JOS 设计文档（lab3范围内）
丁卓成 5120379064

一、概述

本文档为Lab1,Lab2提交的设计文档之续，继续分析JOS的架构，本次Lab的主要内容是实现第一个用户态程序，提供异常处理和系统调用。

二、预备知识

在x86架构的CPU上，为了实现操作系统的基本功能如异常处理和系统调用，必须使用Intel提供的机制，且在课上并未详细讲授这些内容，因此本节首先介绍这些会用到的机制。

首先，此前在Lab2中已经实现了页式内存管理，这也是一种通用的做法，在各种架构的CPU上都会采用，这为实现用户态程序提供了一个基础。而在x86架构中，还有段式内存管理的概念，这只是为了兼容旧版本的CPU而保留的机制，当x86_64架构的CPU运行在IA-32e模式即64位模式下时，其段式内存管理被简化，cs、ds、es、ss所选择的段，其base和limit都会被忽略，硬件会认为这些段从0开始，且从不做limit的检查（因此段的大小可以任意大），使Logical Address和Linear Address等同了起来，但保留了fs、gs的段式内存管理功能，以供操作系统作特殊用途（一般用于存放每个CPU的信息或每个thread的信息）。

在段式内存管理的概念中，Linear Address被分为一个个段，有两大类，一类是code/data segment，一类是system segment，用于描述段的信息的段描述符存放于GDT（Global Descriptor Table）、LDT（Local Descriptor Table）或IDT（Interrupt Descriptor Table）中，由段寄存器中存放的段选择器指出其在表内的偏移，由此可以找到相应的段描述符。段式内存管理提供了权限机制，称为Privilege Level，在段描述符中有一项DPL（Descriptor Privilege Level），CS寄存器的末两位称为CPL（Current Privilege Level），其他段寄存器的末两位称为RPL（Requested Privilege Level），数值从0到3，0级权限最高，3级最低，常称为ring0-ring3。在页式内存管理中只有Supervisor和User的区别，是将ring3视为User，ring0-ring2都视为Supervisor。

对于code/data segment的一些属性，在JOS中并未使用，故不做介绍，唯一需要知道的就是，Intel规定CS中的CPL和SS中的RPL以及SS对应的data segment的DPL必须一致，因为程序运行至少需要代码和栈，这两者的权限必须保持一致，否则会引发12号异常#SS（Stack Fault）。因此，不可能通过一条载入ss寄存器的指令加上一条载入cs寄存器的指令实现Privilege Level的转换。

对于system segment，最简单的一种是LDT descriptor，它只允许出现在GDT中，用于指明一块用于存放LDT的地址空间，在JOS中我们不使用LDT，在初始化时为LDTR加载了0值。还有一种是TSS（Task-state segment） descriptor，用于指明一块用于存放TSS的地址空间。这就引出GDTR、LDTR、IDTR（分别用于保存GDT、LDT、IDT基址，其中LDTR放的是LDT selector）以外的第四个寄存器，即Task Register，用于保存TSS selector。顾名思义，Task Register和TSS是Intel从硬件层面上提供的任务管理机制，可以用于实现操作系统的进程（尽管至少到目前这个Lab为止JOS没有使用，作为JOS原型的xv6是使用了的）。Task Register指向当前的Task的TSS，其中保存了从ring0到ring3的四个栈（SS及ESP），CR3寄存器（存放Page Directory基址），各个段寄存器和通用寄存器（包括eflags），LDT选择子以及Previous Task Link。除这两种以外，还有四种system segment，它们都属于一类，称为Gate，分别是Call Gate、Interrupt Gate、Trap Gate、Task Gate，用于Privilege Level的转移。

接下来介绍这些system segment的具体作用，这要从我们最熟悉的jmp指令和call、ret指令说起。平时常见的jmp、call、ret指令，其更准确的说法称为near jmp、near call、near ret，与其相对还有far jmp等对应的指令，时至今日仍能在一些介绍8086微机的过时的教科书上看到这种概念，包括近指针、远指针等，往往令平时只接触过用户态编程的学生感到困惑。其实，这里的near和far是针对segment而言的，在同一个code segment内的跳转，就称之为near，跳转到不同的code segment的，就称之为far（进行far call时会在栈上压入原本CS的值以保证能返回），同理指向同一个data segment内的就称为near pointer，反之则为far pointer。这些概念在8086的时代很常用，现在则基本不使用，因为这种far的跳转，只能在相同Privilege Level的code segment之间往返，而无法前往不同Privilege Level的code segment（有一种称为Confirming Code Segment的例外，不过我们不关心这些细节）。

为了在不同Privilege Level往返，需要借助Gate的概念。最简单的是Call Gate，只能位于GDT和LDT中，在进行far跳转时将目标段选择子设置为Call Gate相应的选择子即可通过Call Gate进行跨Privilege Level的跳转。Call Gate的Descriptor内存放了目标code segment的选择子以及入口点的offset，跳转指令提供的offset会被忽略。far jmp指令即使通过Gate也无法进入其他Privilege Level，以下不讨论，far call可以通过Call Gate进入更高的Privilege Level（数值更小），并通过far ret返回，只要满足caller的CPL和Call Gate的RPL（即far call指令中指定的selector的RPL）均小于等于Call Gate的DPL即可。跳转后CPL会变为目标code segment的相应级别，因此跳转时栈也需要同时进行变换，换栈时，就从TSS中查找对应的Privilege Level的SS和ESP载入寄存器中。

介绍了Call Gate的概念，另外三种Gate也就很简单了，它们的概念都是类似的。Interrupt Gate和Trap Gate都只能存放于IDT中，它们在一个中断发生时，自动被调用，因此没有RPL的概念，假如CPL<=DPL就能成功被调用，要返回时使用IRET指令返回。它们与Call Gate的区别是，Call Gate只会压栈SS、ESP、CS、EIP，而这两者还会压栈EFLAGS，并且根据中断号的不同，有些中断还会压入Error Code（例如Page Fault）。它们之间的区别在于，通过Interrupt Gate进入Handler以后eflags的IF会被清零，即禁止了外部中断，而Trap Gate则不会，除此之外两者没有区别。Task Gate是用于不同Task之间的切换，在GDT、LDT、IDT中均可存在，切换时当前的状态保存于当前TSS内，随后Task Register切换到目标Task的TSS，读取该TSS内的内容，将它们加载到对应的寄存器内，并设置Previous Task Link为切换前的Task的TSS Selector，以便此后可以通过IRET指令返回原来的Task。为了区分IRET指令是从Interrupt Gate、Trap Gate中返回还是返回上一个Task，在Task切换时会设置EFLAGS中的NT（Nested Task）位，若该位为1则IRET指令会进行Task切换，否则不会。

可以看到从Call Gate到Task Gate一个比一个复杂，而这些复杂的Stack switch乃至Task switch都只需一条far call指令或一条iret指令即可完成，是x86架构CISC特征的典型体现。

三、用户态程序的加载和执行

这是较为简单的一部分，Lab 3中已经为我们准备好了大部分代码，为了运行起一个用户态程序我们只需为其设置页表、加载可执行文件到内存，最后利用iret指令返回用户态即可。

具体而言，由于我们现在并没有文件系统，我们的用户态程序是通过binary的方式链接到kernel的镜像文件上的，这些程序都位于kernel文件的.data节，我们可以通过Lab 3中已经为我们写好的宏来选择一个用户态程序进行加载。Lab 3中提供的宏最终调用了env_create函数，在这个函数中我们调用load_icode函数，将用户程序的内容加载到内存中并设置好这些内容对应的页表，该过程类似于加载kernel镜像的过程，不再详述。加载完毕后，调用env_run函数即可运行该用户态程序。这是通过iret指令实现的，也就是说，尽管没有对应的异常，我们仍可以使用iret指令，只要将栈安排成和通过Interrupt Gate或Trap Gate进入的时候产生的栈一样即可，这是通过struct Trapframe实现的。这个结构体按照产生异常时栈的布局，设置了其成员的类型及顺序，而每个用于表示用户态进程的struct Env中都包含一个struct Trapframe作为成员。这样，只要在初始化的时候（env_init和env_alloc中）设置好相应的初始值，并在加载ELF文件时设置好其入口点，就可以顺利通过几条pop指令和一条iret指令顺利运行一个用户态程序。

四、系统调用的实现

尽管经过以上步骤，一个用户态程序已经可以运行，但其甚至连hello world也无法实现，因为输入输出必须经过内核操作相应的硬件才能实现，因此在用户态势必通过系统调用才能实现。实现系统调用有两种方式，一是采用int $SYSCALL_NUM的形式，选择一个中断号供系统调用使用，另一种是使用专门用于系统调用的指令，如sysenter、sysexit指令，本次Lab规定必须使用sysenter、sysexit指令实现。这两条指令需要操作MSR进行初始化，所谓MSR即Model Specific Register，在不同型号的CPU上可能不同，这次需要设置的是0x174-0x176（MSR寄存器有很多，形成了自己的地址空间）三个寄存器。根据最新的Intel文档，0x175号寄存器用于存放EIP，0x176号寄存器用于存放ESP，而Lab中提供的阅读材料（Linus Torvalds在2002年为Linux打的补丁）显示0x175号用于存放ESP而0x176号用于存放EIP。到底何者正确呢，事实上两者都正确，因为MSR本来就是为了不兼容而诞生的，是Model Specific的，相隔十来年其定义发生了变化也属正常。经过测试，可能由于在我们的Lab中使用的是极早期的QEMU版本，后者的顺序才是正确的，但这并不能代表在其他模拟器或真实的机器上一定也是这样定义的。

通过在kern/trap.c中添加了三条wrmsr指令初始化后，sysenter、sysexit指令即可使用了。我们的用户态程序进行系统调用后，最后都会来到lib/syscall.c中的syscall函数，我们需要在此处写内联汇编调用sysenter指令。注意sysenter指令并不会保存调用者的eip和esp，因此需要手动保存，占用两个寄存器，此外我们还要支持一个系统调用号及五个参数，需要占用6个寄存器进行传参，这样一来寄存器就不够用了（因esp会由于sysenter指令而被覆盖）。为了解决这个问题，我的解决方案是只采用一个寄存器保存返回地址和调用者的esp，方法是使用这一个寄存器（我采用的是ebp）保存调用者的esp，并且在调用者的栈上压入其返回地址，这样可以通过保存在ebp中的esp间接地获取返回地址，由此即可顺利返回。

执行sysenter指令后就来到了kern/trapentry.S中的sysenter_handler（该入口点在此前初始化时指定），它将通过寄存器传来的参数压栈，并设置es、ds，然后就调用syscall函数（位于kern/syscall.c），执行具体的system call。待函数返回后，由于函数调用规约，ebp的值并未受到影响，我们可以恢复es、ds，并从ebp中获得返回地址和要切换到的esp，载入到对应的寄存器（edx和ecx），最后通过一条sysexit指令即可返回用户态。

只要小心地编写汇编和内联汇编，保证以上部分正常工作，系统调用就算是完成了，剩下的工作不涉及用户态和内核态的切换，可以使用C编写，我们只需要关心具体的业务逻辑即可。至此，hello程序已经可以输出hello world了，但仍会在第二次cprintf调用时出错，因为thisenv变量没有设置（这也是Exercise 2完成后运行hello会出错的原因，而不是Lab介绍中写的因为调用了系统调用而出错，因为此时syscall函数中甚至还没有写上sysenter指令），解决方法是在lib/libmain.c中加上一句初始化语句即可（这句语句也需要调用系统调用）。

需要我们实现的的系统调用只有sys_cputs和sys_sbrk，其他均已经为我们实现。

sys_cputs函数默认已经包含一句cprintf，因此已经可以正常工作，此前hello程序已经能正常输出hello world。然而，作为一个系统调用，除了实现用户程序要求的功能，还必须采取不信任用户程序的态度，检查其提供的输入，防止有bug的或恶意的用户程序对内核造成破坏。因此，我们需要回到Lab 2中的kern/pmap.c中，添加一个user_mem_check函数，检查用户是否有权限使用它提供的地址（可能内核具有权限使用该地址，但用户没有）。在sys_cputs中我们利用这个新添加的函数，检查用户的输入是否合法，如果不合法则销毁用户进程，避免其破坏内核的一致性。

sys_sbrk函数可以扩展用户进程的data段，为其提供一种分配内存的基本机制，使用Lab 2中提供的接口即可实现。有了这个函数，我们就可以说用户程序已经具备了堆，这样我们的用户程序已经和通常意义上的用户程序较为接近了，除了不具备动态链接库。事实上，在POSIX兼容的系统上的确有sbrk系统调用，在UNIX和Linux上malloc库的早期实现确实是使用的sbrk系统调用，后来则采用了mmap系统调用作为获取内存的方式，有些库提供了选项可以在两种底层实现中选择一种进行编译。

五、中断（异常）处理的实现

对于中断的处理，需要在kern/trap.c中的trap_init函数中通过SETGATE宏设置IDT表项。尽管原则上这个Lab中处理的属于“Trap”，应该使用Trap Gate，但我使用了Interrupt Gate，这是因为本次Lab中提供的中断处理函数trap要检查中断是否已经关闭（即eflags的IF位为零），为了适应这一点，不能使用Trap Gate。具体的Trap Gate对应的入口点在kern/trapentry.S中定义，它们在压入error code（如果其本身没有error code）和trap number后都会跳转到_alltrap，在此处保存用户的段寄存器和通用寄存器，形成和Trapframe定义一致的栈布局，然后调用trap函数，该函数的参数即为struct Trapframe *tf。

在本次Lab中，需要进行处理的仅有Debug Exception、Breakpoint Exception和Page Fault，对于其他异常均使用默认的应对措施，即若引发异常者为用户，销毁该用户进程，若为内核则产生一个panic。对于Page Fault异常，由专门的函数page_fault_handler处理，不过本次Lab并未提出相关的处理要求，因此对于用户的page fault实际上我们总是销毁该进程而不是试图去处理它。对于Breakpoint异常，这是本Lab中唯一需要将其Interrupt Gate的DPL设置为3的异常，事实上根据Intel的文档来看除了Breakpoint异常允许用户通过INT3指令产生，就只有Overflow异常允许用户通过INTO指令产生，其他任何异常都不应该由用户通过int $trapno的形式产生并调用。用户通过该异常可以进入kernel monitor，我们为其增添了三条命令，分别是c、si和x，即继续执行、单步执行和显示内存。其中c和x都很简单，c命令直接从handler中返回即可，x命令需要parse一下输入，获取用户输入的地址，并检查该地址用户进程是否可以读取。对于单步执行，要分两部分实现，当用户输入si命令时，调用第一部分，将Trapframe中eflags的TF位置为1，然后立即返回用户态。由于此时用户的elfags中TF为1，它只会执行一条指令，接着引发一个Debug Excpetion，此时由第二部分处理该异常，输出当前的eip并通过debuginfo_eip函数获取其在用户程序源码中的位置（注意此时是从用户程序的stabs中获取的调试信息，要先用user_mem_check检查其合法性），最后进入和monitor相同的循环中读取用户命令（不能直接调用monitor函数以免重复输出欢迎信息）。

在课程网站上Lab的描述信息中，提出了一个问题但并未要求在answer-lab3.txt中作答。问题是，在通过INT3指令进入monitor后输入backtrace命令会发生崩溃，请问为何崩溃。答案很简单，backtrace试图追溯函数的参数，不管是否存在该参数（因为它无法获知参数数量），它总会试图输出5个参数。当追溯到libmain函数时，由于libmain函数只有两个参数，再往前追溯则会越过用户的栈进入无效的虚拟地址区域，所以backtrace会引发page fault。而且，有趣的是我们是在内核态处理Breakpoint Excepition，从内核栈却可以追溯到用户栈，这是因为在从用户态进入内核态直到调用trap函数的过程中，ebp寄存器的值一直没有改变，因而把用户栈的old ebp的位置保存在了trap函数的栈上，使得我们可以从内核栈一路追溯到用户栈，直到用户程序的入口点libmain函数。

六、其他

除了以上内容，本次Lab还有一个Exercise，要求修改用户程序evilhello2，使其进入ring0调用一个函数evil。实现该攻击的方式很简单，Lab中提供了sys_map_kernel_page系统调用，可以把kernel的page映射到user的地址空间，因此我们将gdt映射到user的地址空间并修改，利用前文介绍的Call Gate的方式即可进入ring0执行某个给定的函数。能够进行这个攻击的关键，主要是在于存在sys_map_kernel_page这样一个很危险的系统调用，把内核受到保护的空间暴露给了用户，自然会被恶意的用户程序乘虚而入。

在实现该攻击的过程中，只有一点需要注意，由于在修改GDT后还要在恢复，我采用了一个临时变量保存GDT原来的值，接着修改了GDT，进行far call以ring0调用给定的函数，最后返回后再把临时变量中的值写回GDT。由于gcc不知道内嵌汇编的语义（这是由汇编器负责的），它认为在整个过程中没有做任何会造成副作用的操作，只是把GDT的内容改了一下没做任何事马上又改回来了，因此会采取优化将修改GDT的那段代码删除，以提高效率。由于gcc错误的优化，我们事实上并没有修改相应的GDT表项，因此会产生一个General Protection异常而不是我们期望的成功调用了函数。为了避免这种情况，我们可以关闭gcc的优化功能，但更好的解决方案是利用

volatile关键字强制gcc进行执行赋值操作。在加入volatile关键字后，evil函数如预期般被成功调用，该Exercise顺利完成。

七、结论

至此，本次Lab中所有Exercise涉及实现的内容均已介绍完毕，本文档到此结束，但目前还无法处理外部中断，也不支持多进程运行和调度，更多后续内容留待接下来Lab的文档继续介绍。