**Linux中函数调用**

理论实现过程：

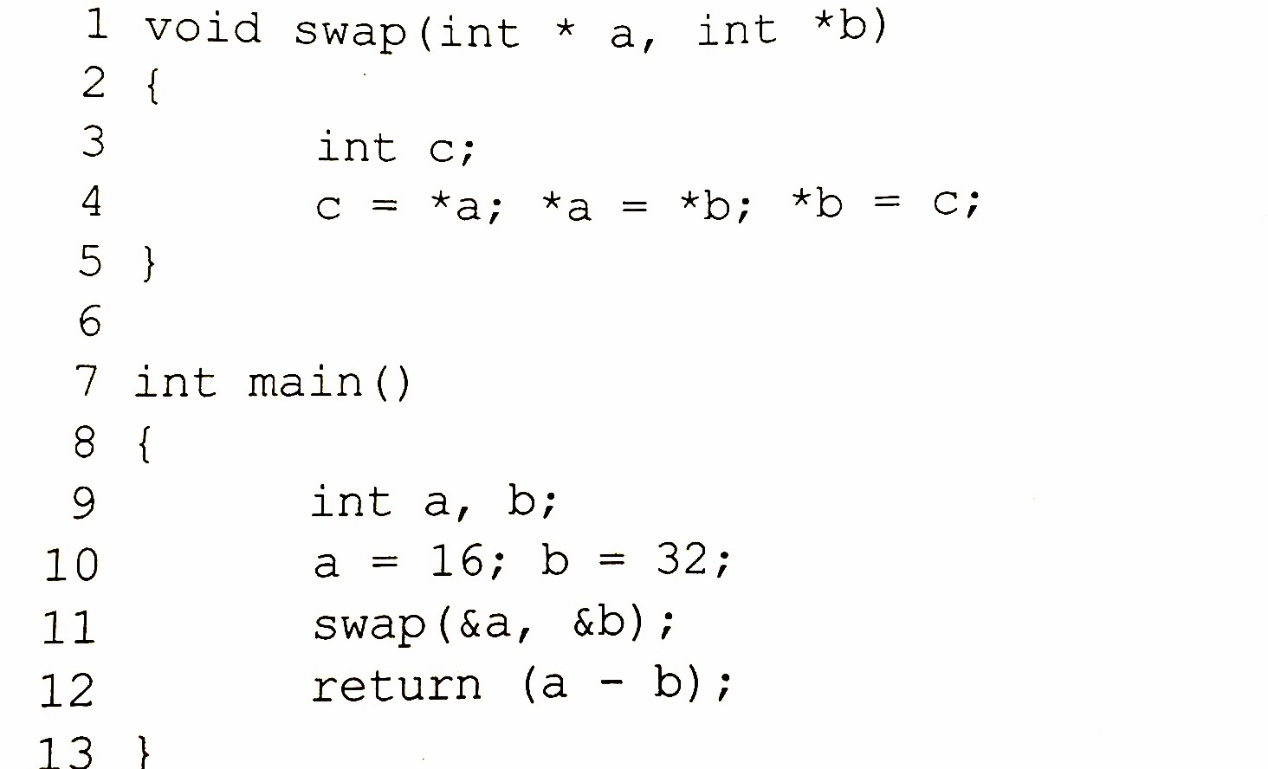
现存大多数的计算机都是用栈来实现的函数之间的调用操作。函数调用主要涉及参数的传递，返回值的返回，调用函数的ret，数据的恢复，被调用函数的call等问题。

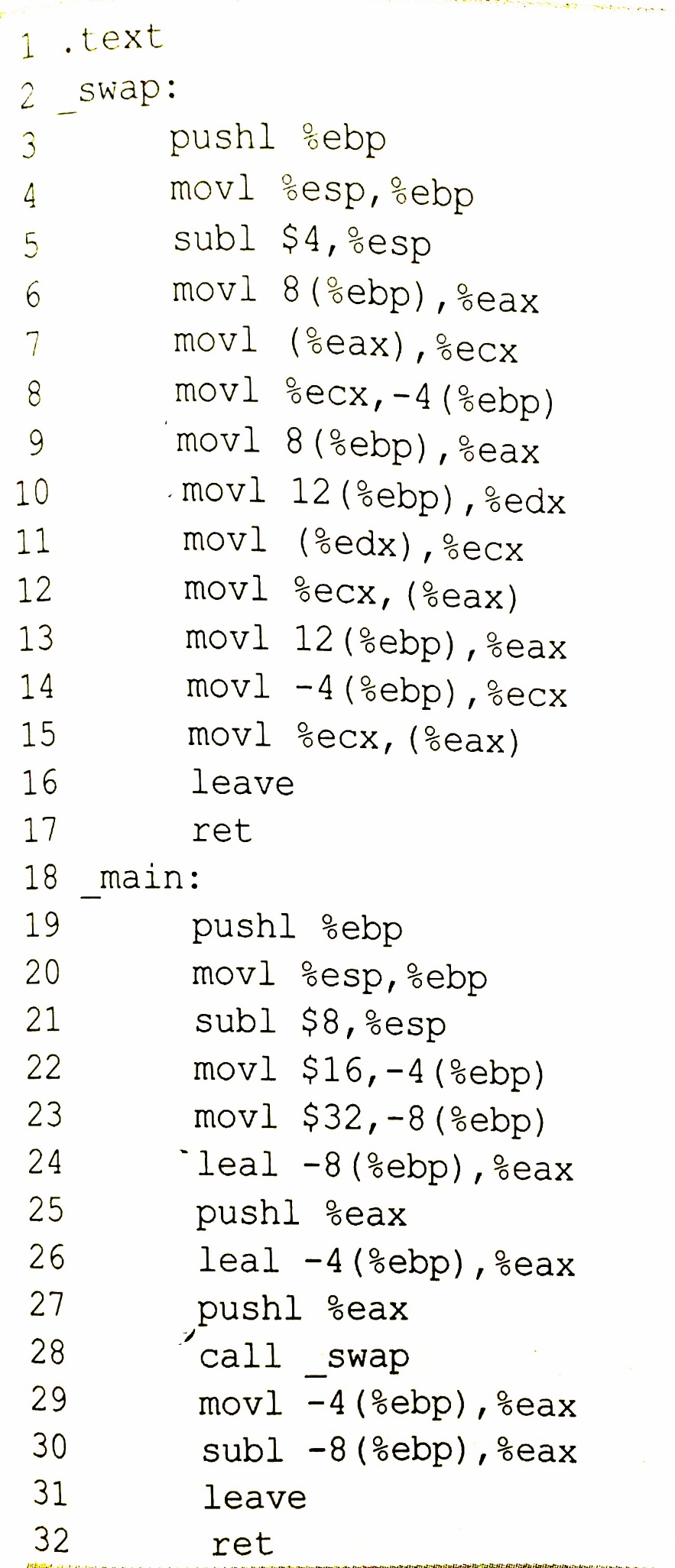
在栈中每一个函数都有一段栈来存储数据，这一段栈叫做栈帧（ebp存储器用来指向每一帧的底部），在每一帧中有一个帧顶的指针esp。当调用一个函数的时候即call的时候，第一步会把调用函数的返回地址push到调用者的帧栈里面，然后在跳到被调用函数的地址执行。通常每个函数的第一步都是push调用函数的ebp以便返回。当执行ret的时候，会将栈中的返回地址pop，并跳转到该地址。所以call和ret的联合使用的时候需要注意栈中的数据清空，以免ret执行pop的时候返回不到正确地址或者call之后未存储上一个函数的ebp回不去栈帧的情况。每call一个函数的时候，需要该函数负责保护某些数据如ebp，若用使用的情况还需保存ebx，esi，edi.每次ret的时候还需要leave（movl %ebp,%esp; popl %ebp;）即恢复上一个调用函数的栈帧指针和栈顶指针。

关于函数的参数的传递，调用函数将从右到左把参数的地址压入栈中，由于栈的增加方向是地址减小的方向，所以被调用函数可以通过8（%ebp）来访问最左边第一个的参数的地址，4\*n来访问第n-1个参数，因为在call的时候会将调用函数的下一条指令的地址（返回地址）posh入栈，所以4（%ebp）的位置存放的是返回地址，（其实此处我还是有些不是很清楚，也很有可能是另一种情况即编译阶段，编译器根据数据类型来确定4，8，12或者其他的数据的地址）。返回的数据放到eax中,可以此来实现函数的返回值传递。

汇编代码：

Swap()实现将ab交换，main()来调用swap()。





可以看到swap(),main()的第一句都是pushl %ebp;这是将上一个调用函数的栈帧保存起来留作ret之前的leave操作可以popl回到栈帧的保证。

第二句都是movl %ebp %esp;

这是将被调用函数的栈帧的初始化，因为刚刚运行的时候并没有数据压入栈中，所以栈顶esp和栈帧ebp在同一个位置。

在每个函数的结尾处都有leave和let，

Leave有两步操作第一步将ebp的值传给esp（此步相当于清空了本函数的栈帧，因为栈顶和栈底指向同一地址），而前面讲的每个函数被调用后第一步就是posh上一个函数的ebp，所以每一个函数的栈底存储的都是返回函数的ebp（栈帧指针的地址）。第二步就合情理了，pop上个函数的ebp到ebp中。

Ret执行的就是pop出main的返回地址。因为swap被调用前call指令会将main的返回地址压入栈，所以当swap保存的main的ebp被leave指令pop后，此时ebp指向的是main的栈帧，esp指向的是main的栈顶，实现了swap的内存清除也同时pop地址。

在main中可以看到call swap之前，main会将swap的参数从右到左压入栈中，swap可以通过n（%ebp）来访问参数。

2015317200406

计科1502

程天赐