

操作系统及实习（实验班）

JOS-Lab3

运行时环境
实习报告

姓名 杜若谷 学号 1300012855

日期 2016.4.6

目录

内容一：总体概述.....	3
内容二：任务完成情况.....	3
任务完成列表（Y/N）	3
具体 Exercise 的完成情况.....	3
内容三：遇到的困难以及解决方法.....	15
内容四：收获及感想.....	15
内容五：对课程的意见和建议.....	16
内容六：参考文献.....	17

内容一：总体概述

这个 Lab 分为 Part A 和 Part B 两部分。其中第一部分主要工作是实现运行时环境。第二部分则是实现中断和异常的处理。

Part A 的工作简而言之就是实现进程，并且最终成功执行第一个进程。首先需要熟悉的概念是 Env，这个结构对应于操作系统课程中 PCB 的概念，剩余的工作则完全围绕这个概念进行。首先是为所有的 Env 分配空间，映射到页表，保存地址在 envs 内，然后初始化。

创建进程需要为 Env 初始化虚拟内存空间，具体来说目前实现的就是将内核页表 UTOP 以上部分复制一份，当然还要单独设置自己的自映射。对于内存空间当然还要实现内存的分配机制。

随后则需要实现从 ELF 文件加载内容到内存空间。由于目前的 JOS 没有文件系统，因此首先 ELF 程序被链接入内核内，由一些标识符标志。最后则是完成运行进程的功能。

Part B 除了实现中断异常处理之外，还需要实现系统调用功能。JOS 已经提供加载中断向量表的功能。但是除此之外我们还需要实现中断处理程序，并且在中断向量表中初始化中断向量。

由于 JOS 的设置是所有中断都由一个函数分发，因此我们需要经常改动这个函数来处理不同的中断异常。

实现系统调用需要完成所有以上部分，并且在统一的位置分发系统调用。在系统调用中还需要注意用户传来的地址是否合法，如果非法需要强制结束进程。

内容二：任务完成情况

任务完成列表 (Y/N)

Exercise1	Exercise2	Exercise3	Exercise4	Exercise5	
Y	Y	Y	Y	Y	
Exercise6	Exercise7	Exercise8	Exercise9	Exercise10	Challenge2
Y	Y	Y	Y	Y	Y

具体 Exercise 的完成情况

第一部分

Exercise1

在第一部分我们需要为 JOS 的 PCB 分配空间。JOS 的 PCB 也就是 Env 结构体，而 JOS 中最多可以同时管理的进程数已经写在变量 NENV 中。因此这里空间的总大小就是

```
sizeof(struct Env) * PGSIZE
```

和之前分配 PageInfo 空间一样，这里的分配同样使用了 boot_alloc() 函数。由于函数中已经进行了页对齐，所以我们传入参数的时候就不必考虑页了。分配完后使用 memset 将这一部分空间清空。

之后则需要将这一部分空间加入页表，具体过程和 PageInfo 完全一致，因此不再赘述。值得注意的是，在这一部分我们使用了 page_insert() 方法一页一页进行插入，因此为了防止最后缺少一页，我们需要在这个地方的 for 循环中进行页对齐，才能获得我们最终使用的页数。

```
6828 decimal is 15254 octal!
Physical memory: 66556K available, base = 640K, extended = 65532K
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_installed_pgdir() succeeded!
kernel panic at kern/env.c:460: env_run not yet implemented
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> 
```

最终结果如上，check_kern_pgdir() 成功，证明之前的修改已经完成。

Exercise2

这一部分需要完成一些和 Env 有关的函数。

首先是初始化函数 env_init()，这个函数和 page_free_list 的初始化很相像。首先将每个 Env 结构体的 env_id 设置为 0，随后将其加入 env_free_list 队列。唯一不同的是这个函数要求 env_free_list 的顺序和在内存中的顺序相同，以便 env_alloc() 能够首先弹出 envs[0]，因此我们不能使用初始化 page_free_list 的方法即从 0 开始加入队列，反过来我们需要从后往前加入队列，这样最终形成的队列和其内存中的顺序相同。

其次是初始化 env_setup_vm() 函数，这个函数的功能是为一个 Env 结构体分配一个页目录，并且初始化设置 UTOP 以上部分，使其与 kern_pgdir 相同。特殊的部分是 UVPT 的页目录，由于需要设置自映射，因此这里需要单独设置。最后还需要注意的是新分配的页目录页的 pp_ref 需要加 1，功能是为了使 env_free() 工作正常。

随后是完成 region_alloc() 函数，其功能是在 Env 的虚拟地址空间内的虚拟地址 va 处分配 len 大小的空间。这个函数虽然看似很简单，但是有很多需要注意的地方，首先 va 和 len 都不一定是页对齐的，因此需要先对 va 进行 ROUNDDOWN，随后还要对 len 进行 ROUNDUP，另外 len 的大小还应该包括 va 被 ROUNDDOWN 的部分。

然后是 load_icode() 函数，其作用是加载内存中存储的一个 ELF 文件，因为 JOS 目前没有文件系统，所以所有的程序都是直接编码在内存中的，因此这一部分和 boot 阶段从硬盘中加载 kernel 的工作有所不同。这个函数根据 ELF 文件中

所确定的位置加载部分段，并且从 ELF 文件中拷贝部分内容，同时清空.bss 段等空间。随后为用户栈分配一页，最后则是设置 env 的 eip 为程序入口。

env_create() 函数。这个函数的功能十分简单，即创建一个进程。这个函数首先调用 env_alloc 函数分配一个 Env，随后调用 load_icode 加载用户程序。

env_run() 函数。这个函数的功能是运行指定 Env 所代表的进程。所以这个函数不会返回。首先判断当前是否有进程在运行（由 curenv 标志），是的话需要改变其状态，随后将当前进程送上 curenv，改变状态和调度数，加载页表。随后调用 env_pop_tf 函数，这个函数的功能是将 Env 中寄存器的值加载到 CPU 上，随后调用 iret 返回用户态，运行用户进程。

系统完成到这个部分应该可以从 kern/init.c 的函数 i386_init 看到已经可以经过 cons_init, mem_init, env_init, trap_init, env_create, env_run 这几个阶段，但是 trap_init 还没有完成，因此在 env_run 中 env_pop_tf 进入用户态后开始运行 hello 程序。根据 Write Up 指示跟踪进入 env_pop_tf 后，查看编译后的 hello.asm 文件，找到 sys_cputs() 函数中的 INT 0x30 指令（看来这个编译没有使用动态链接库）地址在 0x800b9d，在此处设置断点后跟踪，程序跳转到 0xfe05b 继续执行，一段时间后重启，说明遇到问题，应该与中断处理程序还未设置有关。

```
(gdb) break *0x800b9d
Breakpoint 2 at 0x800b9d
(gdb) c
Continuing.
=> 0x800b9d:    int    $0x30
Breakpoint 2, 0x00800b9d in ?? ()
```

Exercise3

这一个 Exercise 的主要工作是阅读 IA32 用户手册，熟悉中断和异常的处理流程。

首先 NMI(NonMaskable Interrupt)和 Exception 已经预先设置为 0-31 号中断。第 32 号到第 255 号可以设置为 Maskable Interrupt。Exception 可以分为 Faults, Traps, Aborts，但是这个分类在这个 Lab 中其实关系不大。

中断描述符表(IDT)是有许多中断描述符组成的数组。中断描述符又称为 Gate。要找到中断所对应的中断描述符，需要从 IDTR 找到中断描述符表的指针，随后根据中断号找到相应的 Gate。

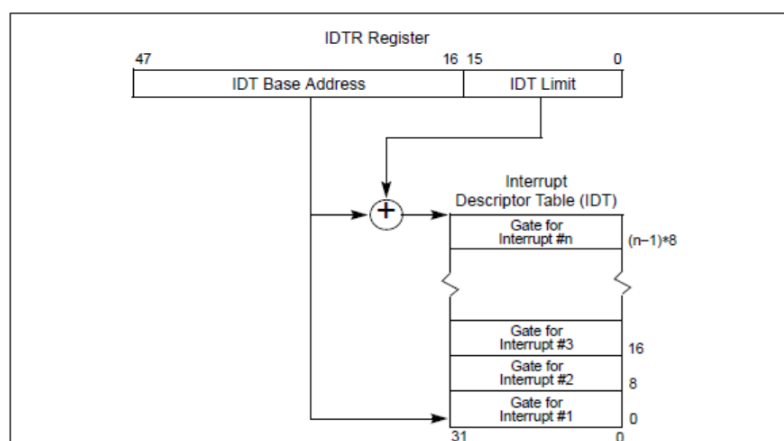
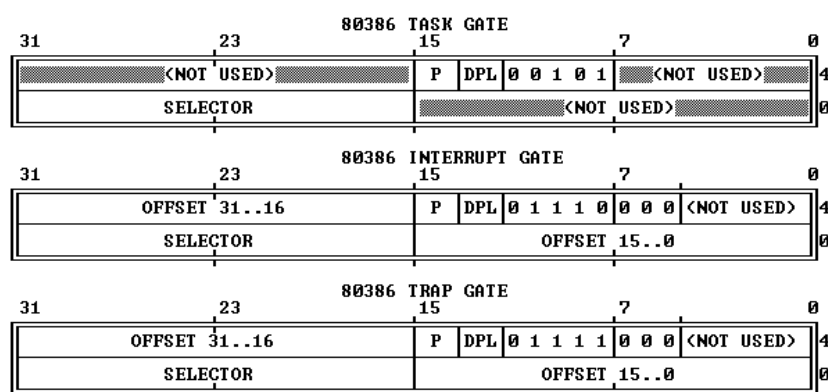


Figure 5-1. Relationship of the IDTR and IDT

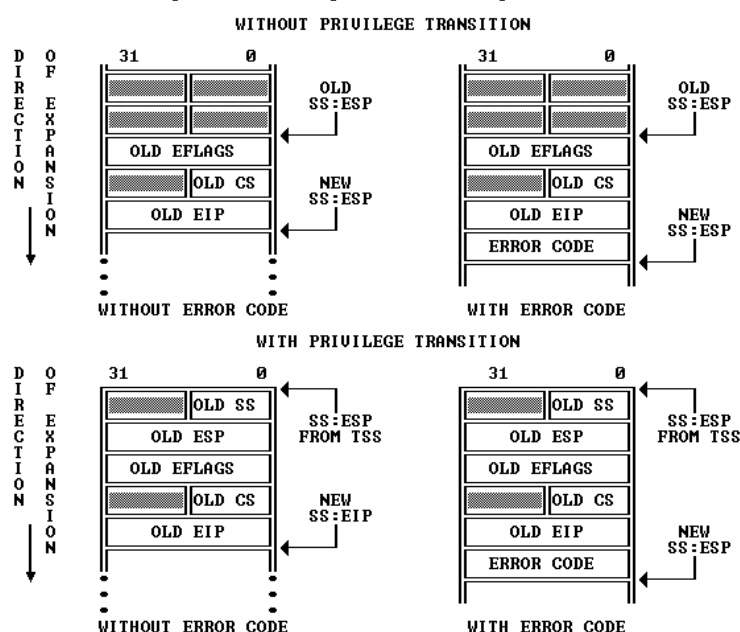
每个 Gate 是由中断处理程序段选择符，偏移和一些权限位组成(如下图)。我们需要设置的是 DPL 位，如果为 0 表明只有内核态能进入这个门，如果为 3 表示用户态也可以进入这个门。

Figure 9-3. 80386 IDT Gate Descriptors



进入中断处理程序之前硬件会先将一些内容压栈，分为有 ERROR CODE，无 ERROR CODE 两种。还分为需要切换堆栈和不需要切换堆栈两种。

Figure 9-5. Stack Layout after Exception of Interrupt



Exercise4

这个 EXERCISE 的主要工作是设置前 32 个中断地 IDT 中断向量表的内容以及所对应的中断处理程序。在 JOS 中这些中断处理程序都是类似的，其工作主要是构建好 Trapframe 结构，并且跳转到一个统一的位置，进行统一的初始化后跳转到 C 语言写的函数 trap()。

在 kern/trapentry.S 中，主要工作是完成中断处理程序。我们可以看到 JOS 已经提供了连个宏 TRAPHANDLER 和 TRAPHANDLER_NOEC。这两个宏的参数为 name 和 num，功能都是自动生成一段代码，起始部分的指针定义为一个名称为 name 的函数，num 则是这个程序所对应的中断向量号。使用 TRAPHANDLER 的中断硬件会自动向栈中压入 error code，使用 TRAPHANDLER_NOEC 则不会。所以在后者中首先会向栈压入 0 以保证栈结构统一，随后两种函数都会压入 trapno，最后所有函数都会跳转到 _alltraps，继续完成 trapframe 剩余的部分，其中有一段代码如下：

```
movw $(GD_KD), %ax
movw %ax, %ds
movw %ax, %es
```

这段代码的作用是切换 ds 和 es 寄存器，使之指向内核的数据段。但是指令集限制我们不能直接向 ds 和 es 加载数据，因此我们使用了 ax 寄存器作为中介。

接下来的工作在 kern/trap.c 中，我们需要完成初始化 IDT 的工作，即完成 trap_init() 函数，由于 trap_init_percpu() 函数已经自动帮我们加载 IDT，我们只需要设置好 IDT 的内容就行了。在这个过程中，我们需要使用 SETGATE 宏，其定义如下。

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

第一个参数 gate 是中断向量，istrap 则为了区分这个中断时 interrupt 还是 trap，它们的区别是 interrupt 处理时会关中断，而 trap 则不会，sel 则是中断处理程序的段选择子，我们所写的中断处理程序的都因该是 GD_KT，off 则是段内偏移，可以直接填入我们所设置的中断处理程序的名字，dpl 则是权限位，x86 中用户态为 3，内核态为 0，所以允许用户态进入的中断应当设置为至少是 3。另外由于我们的中断处理程序都是在 trapentry.S 中实现的，因此我们需要在 trap.c 中声明这些函数。

这些工作完成后，已经可以通过 Part A 的测试 divzero, softint 和 badsegment。

```
divzero: OK (1.3s)
softint: OK (1.3s)
badsegment: OK (1.1s)
Part A score: 30/30
```

Question1

目前为每个中断设置不同的中断处理程序可以实现的功能是根据不同的中断所自动保存在栈内的信息不同，差异化地补充内容并且统一构建 Trapframe，这个过程中需要获取不保存 Error Code 的中断的编号，如果所有的中断都用同一

段程序则非常难实现这个功能。

Question2

这个问题和我在下一个 Exercise 6 遇到的问题很像(我首先完成了 Exercise)，而他们产生的原因也一样。由于 Page Fault 的中断描述符的 DPL 位为 0，所以用户态不能进入这个 Gate，所以会产生一个 General Protection Exception。

经过修改 DPL 位为 3，运行 user/softint.c，结果如下。这种情况导致成功进入 Page Fault 的中断处理程序，但是出错地址目前为 0，证明没有被设置，所以中断处理程序无法处理这个中断。当然目前所有用户态造成的 Page Fault 都会造成进程被回收，所以结果都是一样的。

```
[00000000] new env 00001000
Incoming TRAP frame at 0xeffffbc
Incoming TRAP frame at 0xeffffc0
[00001000] user fault va 00000000 ip 0000001b
TRAP frame at 0xeffffc0
edi 0x00000000
esi 0x00000000
ebp 0xeabfd0
oesp 0xeffffe0
ebx 0x00000000
edx 0x00000000
ecx 0x00000000
eax 0xec00000
es 0x---0023
ds 0x---0023
trap 0x0000000e Page Fault
cr2 0x00000000
err 0x00800038 [kernel, read, not-present]
eip 0x0000001b
cs 0x---0046
flag 0xeabfd0
esp 0x00000023
ss 0x---ff53
[00001000] free env 00001000
Destroyed the only environment - nothing more to do!
Welcome to the JOS kernel monitor!
```

第二部分

Exercise5

在 trap() 函数中我们可以看到首先如果中断发生在用户态，需要把 trapframe 保存在 Env 里，随后调用 trap_dispatch()，这个函数的功能是根据 trapframe 的 trapno，把不同的中断分发到不同的处理程序中去。这个 EXERCISE 的工作就是在这个函数中增加对 Page Fault 的处理，将其分发到 page_fault_handler() 函数。这个改动非常简单，就不赘述了。

查看 page_fault_handler() 函数，发现目前所实现的功能就是打印 trapframe 后回收 Env，进而结束进程。

Exercise6

这个 EXERCISE 所要实现的功能和上一个非常相似，修改 trap_dispatch() 函数，分发 Breakpoint Exception 到 monitor()，从而实现一个简易的 Debug 功能。

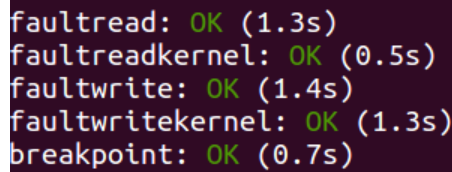
但是完成后无法通过 Test，经过调试查到测试所产生的错误号是 13，并不是 Breakpoint Exception 所对应的 3。查询这个编号对应的中断为：

Interrupt 13—General Protection Exception (#GP)

查看手册发现产生这个中断的原因是违反了中断的保护机制。所以问题就很明显了，由于之前设置 IDT 是 dpl 位设置都是 0，导致所有的中断都只能从内核态触发，因此需要进行修改：

```
SETGATE(idt[2], 0, GD_KT, h_nmiinterrupt, 0);
SETGATE(idt[3], 0, GD_KT, h_brkpoint, 3);
SETGATE(idt[4], 0, GD_KT, h_overflow, 3);
SETGATE(idt[5], 0, GD_KT, h_bdcheck, 3);
```

经过修改后终于可以通过测试了。



```
faultread: OK (1.3s)
faultreadkernel: OK (0.5s)
faultwrite: OK (1.4s)
faultwritekernel: OK (1.3s)
breakpoint: OK (0.7s)
```

Question3

这和我遇到的问题简直一模一样。所以解决方法在上一个 Exercise 已经给出，这里就不赘述了。

Question4

显然这个机制能够防止用户程序恶意造成中断进入内核。

Exercise7

这个 Exercise 的任务是增加 System Call 功能。完成过程可以分成两部分，第一部分是中断处理的部分，第二部分则是具体处理系统调用的部分。

首先第一部分和之前设置 0 到 31 号中断的过程是基本一致的。系统调用的中断向量号是 0x30 即 48，首先在 kern/trapentry.S 中设置中断处理程序，这个程序应当使用的宏是 TRAPHANDLER_NOEC。随后在 kern/trap.c 的 trap_init() 函数使用 SETGATE 宏设置第 48 号中断向量，因为需要用户态能够发出这个中断，所以权限位是 3。随后修改 trap_dispatch() 函数，将 T_SYSCALL 的中断分发到新定义的函数 syscall_handler()，使用这个函数将 Trapframe 的内容对应地填入 syscall() 函数的参数中。

下面进入系统调用处理过程。首先需要了解 kern/syscall.c 和 lib/syscall.c 的区别。前者是在内核处理系统调用的部分，而后者则是用户使用系统调用的部分。前者的执行在中断之后，后者的执行在中断之前。我们需要修改的部分是前者，但是需要参考后者。

在 kern/syscall.c 中，已经给出了处理系统调用所需要的一些函数，目前所需要做的就是函数 syscall() 中根据系统调用号将其分发到各自的函数。由于分发需要知道参数是如何填写的，我们需要查看 lib/syscall.c，这里面有各种用户能够使用的系统调用包装的函数。其中这里的 syscall() 的功能是将系统调用号填入 eax，随后将其他参数对应地填入 edx, ecx, ebx, edi, esi 最多五个寄存器中，

然后执行 `int $0x30`。因此我们可以从如下的函数找到所对应的参数是如何填写的。

最后需要设置返回时的结果到 `Trapframe` 的 `eax` 里，以实现系统调用的返回。

```
void
sys_cputs(const char *s, size_t len)
{
    syscall(SYS_cputs, 0, (uint32_t)s, len, 0, 0, 0);
}

int
sys_cgetc(void)
{
    return syscall(SYS_cgetc, 0, 0, 0, 0, 0, 0);
}

int
sys_env_destroy(envid_t envid)
{
    return syscall(SYS_env_destroy, 1, envid, 0, 0, 0, 0);
}

envid_t
sys_getenvid(void)
{
    return syscall(SYS_getenvid, 0, 0, 0, 0, 0, 0);
}
```

根据如上的信息，我们填写好 `kern/syscall.c` 的 `syscall()` 函数后运行 `testbss`，发现产生如下中断：

```
TRAP frame at 0xeffffe5c
edi  0x0000000e
esi  0x00000000
ebp  0xeffffee0
oesp 0xeffffe7c
ebx  0x00000000
edx  0x00000000
ecx  0x00000000
eax  0xf01068c3
es   0x----0010
ds   0x----0010
trap 0x0000000e Page Fault
cr2  0x0000000d
err  0x00000000 [kernel, read, not-present]
eip  0xf0104923
cs   0x----0008
flag 0x00000002
```

显然产生的 `Page Fault` 并不正常，运行 `make grade` 后有如下问题：

```
testbss: FAIL (1.2s)
...
    cs  0x----0008
    flag 0x00000012
GOOD [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.
qemu: terminating on signal 15 from pid 10348
MISSING 'Making sure bss works right...'
MISSING 'Yes, good. Now doing a wild write off the end...'
MISSING '.00001000. user fault va 00c..... ip 008.....'
QEMU output saved to jos.out.testbss
```

似乎用户程序所使用的所有 `cprintf` 都没有打印出来。经过 `Debug` 后，发现原因是 `kern/syscall.c` 的 `syscall()` 函数在分发时把 `a1`、`a2` 写成了 `a2`、`a3`。造成错误的原因是在 `kern/syscall.c` 和 `lib/syscall.c` 所使用的 `syscall()` 的参数位置不一样，因此造成错误。

```
int32_t
syscall(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t a3, uint32_t a4, uint32_t a5)

static inline int32_t
syscall(int num, int check, uint32_t a1, uint32_t a2, uint32_t a3, uint32_t a4, uint32_t a5)
```

修改后可以通过 testbss 测试。

```
testbss: OK (1.4s)
(Old jos.out.testbss failure log removed)
```

Exercise8

这一个 Exercise 的任务是完成用户程序的 Set Up 部分，由于其实大部分工作已经完成了，这一部分实际上只需要添加让用户程序能够获取目前的 Env 的功能。

用户程序的 Set Up 过程是从 lib/entry.S 开始的。首先设置了 envs, pages, uvpt, uvpd 等变量，然后判断是否有传入参数，从内核开始运行的程序是没有参数的，因此此处需要设置参数为空。随后调用 lib/libmain.c 的 libmain() 函数，这个函数的功能是设置 thisenv 和 binaryname 使用户程序能够获得当前的 Env 和程序名。其中 binaryname 已经设置完成，我们下一部设置 thisenv。

由于我们刚才已经配置好了系统调用，因此这一步中我们只需要使用 sys_getenvid() 来获得当前的环境的 envid，然后使用宏 ENVX，可以从 envid 中获得 envs 的偏移量，然后就能得到当前 Env 的地址了。

随后 libmain() 调用了 umain()，即用户程序的入口，完成后调用 exit() 销毁当前 Env，退出程序。

```
[00000000] new env 00001000
Incoming TRAP frame at 0xefffffb
Incoming TRAP frame at 0xefffffb
hello, world
Incoming TRAP frame at 0xefffffb
i am environment 00001000
Incoming TRAP frame at 0xefffffb
[00001000] exiting gracefully
[00001000] free env 00001000
Destroyed the only environment - nothing more to do!
Welcome to the JOS kernel monitor!
```

上图是运行 make run-hello 的结果，可以看出整个程序一共使用了 4 个 Trap。第一个 trap 为 sys_getenvid() 函数请求 envid，第二、三个为 sys_cputs() 打印字符串，第四个为退出时 sys_env_destroy()。

Exercise9

以下两个 Exercise 的主要工作是处理 Page Fault 和完善内存保护。在程序运行时，可能会访问一个非法的内存地址，原因可能是权限不够或者地址无效。发生这种情况后会产生一个 Page Fault 并且进入一个中断处理过程，如果问题可以被修复则可以处理后让程序继续运行，否则不应该返回原程序。

在用户态和内核态下产生的 Page Fault 应当区别对待，因为如果是内核自己的操作导致了中断，证明内核有 Bug。如果是用户态程序则可能是一些误操作，直接结束程序即可。

但是在内核态下有一个例外，就是在系统调用中用户可以通过 Trap 向内核发送指针以供内核使用，如果这个指针是非法的，则会造成在内核态下发生 Page Fault，但是这并不意味着内核的 Bug，我们还是应当把这种情况当作用户程序发生的问题，进而结束程序即可。这就需要在系统调用中对用户传递的地址进行检查，如果是非法则应当进行处理。

所以这个 Exercise 所进行的具体操作就是完成以上的工作。首先需要修改 kern/trap.c 的 Page Fault 处理程序 page_fault_handler(), 使其在内核态发生错误时让系统进入 panic。在这里不处理系统调用造成的错误的原因是我们一会儿需要修改系统调用, 使其在使用内存地址之前检查权限, 所以不会造成缺页异常。

值得一提的是判断中断前状态的方法, 查询 IA32 手册发现, CS 和 SS 寄存器的后两位称为 CPL 位(Current Privilege Level), 如果是内核态应当为 0, 用户态则应当为 3, 所以通过查看 Trapframe 的 CS 寄存器就可以判断出当前的状态。

```
if ((tf->tf_cs & 3) == 0)
    panic("kernel fault va %08x ip %08x\n", fault_va, tf->tf_eip);
```

接下来我们就需要修改系统调用的部分了, 现在实现的系统调用有 4 种: SYS_cputs, SYS_cgetc, SYS_getenvid, SYS_env_destroy。其中只有 SYS_cputs 需要用户传入内存地址, 因此只修改这个部分即可。

在这里 JOS 已经提供了 kern/pmap.c 的 user_mem_assert() 函数来进行权限检查的工作。但是在这个函数中调用了 user_mem_check() 来进行检查, 这个函数需要我们自己实现。

这个函数中一个比较重要的一点是检查哪些地址。很明显这个权限应当是按页检查的, 所以每页检查一次即可, 但是要检查多少页呢? JOS 的注释认为可能检查 len/PGSIZE 或 len/PGSIZE+1 或 len/PGSIZE+2 页。但是我认为这样过于复杂, 我们其实只需要知道这段地址最后一页的末尾的位置就可以了, 然后之前每一页检查一次即可。使用 ROUNDUP 宏我们可以很轻易地得到这个位置。

```
const void *lim = ROUNDUP(va+len, PGSIZE);
```

然后就是如何检查的问题。在这里我们需要检查三个部分, 第一个是是否有页表项, 即 PTE 是否是空, 如果是空显然这个地址是无效的。第二则是检查权限位, 除了参数给定的 perm 之外, 我们还应该检查 PTE_P, 以确保页表项有效。最后则是应该检查地址是否高于 ULIM, 如果是则这个位置显然不是用户应该访问的。

```
if(pte == NULL
    || (*pte & (perm|PTE_P)) != (perm|PTE_P)
    || p >= (const void *)ULIM) {
```

最后应该把第一个出错的地址存入 user_mem_check_addr, 以便反馈在控制台上。

user_mem_assert() 函数完成后我们就可以在系统调用中使用来检查用户所给地址是否合法。以下是运行 make run-buggyhello 的结果。可以看出系统调用被传入了一个非法地址, 第一个出错的位置是 0x00000001, 查看 inc/memlayout.h 后发现这个位置应当是 Empty, 没有页被分配到这个位置, 因此这个地址是非法的。

```
[00000000] new env 00001000
Incoming TRAP frame at 0xfffffbc
Incoming TRAP frame at 0xfffffbc
[00001000] user_mem_check assertion failure for va 00000001
[00001000] free env 00001000
Destroyed the only environment - nothing more to do!
Welcome to the JOS kernel monitor!
```

随后使用 backtrace 指令查看调用栈, 可以看出 user_mem_assert() 检查出错后调用了 env_destroy(), 最后调用 monitor() 进入控制台。

```

K> backtrace
Stack backtrace:
  ebp effffffe80 eip f0100a24 args 00000001 effffffe98 00000000 f01a1000 f017e9a0
    kern/monitor.c:181: monitor+275
  ebp effffffef0 eip f01038c0 args 00000000 effffff1c 00000010 f01a1000 f01a1000
    kern/env.c:461: env_destroy+41
  ebp effffff10 eip f0103206 args f01a1000 00001000 00000001 00000004 00001000
    kern/pmap.c:642: user_mem_assert+82
  ebp effffff30 eip f01042af args f01a1000 00000001 00000001 00000000 f0103984
    kern/syscall.c:27: syscall+79
  ebp effffff60 eip f010408e args 00000000 00000001 00000001 00000000 00000000
    kern/trap.c:283: syscall_handler+55
  ebp effffff90 eip f010415e args f01a1000 effffffbc f0180000 f0123c5c 00000000
    kern/trap.c:201: trap+199
  ebp effffffb0 eip f0104259 args effffffbc 00000000 00000000 eebfdb0 effffffdc
    kern/trapentry.S:84: <unknown>+0
Incoming TRAP frame at 0xeffffffd7c
kernel panic at kern/trap.c:268: kernel fault va eebfdbb4 ip f010077a

```

Exercise10

这个 Exercise 是为了测试之前所做的保护能否抵御用户使用系统调用蓄意攻击的情况，执行 make run-evilhello 后控制台结果如下：

```

[00000000] new env 00001000
Incoming TRAP frame at 0xeffffffbc
Incoming TRAP frame at 0xeffffffbc
[00001000] user_mem_check assertion failure for va f010000c
[00001000] free env 00001000
Destroyed the only environment - nothing more to do!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> 

```

第一个出错的位置是 0xf010000c，通过查看 inc/memlayout.h 发现这里的是 KERNBASE 以上，用户应当不能访问。

到现在为止，JOS 已经能够通过 lab3 的所有测试。

```

divzero: OK (1.2s)
softint: OK (1.2s)
badsegment: OK (0.7s)
Part A score: 30/30

faultread: OK (1.4s)
faultreadkernel: OK (1.4s)
faultwrite: OK (0.8s)
faultwritekernel: OK (1.1s)
breakpoint: OK (1.0s)
testbss: OK (1.2s)
hello: OK (1.5s)
buggyhello: OK (0.8s)
buggyhello2: OK (1.8s)
evilhello: OK (1.6s)
Part B score: 50/50

Score: 80/80

```


Challenge2

这个 Challenge 需要我们在 Kernel Monitor 中加入 continue 指令，以便从 Breakpoint Exception 进入 Monitor 后可以实现继续调试功能，进而可以实现单步调试功能。

首先完成 continue 功能。增加 Monitor 指令的步骤和之前相同，在 kern/monitor.c 的 commands 数组增加一条指令 continue，并且完成函数 mon_continue()。有一种恢复方法是直接将 Trapframe 覆盖当前的 CPU 状态。但是这样虽然可能会成功继续运行，但是却清空内核栈的内容。所以不能直接使用这个方法。其实 JOS 已经提供了相关的接口，env_pop_tf() 函数首先就是将栈指针指向 Trapframe，随后再将手动保存寄存器的值恢复，最后使用 iret 返回用户程序。所以直接调用这个函数即可。另外需要注意的是需要查看 Trapframe 是否为 NULL，因为非中断状态下也可能是用这个指令，这时应当报错。

运行 user/breakpoint.c 测试结果如下：

```
TRAP frame at 0xf01a1000
edi 0x00000000
esi 0x00000000
ebp 0xeebdfd0
oesp 0xeffffdc
ebx 0x00000000
edx 0x00000000
ecx 0x00000000
eax 0xec00000
es 0x---0023
ds 0x---0023
trap 0x00000003 Breakpoint
err 0x00000000
eip 0x00800037
cs 0x---001b
flag 0x00000046
esp 0xeebdfd0
ss 0x---0023
K> continue
Incoming TRAP frame at 0xeffffbc
[00001000] exiting gracefully
[00001000] free env 00001000
```

下面就是实现单步调试的功能。如果 EFLAGS 的 TF 位为 1，会发生单步中断，中断号为 1，因此我们需要在 trap_dispatch() 函数把 1 号也分发到 monitor()。接下来我们需要再在 Monitor 里增加单步调试 si 指令，过程就不赘述了。而且在 si 中我们还需要打开 TF 位，在 continue 里关闭 TF 位。

为了测试需要修改 user/breakpoint.c:

```
asm volatile("int $3");
asm volatile("movl $8, %eax");
```

接下来测试中断后进入 monitor，输入 si 结果如下，证明已经成功单步调试：

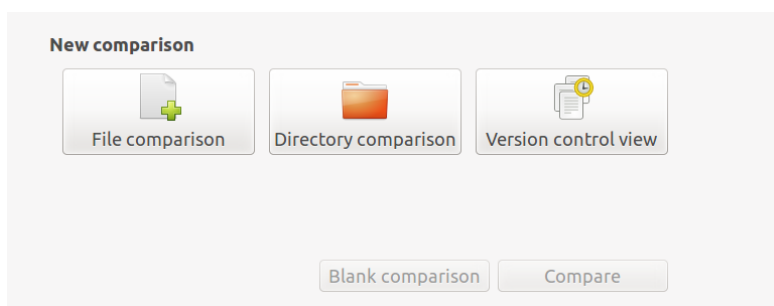
```
K> si
Single Step
Incoming TRAP frame at 0xeffffbc
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
TRAP frame at 0xf01a1000
edi 0x00000000
esi 0x00000000
ebp 0xeebdfd0
oesp 0xeffffdc
ebx 0x00000000
edx 0x00000000
ecx 0x00000000
eax 0x00000008
es 0x---0023
ds 0x---0023
trap 0x00000001 Debug
err 0x00000000
eip 0x0080003c
cs 0x---001b
flag 0x00000146
esp 0xeebdfd0
ss 0x---0023
K> 
```

内容三：遇到的困难以及解决方法

困难 1 Merge Conflict

按照 Write Up 所给的示范,在切换到 lab3 的 branch 后需要与 lab2 进行 merge。但是 merge 发现 kern/kdebug.c 和 kern/monitor.c 两个文件发生了冲突。因此需要 merge tool 来帮助我来 merge 这些文件。

解决方法:首先尝试使用 git 自带的工具,发现没有 GUI,非常难以使用,因此放弃。在查询资料后,许多博客都推荐了工具 meld 作为 merge conflict 的解决方法,尝试后果然非常好用,使用这个软件解决了问题。这个软件同样也是 diff 操作很好用的工具,因此强烈推荐。



内容四：收获及感想

之前在课上我提过一个问题就是直接在程序中执行非系统调用的 INT 指令会造成什么后果。我在这个 Lab 中在 JOS 上,测试了这个做法:

```
void
umain(int argc, char **argv)
{
    //zero = 0;
    //cprintf("1/0 is %08x!\n", 1/zero);
    asm volatile("int $0\n" ::: "memory");
}
```

我修改了 divzero 程序,直接触发 INT 0h。然后通过执行 make run-divzero 查看结果。发现成功进入了中断处理的部分,产生的 trapframe 如下:

```

Incoming TRAP frame at 0xefffffb
TRAP frame at 0xf01a1000
edi 0x00000000
esi 0x00000000
ebp 0xeebdfdf0
oesp 0xefffffdc
ebx 0x00000000
edx 0x00000000
ecx 0x00000000
eax 0x00000000
es 0x----0023
ds 0x----0023
trap 0x0000000d General Protection
err 0x00000002
eip 0x00800036
cs 0x----001b
flag 0x00000046
esp 0xeebdfdf0
ss 0x----0023

```

可以看出这里触发的中断并不是 DIVZERO(0 号)而是 13 号中断 General Protection，这和刚才 Exercise 6 中的 BUG 很相似，由于 divzero 的 dpl 设置为 0，因此用户无法触发这个中断。

那么我们进一步开放这个中断的权限，再次进行测试：

```
SETGATE(idt[0], 0, GD_KT, h_divzero, 3);
```

这时的结果如下

```

TRAP frame at 0xf01a1000
edi 0x00000000
esi 0x00000000
ebp 0xeebdfdf0
oesp 0xefffffdc
ebx 0x00000000
edx 0x00000000
ecx 0x00000000
eax 0x00000000
es 0x----0023
ds 0x----0023
trap 0x00000000 Divide error
err 0x00000000
eip 0x00800038
cs 0x----001b
flag 0x00000046
esp 0xeebdfdf0
ss 0x----0023

```

表明在 JOS 中，使用 INT 指令可以触发任意的中断，但是可能由于权限限制无法进入相应的中断处理程序。

内容五：对课程的意见和建议

对于在系统调用 SYS_cputs 增加检查内存权限的方式我认为可以改进。目前只实现了 4 个系统调用，只有一个系统调用可能实际内存地址的传递，所以只在

这里检查内存权限目前是可行的。但是如果要增加新的系统调用则需要每个需要引用用户传递的指针的调用中进行检查过于繁琐，应当对这些调用设立统一的调度接口，并且在这个接口中进行检查。

内容六：参考文献

IA-32 手册（共 646 页）Vol.3 5-29 中断类型

IA-32 手册（共 646 页）Vol.3 4-10 Privilege Levels

80386 Programmer's Manual Chapter 9