练习5：实现函数调用堆栈跟踪函数（需要编程）。

我们需要在lab1中实现kdebug.c中的函数print\_stackframe，可以通过函数print\_stackframe来跟踪函数调用堆栈中记录的返回地址。如果能够正确实现此函数，可在lab1中执行make qemu后，在qemu模拟器中得到类似如下的输出：

⋮

ebp:0x00007b28 eip:0x00100992 args:0x00010094 0x00010094 0x00007b58 0x00100096

kern/debug/kdebug.c:305: print\_stackframe+22

ebp:0x00007b38 eip:0x00100c79 args:0x00000000 0x00000000 0x00000000 0x00007ba8

kern/debug/kmonitor.c:125: mon\_backtrace+10

ebp:0x00007b58 eip:0x00100096 args:0x00000000 0x00007b80 0xffff0000 0x00007b84

kern/init/init.c:48: grade\_backtrace2+33

ebp:0x00007b78 eip:0x001000bf args:0x00000000 0xffff0000 0x00007ba4 0x00000029

kern/init/init.c:53: grade\_backtrace1+38

ebp:0x00007b98 eip:0x001000dd args:0x00000000 0x00100000 0xffff0000 0x0000001d

kern/init/init.c:58: grade\_backtrace0+23

ebp:0x00007bb8 eip:0x00100102 args:0x0010353c 0x00103520 0x00001308 0x00000000

kern/init/init.c:63: grade\_backtrace+34

ebp:0x00007be8 eip:0x00100059 args:0x00000000 0x00000000 0x00000000 0x00007c53

kern/init/init.c:28: kern\_init+88

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

<unknow>: -- 0x00007d72 –

⋮

请完成实验，看看输出是否与上述显示大致一致，并解释最后一行各个数值的含义。

提示：可阅读 3.3.1 小节“函数堆栈”，了解编译器如何建立函数调用关系的。在完成 lab1 编译后，查看lab1/obj/bootblock.asm，了解 bootloader 源码与机器码的语句和地址等的对应关系；查看 lab1/obj/kernel.asm，了解ucore OS 源码与机器码的语句和地址等的对应关系。

要求完成函数 kern/debug/kdebug.c::print\_stackframe 的实现，提交改进后源代码包(可以编译执行)，并在实验报告中简要说明实现过程，并写出对上述问题的回答。

补充材料

显示完整的栈结构需要解析内核文件中的调试符号，这较为复杂和繁琐。代码中有一些辅助函数可以使用。例如，可以通过调用print\_debuginfo函数完成查找对应函数名并打印至屏幕的功能。具体可以参见kdebug.c代码中的注释。

答：

**（1）函数堆栈[[[1]](#footnote-1)]**

栈是一个很重要的编程概念（编译课和程序设计课都讲过相关内容），与编译器和编程语言有紧密的联系。理解调用栈最重要的两点是：栈的结构，EBP寄存器的作用。一个函数调用动作可分解为：零到多个PUSH指令（用于参数入栈），一个CALL指令。CALL指令内部其实还暗含了一个将返回地址（即CALL指令下一条指令的地址）压栈的动作（由硬件完成）。几乎所有本地编译器都会在每个函数体之前插入类似如下的汇编指令：

pushl %ebp

movl %esp , %ebp

这样在程序执行到一个函数的实际指令前，已经有以下数据顺序入栈：参数、返回地址、ebp寄存器。由此得到类似如下的栈结构（参数入栈顺序跟调用方式有关，这里以C语言默认的CDECL为例）：

|  |
| --- |
| +| 栈底方向 | 高位地址  | ... |  | ... |  | 参数3 |  | 参数2 |  | 参数1 |  | 返回地址 |  | 上一层[ebp] | <-------- [ebp]  | 局部变量 | 低位地址 |

**函数调用栈结构**

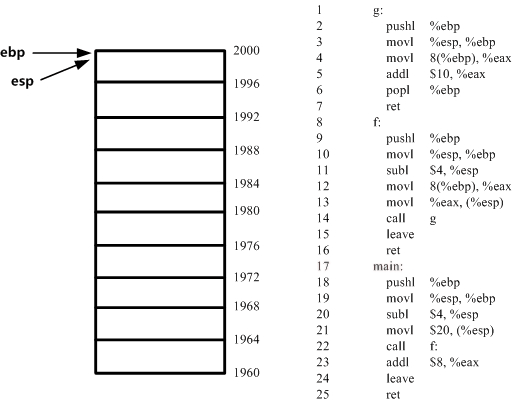
这两条汇编指令的含义是：首先将ebp寄存器入栈，然后将栈顶指针esp赋值给ebp。“mov ebp esp”这条指令表面上看是用esp覆盖ebp原来的值，其实不然。因为给ebp赋值之前，原ebp值已经被压栈（位于栈顶），而新的ebp又恰恰指向栈顶。此时ebp寄存器就已经处于一个非常重要的地位，该寄存器中存储着栈中的一个地址（原ebp入栈后的栈顶），从该地址为基准，向上（栈底方向）能获取返回地址、参数值，向下（栈顶方向）能获取函数局部变量值，而该地址处又存储着上一层函数调用时的ebp值。

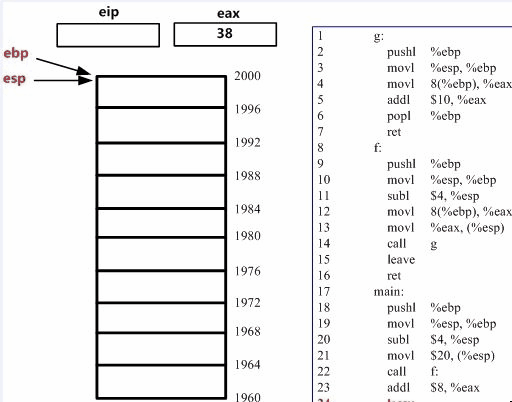
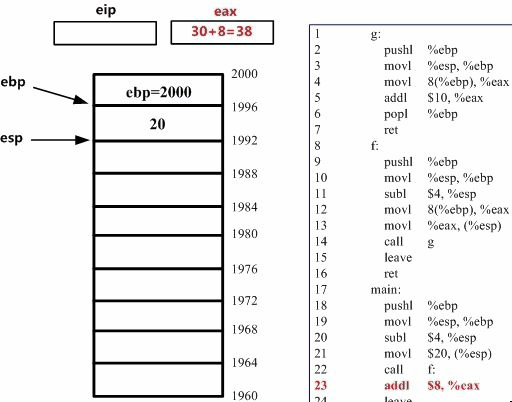
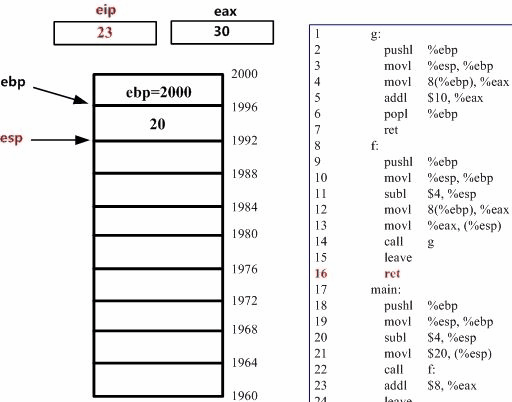
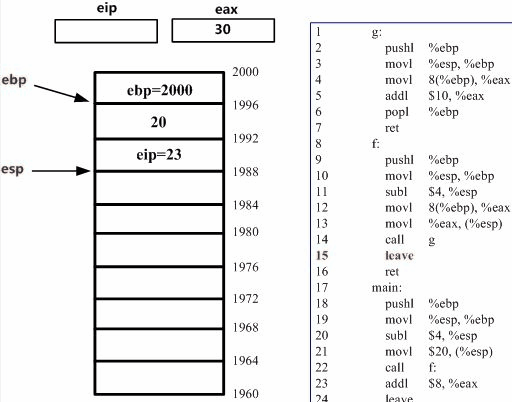
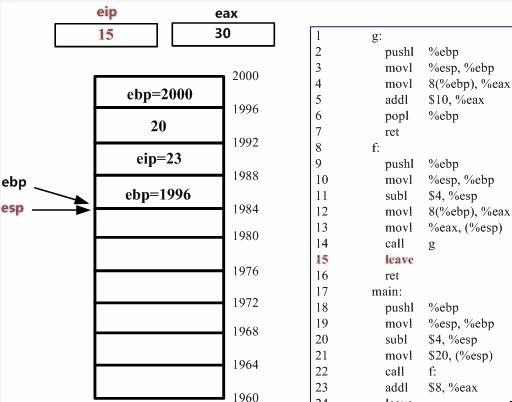
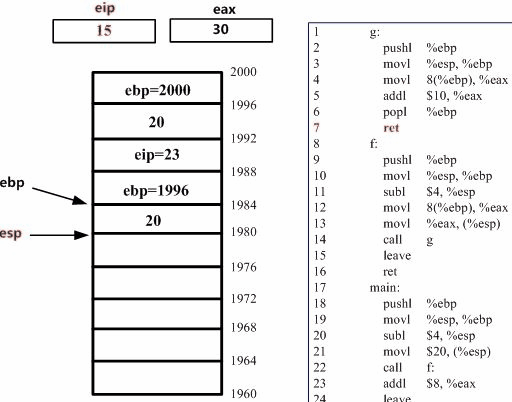
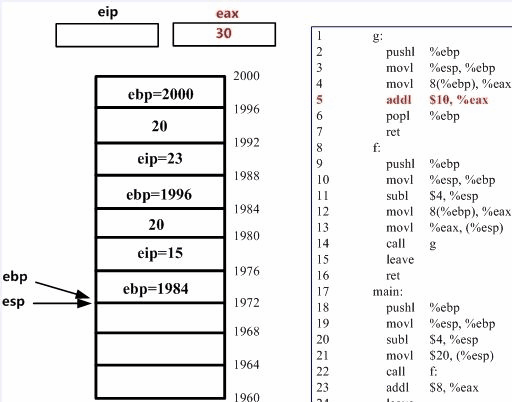
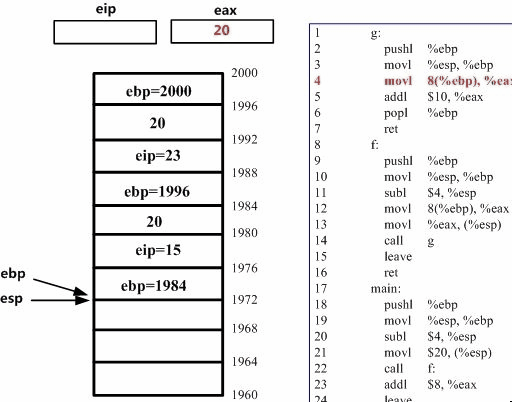
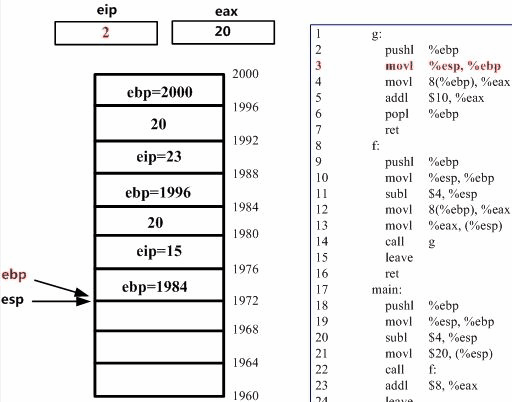
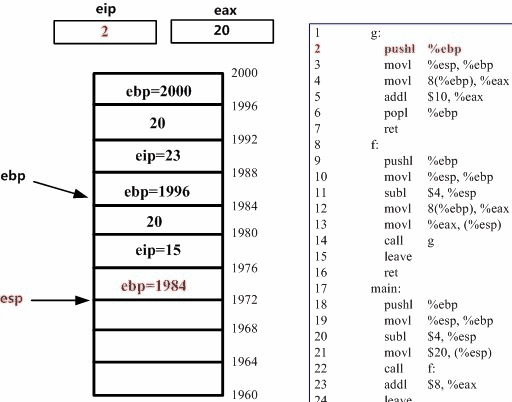
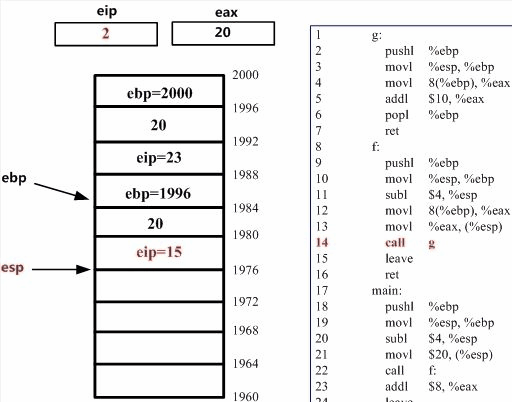
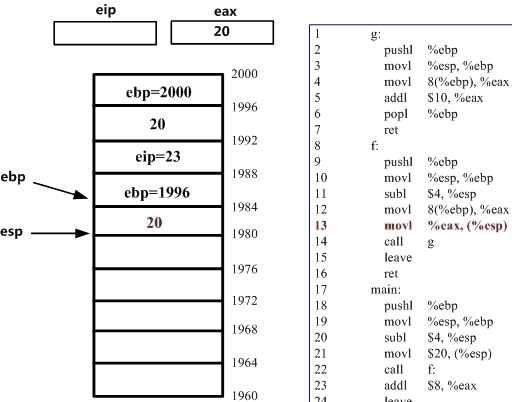
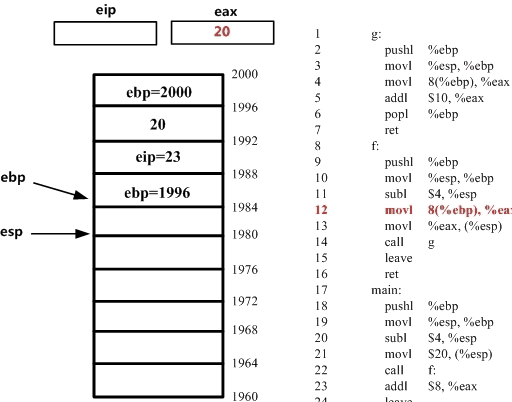
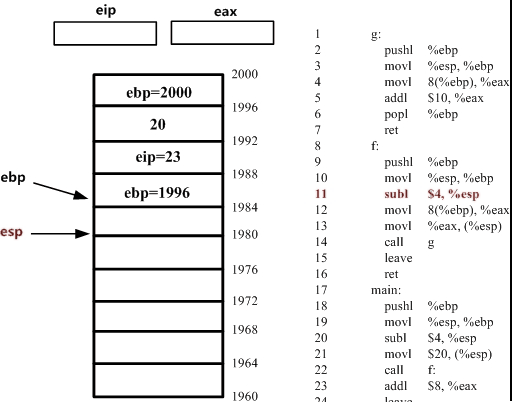
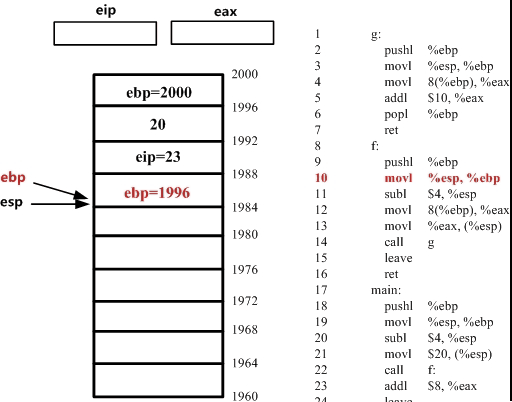
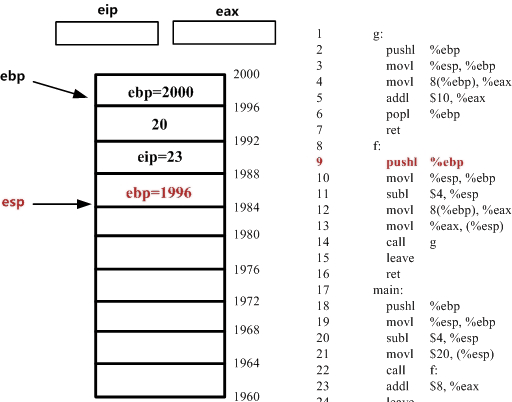
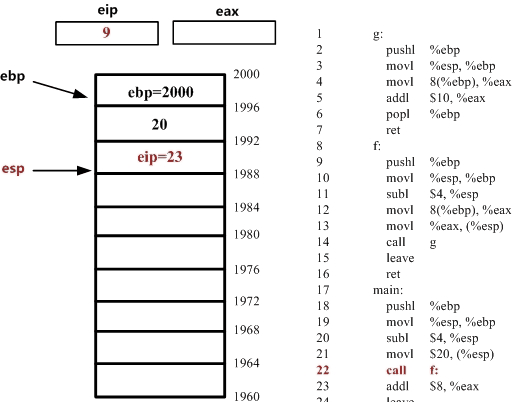
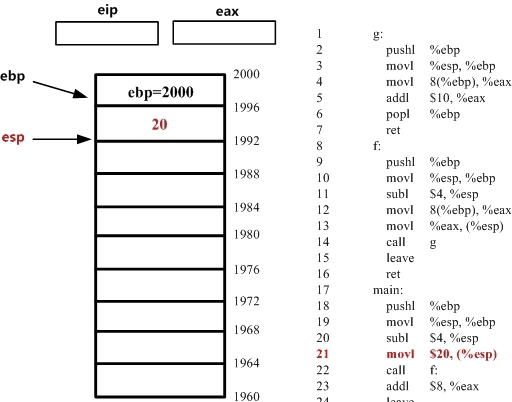
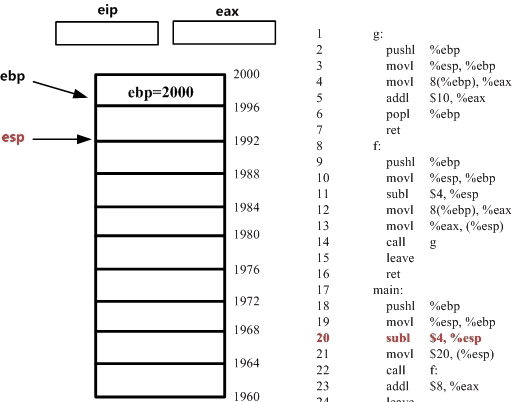
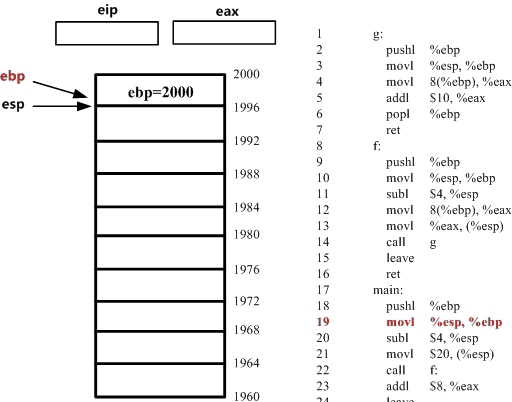
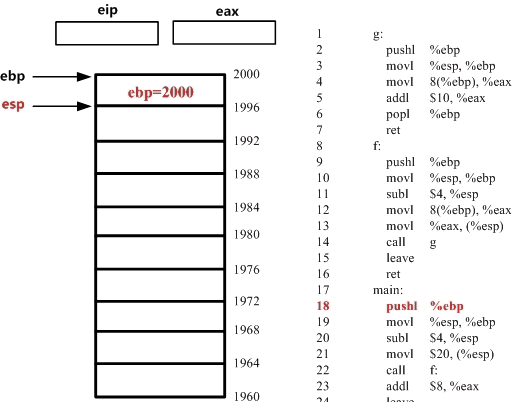
一般而言，ss:[ebp+4]处为返回地址，ss:[ebp+8]处为第一个参数值（最后一个入栈的参数值，此处假设其占用4字节内存），ss:[ebp-4]处为第一个局部变量，ss:[ebp]处为上一层ebp值。由于ebp中的地址处总是“上一层函数调用时的ebp值”，而在每一层函数调用中，都能通过当时的ebp值“向上（栈底方向）”能获取返回地址、参数值，“向下（栈顶方向）”能获取函数局部变量值。如此形成递归，直至到达栈底。这就是函数调用栈。

举一个实际的例子[[[2]](#footnote-2)]查看ebp与esp两个寄存器如何构建出完整的函数栈（其中leave等同于movl %ebp, %esp，popl %ebp两条指令）：

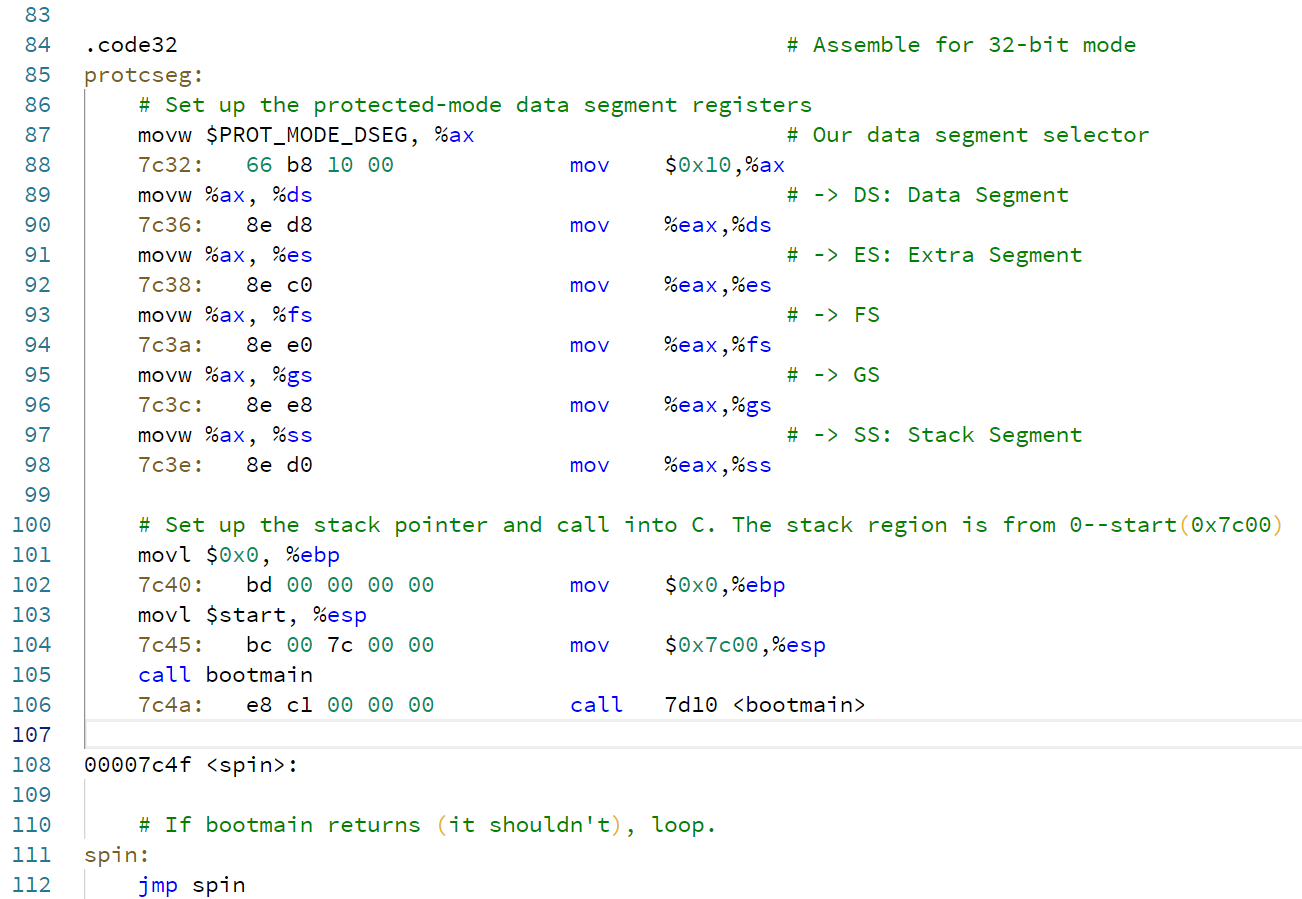
|  |
| --- |
| int g(int x) {  return x + 10;  }  int f(int x) {  return g(x);  }  int main(void) {  return f(20) + 8;  } |

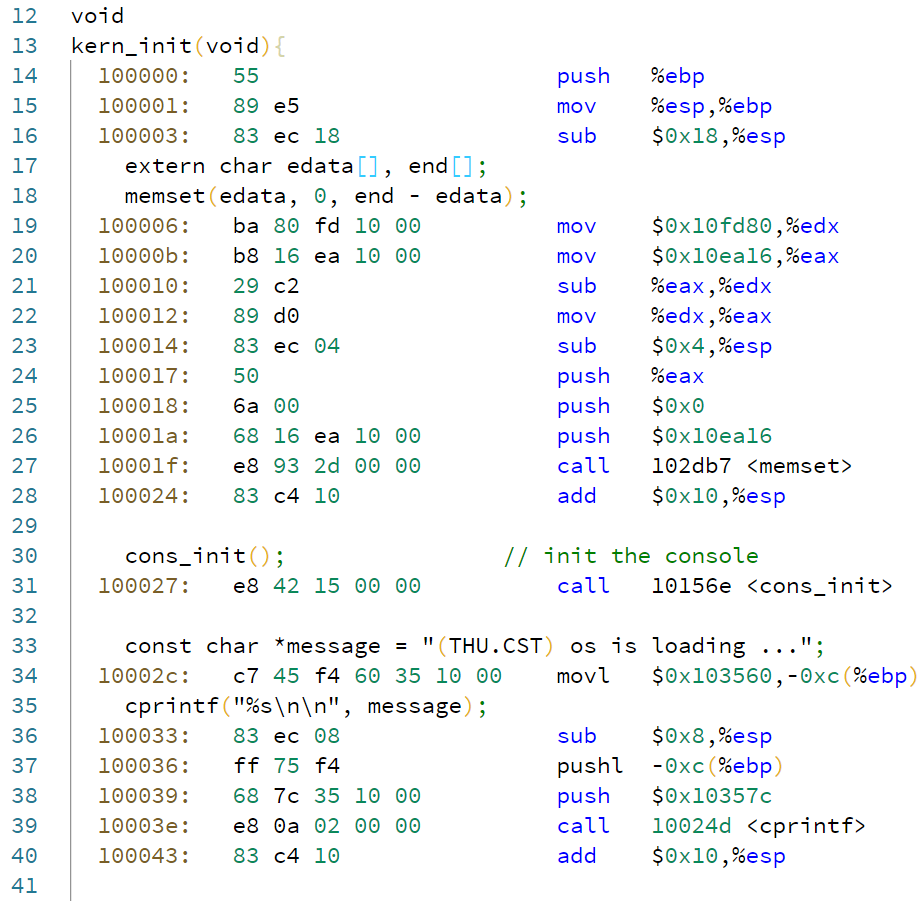
函数调用过程的示意图如下：





**（2）查看lab1/obj目录下的bootblock.asm和kernel.asm**





**（3）实现调用堆栈跟踪函数print\_stackframe[[[3]](#footnote-3)]**

该函数位于lab1/kern/debug/kdebug.c文件中，根据注释实现该函数如下：

|  |
| --- |
| void  print\_stackframe(void) {  uint32\_t ebp = read\_ebp(); //读取ebp的值  uint32\_t eip = read\_eip(); //读取eip的值  int i, j;  for (i = 0; ebp != 0 && i < STACKFRAME\_DEPTH; i++) {  cprintf("ebp:0x%08x eip:0x%08x args:", ebp, eip);  uint32\_t \*args = (uint32\_t \*)ebp + 2; //参数的首地址  for (j = 0; j < 4; j++) { //假设有4个参数，依次输出  cprintf("0x%08x ", args[j]);  }  cprintf("\n");  print\_debuginfo(eip - 1); //打印C语言的函数名和行号  eip = \*((uint32\_t \*)ebp + 1);  ebp = \*((uint32\_t \*)ebp); //转到上一个ebp  }  } |

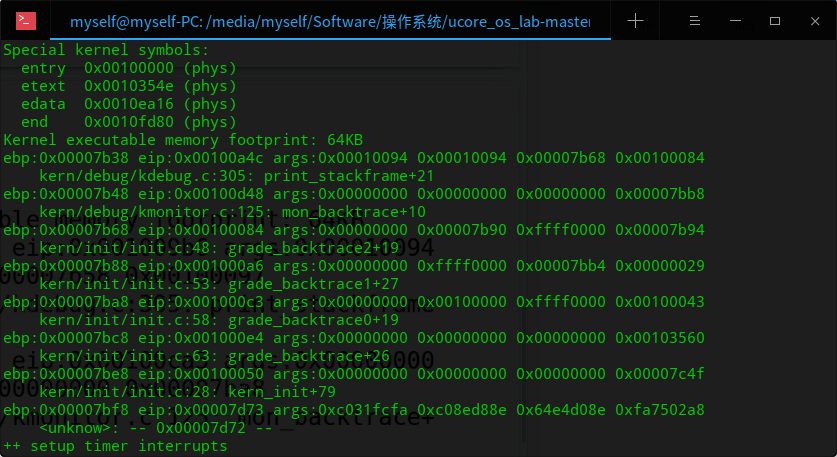
首先定义两个局部变量ebp、esp分别存放ebp、esp寄存器的值。这里将ebp定义为指针，是为了方便后面取ebp寄存器的值；其中cprintf是定义在lab1/libs/stdio.h中负责进行输出的函数。

调用read\_ebp函数来获取执行print\_stackframe函数时ebp寄存器的值，这里read\_ebp必须定义为inline函数，否则获取的是执行read\_ebp函数时的ebp寄存器的值；调用read\_eip函数来获取当前指令的位置，也就是此时eip寄存器的值。这里read\_eip必须定义为常规函数而不是inline函数，因为这样的话在调用read\_eip时会把当前指令的下一条指令的地址（也就是eip寄存器的值）压栈，那么在进入read\_eip函数内部后便可以从栈中获取到调用前eip寄存器的值。

由于变量eip存放的是下一条指令的地址，因此将变量eip的值减去1，得到的指令地址就属于当前指令的范围了；由于只要输入的地址属于当前指令的起始和结束位置之间，print\_debuginfo都能搜索到当前指令，因此这里减去1即可。

以后变量eip的值就不能再调用read\_eip来获取了（每次调用获取的值都是相同的），而应该从ebp寄存器指向栈中的位置再往上一个单位中获取；由于ebp寄存器指向栈中的位置存放的是调用者的ebp寄存器的值，之后通过不断回溯，直到ebp寄存器的值变为0。

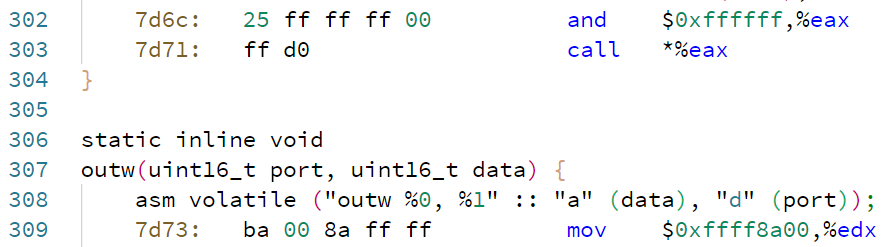
**（4）make qemu**



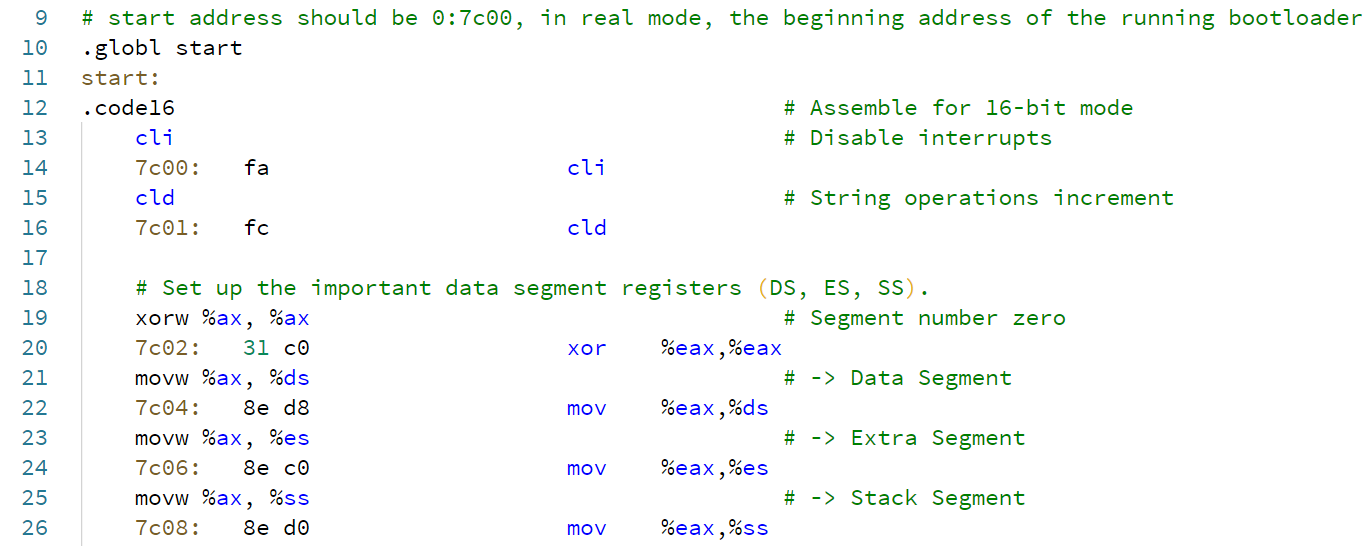
**（5）解释最后一行各个数值的含义[[[4]](#footnote-4)]**

最后一行是ebp:0x00007bf8 eip:0x00007d73 args:0xc031fcfa 0xc08ed88e 0x64e4d08e 0xfa7502a8，共有ebp，eip和args三类参数，下面分别给出解释。

**ebp:0x0007bf8** 此时ebp的值是kern\_init函数的栈顶地址，从obj/bootblock.asm文件中知道整个栈的栈顶地址为0x00007c00，ebp指向的栈位置存放调用者的ebp寄存器的值，ebp+4指向的栈位置存放返回地址的值，这意味着kern\_init函数的调用者（也就是bootmain函数）没有传递任何输入参数给它！因为单是存放旧的ebp、返回地址已经占用8字节了。

**eip:0x00007d73** 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。这与obj/bootblock.asm是相符合的。 

**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。但这里比较特殊，由于bootmain函数调用kern\_init并没传递任何输入参数，并且栈顶的位置恰好在bootloader第一条指令存放的地址的上面，而args恰好是kern\_int的ebp寄存器指向的栈顶往上第2~5个单元，因此args存放的就是bootloader指令的前16个字节！可以对比obj/bootblock.asm文件来验证（验证时要注意系统是小端字节序）。



1. [] 陈渝, 向勇. 操作系统实验指导[M]. 北京, 清华大学出版社, 2013:57-58. [↑](#footnote-ref-1)
2. [] nlskyfree. ucore lab1[OL]. <https://www.jianshu.com/p/969e1bdcb471>, 2018-07-13. [↑](#footnote-ref-2)
3. [] whl1729. 《ucore lab1 exercise5》实验报告[OL]. https://www.cnblogs.com/wuhualong/p/ucore\_lab1\_exercise5\_report.html, 2019-03-04. [↑](#footnote-ref-3)
4. [] whl1729. 《ucore lab1 exercise5》实验报告[OL]. https://www.cnblogs.com/wuhualong/p/ucore\_lab1\_exercise5\_report.html, 2019-03-04. [↑](#footnote-ref-4)