编译原理　第十三周作业　12 月 10 日　周四

PB18151866 龚小航

6.5 一个 C 语言程序如下:

```
typedef struct_a{
    short i;
    short j;
    short l;
}a;
typedef struct_b{
    long i;
    short k;
}b;
main(){
    printf("Size of short, long,a and b=%d,%d,%d,%d\n",
      sizeof(short),sizeof(long),sizeof(a),sizeof(b));
}
```

该程序在 Ubuntu 12.04.2 LTS(GNU/Linux 3.2.0 − 42 − generic x86_64) 系统上，经过编译器 GCC: (Ubuntu/Linaro 4.6.3 − 1 ubuntu5) 4.6.3 编译后，运行结果如下:

$$\text{Size of short, long, a and b} = 2, 8, 6, 16$$

已知 short 类型和 long 类型分别对齐到 2 的倍数和 8 的倍数。试问，为什么类型 b 的 size 会等于 16?

解: 要判断一个结构体所占的空间大小，大体来说分三步走:

1. 先确定实际对齐单位，其由以下三个因素决定

(1) CPU 周期

WIN vs qt 默认 8 字节对齐

Linux 32 位 默认 4 字节对齐，64 位默认 8 字节对齐

(2) 结构体最大成员(基本数据类型变量)

(3) 预编译指令#pragma pack(n)手动设置　　n--只能填 1 2 4 8 16

上面三者取最小的,就是实际对齐单位

2. 除结构体的第一个成员外，其他所有的成员的地址相对于结构体地址(即它首个成员的地址)的偏移量必须为实际对齐单位或自身大小的整数倍(取两者中小的那个)

3. 结构体的整体大小必须为实际对齐单位的整数倍。

结合本例，先确定实际对齐单位，Linux 64 位机默认对齐到 8 字节，而结构体 a 最大成员为 2 字节，结构体 b 最大成员为 8 字节，因此结构体 a 实际对齐单位 2 字节，结构体 b 实际对齐单位 8 字节；再根据对齐规则作出 a, b 内存中所占空间的示意图:

a	i_l	i_h	j_l	j_h	l_l	l_h												
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
b	i_7	i_6	i_5	i_4	i_3	i_2	i_1	i_0	k_l	k_h	p	p	p	p	p	p			

其中 p 表示衬垫区。由规则 2，结构体 a 的每个成员都必须从 2 的整数倍地址开始存储，结构体 b 的 short k 成员必须从地址 8 开始存储。由规则 3，结构体 a 的整体大小必须为 2 的整数倍，结构体 b 的大小必须为 8 的整数倍。因此由图可知 $\text{size(a)} = 6, \text{size(b)} = 16$

需要字节对齐的根本原因在于 CPU 访问数据的效率问题。现代计算机中内存空间都是按照 Byte 划分的，从理论上讲似乎对任何类型的变量的访问可以从任何地址开始，但实际情况是在访问特定类型变量的时候经常在特定的内存地址访问。各个硬件平台对存储空间的处理上有很大的不同。一些平台对某些特定类型的数据只能从某些特定地址开始存取。比如有些架构的 CPU 在访问一个没有进行对齐的变量的时候会发生错误,那么在这种架构下编程必须保证字节对齐.其他平台可能没有这种情况，但是最常见的是如果不按照适合其平台要求对数据存放进行对齐，会在存取效率上带来损失。比如有些平台每次读都是从偶地址开始，如果一个 int 型（假设为 32 位系统）如果存放在偶地址开始的地方，那么一个读周期就可以读出这 32bit，而如果存放在奇地址开始的地方，就需要 2 个读周期，并对两次读出的结果的高低字节进行拼凑才能得到该 32bit 数据。显然在读取效率上下降很多。

6.6 下面是C语言两个函数 f 和 g 的概略（它们不再有其他的局部变量）：

```
int f(int x){int i;...return i + 1;...}
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。

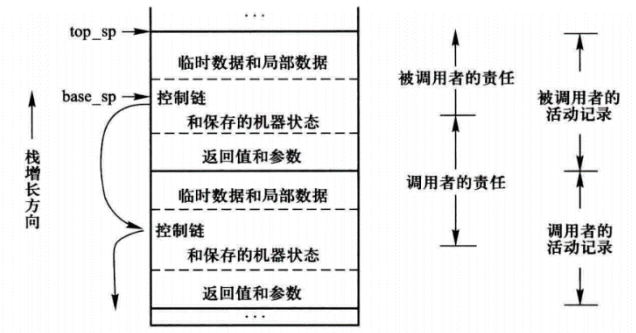
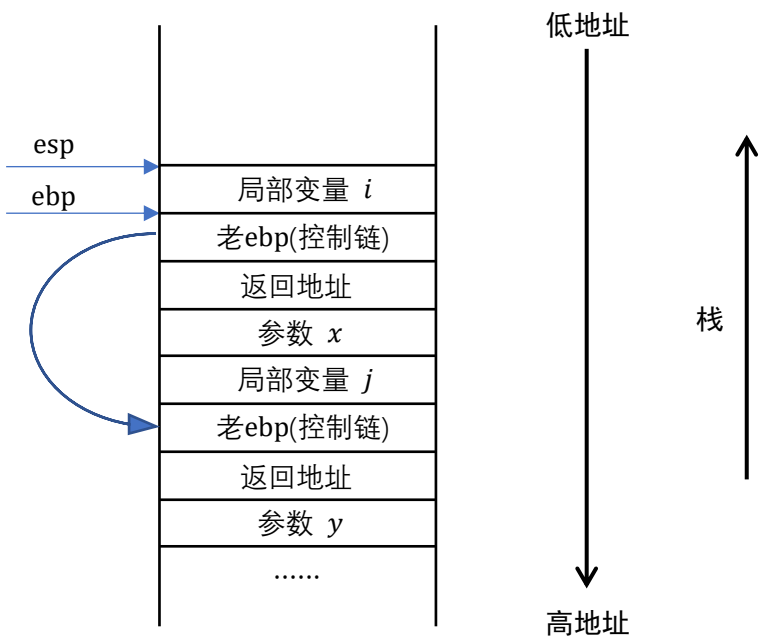


图 6.10 调用者和被调用者之间的任务划分

图 6.11 活动记录的内容及相关信息

解：参考图 6-10，6-11，即可画出活动记录栈的内容及相关信息。



6.9 C语言函数f的定义如下：

```
int f(int x, int *py, int **ppz){
    **ppz+=1;*py+=2;x+=3;return x+*py+***ppz;
}
```

变量 a 是指向 b 的指针，变量 b 是指向 c 的指针，c 是整型变量并且当前值是4。那么执行 $f(c,b,a)$的返回值是多少？

解： 由题意，$c=4$，$b=\&c$，$a=\&b=\&\&c$，x 为形参，作用域与生存周期仅在函数中。

在原程序中标出各条语句的执行结果，如下所示：

```
int f(int x(=4), int *py(*&c), int **ppz(**&&c)){
    **ppz+=1;*py+=2;x+=3;return x+*py+***ppz;
    (c=c+1) (c=c+2) (x=x+3)     (7)+(7)+(7)
}
```

可见最后返回值为 21，局部变量 x 在调用结束后消亡。

6.10 一个 C 语言程序如下：

```
func(i1, i2, i3)long i1,i2,i3;{
    long j1,j2,j3;
    printf("Addresses of i1,i2,i3=%o,%o,%o\n",&i1,&i2,&i3);
    printf("Addresses of j1,j2,j3=%o,%o,%o\n",&j1,&j2,&j3);
}
main(){
    long i1,i2,i3;
    func(i1,i2,i3);
}
```

该程序在 x86/Linux 系统上，经某编译器编译后的运行结果如下：

```
Addresses of i1,i2,i3=27777775460,27777775464,27777775470
Addresses of j1,j2,j3=27777775444,27777775440,27777775434
```

从上面的结果可以看出，func 函数的三个形式参数的地址依次升高，而三个局部变量的地址依次降低。试说明为什么会有这个区别。注意，输出的数据是八进制的。

解：函数调用需要的信息存储在活动记录栈上。由图 6-11，局部变量存储在低地址，而参数存储在高地址，因此总体上 i_1, i_2, i_3 的地址较高；而存储参数和局部变量的过程都是依次入栈的过程，它们之间的区别是 C 语言编译器处理局部变量声明时是从左到右处理，而处理传入参数时是从右到左处理。这一点就体现在入栈顺序上，因此 i, j 的地址增长方向相反。

图 6.11 活动记录的内容及相关信息

6.13 一个 C 语言程序如下：

```
int fact(i)int i;{
    if(i==0)
        return 1;
    else
        return i*fact(i-1);
}
main(){
    printf("%d\n",fact(5));
    printf("%d\n",fact(5,10,15));
    printf("%d\n",fact(5.0));
    printf("%d\n",fact());
}
```

该程序在 x86/Linux 系统上，用编译器 GCC:(GUN)egcs-2.91.66 19990314/Linux(egcs-1.2.2 release)编译后，运行结果如下：

```
120
120
1
Segmentation fault(core dumped)
```

请解释下面问题：

(***a***) 第二个 fact 调用：结果为什么没有受参数过多的影响?

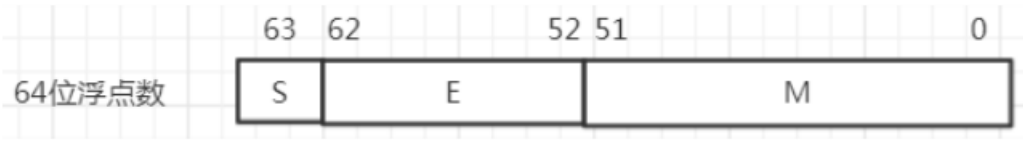
(***b***) 第三个 fact 调用：为什么用浮点数 5.0 作为参数时结果变成 1?

(***c***) 第四个 fact 调用：为什么没有提供参数时会出现Segmentation fault?

解：对三问分别分析：

(***a***) C 语言编译器对函数调用的参数采取从右到左处理的策略。在上一题中可以看到参数的地址是递增的，而栈的增长方向是从高地址向低地址的，因此参数 “5” 在栈顶，函数调用过程根据定义的 int i 从栈顶取四字节的数据并将它解释成整形数据，此时恰好取到的是 5.

(***b***) intel 系列 CPU 通常采用小端法管理存储空间，即高位数据存在高地址，低位数据存储在低地址。



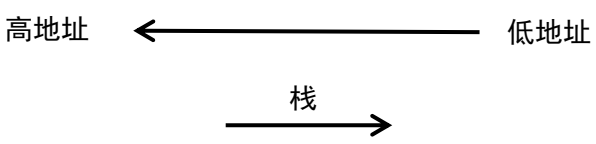
C 语言默认的浮点类型是 double 型，占用 8 字节空间。64 位浮点型数据格式如下：

其中 S 为符号位，E 为阶码，表示 2^E，M 为尾数，是数据的有效数字部分。

上述程序将双精度浮点数 5.0 压入栈中，如下所示：

0100 0000 0001 0100 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

栈顶 4 字节数据



由示意图可知，双精度 5.0 存入栈时，栈顶的四个字节都为0，因此函数调用取到的参数为 0. 按函数逻辑，输出为 1

(***c***) 当不提供任何参数时，函数调用仍然会从活动记录栈的栈顶取 4 字节长的数据并将它认定为整形数据。由于没有预先存入立即数或表达式，这个栈顶数据是不可知不可控的。此时编译器或动态分析器可以认为这个访问是越权的，即访问到了不属于它的内存空间，对于系统是及其危险的。另一方面，若动态检查不报错，即系统允许这种危险的访问，那么函数取到的是一个不可知的 4 字节数据。按 int 能表示的范围来看，大概 20! 就已经溢出了，而 int 型能表示的数据共有 2^{32} 个，取到不报段错误的数据的概率大约为 $20/2^{32} \approx 0.0000000046566$，几乎是不可能事件。