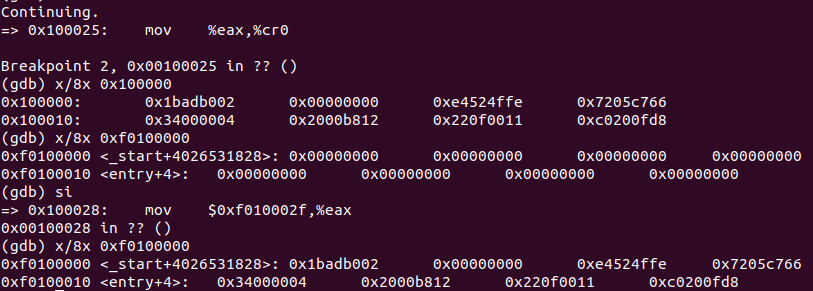
**PartC**

**Exercise 7**

**Use QEMU and GDB to trace into the JOS kernel and find where the new virtual-to-physical mapping takes effect. Then examine the Global Descriptor Table (GDT) that the code uses to achieve this effect, and make sure you understand what's going on.What is the first instruction *after* the new mapping is established that would fail to work properly if the old mapping were still in place? Comment out or otherwise intentionally break the segmentation setup code in kern/entry.S, trace into it, and see if you were right.**



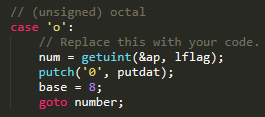
如图中所示，在执行mov %eax, %cr0之前，0xf0100000中的内容和0x100000中的内容并不一致，执行完该命令后，页表启动，0xf0100000被映射到0x100000，两者内容一致

执行下一步的跳转之后可以发现，指令的地址已经变为了0xf010002f，如下图所示，因此在编码中relocated被编码为了一个高地址，如果没有完成地址映射，则无法正确地跳转至该处的代码。

**Exercise 8.**

**We have omitted a small fragment of code - the code necessary to print octal numbers using patterns of the form "%o". Find and fill in this code fragment. Remember the octal number should begin with '0'.**

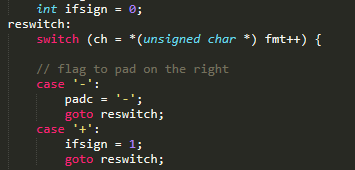
输出八进制的代码在函数vprintfmt中，遇到%符号后会跳出如一个switch表中，其中有case ‘o’的情况，参考u即无符号十进制数的实现即可知道应该在无符号八进制处填什么代码，首先通过getuint函数获取无符号数的值，先向putdat写入开头字符0，之后将base设置为8并跳转至number处即可。如下所示：



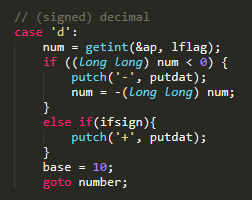
**Exercise 9.**

**You need also to add support for the "+" flag, which forces to precede the result with a plus or minus sign (+ or -) even for positive numbers.**

首先添加一个处理%之后+号的情况，并添加一个变量表示是否需要添加正负号，代码如下所示：



之后在输出有符号十进制的时候再做判断，如果数字大于等于0则输出一个正号，无符号八进制和十六进制不需要考虑，如下所示：



1. printf.c和console.c的接口是cputchar，该函数向console中输出一个字符。
2. 当console屏幕溢出时的操作，CRT\_SIZE为屏幕能够显示的总字数，为2000，CRT\_COL为一行能够显示的总字数，溢出后，将第二行开始的内容移动到第一行，即原来的第二行变为第一行，下一行内容用黑色的空格填满，并调整光标后退一行。
3. (1)fmt指向格式化字符串，即冒号中的字符串，ap指向后面的参数，即x、y和z；

(2)cons\_putc打印一个字符到console中，va\_arg在ap处获取一个指定类型的数据并将ap指向下一处，vcprintf则根据格式和参数打印字符串。函数调用顺序如下：

* vcprintf，fmt为格式化字符串，ap指向参数x的地址
* cons\_putc连续两次，分别打印x和空格
* va\_arg，获取x的值，调用前ap指向x，调用后指向y
* cons\_putc连续五次，打印1、逗号、空格、y和空格
* va\_arg，获取y的值，调用前ap指向y，调用后指向z
* cons\_putc连续五次，打印3、逗号、空格、z和空格
* va\_arg，获取z的值，调用前ap指向z，调用后指向z之后的位置
* cons\_putc连续两次，打印4和换行符

1. (1)显然打印的字符是He110 World，57616对应的16进制是e110，i的内容依照小端序转换为字符串应该是72 6c 64 00，对应的四个字符串分别为r l d \0。

(2)如果是大端序显然将0x00646c72换为0x726c6400即可，57616不需要变化。

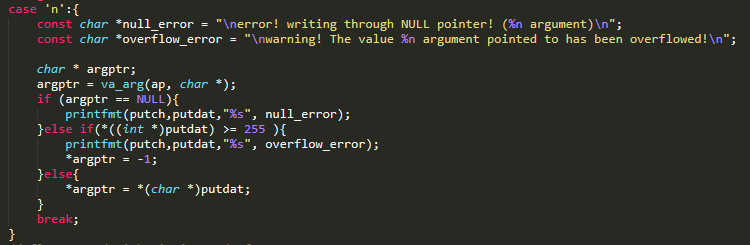
5. y值为x后面紧接着存储的int型整数。

6. 最右增加一个用于表示参数数量的数字即可。

**Exercise 10.**

**Enhance the cprintf function to allow it print with the %n specifier, you can consult the %n specifier specification of the C99 printf function for your reference by typing "man 3 printf" on the console. In this lab, we will use the char \* type argument instead of the C99 int \* argument, that is, "the number of characters written so far is stored into the signed char type integer indicated by the char \* pointer argument. No argument is converted." You must deal with some special cases properly, because we are in kernel, such as when the argument is a NULL pointer, or when the char integer pointed by the argument has been overflowed. Find and fill in this code fragment.**

这一部分要求实现输出%n，即将已经输出的字符个数写入到当前ap指向的参数中，题目要求是char\*参数，需要判断是否该指针为空或要求写入的数字溢出。观察putch函数可以看出，putdat参数表示的就是当前以输出字符个数，因此 直接将其赋值给ap指向的参数即可，代码如下：

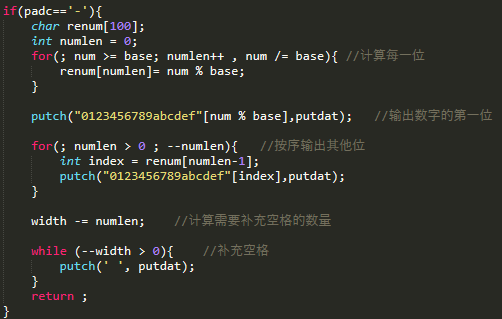


**Exercise 11.**

**Modify the function printnum() in lib/printfmt.c to support "%-" when printing numbers. With the directives starting with "%-", the printed number should be left adjusted. (i.e., paddings are on the right side.) For example, the following function call:cprintf("test:[%-5d]", 3)**

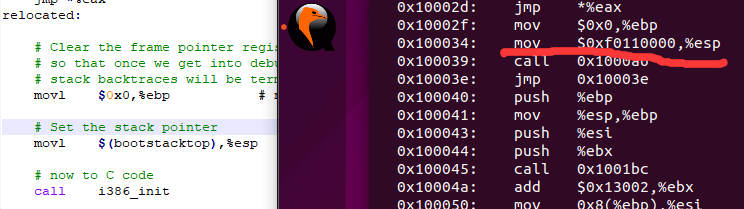
**, should give a result as"test:[3 ]"(4 spaces after '3'). Before modifying printnum(), make sure you know what happened in function vprintffmt().**

本题要求实现%-，用空格在右面补齐要求的宽度，printnum函数是递归至算出数字的最高位，然后依次输出。可以单独在printnum中对padc为’-’的情况做处理，循环地将数字计算在数字中并记录数字的长度，之后完成对输出即可，考虑到数字位数不会超过100位，用100位 长的字符组存储。代码如下所示：



**Exercise 12.**

**Determine where the kernel initializes its stack, and exactly where in memory its stack is located. How does the kernel reserve space for its stack? And at which "end" of this reserved area is the stack pointer initialized to point to?**



如上图所示，在调用C之前设置了EBP和ESP指针的值，一个指向栈顶一个指向栈底，分别为0x0和0xf0110000。进入C函数后，执行初始化push ebp, mov esp, ebp后ebp理论上应该指向了0xf010ffb，之后再修改esp申请栈空间。

**Exercise 13.**

**To become familiar with the C calling conventions on the x86, find the address of the test\_backtrace function in obj/kern/kernel.asm, set a breakpoint there, and examine what happens each time it gets called after the kernel starts. How many 32-bit words does each recursive nesting level of test\_backtrace push on the stack, and what are those words?Note that, for this exercise to work properly, you should be using the patched version of QEMU available on the**[**tools**](https://ipads.se.sjtu.edu.cn/courses/os/2015/tools.html)**page. Otherwise, you'll have to manually translate all breakpoint and memory addresses to linear addresses.**

esi是索引寄存器，ebx是基址寄存器，调用get\_pc\_thunk.bx函数给ebx赋值使得可以通过偏移来访问全局变量，在本题中即指需要打印的字符串常量。通过mov指令将参数x写入了esi寄存器。之后将esi压栈，将字符串常量的地址写入eax中压栈，调用了printf函数打印出信息。之后压栈的eax则是在递归调用之前，传送的是x减一的值。最后两个压栈则分别是当前参数和离开的常量字符串。

**Exercise 14.**

**Implement the backtrace function as specified above. Use the same format as in the example, since otherwise the grading script will be confused. When you think you have it working right, run make grade to see if its output conforms to what our grading script expects, and fix it if it doesn't. *After* you have handed in your Lab 1 code, you are welcome to change the output format of the backtrace function any way you like.**

ebp的上一个是返回地址，返回地址再往上是参数，调用read\_ebp()获取当前ebp，之后按要求打印出当前ebp、eip和args。当前ebp存放的是上一个ebp的值，可以向前递归访问。初始ebp值为0，当ebp为0后则全部回溯完毕。

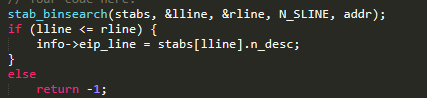


**Exercise 15.**

**Modify your stack backtrace function to display, for each eip, the function name, source file name, and line number corresponding to that eip.**

**In debuginfo\_eip, where do \_\_STAB\_\* come from? This question has a long answer; to help you to discover the answer, here are some things you might want to do:**

首先需要完善kdebug.c中的debuginfo\_eip()函数，观察代码可以发现只需要我们获取代码段行号即可，lline和rline的初值已经设好了，直接调用stab\_binsearch函数即可，因为是找代码段的行数，所以第四个参数用inc/stab.h中定义的N\_SLINE。如果找到行号，则将其赋值给info->eip\_line。代码如下所示：



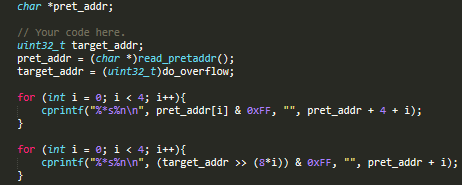
之后在mon\_backtrace()中按要求打印出信息即可，代码如下所示：



**Exercise 16.**

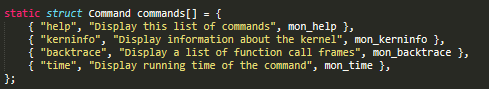
**Recall the buffer overflow attack in ICS Lab. Modify your start\_overflow function to use a technique similar to the buffer overflow to invoke the do\_overflow function. You must use the above cprintf function with the %n specifier you augmented in "Exercise 10" to do this job, or else you won't get the points of this exercise, and the do\_overflow function should return normally.**

使用缓冲区溢出技术修改start\_overflow来调用do\_overflow，需要使用练习10中实现的%n来完成，即利用%\*s动态控制写入字符宽度，然后将其写入需要覆盖的地址。为了能够正常返回，先将正常返回地址移到其上一位 ，之后将正常返回地址修改为do\_overflow()的地址，因为%n能写的只是char所以需要分四次完成，代码如下图所示：



**Exercise 17. There is a "time" command in Linux. The command counts the program's running time. In this exercise, you need to implement a rather easy "time" command. The output of the "time" is the running time (in clocks cycles) of the command. The usage of this command is like this: "time [command]".**

首先需要将time加入到命令集中，如下图所示：



之后实现mon\_time函数即可，mon\_time需要调用rdtsc获取时间标记来测量时间，rdtsc的实现可以参考提示。mon\_time具体需要实现的是测量一条语句的执行时间，首先遍历指令列表，如果找到了对应的指令，则执行并测耗时，如果没找到则打印相应的错误信息，代码如下所示：

