

# 汇编第二次作业

李晨昊 2017011466

2019-8-14

## 目录

1	2
2	5
3	7

# 1

C 函数返回 struct 类型是如何实现的?

C 函数是如何传入 struct 类型参数的?

请针对讲义“C 与汇编语言-4”中最后两张 slides 的内容,回答上述问题。具体要求:语言描述的同时请画出相应的程序栈的内容示意图。

(假设不考虑直接用寄存器返回/传入小 struct 的情况)

1. 返回 struct 类型: 在函数参数表中加入一个 struct 指针类型,调用者负责在栈上分配空间,把指针传入本函数,本函数填好 struct 的成员后再返回 struct 的指针

c 与汇编代码对照如下:

```
typedef struct{
    int age; int bye; int coo; int ddd; int eee;
} TEST_Struct;

int i = 2;

TEST_Struct __cdecl return_struct(int n) {
    TEST_Struct local_struct;
    local_struct.age = n;
    local_struct.bye = n;
    local_struct.coo = 2 * n;
    local_struct.ddd = n;
    local_struct.eee = n;
    i = local_struct.eee + local_struct.age * 2;
    return local_struct;
}

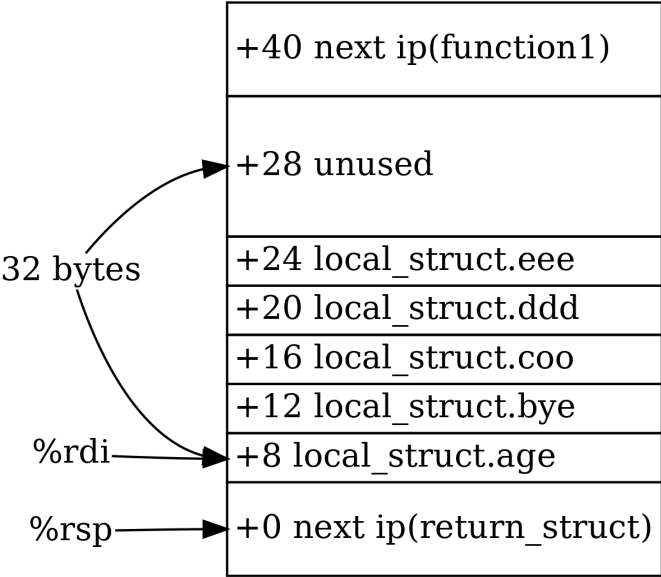
int function1() {
    TEST_Struct main_struct = return_struct(i);
    return 0;
}
```

```
return_struct:
    movq %rdi, %rax ; 设置返回值为传入的 struct 指针
    movl %esi, (%rdi) ; local_struct.age = n
    movl %esi, 4(%rdi) ; local_struct.bye = n
    leal (%rsi,%rsi), %edx ; 2 * n
```

```
movl %edx, 8(%rdi) ; local_struct.coo = 2 * n
movl %esi, 12(%rdi) ; local_struct.ddd = n
movl %esi, 16(%rdi) ; local_struct.eee = n
addl %edx, %esi ; local_struct.age * 2
movl %esi, i(%rip) ; i = local_struct.eee + local_struct.age * 2
ret

function1:
    subq $32, %rsp ; 分配了 32byte 的空间
    movl i(%rip), %esi ; 传第二个参数: n = i
    movq %rsp, %rdi ; 传第一个参数: struct 指针 = 栈顶指针
    call return_struct
    movl $0, %eax
    addq $32, %rsp
    ret
```

在函数 `return_struct` 中，栈的结构如下：



2. 传入 `struct` 类型: 删除参数表中的 `struct` 参数, 调用者负责在栈上分配空间并填好 `struct` 的成员, 本函数中直接根据栈上的偏移量来访问字段

c 与汇编代码对照如下:

```
...
```

```
int input_struct(TEST_Struct in_struct) {
    return in_struct.eee + in_struct.age * 2;
}
```

```
int function2() {
    TEST_Struct main_struct;
    main_struct.age = i;
    main_struct.bye = i;
    main_struct.coo = 2 * i;
    main_struct.ddd = i;
    main_struct.eee = i;
    return input_struct(main_struct);
}
```

input\_struct:

```
movl 8(%rsp), %eax ; in_struct.age
addl %eax, %eax ; in_struct.age * 2
addl 24(%rsp), %eax ; in_struct.eee + in_struct.age * 2
ret
```

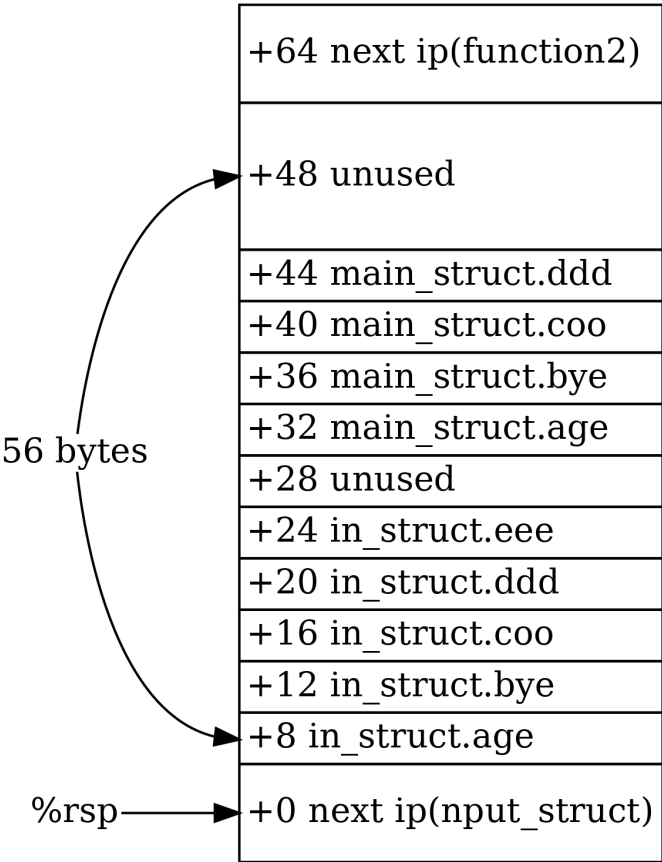
function2:

```
subq $56, %rsp ; 分配 56bytes 的空间
movl i(%rip), %eax ; i
movl %eax, 24(%rsp) ; main_struct.age = i
movl %eax, 28(%rsp) ; main_struct.bye = i
leal (%rax,%rax), %edx ; 2 * i
movl %edx, 32(%rsp) ; main_struct.coo = 2 * i
movl %eax, 36(%rsp) ; main_struct.ddd = i
movq 24(%rsp), %rdx ; rdx = main_struct.bye:main_struct.age
movq %rdx, (%rsp) ; in_struct.bye:in_struct.age = rdx
movq 32(%rsp), %rdx ; rdx = main_struct.ddd:main_struct.coo
movq %rdx, 8(%rsp) ; in_struct.ddd:in_struct.coo = rdx
movl %eax, 16(%rsp) ; in_struct.eee = i
call input_struct
addq $56, %rsp
ret
```

看起来汇编有点冗余，也许这个等级的优化下产生了类似于把 struct 参数复制一遍的行为（但

是又不是真的完全复制了)，其实只要填好参数就行了，main\_struct 可以不用存在。

在函数 nput\_struct 中，栈的结构如下：



2

有如下 C 代码（x86-64 Linux 系统），编译成对应的汇编代码，请对照两边代码给出 M、N 的值。同时请在汇编代码段中标出哪些语句的作用是直接将源结构的字段复制给目标结构（每一个字段的复制指令都需分别标出）。

```
typedef struct typeTAG {
    double attribute_3;
    char attribute_2;
    int attribute_1;
} TAG;

TAG mat1[M][N];
```

```
TAG mat2[N][M];
```

```
int copy_element(int i, int j) {  
    mat1[i][j] = mat2[j][i];  
    return 0;  
}
```

```
copy_element:
```

```
    pushq    %rbp  
    movq     %rsp, %rbp  
    movl     %edi, -4(%rbp) ; -4(rbp) = i  
    movl     %esi, -8(%rbp) ; -8(rbp) = j  
    movl     -8(%rbp), %eax  
    movslq   %eax, %rcx ; rcx = j  
    movl     -4(%rbp), %eax  
    movslq   %eax, %rdx ; rdx = i  
    movq     %rdx, %rax ; rax = i  
    addq     %rax, %rax ; rax = 2 * i  
    addq     %rdx, %rax ; rax = 3 * i  
    salq     $2, %rax ; rax = 12 * i  
    addq     %rdx, %rax ; rax = 13 * i  
    addq     %rcx, %rax ; rax = 13 * i + j  
    salq     $4, %rax ; rax = (13 * i + j) * 16  
    leaq     mat1(%rax), %rcx ; rcx = mat1 + (13 * i + j) * 16  
    movl     -4(%rbp), %eax  
    movslq   %eax, %rsi ; rsi = i  
    movl     -8(%rbp), %eax  
    movslq   %eax, %rdx ; rdx = j  
    movq     %rdx, %rax ; rax = j  
    salq     $3, %rax ; rax = 8 * j  
    subq     %rdx, %rax ; rax = 7 * j  
    addq     %rsi, %rax ; rax = i + 7 * j  
    salq     $4, %rax ; rax = (i + 7 * j) * 16  
    addq     $mat2, %rax ; rax = mat2 + (i + 7 * j) * 16  
    movq     8(%rax), %rdx ; rdx = (mat2 + (i + 7 * j) * 16)[16:8], attribute_2/1  
    movq     (%rax), %rax ; rax = (mat2 + (i + 7 * j) * 16)[8:0], attribute_3  
    movq     %rax, (%rcx) ; 复制 attribute_3
```

```

movq    %rdx, 8(%rcx) ; 复制 attribute_2/1
movl    $0, %eax
popq    %rbp
ret

```

故  $M = 7$ ,  $N = 13$

### 3

源码见附件，这里提供截图和设计说明文字。

#### 1. countln

结果截图如下：

```

countln.asm
84 jne .COUNT_LOOP
85 cmpb $LF, -1(%rax) # t
86 setne %cl # end of file
87 movzbl %cl, %ecx
88 addl %ecx, %ebx
89 .COUNT_DONE:
90 # call open() to open
91 movl $SYS_OPEN, %eax
92 movq 24+SIZEOF_STAT(%r
93 movl $(O_WRONLY | O_CR
94 movl $OPEN_MODE, %edx
95 syscall
96 movl %ebx, %edi
97 movl %eax, %esi # fd2
98 call print_int
99 # call exit()
100 movq $SYS_EXIT, %rax
101 movq $0, %rdi
102 syscall
103 # doesn't need to recd
104

input.txt
1 MashPlant
2 M
3 a
4 s
5 h
6 P
7 l
8 a
9 n
10 t

output_line_num.txt
1 10

```

```

[mashplant@mashplant asm]$ as countln.o -o countln.o -g && ld countln.o -o countln
[mashplant@mashplant asm]$ ./countln input.txt output_line_num.txt
[mashplant@mashplant asm]$

```

简要说明设计思路：

总共编写了两个函数，一个用于输出整数，另一个就是入口函数，处理了所有逻辑。输出整数的函数接受一个 `unsigned int` 值输出，和一个 `int` 文件描述符，将整数值输出到文件描述符指向的文件中，代码如下（实际代码中注释是用英文写的，为了写报告方便这里改用中文）：

```

# print_int(unsigned int value, int fd)
.section .text
.globl print_int
print_int:
    # 把整数 parse 为字符串临时存储在栈上，但不需要分配栈空间，因为占用空间不会超过 red zone
    # 存储方式是，每生成一个字符就让栈向下增长 1byte 来存放它
    movl %edi, %eax
    movl $0, %ecx

```

```

# div 指令不接受立即数作为参数，这是合理的，任何正常的编译器都不会生成除立即数的指令，
# 一定会用其它运算来代替，但是其中原理比较复杂，这里就不实现这个优化了，直接除寄存器即可
movl $10, %edi
.DIV_LOOP:
    movl $0, %edx
    # edx:eax / edi = eax(商) ... edx(余数)
    divl %edi
    addl $ASCII_0, %edx # 除 10 余数 + '0' = 字符
    decq %rcx
    movb %dl, (%rsp, %rcx) # 存放字符
    testl %eax, %eax
    jne .DIV_LOOP
    # 使用系统调用 write() 来输出
    movl $SYS_WRITE, %eax
    movl %esi, %edi
    leaq (%rsp, %rcx), %rsi
    negq %rcx # 每放一个 byte, rcx 就减 1, 所以取负得到长度
    movl %ecx, %edx
    syscall
    ret

```

关于 linux 系统调用的传参，我参考了Linux\_System\_Call\_Table\_for\_x86\_64。此外，根据资料，系统调用只会修改%r11 和 rcx 这两个寄存器。

入口函数 \_start 稍长一些，这里就不贴了。大致逻辑是首先使用系统调用 open() 打开文件，然后使用 mmap() 来读取文件，从我以往的经验来看，这是一种非常高效的读入方法。但是需要解决给 mmap() 传参的问题，平时写的 c 代码大致是：

```

int fd = open(...);
struct stat st;
fstat(fd, &st);
char *a = (char *)mmap(0, st.st_size, PROT_READ, MAP_PRIVATE, fd, 0);

```

这里涉及到了一个开在栈上 struct 和一次 struct 成员访问。我借助 c 代码输出了我的平台上 stat 的大小和 st.st\_size 的偏移量，硬编码到了汇编中。这样也许可移植性不是很好，不过 linux 的兼容性应该比较强，不会有什么问题。

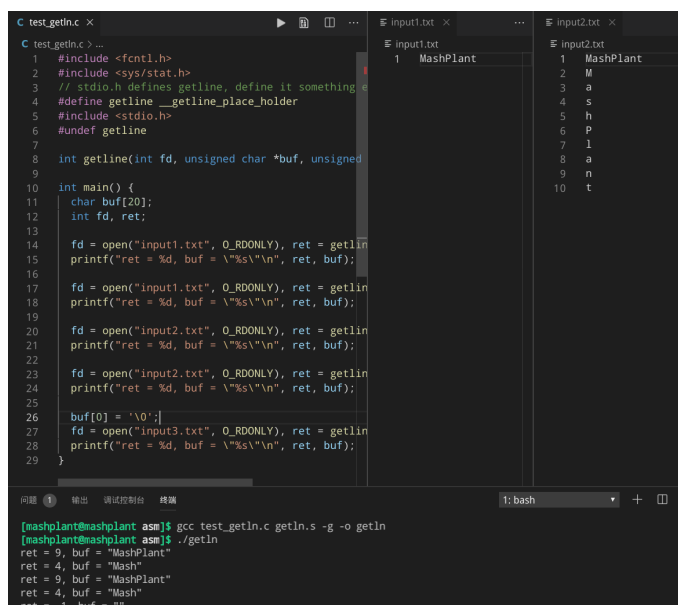
得到了这些信息后就可以直接循环计数了（再利用一次 st.st\_size）。计数结束之后打开另一个文件，调用 print\_int 写入行数，最后调用 exit() 即可。这里应该没有必要把栈和寄



存器之类的东西复原，因为调用 `exit()` 后回到内核态，内存资源都会被回收，这里设置与否没有意义。此外，打开的文件描述符和创建的内存映射应该也都没有回收的必要，因为 `os` 一定会做必要的清理工作。这与平时和 `libc` 打交道不同，如果程序结束时没有关闭 `FILE *` 之类的东西，`libc` 可能有还有没有刷到内核的缓冲区，所以输出可能会出问题，但是这里不经过 `libc` 这一层，所以不必担心这种错误。

## 2. `getline`

结果截图如下：



The screenshot shows a C program named `test_getlin.c` and its execution results. The program uses `getline` to read lines from `input1.txt` and `input2.txt`, and then from a non-existent `input3.txt`. The terminal output shows the program successfully reading lines from the first two files and returning -1 for the third file.

```
C test_getlin.c x
1 #include <fcntl.h>
2 #include <sys/stat.h>
3 // stdio.h defines getline, define it something else
4 #define getline __getline_placeholder
5 #include <stdio.h>
6 #undef getline
7
8 int getline(int fd, unsigned char *buf, unsigned
9
10 int main() {
11     char buf[20];
12     int fd, ret;
13
14     fd = open("input1.txt", O_RDONLY), ret = getlin
15     printf("ret = %d, buf = \"%s\"\n", ret, buf);
16
17     fd = open("input1.txt", O_RDONLY), ret = getlin
18     printf("ret = %d, buf = \"%s\"\n", ret, buf);
19
20     fd = open("input2.txt", O_RDONLY), ret = getlin
21     printf("ret = %d, buf = \"%s\"\n", ret, buf);
22
23     fd = open("input2.txt", O_RDONLY), ret = getlin
24     printf("ret = %d, buf = \"%s\"\n", ret, buf);
25
26     buf[0] = '\0';
27     fd = open("input3.txt", O_RDONLY), ret = getlin
28     printf("ret = %d, buf = \"%s\"\n", ret, buf);
29 }
```

```
[mashplant@mashplant asm]$ gcc test_getlin.c getlin.s -g -o getlin
[mashplant@mashplant asm]$ ./getlin
ret = 9, buf = "MashPlant"
ret = 4, buf = "Mash"
ret = 9, buf = "MashPlant"
ret = 4, buf = "Mash"
ret = -1, buf = ""
```

我使用了 `c` 语言来编写测试，进一步证明了编写的汇编代码是符合调用约定的。其中的 `input3.txt` 是一个不存在的文件，`getline` 正确地返回了-1(并且不会往 `buf` 内写任何内容)。

代码中注释比较清楚了，这里就简单说明思路。首先判断传入的 `size`，若为 0 则不做任何事，若为 1 则填一个 0。若 `size` 大于 1，则进入读入的循环，每次都使用系统调用 `read()` 读取一个字符，如果出错或者读到 `LF` 就返回，但二者的处理不同，前者返回了-1，并且没有填上末尾的 0(可以认为这次函数调用失败了，结果都没有意义)；后者会填上 0 后正常返回。读入至多 `size - 1` 个字符后结束，填上 0 后正常返回。