free_list;  // 空闲的 environment list
```

内核用这三个结构来控制用户环境

内核运行开始时, `envs` 指向一个 Env数组(该数组一一对应所有的environment) 和之前Page的思想类似

在jos里,作者设计的是最多同时运行 `NENV` 个environment，也就是说初始化时NENV是数组的个数。

Environment State

```
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
};
```

变量说明

```
env_tf:
      这个结构体用于保存寄存器,内核切换environment时用的。
env_link:
      用于env_free_list,指向下一个空闲environment
env_id:
     内核用该值储存一个唯一标识. 在一个用户environment终止后,内核可用重申请同一个Env结构用于不同的environment,新的environment会有不同的env_id.
env_parent_id:
     内核用该值存创建该environment的environment, 这样environment就可以形成一个树,可以用于做安全判断environment是否被允许对某物做某事.
env_type:
     用于区分ENV_TYPE_USER(大多数)和ENV_TYPE_IDLE,在未来的lab里会用
env_status:
     ENV_FREE:          非活跃 在env_free_list中
     ENV_RUNNABLE:      活跃   已准备好 等待run
     ENV_RUNNING:       活跃   当前正在 run
     ENV_NOT_RUNNABLE:  活跃   但未准备好, 比如在等待另一个environment的interprocess communication (IPC)
env_pgdir:
     保存kernel virtual address of this environment's page directory.
env_cr3:
      保存对应的environment's page directory的物理地址
```

Exercise 1

```
Exercise 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.
```

修改`kern/pmap.c`中的`mem_init()`申请并映射envs数组. 数组元素为NENV个,同时envs的权限为(user read-only)只读,最终用`check_kern_pgdir()`检测是否正确。

分析:

查看`inc/memlayout.h`看到UENVS这一块大小为PTSIZE

所以新增两段代码为(在对应提示的位置)

```
  //////////////////////////////////////////////////////////////////////
  // Make 'envs' point to an array of size 'NENV' of 'struct Env'.
  // LAB 3: Your code here.
  envs = (struct Env*) boot_alloc(sizeof(struct Env) * NENV); //allocated
```

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

运行`make qemu-nox`可以看到

```
check_kern_pgdir() succeeded!
check_page_installed_pgdir() succeeded!
kernel panic at kern/env.c:460: env_run not yet implemented
```

Creating and Running Environments

因为我们尚无文件系统,我们要运行的程序都是嵌入在kernel内部的,作为elf镜像嵌入,我们将要运行的源程序都在`user/`里,编译后的在`obj/user/`里。

Exercise 2

```
Exercise 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 to 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".
```

在`kern/env.c`中实现特定的函数

1 env_init()

`env_init()` 初始化envs并且把它们加入`env_free_list`. 并调用`env_init_percpu()`(它配置段硬件并配置权限0(内核)权限3(用户)),参照`pages_init()`实现如下

```
void
env_init(void)
{
  // Set up envs array
  // LAB 3: Your code here.
  int i;
  for(i = NENV - 1;i >= 0; i--){
    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

`env_setup_vm()` 为新环境申请一个页目录， 并初始化这个新的environment的内核地址空间部分。

这里建立一个独自的pgdir,但只拷贝UTOP以上,故可以用`memmove(e->env_pgdir, kern_pgdir, PGSIZE)`把`kern_pgdir`复制一份放到`e->env_pgdir`并设置`UVPT`在PD中的对应位置的PDE。

```
// Initialize the kernel virtual memory layout for environment e.
// Allocate a page directory, set e->env_pgdir accordingly,
// and initialize the kernel portion of the new environment's address space.
// Do NOT (yet) map anything into the user portion
// of the environment's virtual address space.
//
// Returns 0 on success, < 0 on error.  Errors include:
//  -E_NO_MEM if page directory or table could not be allocated.
//
static int
env_setup_vm(struct Env *e)
{
  int i;
  struct PageInfo *p = NULL;

  // Allocate a page for the page directory
  if (!(p = page_alloc(ALLOC_ZERO)))
    return -E_NO_MEM;

  // Now, set e->env_pgdir and initialize the page directory.
  //
  // Hint:
  //    - The VA space of all envs is identical above UTOP
  //  (except at UVPT, which we've set below).
  //  See inc/memlayout.h for permissions and layout.
  //  Can you use kern_pgdir as a template?  Hint: Yes.
  //  (Make sure you got the permissions right in Lab 2.)
  //    - The initial VA below UTOP is empty.
  //    - You do not need to make any more calls to page_alloc.
  //    - Note: In general, pp_ref is not maintained for
  //  physical pages mapped only above UTOP, but env_pgdir
  //  is an exception -- you need to increment env_pgdir's
  //  pp_ref for env_free to work correctly.
  //    - The functions in kern/pmap.h are handy.

  // LAB 3: Your code here.
  p->pp_ref++;
  e->env_pgdir = (pde_t*) page2kva(p);
  memcpy(e->env_pgdir,kern_pgdir,PGSIZE);//kern_pgdir as template

  // UVPT maps the env's own page table read-only.
  // Permissions: kernel R, user R
  e->env_pgdir[PDX(UVPT)] = PADDR(e->env_pgdir) | PTE_P | PTE_U;
  return 0;
}
```

3 region_alloc()

`region_alloc(struct Env *e, void *va, size_t len)`申请len字节的物理地址地址空间,并把它映射到虚拟地址va上。

注释提示不要初始化为0或其它操作,页需要能被用户写,如果任何步骤出错panic，另外需要注意页对齐。

回想lab2 我们有把物理地址和虚拟地址的映射的函数,也有把虚拟地址和PageInfo做映射的函数，实现思路为:

- 计算va起始和末尾
- for(起始->末尾){
- 申请页//不要做任何初始化操作
- 申请失败则panic
- 把该页和for的va映射 注意权限位
- }

实现如下:

```
// Allocate len bytes of physical memory for environment env,
// and map it at virtual address va in the environment's address space.
// Does not zero or otherwise initialize the mapped pages in any way.
// Pages should be writable by user and kernel.
// Panic if any allocation attempt fails.
//
static void
region_alloc(struct Env *e, void *va, size_t len)
{
  // LAB 3: Your code here.
  // (But only if you need it for load_icode.)
  //
  // Hint: It is easier to use region_alloc if the caller can pass
  //   'va' and 'len' values that are not page-aligned.
  //   You should round va down, and round (va + len) up.
  //   (Watch out for corner-cases!)
  uintptr_t va_start = (uintptr_t)ROUNDDOWN(va , PGSIZE);
  uintptr_t va_end = (uintptr_t)ROUNDUP(va + len, PGSIZE);
  uintptr_t i;
  for(i = va_start; i < va_end; i += PGSIZE){
    struct PageInfo *pg = page_alloc(0);//no initialization
    if(!pg){
      panic("region_alloc failed!");
    }
    page_insert(e->env_pgdir, pg, (void*)i, PTE_W|PTE_U);//read and write
  }
}
```

4 load_icode()

`load_icode(struct Env *e, uint8_t *binary, size_t size)`是用来解析ELF二进制文件的,和boot loader已经完成了的工作很像,把文件内容装载进新的用户环境。该函数只会在内核初始化时,第一个用户模式环境开始前,调用。

该函数把ELF文件装载到适当的用户环境中,从适当的虚拟地址位置开始执行,清零程序头部标记的段,可以参考`boot/main.c`(该部分是从磁盘读取的)

阅读注释，我们需要加载 `ph->p_type == ELF_PROG_LOAD` 的`ph`的,

- 每个段的虚拟地址 `ph->p_va`
- 内存大小 `ph->p_memsz` 字节
- 文件大小 `ph->p_filesz` 字节
- 需要把 `binary + ph->p_offset` 拷贝到`ph->p_va`,其余的内存需要被清零
- 设置其对于用户可写
- ELF 段不必页对齐,可以假设没有两个段会使用同一个虚拟页面
- 建议函数`region_alloc`

参考`boot/main.c`中的`bootmain()`函数,其中readseg为读磁盘,之后为获得ph,把`ph~eph`读入了内存,然后执行,那我们的实现思路为:

- 读取binary 判断是否为ELF
- 切换cr3
- for(ph~eph){
- 注意判断是否为`ELF_PROG_LOAD`
- 通过`region_alloc`来申请va以及memsz
- 清零所有
- 复制程序的filesz
- }
- 复原cr3 为`kern_pgdir`

实现如下

```
// Set up the initial program binary, stack, and processor flags
// for a user process.
// This function is ONLY called during kernel initialization,
// before running the first user-mode environment.
//
// This function loads all loadable segments from the ELF binary image
// into the environment's user memory, starting at the appropriate
// virtual addresses indicated in the ELF program header.
// At the same time it clears to zero any portions of these segments
// that are marked in the program header as being mapped
// but not actually present in the ELF file - i.e., the program's bss section.
//
// All this is very similar to what our boot loader does, except the boot
// loader also needs to read the code from disk.  Take a look at
// boot/main.c to get ideas.
//
// Finally, this function maps one page for the program's initial stack.
//
// load_icode panics if it encounters problems.
//  - How might load_icode fail?  What might be wrong with the given input?
//
static void
load_icode(struct Env *e, uint8_t *binary)
{
  // Hints:
  //  Load each program segment into virtual memory
  //  at the address specified in the ELF segment header.
  //  You should only load segments with ph->p_type == ELF_PROG_LOAD.
  //  Each segment's virtual address can be found in ph->p_va
  //  and its size in memory can be found in ph->p_memsz.
  //  The ph->p_filesz bytes from the ELF binary, starting at
  //  'binary + ph->p_offset', should be copied to virtual address
  //  ph->p_va.  Any remaining memory bytes should be cleared to zero.
  //  (The ELF header should have ph->p_filesz <= ph->p_memsz.)
  //  Use functions from the previous lab to allocate and map pages.
  //
  //  All page protection bits should be user read/write for now.
  //  ELF segments are not necessarily page-aligned, but you can
  //  assume for this function that no two segments will touch
  //  the same virtual page.
  //
  //  You may find a function like region_alloc useful.
  //
  //  Loading the segments is much simpler if you can move data
  //  directly into the virtual addresses stored in the ELF binary.
  //  So which page directory should be in force during
  //  this function?
  //
  //  You must also do something with the program's entry point,
  //  to make sure that the environment starts executing there.
  //  What?  (See env_run() and env_pop_tf() below.)

  // LAB 3: Your code here.
  struct Elf * elf = (struct Elf *) binary;
  struct Proghdr *ph, *eph;
  //valid elf?
  if(elf->e_magic != ELF_MAGIC){
    panic("load_icode failed! invalid ELF!");
  }

  ph = (struct Proghdr *) ((uint8_t *) elf + elf->e_phoff);
  eph = ph +elf->e_phnum;
  //switch cr3
  lcr3(PADDR(e->env_pgdir));

  for(;ph < eph;ph++){
    if(ph->p_type == ELF_PROG_LOAD){
      region_alloc(e,(void*)ph->p_va,ph->p_memsz);
      memset((void*)ph->p_va,0,ph->p_memsz);
      memmove((void*)ph->p_va,
      binary+ph->p_offset,ph->p_filesz);
    }
  }
  lcr3(PADDR(kern_pgdir));
  e->env_tf.tf_eip = elf->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);
}
```

5 env_create()

`env_create()` 申请 environment并调用 `load_icode` 来装载ELF binary到申请的environment中，只会在kernel初始化的时候执行一次,并且new environment的parent id设为0。注释提示使用函数 `env_alloc()`

实现代码为:

```
// Allocates a new env with env_alloc, loads the named elf
// binary into it with load_icode, and sets its env_type.
// This function is ONLY called during kernel initialization,
// before running the first user-mode environment.
// The new env's parent ID is set to 0.
//
void env_create(uint8_t *binary, enum EnvType type)
{
  // LAB 3: Your code here.
  struct Env *e;
  int i = env_alloc(&e,0);
  if(i != 0){
    panic("env_create: %e",i);
  }
  e->env_type = type;
  load_icode(e,binary);
}
```

6 env_run()

`env_run(struct Env *e)` 在用户态运行给定的environment。

查看注释

- 如果有正在运行的程序并且不是它自己,则把当前运行的程序设为RUNNABLE (`curenv`,`env_status`)
- 设置当前正在运行的为e (`curenv`,`env_status`)
- 更新`e->env_runs`的计数
- 和上面一样用`lcr3()`切换cr3
- 用`env_pop_tf()`重新加载环境寄存器
- 检查之前的函数`e->env_tf`的值是否正确

实现如下:

```
// Context switch from curenv to env e.
// Note: if this is the first call to env_run, curenv is NULL.
//
// This function does not return.
//
void
env_run(struct Env *e)
{
  // Step 1: If this is a context switch (a new environment is running):
  //     1. Set the current environment (if any) back to
  //        ENV_RUNNABLE if it is ENV_RUNNING (think about
  //        what other states it can be in),
  //     2. Set 'curenv' to the new environment,
  //     3. Set its status to ENV_RUNNING,
  //     4. Update its 'env_runs' counter,
  //     5. Use lcr3() to switch to its address space.
  // Step 2: Use env_pop_tf() to restore the environment's
  //     registers and drop into user mode in the
  //     environment.

  // Hint: This function loads the new environment's state from
  //  e->env_tf.  Go back through the code you wrote above
  //  and make sure you have set the relevant parts of
  //  e->env_tf to sensible values.

  // LAB 3: Your code here.
  if(curenv != e){
    if(curenv && curenv->env_status == ENV_RUNNING){
      curenv->env_status = ENV_RUNNABLE;
    }
    curenv = e;
    e->env_status = ENV_RUNNING;
    e->env_runs++;
    lcr3(PADDR(e->env_pgdir));

  }
  env_pop_tf(&e->env_tf);
```

```
    //panic("env_run not yet implemented");
  }
```

至此 我们实现了用户环境的装入、运行、切换。

用户程序的唤起顺序如下:

- `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`

现在我们可以使用gdb查看函数是否工作正常:

```
(gdb) b env_pop_tf
Breakpoint 1 at 0xf01039ee: file kern/env.c, line 469.
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0xf01039ee <env_pop_tf>:    push   %ebp

Breakpoint 1, env_pop_tf (tf=0xf01b2000) at kern/env.c:469
469    {
(gdb) s
=> 0xf0103a00 <env_pop_tf+18>:    mov    0x8(%ebp),%esp
470         asm volatile(
(gdb) si
=> 0xf0103a03 <env_pop_tf+21>:    popa
0xf0103a03    470         asm volatile(
(gdb) si
=> 0xf0103a04 <env_pop_tf+22>:    pop    %es
0xf0103a04 in env_pop_tf (
    tf=<error reading variable: Unknown argument list address for `tf'.>)
    at kern/env.c:470
470         asm volatile(
(gdb) si
=> 0xf0103a05 <env_pop_tf+23>:    pop    %ds
0xf0103a05    470         asm volatile(
(gdb) si
=> 0xf0103a06 <env_pop_tf+24>:    add    $0x8,%esp
0xf0103a06    470         asm volatile(
(gdb) si
=> 0xf0103a09 <env_pop_tf+27>:    iret
0xf0103a09    470        asm volatile(
(gdb) si
=> 0x800020:    cmp    $0xeebfe000,%esp
0x00800020 in ?? ()
(gdb) si
=> 0x800026:    jne    0x80002c
```

至目前为止工作正常。

Handling Interrupts and Exceptions

到目前这步， `int $0x30`system的用户系统调用是一个死循环,一旦从内核态进入用户态就回不来了,现在要实现基本的 exception和系统调用的处理,以致内核能从用户态代码拿回处理器控制权。

Excerices 3

```
Exercise 3. Read Chapter 9, Exceptions and Interrupts in the 80386
Programmer's Manual (or Chapter 5 of the IA-32 Developer's Manual), if you
haven't already.
```

Basics of Protected Control Transfer

Exceptions 和 interrupts 都是`protected control transfers`,都是让处理器从用户态(CPL=3) 转为内核态(CPL=0)。 同时也不会给用户态代码任何干扰内核运行的机会,在intel的术语中interrupt通常为处理器外部异步事件引起的`protected control transfers`，比如外部I/O活动。作为对比，exception为同步事件引起的`protected control transfers`，例如除0、访问无效内存等。

为了确保这些`protected control transfers`能真正的起到保护作用,因此设计的是当exception或interrupt发生时,并不是任意的进入内核,而是处理器要确保内核能控制才会进入,用了以下两个方法:

1. The Interrupt Descriptor Table

也就是IDT,该表让processer设置内核对特定中断的特定的入口点，而不会继续执行错误的代码，x86系统允许256个不同的interrupt/exception入口点，即interrupt vector (也就是0~255的整数)，中断类型决定一个数值，CPU用interrupt vector的值作为index在IDT中找值放入eip，也就是指向内核处理该错误的函数入口。另外，加载到代码段(CS)寄存器中的值,第0-1位包括要运行异常处理程序的权限级别。(在JOS中,所有异常都在内核模式下处理,权限级别为0)。

简单的说就是不同的错误(interrupt/exception)会发出不同的值(0~255)，然后cpu再根据该值在IDT中找处理函数入口，所以我们的任务要去配置IDT表，以及实现对应的处理函数。

2. The Task State Segment.

在中断前需要保存当前程序的寄存器等，在处理完后回重新赋值这些寄存器，所以保存的位置需要不被用户修改 否则在重载时可能造成危害。

因此x86在 处理interrupt/trap，模式从用户转换到内核时,它还会转换到一个内核内存里的栈(一个叫做TSS(task state segment)的结构体)，处理器把SS, ESP, EFLAGS, CS, EIP, 和一个 optional error code push到这个栈上,然后它再从IDT的配置里设置CS和EIP的值，再根据新的栈设置ESP和SS。

虽然 TSS很大并有很多用途,但对于jos的lab我们只用它来定义处理器在从用户模式转换到内核模式时,应切换的堆栈。x86上JOS在kernel态的权限级别为0，在进入内核模式时，处理器用TSS的ESP0 和SS0两个字段来定义内核栈， JOS不使用其它的TSS字段。

Types of Exceptions and Interrupts

x86 能生成的所有同步exceptions的值(interrupt vector) 在0~31,比如page fault 会触发14号,＞31的部分是给软件中断使用的,可以由int 指令生成或外部异步硬件产生。

An Example

用一个example来说明,如果用户程序执行除0

1. 处理器根据TSS的SS0和ESP0字段切换栈(这两个字段在JOS会分别设为`GD_KD`和`KSTACKTOP`)
2. 处理器按照以下格式push exception参数到 内核栈上

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

3. 因为我们要处理 除0 错误,对应 interrupt vector的值为0,在x86中处理器去读配置的IDT表配置的0号的入口，然后设置CS:EIP指向该入口
4. 然后该处理函数处理，比如终止用户程序。

对于明确的错误 比如上面的除0 ,处理器还会把 错误号push上去 也就是interrupt vector

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

Nested Exceptions and Interrupts

处理器在内核态和用户态都可以处理 exceptions和interrupts。但只有从用户态转换到内核态时，在push old 寄存器前，处理器会自动转换栈，并根据IDT配置调用适当的exception处理程序。如果发生interrupt/exception时 已经在内核态(CS寄存器的低两位为0)，那么CPT只会push 更多的值在同一个内核栈上，这种情况下，内核可以优雅的处理内核自己引发的嵌套exceptions。这种能力是实现保护的重要途径。

如果说发生exceptions或interrupts时本身就在内核态，那么也就不需要储存old SS和EIP，以不push error code 的exception/interrupt为例,内核栈长这样

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

一个需要注意的点是，内核处理嵌套exceptions能力有限，若已经在内核态并接受到一个exception，而且还不能push它old state到内核栈上(比如栈空间不够了)，那么这样的处理无法恢复，因此它会简单地reset它自己,我们不应让这样的情况发生。

Setting Up the IDT

接下来，我们将要设置 IDT的0~31,一会还要设置system call interrupt,在未来的lab会设置32~47(device IRQ)

阅读`inc/trap.h`和`kern/trap.h`，`inc/trap.h`定义了 一些interrupt vector的常量宏和两个数据结构PushRegs和Trapframe。`kern/trap.h`则定义的是两个全局变量`extern struct Gatedesc idt[];`和`extern struct Pseudodesc idt_pd;`以及一堆函数的申明

关于`Gatedesc`和`Pseudodesc`这两个结构体的定义可以在inc/mmu.h中找到

Note: 0~31 有些是保留定义的,但在lab里并不会由processer产生,它们的处理函数怎么写也无所谓,可以按我们认为最简洁的处理。

整个流程如下所画：

```
                  IDT                   trapentry.S         trap.c

     +----------------+
     |   &handler1    |---------> handler1:          trap (struct Trapframe
*tf)
     |                |             // do stuff      {
     |                |             call trap          // handle the
exception/interrupt
     |                |             // ...           }
     +----------------+
     |   &handler2    |--------> handler2:
     |                |            // do stuff
     |                |            call trap
     |                |            // ...
     +----------------+
                 .
                 .
                 .
     +----------------+
     |   &handlerX    |--------> handlerX:
     |                |             // do stuff
     |                |             call trap
     |                |             // ...
     +----------------+
```

每一个 exception/interrupt 需要在trapentry.S有它自己的handler, `trap_init()`函数要做的是把 IDT中填上这些handler函数的地址,每一个handler需要建立一个`struct Trapframe` 在`//do stuff`的位置,然后 调用trap.c中的trap函数,然后trap再处理具体的exception/interrupt或者分发给更具体的处理函数。

Exercise 4

```
Exercise 4. 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 _alltraps which 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 SETGATE macro 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.
```

编辑trapentry.S和trap.c实现上面描述的。 在trapentry.S中的宏`TRAPHANDLER`和`TRAPHANDLER_NOEC`对我们会有帮助。我们需要用inc/trap.h中定义的宏，在trapentry.S定义入口函数，我们还需要编辑TRAPHANDLER 宏引用的`_alltraps`和`trap_init()`初始化idt 来指向每一个 trapentry.S中拟定已的entry point; 另外，阅读`SETGATE`宏会有帮助。

我们的`_alltraps`应该满足:

1. 按照Trapframe 的结构push值
2. 装载`GD_KD` 到 %ds和 %es
3. pushl %esp 传递一个指向 Trapframe 的指针作为 trap()的参数
4. call trap

对于`_alltraps`我们可以考虑使用`pushal` 指令，它和struct Trapframe很契合。

用user/下的代码测试我们写的处理程序，它们会引发一些 exceptions 比如 用户/除零，我们需要能够 `make grade`通过 divzero, softint, 和badsegment tests。

分析:

先看宏`TRAPHANDLER(name, num)`注释中说需要在trap.c中定义一个类似`void NAME();`的函数，然后把NAME作为参数传给`TRAPHANDLER(name, num)`，num为错误号，`TRAPHANDLER_NOEC` 是NO ERROR CODE的版本。这两个宏实际是函数模板,这里我们用这两个宏来实现上面图中trapentry.S的handlerX的部分,关于哪个vector会push 错误号，可以参考下图:

再根据trap.h里定义的,我们的实现为

```
/*
 * Lab 3: Your code here for generating entry points for the different traps.
 */
```

```
TRAPHANDLER_NOEC(DivideEntry, T_DIVIDE) // 0 divide error
TRAPHANDLER_NOEC(DebugEntry, T_DEBUG) // 1 debug exception
TRAPHANDLER_NOEC(NoMaskEntry, T_NMI) // 2 non-maskable interrupt
TRAPHANDLER_NOEC(BreakEntry, T_BRKPT) // 3 break point
TRAPHANDLER_NOEC(OverFlowEntry,T_OFLOW) // 4 over flow
TRAPHANDLER_NOEC(BoundsEntry, T_BOUND) // 5 bounds check
TRAPHANDLER_NOEC(OpCodeEntry, T_ILLOP) // 6 illegal opcode
TRAPHANDLER_NOEC(DeviceEntry, T_DEVICE) // 7 device not available
TRAPHANDLER(SysErrorEntry,  T_DBLFLT) // 8 system error
TRAPHANDLER(TaskSwitchEntry, T_TSS) // 10 invalid task switch segment
TRAPHANDLER(SegmentEntry, T_SEGNP) // 11 segment not present
TRAPHANDLER(StackEntry, T_STACK) // 12 stack exception
TRAPHANDLER(ProtectEntry, T_GPFLT) // 13 general protection error
TRAPHANDLER(PageEntry, T_PGFLT) // 14 page fault
TRAPHANDLER_NOEC(FloatEntry, T_FPERR) // 16 floating point error
TRAPHANDLER_NOEC(AlignEntry, T_ALIGN) // 17 aligment check
TRAPHANDLER_NOEC(MachineEntry, T_MCHK) // 18 machine check
TRAPHANDLER_NOEC(SIMDFloatEntry, T_SIMDERR) // 19 SIMD floating point error
TRAPHANDLER_NOEC(SysCallEntry, T_SYSCALL) // 48 system call
```

这样 我们 完成了handlerX中具有每个特性的东西。

下面在`_alltraps`中实现它们共性的东西——按照Trapframe结构push数据，`Trapframe`的结构如下:

```
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));
```

`below`注释以下的是硬件push的,`tf_trapno`是我们刚刚 用TRAPHANDLER模板实现的,那我们现在还剩下的就是把 顶部5个push,设置ds和es,再把最后的栈顶地址作为结构体首部地址压栈。调用trap(也就是上面写的应该满足)。

实现如下

```
/*
 * 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
```

接下来，开始实现trap函数`trap_init`，需要用到宏`SETGATE(gate, istrap, sel, off, dpl)`，该宏定义在`inc/mmu.h`中。

首先声明前面汇编中的函数名

```
  void DivideEntry ();// 0 divide error
  void DebugEntry  ();//1 debug exception
  void NoMaskEntry    ();//  2 non-maskable interrupt
  void BreakEntry  ();//  3 breakpoint
  void OverFlowEntry  ();//  4 overflow
  void BoundsEntry  ();//  5 bounds check
  void OpCodeEntry  ();//  6 illegal opcode
  void DeviceEntry ();//  7 device not available
  void SysErrorEntry ();//  8 double fault
  void TaskSwitchEntry    ();//10 invalid task switch segment
  void SegmentEntry  ();// 11 segment not present
  void StackEntry  ();// 12 stack exception
  void ProtectEntry  ();// 13 general protection fault
  void PageEntry  ();// 14 page fault
  void FloatEntry  ();// 16 floating point error
  void AlignEntry  ();// 17 aligment check
  void MachineEntry  ();// 18 machine check
  void SIMDFloatEntry();// 19 SIMD floating point error
  void SysCallEntry();// 48 system call
```

然后配置IDT表。

```
  SETGATE(idt[T_DIVIDE ],0,GD_KT,DivideEntry ,0);
  SETGATE(idt[T_DEBUG  ],0,GD_KT,DebugEntry  ,0);
  SETGATE(idt[T_NMI    ],0,GD_KT,NoMaskEntry    ,0);
  SETGATE(idt[T_BRKPT  ],0,GD_KT,BreakEntry  ,3);
  SETGATE(idt[T_OFLOW  ],0,GD_KT,OverFlowEntry  ,0);
  SETGATE(idt[T_BOUND  ],0,GD_KT,BoundsEntry  ,0);
  SETGATE(idt[T_ILLOP  ],0,GD_KT,OpCodeEntry  ,0);
  SETGATE(idt[T_DEVICE ],0,GD_KT,DeviceEntry ,0);
  SETGATE(idt[T_DBLFLT ],0,GD_KT,SysErrorEntry ,0);
  SETGATE(idt[T_TSS    ],0,GD_KT,TaskSwitchEntry    ,0);
  SETGATE(idt[T_SEGNP  ],0,GD_KT,SegmentEntry  ,0);
  SETGATE(idt[T_STACK  ],0,GD_KT,StackEntry  ,0);
  SETGATE(idt[T_GPFLT  ],0,GD_KT,ProtectEntry  ,0);
  SETGATE(idt[T_PGFLT  ],0,GD_KT,PageEntry  ,0);
  SETGATE(idt[T_FPERR  ],0,GD_KT,FloatEntry  ,0);
  SETGATE(idt[T_ALIGN  ],0,GD_KT,AlignEntry  ,0);
  SETGATE(idt[T_MCHK   ],0,GD_KT,MachineEntry   ,0);
  SETGATE(idt[T_SIMDERR],0,GD_KT,SIMDFloatEntry,0);
  SETGATE(idt[T_SYSCALL],0,GD_KT,SysCallEntry,3);
```

至此 若用户除零中断发生则->硬件检测并push需要push的值->硬件根据我们在`trap_init()`中SETGATE配的IDT表找到我们的处理函数入口-> 该处理函数是由trapentry.S中TRAPHANDLER模板实现,并调用`_alltraps`->`_alltraps`在之前push的基础上再push上Trapframe结构体相复合的数据，放置其头部地址(指针)->调用trap(已经由作者实现)->调用`trap_dispatch`(需要我们补充)。在这里`divzero`,`softint`以及`badsegment`的处理都只是`print_trapframe`+`env_destroy`。

执行`make grade` 根据检测代码，PartA的测试已经通过。

```
divzero: OK (2.7s)
softint: OK (1.1s)
badsegment: OK (0.9s)
Part A score: 30/30
```

Challenge

```
Challenge! You probably have a lot of very similar code right now, between the
lists of TRAPHANDLER in trapentry.S and their installations in trap.c. Clean
this up. Change the macros in trapentry.S to automatically generate a table
for trap.c to use. Note that you can switch between laying down code and data
in the assembler by using the directives .text and .data.
```

我们现在肯定有许多相似度很高的代码，你可以修改`trapentry.S`中的宏以及`trap.c`中的IDT设置，让代码更简洁、自然。提示：可以使用`laying down code`以及`data in the assembler`，具体为`directives .text and .data`。

分析：

参考xv6中的`trap_init()`：

```
extern uint32_t vectors[];
for(int i = 0; i < T_SYSCALL; i++)
  SETGATE(idt[i], 0, GD_KT, vectors[i], 0);
SETGATE(idt[T_SYSCALL], 1, GD_KT, vectors[T_SYSCALL], 3);
```

我们可以在`trapentry.S`中定义一个`funtion array`：`vectors`

```
.data
    .globl  vectors
vectors:
```

通过使用`directives .text and .data`，我们可以实现`laying down code`以及`data in the assembler`。我们对宏`TRAPHANDLER`进行如下修改，使之兼容`TRAPHANDLER_NOEC`：

```
#define TRAPHANDLER(name, num)            \
    .data;  \
        .long name; \
    .text;  \
        .globl name;    /* define global symbol for 'name' */ \
      .type name, @function;  /* symbol type is function */   \
      .align 2;   /* align function definition */   \
  name:     /* function starts here */    \
        .if !(num == 8 || num == 17 || (num >= 10 && num <= 14));   \
        pushl $0;   \
        .endif;     \
      pushl $(num);             \
      jmp _alltraps
```

然后我们对这两个宏的使用进行修改，修改为：

```
    TRAPHANDLER(vector0, T_DIVIDE) // 0 divide error
    TRAPHANDLER(vector1, T_DEBUG) // 1 debug exception
    TRAPHANDLER(vector2, T_NMI) // 2 non-maskable interrupt
    TRAPHANDLER(vector3, T_BRKPT) // 3 break point
    TRAPHANDLER(vector4,T_OFLOW) // 4 over flow
    TRAPHANDLER(vector5, T_BOUND) // 5 bounds check
    TRAPHANDLER(vector6, T_ILLOP) // 6 illegal opcode
    TRAPHANDLER(vector7, T_DEVICE) // 7 device not available
    TRAPHANDLER(vector8,  T_DBLFLT) // 8 system error
    TRAPHANDLER(vector9, 9)
    TRAPHANDLER(vector10, T_TSS) // 10 invalid task switch segment
    TRAPHANDLER(vector11, T_SEGNP) // 11 segment not present
    TRAPHANDLER(vector12, T_STACK) // 12 stack exception
    TRAPHANDLER(vector13, T_GPFLT) // 13 general protection error
    TRAPHANDLER(vector14, T_PGFLT) // 14 page fault
    TRAPHANDLER(vector15, 15)
    TRAPHANDLER(vector16, T_FPERR) // 16 floating point error
    TRAPHANDLER(vector17, T_ALIGN) // 17 aligment check
    TRAPHANDLER(vector18, T_MCHK) // 18 machine check
    TRAPHANDLER(vector19, T_SIMDERR) // 19 SIMD floating point error
    TRAPHANDLER(vector20, 20)
    TRAPHANDLER(vector21, 21)
    TRAPHANDLER(vector22, 22)
    TRAPHANDLER(vector23, 23)
    TRAPHANDLER(vector24, 24)
    TRAPHANDLER(vector25, 25)
    TRAPHANDLER(vector26, 26)
    TRAPHANDLER(vector27, 27)
    TRAPHANDLER(vector28, 28)
    TRAPHANDLER(vector29, 29)
    TRAPHANDLER(vector30, 30)
    TRAPHANDLER(vector31, 31)
    TRAPHANDLER(vector32, 32)
    TRAPHANDLER(vector33, 33)
    TRAPHANDLER(vector34, 34)
    TRAPHANDLER(vector35, 35)
    TRAPHANDLER(vector36, 36)
    TRAPHANDLER(vector37, 37)
    TRAPHANDLER(vector38, 38)
    TRAPHANDLER(vector39, 39)
    TRAPHANDLER(vector40, 40)
    TRAPHANDLER(vector41, 41)
    TRAPHANDLER(vector42, 42)
    TRAPHANDLER(vector43, 43)
    TRAPHANDLER(vector44, 44)
    TRAPHANDLER(vector45, 45)
    TRAPHANDLER(vector46, 46)
    TRAPHANDLER(vector47, 47)
    TRAPHANDLER(vector48, T_SYSCALL) // 48 system call
```

最后修改`trap.c`中对应的代码，改为：

```
  //Challenge
  extern uint32_t vectors[];
  int i;
  for(i = 0; i <= T_SYSCALL; i++){
    switch(i){
      case T_SYSCALL:
        SETGATE(idt[i],0,GD_KT,vectors[i],3);
        break;
      default:
        SETGATE(idt[i],0,GD_KT,vectors[i],0);
    }
  }
```

Question 1 & Answer

```
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?)
```

A1: 因为不同的 exception和interrupt有不同的处理机制，每个exception/interrupt需要各自独立的处理函数。如果用同一个处理函数则无法区分错误类型。

Question 2 & Answer

```
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)?
```

A2: 当前softint是运行在用户模式下，dlp为3，但是int 指令是系统级别指令，它的dlp为0，所以用户不能产生int $14，而是引发general protection(trap 13)。如果要让 softint产生int $14,那就把对应的权限位dpl 设为3即`SETGATE(idt[T_PGFLT ],0,GD_KT,ENTRY_PGFLT ,3)`,但是这样会让用户有管理内存的权限，这是越权的。

```

```

以上便是PartA的内容，再回顾一下excption/interrupt的处理流程：

错误发生->硬件int->硬件根据IDTR寄存器的前部分得到IDT的地址->硬件根据IDT找函数入口->硬件push->硬件进入函数入口->处理函数push->trap()。

Part B: Page Faults, Breakpoints Exceptions, and System Calls

现在有基本处理机制了，现在需要实现更强大的exception的处理机制。

Handling Page Faults

`The page fault exception`,`interrupt vector 14`(`T_PGFLT`), 是一个非常重要的一个中断，(前面的lab只实现了页的工作相关函数，测试都是用硬编码测试没有中断机制)。当处理器产生了一个page fault,，它会保存引发错误的线性/虚拟地址到CR2。作者已经在`trap.c`中的`page_fault_handler()`实现了部分(kernel态的page fault没有处理)，我们之后需要完全实现它。

Exercise 5

```
Exercise 5. 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. For
instance, make run-hello-nox runs the hello user program.
```

编辑`trap_dispatch()`来分发页错误到`page_fault_handler()` 然后需要通过`make grade`的faultread, faultreadkernel, faultwrite, 和 faultwritekernel tests.测试，你可以用`make run-x`或`make run-x-nox`来运行特殊的用户程序 ,比如`make run-faultread-nox`.

实现如下 值得注意的是`page_fault_handler` 是无返回的 它会销毁当前的用户程序 所以 这里有没有break是一样的

```
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);
    return;
  default:
    break;
  }
 }
```

The Breakpoint Exception

断点异常 也就是`int 3`(`T_BRKPT`),是允许调试程序向用户代码中临时"插入/取代"的断点指令，在Lab中我们将轻微地滥用该指令，将它转化为任何用户程序都可以用来调用内核monitor的 原始伪系统，这种方法也有它的合理性，比如你可以直接把jos kernel看成一个原始调试器, 比如用户模式下`lib/panic.c`中的`panic`在输出panic信息后会`while(1){int 3}`

Exercise 6

```
Exercise 6. 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.
```

编辑`trap_dispatch()`让断点异常能调用kernel monitor.你现在需要通过`make grade` 的`breakpoint`测试。

先看`kern/monitor.c`的`monitor(struct Trapframe *tf)`接受参数tf，那在`trap_dispatch`中加上

```
case T_BRKPT:
  cprintf("trap T_BRKPT:breakpoint\n");
  monitor(tf);
  return ;
```

然后我们需要在`trap.c`中修改`T_BRKPT`的`dpl`使得用户有权限触发`breakpoint exceptions`

```
extern uint32_t vectors[];
  int i;
  for(i = 0; i <= T_SYSCALL; i++){
    switch(i){
      case T_BRKPT:
      case T_SYSCALL:
        SETGATE(idt[i],0,GD_KT,vectors[i],3);
        break;
      default:
        SETGATE(idt[i],0,GD_KT,vectors[i],0);
    }
  }
```

`make grade`得到`55/80`

Question 3 & Answer

```
Q3: 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?
```

A3: 将break point exception 的dlp设为3即可。

Question 4 & Answer

```
Q4: What do you think is the point of these mechanisms, particularly in light
of what the user/softint test program does?
```

A4: 这些机制的重点就是保护，保护内核环境不被其他用户环境所破坏，所以用dlp对用户权限进行限制。

System Call

用户程序通过使用系统调用来让内核帮它们完成它们自己权限所不能完成的事情，当用户程序调用`System Call`时 处理器进入内核态，处理器+内核合作一起保存用户态的状态，内核再执行对应的`System Call`的代码，完成后再返回用户态。但用户如何调用`System Call`的内容和过程因系统而异。

程序会用寄存器传递系统调用号和系统调用参数，系统调用号放在%eax中，参数依次放在`%edx`，`%ecx`，`%ebx`，`%edi`中，内核执行完后返回值放在%eax中，在`lib/syscall.c`的`syscall()`函数中已经写好了汇编的系统调用函数的一部分。

Exercise 7

```
Exercise 7. Add a handler in the kernel for interrupt vector T_SYSCALL. You
will have to edit kern/trapentry.S and kern/trap.c's trap_init(). You also
need to change trap_dispatch() to handle the system call interrupt by calling
syscall() (defined in kern/syscall.c) with the appropriate arguments, and then
arranging for the return value to be passed back to the user process in %eax.
Finally, you need to implement syscall() in kern/syscall.c. Make sure
syscall() returns -E_INVAL if the system call number is invalid. You should
read and understand lib/syscall.c (especially the inline assembly routine) in
order to confirm your understanding of the system call interface. Handle all
the system calls listed in inc/syscall.h by invoking the corresponding kernel
function for each call.

Run the user/hello program under your kernel (make run-hello). It should print
"hello, world" on the console and then cause a page fault in user mode. If
this does not happen, it probably means your system call handler isn't quite
right. You should also now be able to get make grade to succeed on the testbss
test.
```

为内核加上`T_SYSCALL`的处理方法，我们需要修改`kern/trapentry.S`和`kern/trap.c's trap_init()`，并且要使得`trap_dispatch()`可以处理`syscall()`引起的中断，然后考虑返回给`user process`的`%eax`。最后，完成`syscall()`函数，如果`syscall()`的`number`是非法的，也需要返回`-E_INVAL`。

可以阅读`lib/syscall.c`的代码，以便更好地理解，我们需要处理`inc/syscall.h`列举的所有`system calls`。

首先我们完善`syscall`函数

```
// Dispatches to the correct kernel function, passing the arguments.
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((envid_t) a1);
  default:
    return -E_INVAL;
  }
}
```

然后我们需要调用`user_mem_assert`确认用户模式下是否有权限在`sys_cputs`中读取内存

```
// Print a string to the system console.
// The string is exactly 'len' characters long.
// Destroys the environment on memory errors.
static void
sys_cputs(const char *s, size_t len)
{
  // Check that the user has permission to read memory [s, s+len).
  // Destroy the environment if not.

  // LAB 3: Your code here.
  if(curenv->env_tf.tf_cs & 3){
    user_mem_assert(curenv,s,len,0);
  }
  // Print the string supplied by the user.
  cprintf("%.*s", len, s);
}
```

最后，我们修改`trap_dispatch`函数，加上一段代码，使之能处理`T_SYSCALL`中断：

```
if(tf->tf_trapno == T_SYSCALL){
    cprintf("SYSTEM CALL\n");
    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;
  }
```

User-mode startup

用户程序开始运行于 `lib/entry.S`。 在一些配置后代码调用发生在`lib/libmain.c`中的`libmain()`。 我们需要修改`libmain()`： 初始化全局指针`thisenv`指向当前用户环境的Env. (提示：Part A的 `lib/entry.S`已经定义了 envs 指向UENVS个environments。可以查看`inc/env.h`并使用`sys_getenvid`)。

`libmain()`之后会调用`umain()`,也就是 每一个函数的`主函数`， `user/hello.c`在主函数结束后会尝试访问`thisenv->env_id`。之前我们没有实现，这里会报错。现在应该正确了，如果还报错请检查它的是否是用户可读

Exercise 8

```
Exercise 8. Add the required code to the user library, then boot your kernel.
You should see user/hello print "hello, world" and then print "i am
environment 00001000". user/hello then attempts to "exit" by calling
sys_env_destroy() (see lib/libmain.c and lib/exit.c). Since the kernel
currently only supports one user environment, it should report that it has
destroyed the only environment and then drop into the kernel monitor. You
should be able to get make grade to succeed on the hello test.
```

在`libmain()`添加代码，让user/hello能输出 "i am environment 00001000"。 user/hello 然后尝试`sys_env_destroy()`来退出 (see lib/libmain.c and lib/exit.c)。 因为现在只有一个用户环境，因此内核应该报告销毁了唯一用户环境并进入内核monitor。

需要`make grade` 通过 hello 测试。

`sys_getenvid()` 可以得到当前`env_id`,通过`kern/env.c`中`envid2env()`函数中的方式，实现如下

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

Page faults and memory protection

内存保护是操作系统的一个重要的功能,确保一个程序的bug不会破坏操作系统和其它程序。操作系统总是依赖于硬件的支持来实现内存保护。操作系统记录了哪些虚拟地址是有效的，哪些是无效的。 当一个程序尝试访问无效的地址或者它没有权限的地址，处理器在这个引发fault的程序的指令位置停止然后trap进内核 with information about the attempted operation。如果fault可以修复，则内核可以修复它让程序继续运行，如果不能修复则不会执行该指令及以后的指令。

举一个修复的例子，考虑自动增加的stack。在很多系统中内核初始化只申请了一个stack页, 如果一个程序访问的超过了页大小,内核需要申请新的页让程序继续。通过这样，内核只申请这个程序真实需要的stack内存, 但在程序看来它一直有很大内存。

系统调用给内存保护带来了一个有趣的问题。大多数系统调用接口允许用户传递指向内核传递了一个指针，这些指针指向用户的用来读或者写的buffer，然后内核使用这些指针工作，这样有两个问题：

1、内核里的页错误相对于用户的页错误是有更大的潜在危险。如果管理数据结构时内核页错误，那会引起内核的bug, 并且错误会使整个内核异常。但是当内核使用这些用户给的指针时，应该只属于用户的行为错误，不应产生内核bug。

2、 内核有更多的内存读写权限。用户传来的指针也可能指向一个内核才有权限的地址，内核需要能分辨它对用户的权限是否满足要求。

根据上面两个原因，我们都应该小心的处理用户程序。

可以用审查所有从用户传给内核的指针的方法来解决这两个问题，检查它是否是用户可访问的以及它是否已经分配。

如果本身内核的页错误，那内核应该panic并终止。

Exercise 9

```
Exercise 9. Change kern/trap.c to panic if a page fault happens in kernel
mode.

Hint: to determine whether a fault happened in user mode or in kernel mode,
check the low bits of the tf_cs.

Read user_mem_assert in kern/pmap.c and implement user_mem_check in that same
file.

Change kern/syscall.c to sanity check arguments to system calls.

Boot your kernel, running user/buggyhello. The environment should be
destroyed, and the kernel should not panic. You should see:

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

Finally, change debuginfo_eip in kern/kdebug.c to call user_mem_check on usd,
stabs, and stabstr. If you now run user/breakpoint, you should be able to run
backtrace from the kernel monitor and see the backtrace traverse into
lib/libmain.c before the kernel panics with a page fault. What causes this
page fault? You don't need to fix it, but you should understand why it
happens.
```

修改`kern/trap.c` 使之若在kernel mode 发生页错误则panic，提示通过 `tf_cs`的低位检测当前处于什么模式

阅读`kern/pmap.c`中的`user_mem_assert()`函数并实现`user_mem_check()`函数.

修改`kern/syscall.c`以至能健全的检查系统调用的参数.

运行`user/buggyhello` 用户环境应当被销毁，但内核不应panic. 你应该看到:

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

最后修改`kern/kdebug.c`中的`debuginfo_eip` 让它在 `usd`,`stabs`,和 `stabstr`上调用调用`user_mem_check` 。

如果你运行`user/breakpoint`，你必须在终端上可以运行`backtrace`在`page fualt`产生`panic`之前来跟踪`lib/libmain.c`。

提示：你刚刚实现的机制对`malicious user applications` 也试用(such as user/evilhello)

一步一步，先`kern/trap.c`的`page_fault_handler`加上 是否是内核态的检测

```
if ((tf->tf_cs & 0x3) == 0)
  panic("kernel page fault");
```

然后看`user_mem_assert` 发现它是对`user_mem_check`的一个封装，如果`user_mem_check`出错`user_mem_assert`就直接destroy用户环境了

`user_mem_check`说 va 和len都没有页对齐，我们应该检查它覆盖的所有部分，权限应满足`perm | PTE_P`， 地址应小于ULIM。如果出错设置`user_mem_check_addr`的值为第一个出错的虚拟地址，正确则返回0，失败返回`-E_FAULT`

回顾`pte_t * pgdir_walk(pde_t *pgdir, const void *va, int create)`函数 传入 (页目录,虚拟地址,是否新建) 返回 页表项，我们对`user_mem_check`实现如下

```
int
user_mem_check(struct Env *env, const void *va, size_t len, int perm)
{
  // LAB 3: Your code here.
  uint32_t start = (uint32_t)ROUNDDOWN((char *)va, PGSIZE);
  uint32_t end = (uint32_t)ROUNDUP((char *)va+len, PGSIZE);
  for(; start < end; start += PGSIZE) {
    pte_t *pte = pgdir_walk(env->env_pgdir, (void*)start, 0);
    if((start >= ULIM) || (pte == NULL) || !(*pte & PTE_P) || ((*pte & perm)
!= perm)) {
      user_mem_check_addr = (start < (uint32_t)va ? (uint32_t)va : start);
      return -E_FAULT;
    }
  }
  return 0;
}
```

最后修改`kern/kdebug.c`函数，加上对usd，stabs,stabstr的地址的检测，实现如下：

```
if (user_mem_check(curenv, usd, sizeof(struct UserStabData), PTE_U))
    return -1;

  ...

  if (user_mem_check(curenv, stabs, sizeof(struct Stab), PTE_U))
    return -1;

  if (user_mem_check(curenv, stabstr, stabstr_end-stabstr, PTE_U))
    return -1;
```

至此`make grade`已经`80/80`

我们在终端中调用`bakctrace`，可以看到`page fault`是由`accessing memory 0xeebfe000`引起的：

```
K> backtrace
Stack backtrace:
ebp effffff20 eip f01008df args 00000001 effffff38 f01a0000 00000000 f017da40
       kern/monitor.c:147: monitor+276
ebp effffff90 eip f01034ff args f01a0000 efffffbc 00000000 00000082 00000000
       kern/trap.c:161: trap+153
ebp efffffb0 eip f0103733 args efffffbc 00000000 00000000 eebfdfd0 efffffdc
       kern/syscall.c:69: syscall+0
ebp eebfdfd0 eip 00800073 args 00000000 00000000 eebfdff0 00800049 00000000
       lib/libmain.c:28: libmain+58
ebp eebfdff0 eip 00800031 args 00000000 00000000Incoming TRAP frame at
0xeffffea4
kernel panic at kern/trap.c:240: Page fault in kernel mode
```

Exercise 10

```
Exercise 10. Boot your kernel, running user/evilhello. The environment should
be destroyed, and the kernel should not panic. You should see:

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

Exercise 9完成后，10也就自动完成了。

参考文档

- panic source code
- iret
- Interrupt vector table
- IDT
- How does the kernel know if the CPU is in user mode or kenel mode?
- Where is the mode bit?
- PUSHA
- X86 Assembly/Other Instructions
- X86 Registers
- How are the segment registers (fs, gs, cs, ss, ds, es) used in Linux?
- LInux 描述符GDT, IDT & LDT结构定义
- The difference between Call Gate, Interrupt Gate, Trap Gate?
- ARM GCC Inline Assembler Cookbook
- GCC-Inline-Assembly-HOWTO
- Labels in GCC inline assembly
- eflags
- Trap flag
- far call

- Call gate