**练习4：分析bootloader加载ELF格式的OS的过程。（要求在报告中写出分析）**

*通过阅读bootmain.c，了解bootloader如何加载ELF文件。通过分析源代码和通过qemu来运行并调试bootloader&OS，*

先读bootmain函数：

bootmain(void) {

readseg((uintptr\_t)ELFHDR, SECTSIZE \* 8, 0);

if (ELFHDR->e\_magic != ELF\_MAGIC) {

goto bad;

}

struct proghdr \*ph, \*eph;

ph = (struct proghdr \*)((uintptr\_t)ELFHDR + ELFHDR->e\_phoff);

eph = ph + ELFHDR->e\_phnum;

for (; ph < eph; ph ++) {

readseg(ph->p\_va & 0xFFFFFF, ph->p\_memsz, ph->p\_offset);

}

((void (\*)(void))(ELFHDR->e\_entry & 0xFFFFFF))();

bad:

outw(0x8A00, 0x8A00);

outw(0x8A00, 0x8E00);

while (1);

}

* **bootloader如何读取硬盘扇区的？**

根据bootmain函数，先由readseg函数读取硬盘扇区，readseg函数循环调用真正读取硬盘扇区的函数readsect，来每次读出一个扇区。其实就是把硬盘上的kernel，读取到内存中

readsect(void \*dst, uint32\_t secno) {

waitdisk(); // 等待硬盘就绪

// 写地址0x1f2~0x1f5,0x1f7,发出读取磁盘的命令

outb(0x1F2, 1);

outb(0x1F3, secno & 0xFF);

outb(0x1F4, (secno >> 8) & 0xFF);

outb(0x1F5, (secno >> 16) & 0xFF);

outb(0x1F6, ((secno >> 24) & 0xF) | 0xE0);

outb(0x1F7, 0x20);

waitdisk();

insl(0x1F0, dst, SECTSIZE / 4);//读取一个扇区

}

步骤如下：

1. 等待硬盘空闲。waitdisk的函数实现只有一行：

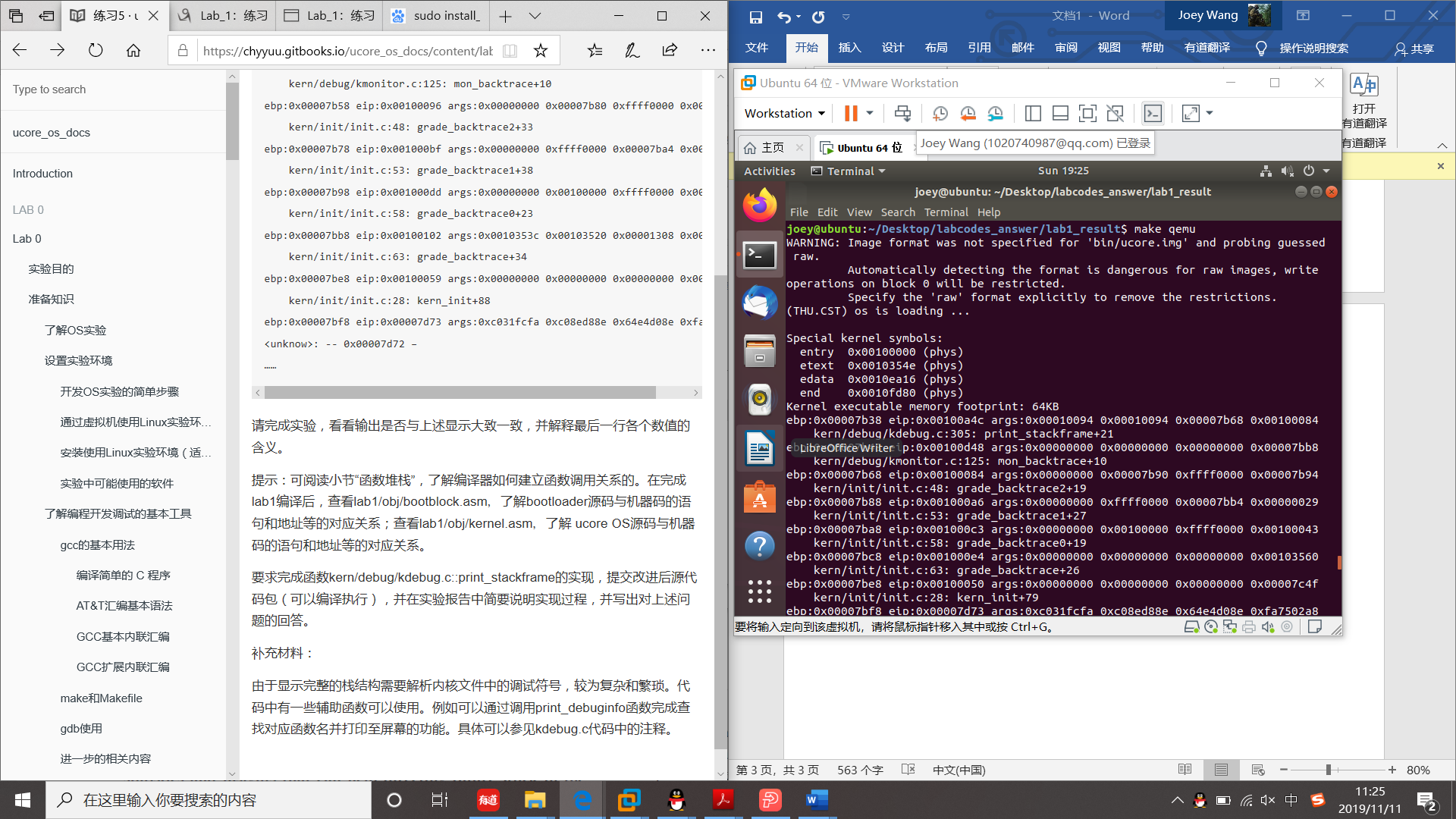
while ((inb(0x1F7) & 0xC0) != 0x40)

意思是不断查询读0x1F7寄存器的最高两位，直到最高位为0、次高位为1（这个状态应该意味着磁盘空闲）才返回。

1. 硬盘空闲后，发出读取扇区的命令。对应的命令字为0x20，放在0x1F7寄存器中；读取的扇区数为1，放在0x1F2寄存器中；读取的扇区起始编号共28位，分成4部分依次放在0x1F3~0x1F6寄存器中。
2. 发出命令后，再次等待硬盘空闲。
3. 硬盘再次空闲后，开始从0x1F0寄存器中读数据

* **bootloader是如何加载ELF格式的OS？**
  + 1. 从硬盘读了8个扇区数据到内存0x10000处，并把这里强制转换成elfhdr使用；
  1. 校验e\_magic字段；
  2. 根据偏移量分别把程序段的数据读取到内存中。

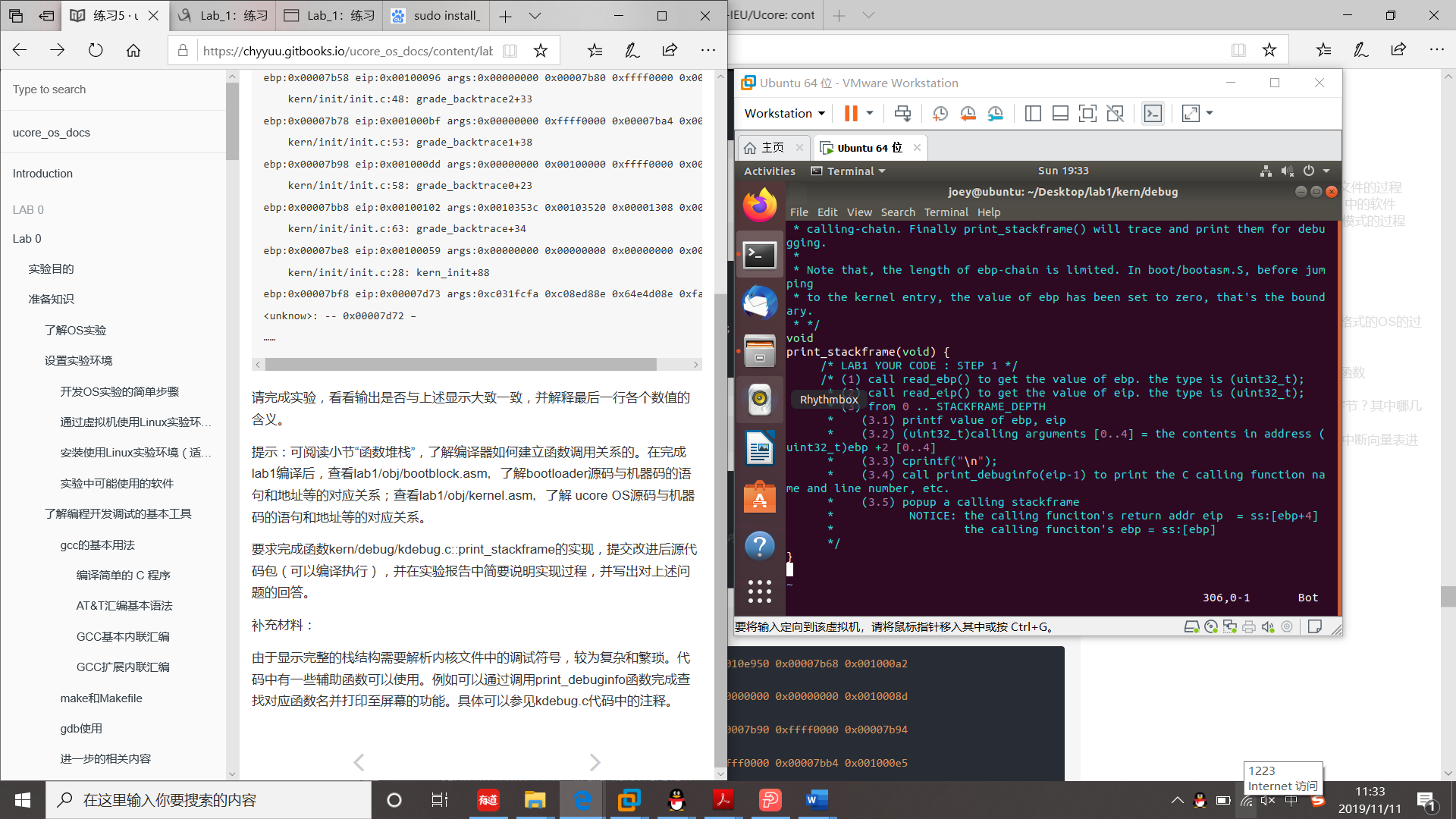
**练习5：实现函数调用堆栈跟踪函数 （需要编程）**



执行make qume结果如上，最后一行是ebp，eip和args三类参数：

***ebp:0x00007bf8 eip:0x00007d73 args:0xc031fcfa 0xc08ed88e 0x64e4d08e 0xfa7502a8***

***<unknow>: -- 0x00007d72 --***



找到print\_stackframe函数，发现函数里面的注释已经提供了十分详细的步骤，基本上按照提示来做就行了。代码如下所示:

***/\* LAB1 YOUR CODE : STEP 1 \*/***

***/\* (1) call read\_ebp() to get the value of ebp. the type is (uint32\_t);***

***\* (2) call read\_eip() to get the value of eip. the type is (uint32\_t);***

***\* (3) from 0 .. STACKFRAME\_DEPTH***

***\* (3.1) printf value of ebp, eip***

***\* (3.2) (uint32\_t)calling arguments [0..4] = the contents in address (uint32\_t)ebp +2 [0..4]***

***\* (3.3) cprintf("\n");***

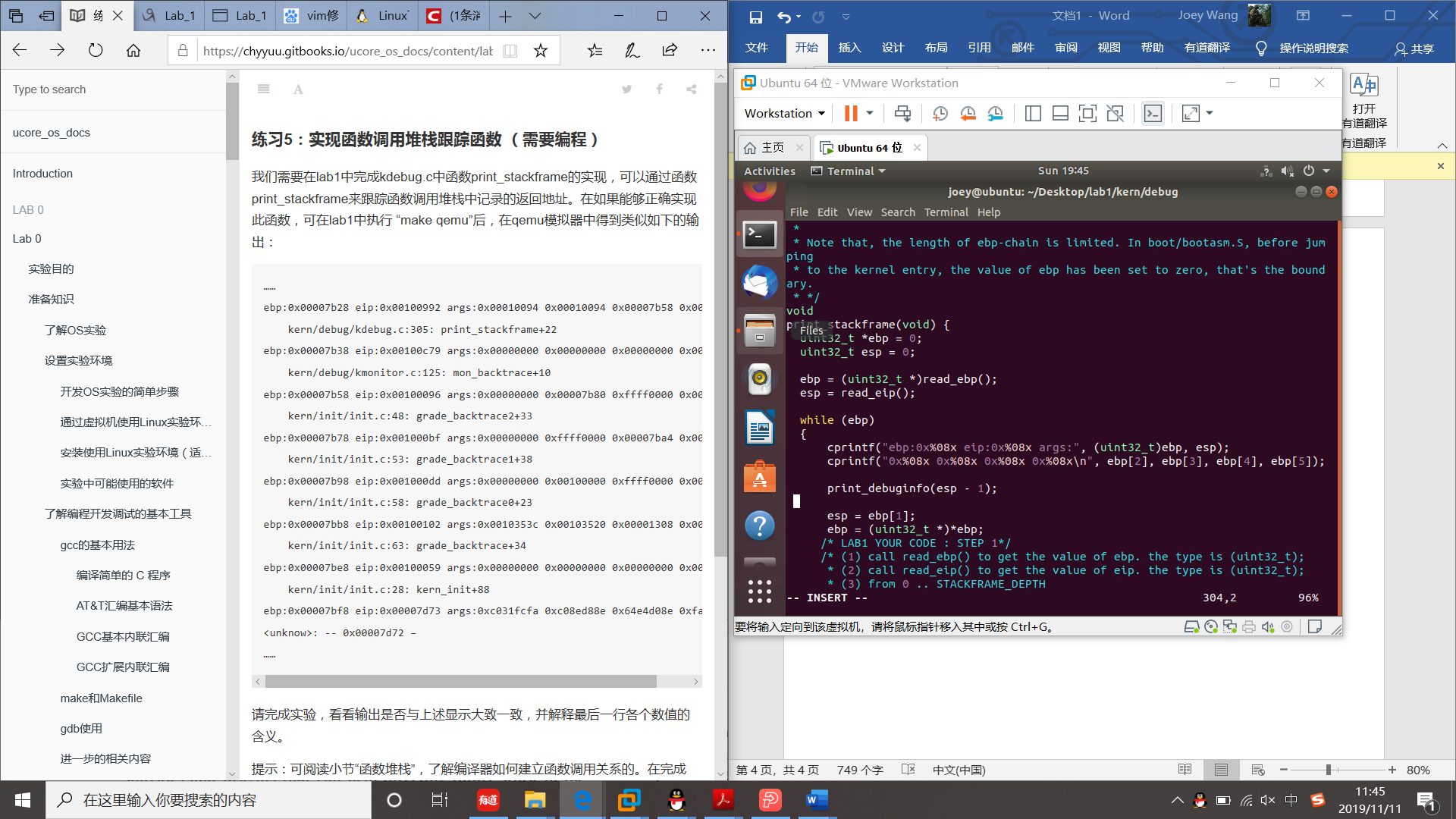
***\* (3.4) call print\_debuginfo(eip-1) to print the C calling function name and line number, etc.***

***\* (3.5) popup a calling stackframe***

***\* NOTICE: the calling funciton's return addr eip = ss:[ebp+4]***

***\* the calling funciton's ebp = ss:[ebp]***

***\*/***



***代码如下：***

***uint32\_t \*ebp = 0;***

***uint32\_t esp = 0;***

***ebp = (uint32\_t \*)read\_ebp();***

***esp = read\_eip();***

***while (ebp)***

***{***

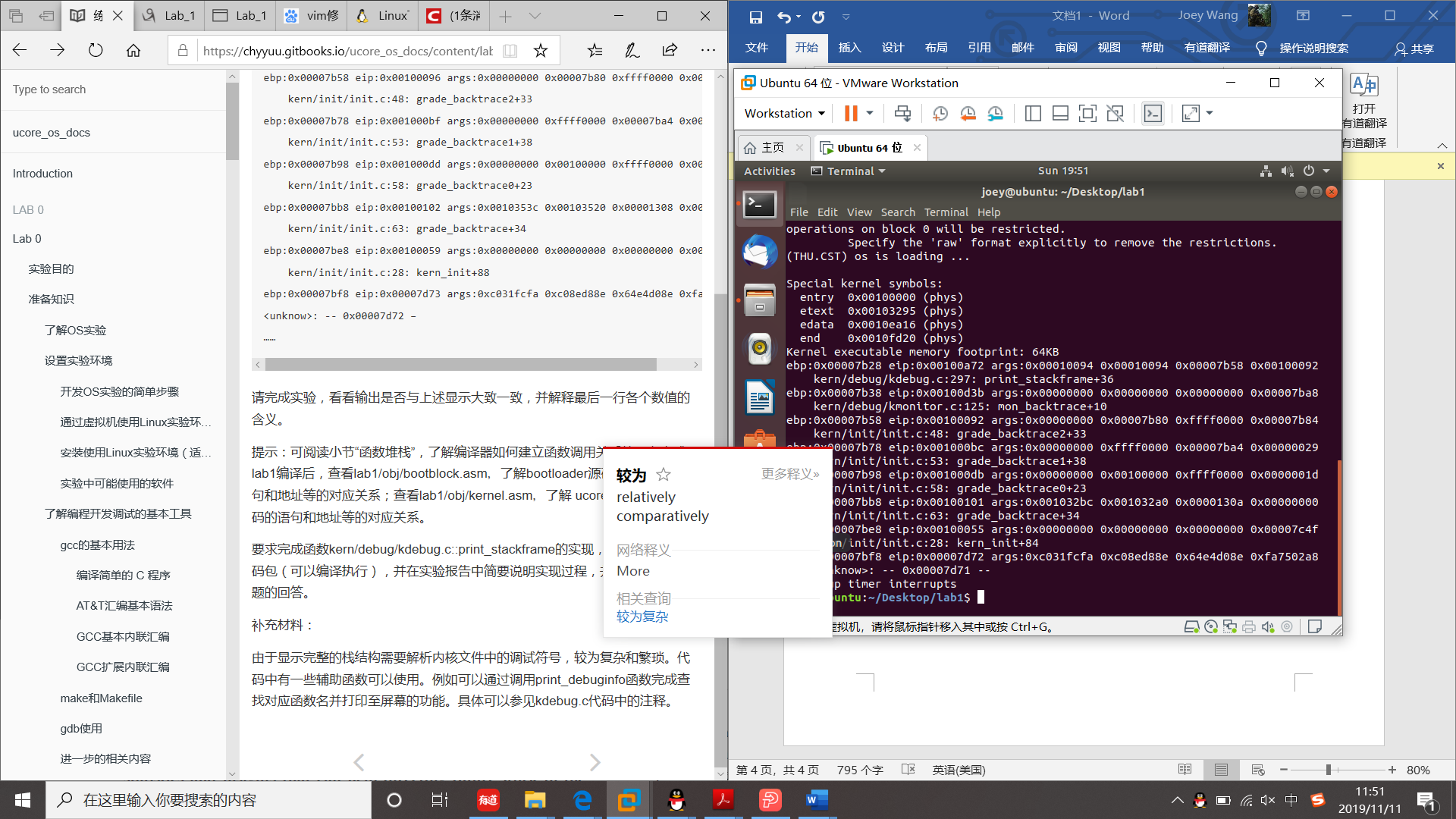
***cprintf("ebp:0x%08x eip:0x%08x args:", (uint32\_t)ebp, esp);***

***cprintf("0x%08x 0x%08x 0x%08x 0x%08x\n", ebp[2], ebp[3], ebp[4], ebp[5]);***

***print\_debuginfo(esp - 1);***

***esp = ebp[1];***

***ebp = (uint32\_t \*)\*ebp;***



再次执行make qemu后得出结果如上，最后一行是ebp:0x00007bf8 eip:0x00007d72 args:0xc031fcfa 0xc08ed88e 0x64e4d08e 0xfa7502a8。

解释：

1. ebp:0x00007bf8：此时ebp的值是kern\_init函数的栈顶地址。
2. eip:0x00007d72：eip的值是kern\_init函数的返回地址，也就是bootmain函数调用kern\_init对应的指令的下一条指令的地址。
3. args:0xc031fcfa 0xc08ed88e 0x64e4d08e 0xfa7502a8 一般来说，args存放的4个dword是对应4个输入参数的值。