操作系统实验lab3

**PART A**

**exercise one**

修改pamp.c文件，其中主要有两个部分需要增加，即分配空间与进行相应的映射。

其实际过程是与之前通过mem\_init中为pages[]分配内存空间的方法类似，首先需要调用boot\_alloc函数，设定需要分配的内存大小，然后通过memset函数初始化整个分配的内存块中的内容，然后为其分配相应的映射，根据代码中相应的提示，**perm = PTE\_U | PTE\_P**，因此，在调用boot\_map\_region时，给它赋上相应的permission（权限）



**exercise two**

查看kern/init.c中的i386\_init函数，可以看到，针对lab3，其调用了env\_init和trap\_init两个函数，主要是实现本次实验的part A和 part B两个部分，其中part A部分主要是对于用户进程环境的创建。

**env\_init()**

初始化全部envs数组中的Env结构体，并将它们加入到env\_free\_list中。还要调用env\_init\_percpu，这个函数会通过配置段硬件，将其分隔为特权等级 0 (内核) 和特权等级 3（用户）两个不同的段。

其中，env\_free\_list也是一个Env结构体，正如我们之前所看到的那样。 同时可以查看env\_init\_percpu函数，看到此函数实际上是通过内联汇编代码，将不同的内存地址（段）分配给不同的权限。

在此处完成env\_init函数：



可以看到，整个env\_init函数，主要分成了两个部分，一为初始化整个需要分配的Env结构体，二为调用env\_init\_percpu函数完成划分。其中，第一个部分倒序进行的原因是它相当于一个压栈过程，故而需要把最早使用的放在env\_free\_list 的最后，也会最早被调用。

**env\_setup\_vm()**

为新的进程分配一个页目录，并初始化新进程的地址空间对应的内核部分。给一个进程分配页目录，即需要构造一个PageInfo结构体，并初始化其中的内容，即需要将页目录++，并通过地址转换找到分配的页目录对应的地址，并将其拷贝到内核中，形成一个对应的内核进程，并为其赋予相应的权限。



**region\_alloc()**

为进程分配和映射物理内存。 要为进程分配和映射物理内存，需要通过进程的相应结构和虚拟地址找到其需要映射的空间大小。



通过虚拟地址va和提供的进程大小len，利用ROUNDDOWN和ROUNDUP可以相应需要映射的初始地址和结束地址位置，根据相应的地址，调用page\_alloc为其分配空间，然后将分配好的物理页插入到进程e的env\_pgdir中，并设置权限为PTE\_W | PTE\_U。

**load\_icode()**

需要处理 ELF 二进制映像，就像是引导加载程序(boot loader)已经做好的那样，并将映像内容读入新进程的用户地址空间。 这里实际上就是需要将得到的像ELF这样的二进制映像，在这里是binary处理为**ELF**文件，然后根据**ELF**映像进行处理。根据lab1中所提到的**ELF**中的header对于**ELF**文件进行处理。找到相应的ELF开始和结束的位置，在调用region\_alloc函数映射内存前，需要先设置cr3寄存器内容为进程的页目录物理地址，设置完之后再映射回kern\_pgdir的物理地址。

具体的结构体ELF和Proghdr定义在inc/elf.h之中.

最终的结果为：



**env\_create()**

通过调用env\_alloc分配一个新进程，并调用load\_icode读入 ELF 二进制映像。故而，实际上这个函数的实现非常简单，首先根据env\_alloc分配一个新的进程，其中，$id \ge 0$，否则直接panic，并且为其赋上相应的type，最后，调用load\_icode函数加载binary 中的映像。



**env\_run()**

启动给定的在用户模式运行的进程。根据注释，可以看到整个实际过程需要完成的工作就是进程的切换，将当前正在运行的进程切换到就绪态中，然后将一个新的进程切换进行开始运行，并且更新变量env\_runs，最后利用lcr3函数切换地址空间为运行的当前的页目录的物理地址。而step 2 则需要调用env\_pop\_tf来保存相应的进程信息（退出的进程的相应信息）。

最终实现的代码为：



此时，调用make qemu可以看到相应的结果为：



可以看到，这里有一个错误为：Triple fault，如果去看user\hello.c的话，可以看到其代码为：



故实际上，这里调用cprintf函数，实际上会从用户态陷入到内核态，而此过程，需要调用**int $0x30**指令，但显然我们还没有处理这些东西。用另一种方式来检测现在是否正确： 使用make qemu-gdb并在env\_pop\_tf处设置一个 GDB 断点，这应当是真正进入用户模式前所执行的最后一个内核函数。 用si指令在这个函数中步进。处理器应该在一个iret指令后进入用户模式。 接下来，你应该能够看见用户进程的可执行代码的第一个指令：在lib/entry.S中start标签的cmpl指令。具体执行的结果为：



**exercise three**

在本次实验中，我们大体上是遵照 Intel 所采用的关于中断、异常或者别的什么的术语。然而，像是 exception, trap, interrupt, fault 和 abort 这样的词，在不同架构体系或者操作系统中也没有什么标准含义。即使在某个特定的架构，比如 x86，用起它们来也通常不管它们间到底有什么细微的差别。当你在本次实验之外见到它们的时候，它们的含义也许会有些许不同。

异常和中断都是"保护控制转移”，这些导致处理器从用户态转移到内核模式 ([CPL](https://en.wikipedia.org/wiki/Protection_ring#Privilege_level) = 0)，用户模式代码在这一过程中没有任何机会来干预内核或者其他进程的行为。在 Intel 的术语中，**中断**是一个由异步事件造成的保护控制转移，这一事件通常是在处理器外部发生的，例如外接设备的 I/O 活动通知。相反，**异常**是一个由当前正在运行的代码造成的同步保护控制转移，例如除零或者不合法的内存访问。

为了确保这些保护控制转移确实是受**保护**的，处理器的中断/异常处理机制被设计成当发生中断或异常时当前运行的代码 没有机会任意选择从何处陷入内核或如何陷入内核，而是由处理器确保仅在小心控制的情况下才能进入内核。在 x86 架构中，两种机制协同工作来提供这一保护：

* **The Interrupt Descriptor table / 中断描述符表** 处理器确保中断和异常只能导致内核进入一些确定的、设计优良的、 由内核自身决定的 入口点，而不是在发生中断或异常时由正在运行的代码决定。 x86 允许最多 256 个不同的进入内核的中断或者异常入口点，每个有不同的 **中断向量**。向量是指从 0 到 255 的数字。一个中断的向量是由中断源决定的：不同的设备，不同的错误情况，或者应用向内核的不同请求会生成带有不同向量的中断。CPU 将向量作为进入处理器 **中断描述符表** 的索引，而这个中断描述符表是由内核在内核私有内存区域建立的，就像 GDT 一样。从这个表中对应的入口，处理器会读取：
* 一个读入指令寄存器(EIP)的值，它指向用于处理这一类型异常的内核代码。
* 一个读入代码段寄存器(CS)的值，其中包含一些 0-1 位来表示异常处理代码应该运行在哪一个特权等级（在 JOS 中，所有的异常都在内核模式处理，特权等级为0）。
* **The Task State Segment / 任务状态段** 处理器需要一处位置，用来在中断或异常发生前保存旧的处理器状态，比如，在处理器调用异常处理函数前的 **EIP** 和 **CS** 的值，使得随后异常处理函数可以恢复旧的状态并从中断的地方继续。但用于保存旧处理器状态的区域必须避免被非特权的用户模式代码访问到，否则有错误的或恶意的用户模式代码可能危及内核安全。 因此，当 x86 处理器遇到使得特权等级从用户模式切换到内核模式的中断或陷阱时，它也会将栈切换到内核的内存中的栈。一个被称作 **任务状态段, TSS** 的结构体来描述这个栈所处的[段选择子](https://en.wikipedia.org/wiki/X86_memory_segmentation)和地址。处理器将 **SS**, **ESP**, **EFLAGS**, **CS**, **EIP** 和一个可能存在的错误代码压入新栈，接着它从中断向量表中读取 **CS** 和 **EIP**，并使 **ESP** 和 **SS** 指向新栈。 即使 TSS 很大，可以服务于多种不同目的，JOS只将它用于定义处理器从用户模式切换到内核模式时的内核栈。因为 JOS 的 “内核模式” 在 x86 中是特权等级 0，当进入内核模式时，处理器用 TSS 结构体的 **ESP0** 和 **SS0** 字段来定义内核栈。JOS 不使用 TSS 中的其他任何字段。

x86 处理器可产生的全部同步异常内部使用 0 ~ 31 作为中断向量，因此被映射为中断描述符表入口的 0 ~ 31。例如，一个缺页(page fault, 下同)总会通过向量 14 造成异常。大于 31 的中断向量只被用于 软件中断，这些中断可以用 int 指令生成，或者被用于 异步硬件中断，当外部设备需要提请注意时由其生成。

**exercise four**

在trap.c中，需要初始化中断向量，其中，在inc/mmu.h中提供了相应的初始化中断向量的函数，因此，在trap.c中，只需要声明相应的处理函数handlerx将相应的函数变量与中断向量连接起来，进行初始化。



而在trapentry.S中初始化中断处理函数，首先可以看到trapentry.S的TRAPHANDLER和TRAPHANDLER\_NOEC两者之间的不同只在于TRAPHANDLER\_NOEC中加上了一个pushl $0，实际上这是对于不同的异常和中断处理，是因为对于某些中断时需要压入一个错误码，而另一些不需要。例如对于T\_DIVIDE和T\_PGFLT这两种不同情况的处理，因为T\_DIVIDE不需要压入错误码（保存在$0之中），而T\_PGFLT则需要这样做。

调用\*\_alltraps\*是为了保存出现异常时的那些参数，即将ds和es这两个段保存在其中。根据要求，\_alltraps应该：

* 将一些值压栈，使栈帧看起来像是一个struct Trapframe
* 将GD\_KD读入%ds和%es
* push %esp来传递一个指向这个Trapframe的指针，作为传给trap()的参数
* call trap

显然，这个调用时不需要返回的，因为对于此的调用过程是直接对于trap的处理。



做这些处理的作用是在内核栈中构造Trapframe的结构，这样在\_alltraps之后，trap(Trapframe tf)中参数tf指向的内核栈，而栈中内容正好是一个完整的Trapframe结构。



**问题1**

对每一个中断/异常都分别给出中断处理函数的目的是什么？换句话说，如果所有的中断都交给同一个中断处理函数处理，现在我们实现的哪些功能就没办法实现了？

答：这是为了区分不同的异常/中断类型，TRAPHANDLER在栈中压入了中断向量trapno和错误码errno，在以方便后面根据异常/中断类型做对应处理。正如我们看到的那样，对于不同的异常和中断，其实质处理过程是不同的，是否要压入Error Code，是否要到内核态进行处理等等，需要的内容都是不同的，因此，如果都交给同一个中断处理函数进行处理，这些功能要不就没法实现，要不就只能到内核态再去进行判断，在内核态运行得越久，操作系统出问题的可能性就越大。

**问题2**

你有没有额外做什么事情让user/softint这个程序按预期运行？打分脚本希望它产生一个一般保护错(陷阱 13)，可是softint的代码却发送的是int $14。为什么这个产生了中断向量 13 ？如果内核允许softint的int $14 指令去调用内核中断向量 14 所对应的缺页处理函数，会发生什么？

答：这是因为我们在SETGATE中对中断向量14设置的DPL为0，从而由于用户程序CPL=3，触发了13异常。如果要允许，可以设置中断向量14的DPL为3，但是我们是不希望用户程序来操作内存的。如果允许softint的int $14指令调用内核中断向量14所对应的缺页处理函数，那么，用户态就可以操作内存了，这是我们不想看到的一个结果。

至此，除了challenge 之外，PART A 就已经完成了，如果make qemu的话，可以看到没有没有报triple fault了，但是由于user\_hello运行时用了int 0x30触发了中断，而我们的trap()函数并没有针对中断做处理，于是会销毁该用户进程并进入 monitor()。而用make grade可以看到divzero, softint, badsegment这几个测试通过。

**PART B**

**exercise five**

修改trap\_dispatch()，将缺页异常分发给page\_fault\_handler()。

首先要解决的是由于中断向量 14,T\_PGFLT.对应的缺页异常。由于实际上已经提供了page\_fault\_handler()函数，那么我们只需要判断当前的Trapframe中是否有该异常即可，那么代码相应的，就很简单：



对于page\_fault\_handler()函数，其具体为：



**exercise six**

由于相当于将T\_BKPT看作对于唤起一个JOS内核监视器的伪系统调用，那么当遇到该情况时，只需要进行monitor()的调用即可。故而只需要添加如下代码：



此时，可以看到相应的，make grade之后，faultread，faultreadkernel，faultwrite和faultwritekernel都能够成功通过，然后breakpoint也能够通过。

**问题3**

断点那个测试样例可能会生成一个断点异常，或者生成一个一般保护错，这取决你是怎样在 IDT 中初始化它的入口的（换句话说，你是怎样在 trap\_init 中调用 SETGATE 方法的）。为什么？你应该做什么才能让断点异常像上面所说的那样工作？怎样的错误配置会导致一般保护错？

答：breakpoint.c中生成了一个int $3的结果，即发生了一个断点处理的异常。这是由于在IDT中，我设置了**SETGATE(idt[T\_BRKPT], 1, GD\_KT, handler3, 3);其中T\_BRKPT**的DPL = 3 ，也就是可以直接运行在用户态。如果我错误配置其DPL = 0，那么它将会导致一个一般保护错。

**问题4**

你认为这样的机制意义是什么？尤其要想想测试程序 user/softint 的所作所为 / 尤其要考虑一下 user/softint 测试程序的行为。

答：这样的机制在于保护操作系统的安全性，正如之前所提到的那样，我们不能赋予用户态过高的权限，以免产生不必要的意外。

**exercise seven**

首先查看lib/syscall.c中相关的代码，在之前的过程中，已经添加了相关的系统调用0x30在kern/trapentry.S和kern/trap.c的trap\_init()方法的处理，故，现在还是需要处理两个地方，即trap\_dispath()和kern/syscall.c。

首先查看kern/syscall.c中相关的操作，可以实现代码如下：



对于trap\_dispatch中的相应处理，根据lib/syscall.c所指示的那样，将不同段的数据传入即可。具体代码如下所示：



此时进行make grade可以看到，testbss已经可以通过，但是hello还是无法通过，通过执行make run-hello可以看到，已经可以打出*hello，world*，但是后续会有一个**page default**的错误。

**exercise eight**

首先查看inc/env.h，可以看到Struct Env中的一些相应内容。

然后，查看lib/libmain.c中的libmain()，并且完成它：



实际上只是调用了sys\_getenvid()函数，然后将*thisenv*指向它即可。

此时，可以运行指令make run-hello并且不再报错。

内存保护是操作系统一个至关重要的功能，用以保证一个程序的错误不会导致其他程序崩溃或者操作系统自身崩溃。

操作系统通常依赖硬件支持来实现内存保护。OS 保持与硬件同步哪些虚拟地址是有效的哪些是无效的。当一个程序试图访问无效的地址或者它无权访问的地址时，处理器可以在出错的指令处停止这个程序，并带着它要进行的操作等信息陷入内核。如果错误可以修正，内核可以修复它并让程序继续运行。如果错误不可修正，程序不能继续运行，因为它永远也不能通过这个造成异常的指令。

系统调用提出了一个与内存保护有关的非常有趣的问题，多数系统调用接口允许用户程序向内核传递指针。这些指针指向提供给内核读取或写入的用户缓冲区。内核在执行系统调用时要对这些指针解引用，这样做存在两个问题：

* 内核发生缺页可能要比在用户程序中缺页有更严重的潜在问题。如果内核在操作自身数据结构的时候发生缺页（内核错误），缺页处理函数应该让内核恐慌（也就是整个系统都会崩溃）。因此当内核对用户提供的指针解引用时，它需要一种办法记住发生的任何缺页事实上都是用户程序造成的。
* 内核通常比用户程序拥有更多的内存权限。用户程序也许会为系统调用传递一个内核可以读写，那个用户程序却无权读写的的内存指针。内核必须小心不要被欺骗并解引用这样的指针，因为这样可能会泄露私有信息或者破坏内核的完整性。

因为这两个原因，内核在处理用户程序提供的指针时必须格外小心。

查看 kern/pmap.c 中的user\_mem\_assert



可以看到，根据user\_mem\_assert函数的实现，可以看到，实际上如果user\_mem\_check函数过程中，发生小于0的情况，则代表检测失败，有错误发生。

故而，对其实现，应该为：



首先，类似的，根据虚拟地址找到相应的页的开始和结束，要调用ROUNDDOWN和ROUNDUP来实现，然后为其进行相应的映射，如果超出了相应的地址（出现在内核地址的位置），那么返回\*-E\_FAULT\*。

然后，调整kern/syscall.c来验证系统调用的参数。

修改为：



主要是在这里进行相应的检测。

此时，按照要求，在kern/kdebug.c的debuginfo\_eip()中加入检查，代码如下：



**exercise ten**

按照要求，执行相应指令，得到想要的结果。

至此，make grade，全部exercise 已完成。