HW8

1

1. 教材6.6 下面是C语言两个函数f和g的概略（它们不再有其它的局部变量）：

```
1.  int f (int x) { int i; …return i +1; … }
2.  int g (int y) {int j; … f (j +1); … }
```

请按照图6.11的形式，画出函数g调用f，f的函数体正在执行时，活动记录栈的内容及相关信息，并按图6.10左侧箭头方式画出控制链。假定函数返回值是通过寄存器传递的。

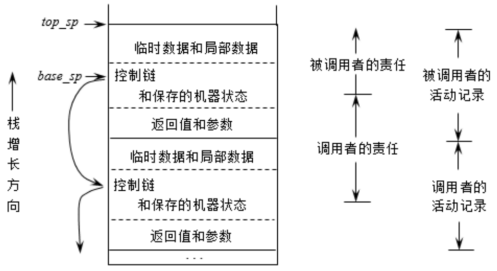


图 6.10 调用者和被调用者之间的任务划分

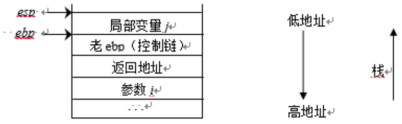
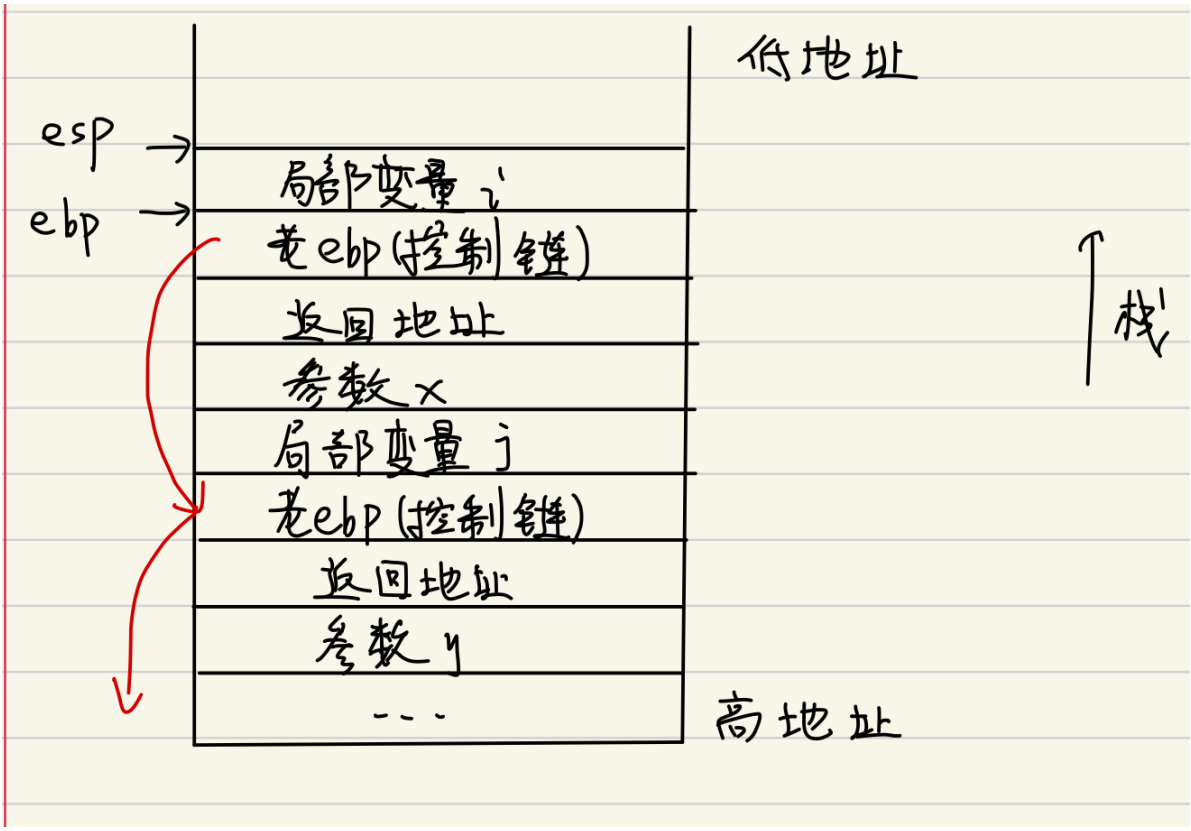


图 6.11 活动记录的内容及相关信息



2

2. 教材6.18 下面是一个C语言程序：

```c
#include <stdio.h>
int main() {
    long i;
    long a[0][4];
    long j;
    i = 4; j = 8;
    printf("%ld, %d\n", sizeof(a), a[0][0]);
}
```

虽然出现long a[0][4]这样的声明，但在x86/Linux系统上，用编译器GCC 7.5.0 (Ubuntu 7.5.0-3ubuntu1~16.04)编译时，该程序能够通过编译并生成目标代码。请在你自己的机器上实验，回答下面两个问题（说明你使用的编译器及版本并给出汇编码）：
(a) sizeof(a)的值是多少，请说明理由。
(b) a[0][0]的值是多少，请说明理由。

- `gcc version 9.3.0 (Ubuntu 9.3.0-17ubuntu1~20.04)`
- 汇编码

```
    .file   "2.c"
    .text
    .section    .rodata
.LC0:
    .string "%ld, %d\n"
    .text
    .globl  main
    .type   main, @function
main:
.LFB0:
    .cfi_startproc
    endbr64
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $32, %rsp
    movq    %fs:40, %rax
    movq    %rax, -8(%rbp)
    xorl    %eax, %eax
    movq    $4, -32(%rbp)   #i
    movq    $8, -24(%rbp)   #j
    movq    -16(%rbp), %rax #数组a在-16(%rbp)的位置
    movq    %rax, %rdx
    movl    $0, %esi
    leaq    .LC0(%rip), %rdi
    movl    $0, %eax
    call    printf@PLT
    movl    $0, %eax
    movq    -8(%rbp), %rcx
    xorq    %fs:40, %rcx
    je  .L3
    call    __stack_chk_fail@PLT
.L3:
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size   main, .-main
    .ident  "GCC: (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0"
    .section    .note.GNU-stack,"",@progbits
```

```
    .section    .note.gnu.property,"a"
    .align 8
    .long    1f - 0f
    .long    4f - 1f
    .long    5
0:
    .string  "GNU"
1:
    .align 8
    .long    0xc0000002
    .long    3f - 2f
2:
    .long    0x3
3:
    .align 8
4:
```

- 多次执行，结果如下

```
root@LAPTOP-HRJHHKLT:/mnt/d/mywork/compiler_lab/hw8# ./2
0, 1970767920
root@LAPTOP-HRJHHKLT:/mnt/d/mywork/compiler_lab/hw8# ./2
0, -386038272
root@LAPTOP-HRJHHKLT:/mnt/d/mywork/compiler_lab/hw8# ./2
0, 2096756816
root@LAPTOP-HRJHHKLT:/mnt/d/mywork/compiler_lab/hw8# ./2
0, -899516208
```

1. `sizeof(a)`的值为0，根据`sizeof`针对数组的计算公式，得到

   $0 \times 4 \times sizeof(int) = 0$

2. `a[0][0]`的值每次都不一样。

   原因：虽然`sizeof(a) = 0`，但是a数组仍有起始地址，且`a[0][0]`的地址即为起始地址，这由汇编码也可以看出

   ```
   movq    -16(%rbp), %rax #数组a在-16(%rbp)的位置
   ```

   a的起始地址位于-16（%rbp）的位置。而由于该地址对应的值未被初始化，所以访问`a[0][0]`得到的值是未定义的值

3

```
int main() {
 int i,j;
 while ( (i || j) && (j > 5) ) {
     i = j;
 }
}
```

```
 .file   "ex7-9.c"
     .text
     .globl  main
     .type   main, @function
main:
.LFB0:
     pushq   %rbp        #rbp入栈
     movq    %rsp, %rbp #将当前的栈顶指针放入rbp中
     jmp     .L2         #跳转到L2
.L5:
     movl    -4(%rbp), %eax #eax = j
     movl    %eax, -8(%rbp) #i = j
.L2:
     cmpl    $0, -8(%rbp)    #计算i的布尔值
     jne     .L3             #如果i为真，则跳转到L3
     cmpl    $0, -4(%rbp)    #否则，计算j的布尔值
     je      .L4             #如果j为真，则跳转到L3，否则跳转到L4
.L3:
     cmpl    $5, -4(%rbp)    #计算j > 5的布尔值
     jg      .L5             #如果j > 5为真，则跳转到L5
.L4:
     movl    $0, %eax    #eax = 0
     popq    %rbp        #rbp出栈
     ret                 #返回
.LFE0:
     .size   main, .-main
     .ident  "GCC: (Ubuntu 7.5.0-3ubuntu1~16.04) 7.5.0"
```

可以看到，对于`(i || j) && (j > 5)`，首先会计算 i 的布尔值，如果 i 为真则不用计算 j 的布尔值，直接计算 j > 5的布尔值；只有当 i 为假时才需要计算 j 的布尔值，如果 j 为真则需要进一步计算 j > 5 的布尔值；若 j 为加则不用计算 j > 5的布尔值，整个条件肯定为假

上述过程体说明了C语言确实是用短路计算方式来完成布尔表达式计算的。

4

4. 对于如下C程序

```
1. int main()
2. {
3.     char *cp1, *cp2;
4.
5.     cp1 = "12345";
6.     cp2 = "abcdefghij";
7.     strcpy(cp1,cp2);
8.     printf("cp1 = %s\ncp2 = %s\n", cp1, cp2);
9. }
```

1）在某些系统上的运行结果是：

```
1.  cp1 = abcdefghij
2.  cp2 = ghij
```

为什么cp2所指的串被修改了？
2）在某些系统上运行会输出段错误，为什么？

1. 字符串常量"12345"和"abcdefghij"在`.data`区（即常数区）是连续存放的，执行如下两条语句后，

```
cp1 = "12345";
cp2 = "abcdefghij";
```

存放如下（按照两个字符串在程序中出现的先后次序）

```
1 2 3 4 5 \0 a b c d e  f g h i j \0
```

其中cp1的值为1所在的地址，cp2的值为a所在的地址

执行如下`strcpy(cp1,cp2)`后，存放如下

```
a b c d e f g h i j \0 f g h i j \0
```

cp1、cp2所指向的地址并没有发生变化，当前cp2指向第一个g，故输出cp2得到的是字符串“ghij”(`\0`标志当前字符串结束)

2. 某些系统的编译器会把程序中的字符串常量单独分配在一个段中（一般位于`.rodata`只读数据区），将它们和其他常数分开，且该段的内容在程序运行时不能被修改（权限为只读)。故在执行`strcpy`时，会输出段错误

5

```
#include <stdio.h>
void funcOld(i,j,f,e)
short i, j; float f, e;
{
 short i1,j1; float f1,e1;
 printf("%p, %p, %p, %p\n", &i,&j,&f,&e);
 printf("%p, %p, %p, %p\n", &i1,&j1,&f1,&e1);
}
void func(short i, short j, float f, float e)
{
 short i1,j1; float f1,e1;
 printf("%p, %p, %p, %p\n", &i,&j,&f,&e);
 printf("%p, %p, %p, %p\n", &i1,&j1,&f1,&e1);
}
int main()
{
 short i,j; float f,e;
 func(i,j,f,e);
 funcOld(i,j,f,e);
}
```

注意：%p 常用于指针类型，用于输出一个指针的地址

`funcOld`的参数声明方式是一种古老的K&R风格，较为特殊，有些现代编译器已经不支持这种风格

输出如下

```
0x7fffbec47f1c, 0x7fffbec47f18, 0x7fffbec47f14, 0x7fffbec47f10
0x7fffbec47f2c, 0x7fffbec47f2e, 0x7fffbec47f30, 0x7fffbec47f34
0x7fffbec47f1c, 0x7fffbec47f18, 0x7fffbec47f10, 0x7fffbec47f08
0x7fffbec47f2c, 0x7fffbec47f2e, 0x7fffbec47f30, 0x7fffbec47f34
```

在我的电脑上的输出（和上面类似，只是地址不同）（版本：`gcc version 9.3.0 (Ubuntu 9.3.0-17ubuntu1~20.04)`）

```
0x7ffd0df30c1c, 0x7ffd0df30c18, 0x7ffd0df30c14, 0x7ffd0df30c10
0x7ffd0df30c2c, 0x7ffd0df30c2e, 0x7ffd0df30c30, 0x7ffd0df30c34
0x7ffd0df30c1c, 0x7ffd0df30c18, 0x7ffd0df30c10, 0x7ffd0df30c08
0x7ffd0df30c2c, 0x7ffd0df30c2e, 0x7ffd0df30c30, 0x7ffd0df30c34
```

汇编代码

```
    .file   "5.c"
    .text
    .section    .rodata
.LC0:
    .string "%p, %p, %p, %p\n"
    .text
    .globl  funcOld
    .type   funcOld, @function
funcOld:
.LFB0:
    .cfi_startproc
    endbr64
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $64, %rsp
    movl    %edi, %edx
    movl    %esi, %eax
    movw    %dx, -36(%rbp)
    movw    %ax, -40(%rbp)
    cvtsd2ss    %xmm0, %xmm0
    movss   %xmm0, -48(%rbp)
    cvtsd2ss    %xmm1, %xmm0
    movss   %xmm0, -56(%rbp)
    movq    %fs:40, %rax
    movq    %rax, -8(%rbp)
    xorl    %eax, %eax
    leaq    -56(%rbp), %rsi
    leaq    -48(%rbp), %rcx
    leaq    -40(%rbp), %rdx
    leaq    -36(%rbp), %rax
    movq    %rsi, %r8
    movq    %rax, %rsi
    leaq    .LC0(%rip), %rdi
    movl    $0, %eax
    call    printf@PLT
    leaq    -12(%rbp), %rsi
    leaq    -16(%rbp), %rcx
    leaq    -18(%rbp), %rdx
    leaq    -20(%rbp), %rax
    movq    %rsi, %r8
    movq    %rax, %rsi
    leaq    .LC0(%rip), %rdi
    movl    $0, %eax
```

```
    call    printf@PLT
    nop
    movq    -8(%rbp), %rax
    xorq    %fs:40, %rax
    je  .L2
    call    __stack_chk_fail@PLT
.L2:
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size   funcOld, .-funcOld
    .globl  func
    .type   func, @function
func:
.LFB1:
    .cfi_startproc
    endbr64
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $48, %rsp
    movl    %edi, %edx
    movl    %esi, %eax
    movss   %xmm0, -44(%rbp)
    movss   %xmm1, -48(%rbp)
    movw    %dx, -36(%rbp)
    movw    %ax, -40(%rbp)
    movq    %fs:40, %rax
    movq    %rax, -8(%rbp)
    xorl    %eax, %eax
    leaq    -48(%rbp), %rsi
    leaq    -44(%rbp), %rcx
    leaq    -40(%rbp), %rdx
    leaq    -36(%rbp), %rax
    movq    %rsi, %r8
    movq    %rax, %rsi
    leaq    .LC0(%rip), %rdi
    movl    $0, %eax
    call    printf@PLT
    leaq    -12(%rbp), %rsi
    leaq    -16(%rbp), %rcx
    leaq    -18(%rbp), %rdx
    leaq    -20(%rbp), %rax
    movq    %rsi, %r8
    movq    %rax, %rsi
    leaq    .LC0(%rip), %rdi
    movl    $0, %eax
    call    printf@PLT
    nop
    movq    -8(%rbp), %rax
    xorq    %fs:40, %rax
    je  .L4
    call    __stack_chk_fail@PLT
.L4:
```

```
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE1:
    .size   func, .-func
    .globl  main
    .type   main, @function
main:
.LFB2:
    .cfi_startproc
    endbr64
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $16, %rsp
    movswl  -12(%rbp), %edx
    movswl  -10(%rbp), %eax
    movss   -4(%rbp), %xmm0
    movl    -8(%rbp), %ecx
    movaps  %xmm0, %xmm1
    movd    %ecx, %xmm0
    movl    %edx, %esi
    movl    %eax, %edi
    call    func
    cvtss2sd    -4(%rbp), %xmm1
    cvtss2sd    -8(%rbp), %xmm0
    movswl  -12(%rbp), %edx
    movswl  -10(%rbp), %eax
    movl    %edx, %esi
    movl    %eax, %edi
    movl    $2, %eax
    call    funcOld
    movl    $0, %eax
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE2:
    .size   main, .-main
    .ident  "GCC: (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0"
    .section    .note.GNU-stack,"",@progbits
    .section    .note.gnu.property,"a"
    .align 8
    .long    1f - 0f
    .long    4f - 1f
    .long    5
0:
    .string  "GNU"
1:
    .align 8
    .long    0xc0000002
    .long    3f - 2f
2:
    .long    0x3
3:
```

```
    .align 8
4:
```

对比二者的汇编码，可以看出二个函数第二个`printf`的汇编码是完全一样的，这也从输出结果中得到了印证。主要区别在于第一个`printf`部分

`funcOld`

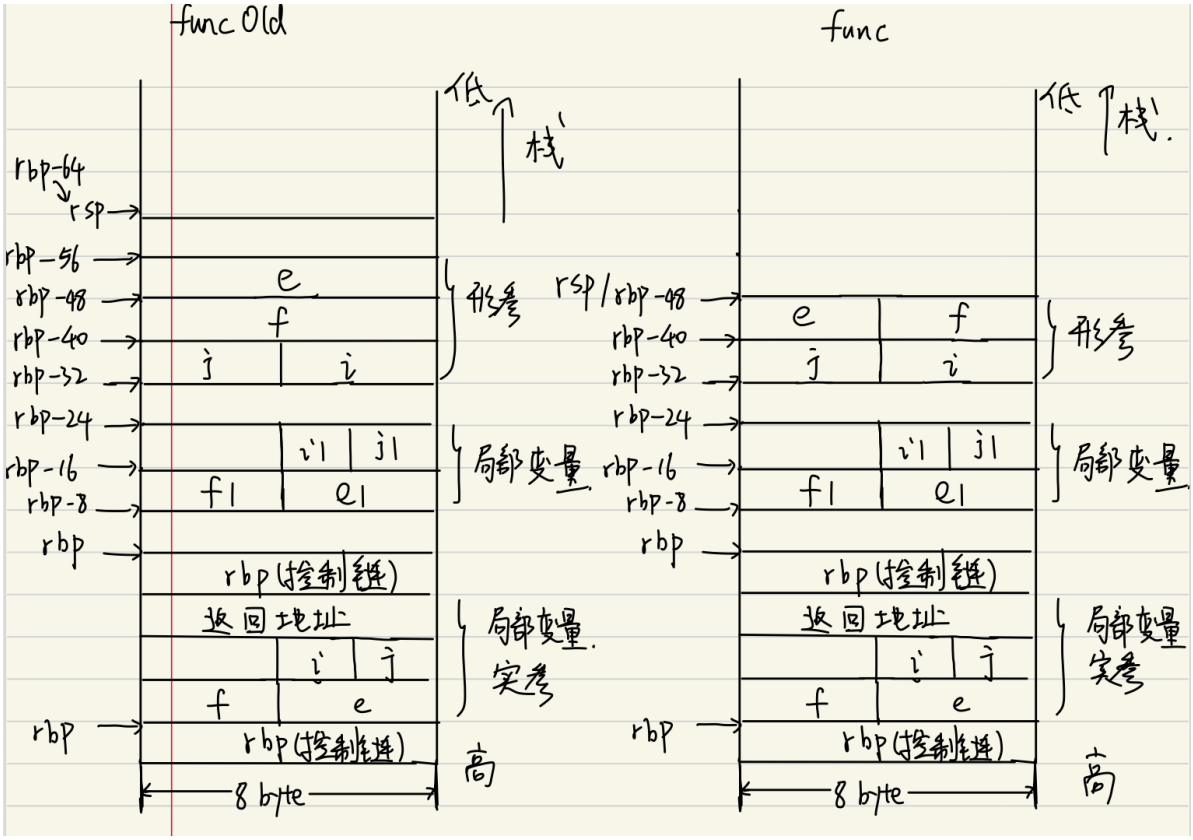
```
    subq    $64, %rsp
    movl    %edi, %edx
    movl    %esi, %eax
    movw    %dx, -36(%rbp)
    movw    %ax, -40(%rbp)
    cvtsd2ss    %xmm0, %xmm0
    movss   %xmm0, -48(%rbp)
    cvtsd2ss    %xmm1, %xmm0
    movss   %xmm0, -56(%rbp)
    movq    %fs:40, %rax
    movq    %rax, -8(%rbp)
    xorl    %eax, %eax
    leaq    -56(%rbp), %rsi
    leaq    -48(%rbp), %rcx
    leaq    -40(%rbp), %rdx
    leaq    -36(%rbp), %rax
    movq    %rsi, %r8
    movq    %rax, %rsi
    leaq    .LC0(%rip), %rdi
    movl    $0, %eax
    call    printf@PLT
```

`func`

```
    subq    $48, %rsp
    movl    %edi, %edx
    movl    %esi, %eax
    movss   %xmm0, -44(%rbp)
    movss   %xmm1, -48(%rbp)
    movw    %dx, -36(%rbp)
    movw    %ax, -40(%rbp)
    movq    %fs:40, %rax
    movq    %rax, -8(%rbp)
    xorl    %eax, %eax
    leaq    -48(%rbp), %rsi
    leaq    -44(%rbp), %rcx
    leaq    -40(%rbp), %rdx
    leaq    -36(%rbp), %rax
    movq    %rsi, %r8
    movq    %rax, %rsi
    leaq    .LC0(%rip), %rdi
    movl    $0, %eax
    call    printf@PLT
```

其中，movb（8位）、movw（16位）、movl（32位）、movq（64位）

从汇编码可以得到如下内存布局



即编译器对于两种传参的处理不同，主要是对形参空间的分配不同

- 分配策略
  - i j f e作为形参，在栈中是从高地址到低地址布局的
    - `func` 中给 short 和 float 的形参都分配了4个字节的空间
    - `funcOld` 中给short类型的形参分配了4个字节的空间，给float类型的形参分配了8个字节的空间
  - i1 j1 f1 e1作为局部变量，在栈中是从低地址到高地址布局的
    - 两个函数对于内部声明的局部变量空间的分配策略是相同的，即为short类型的局部变量分配2个字节，为float类型的局部变量分配4个字节
- 上述分配策略导致了如下三种不同的输出

```
0x7fffbec47f1c, 0x7fffbec47f18, 0x7fffbec47f14, 0x7fffbec47f10
0x7fffbec47f2c, 0x7fffbec47f2e, 0x7fffbec47f30, 0x7fffbec47f34
0x7fffbec47f1c, 0x7fffbec47f18, 0x7fffbec47f10, 0x7fffbec47f08
0x7fffbec47f2c, 0x7fffbec47f2e, 0x7fffbec47f30, 0x7fffbec47f34(和第2行相同)
```

第一行地址依次减小，且均相差4字节

第三行地址依次减小，分别相差4字节、8字节、8字节

第二、四行地址依次增大，分别相差2字节、2字节、4字节