



山东大学  
SHANDONG UNIVERSITY

## 编译原理

# 第六章 运行时存储空间组织

授 课 教 师 : 郑艳伟  
手 机 : 18614002860 (微信同号)  
邮 箱 : zhengyw@sdu.edu.cn

## 第六章 运行时存储空间组织

### □ 6.1 目标程序运行时的活动

- 6.1.1 运行时存储空间访问
- 6.1.2 栈帧结构
- 6.1.3 存储空间分配策略

### □ 6.2 过程调用规范

- 6.2.1 高级程序参数传递
- 6.2.2 std call
- 6.2.3 C调用规范
- 6.2.4 x64调用规范
- 6.2.5 寄存器保护
- 6.2.6 地址计算
- 6.2.7 ARM规范

### □ 6.3 运行时库

- 6.3.1 使用C运行时库输入输出
- 6.3.2 编译器生成输入输出代码

### ➤ 6.3.3 幂运算

### ➤ 6.3.4 跨文件调用

### ➤ 6.3.5 封装库

### □ 6.4 嵌套过程栈帧结构

### ➤ 6.4.1 静态链

### ➤ 6.4.2 静态链构建

### ➤ 6.4.3 外层变量访问

### ➤ 6.4.4 嵌套层次显示表

### ➤ 6.4.5 Display表构建

### ➤ 6.4.6 通过display访问变量

### □ 6.5 堆式存储分配

### ➤ 6.5.1 定长块管理

### ➤ 6.5.2 保留元数据

### ➤ 6.5.3 变长块管理

### ➤ 6.5.4 存储回收

## 第六章 运行时存储空间组织

- **运行时存储空间组织**：代码运行时刻，源代码中的各种变量、常量等用户定义的量是如何存放的，如何去访问它们？
  - 在程序语言中，程序中使用的**存储单元都由标识符表示**，它们对应的**内存地址**由编译程序在**编译时**或由其生成的目标程序在**运行时**分配；
  - 存储组织与管理，就是**将标识符和存储单元关联起来**，进行存储分配、访问和释放。

## 第六章 运行时存储空间组织

### □ 6.1 目标程序运行时的活动

#### ➤ 6.1.1 运行时存储空间访问

#### ➤ 6.1.2 栈帧结构

#### ➤ 6.1.3 存储空间分配策略

### □ 6.2 过程调用规范

#### ➤ 6.2.1 高级程序参数传递

#### ➤ 6.2.2 std call

#### ➤ 6.2.3 C调用规范

#### ➤ 6.2.4 x64调用规范

#### ➤ 6.2.5 寄存器保护

#### ➤ 6.2.6 地址计算

#### ➤ 6.2.7 ARM规范

### □ 6.3 运行时库

#### ➤ 6.3.1 使用C运行时库输入输出

#### ➤ 6.3.2 编译器生成输入输出代码

#### ➤ 6.3.3 幂运算

#### ➤ 6.3.4 跨文件调用

#### ➤ 6.3.5 封装库

### □ 6.4 嵌套过程栈帧结构

#### ➤ 6.4.1 静态链

#### ➤ 6.4.2 静态链构建

#### ➤ 6.4.3 外层变量访问

#### ➤ 6.4.4 嵌套层次显示表

#### ➤ 6.4.5 Display表构建

#### ➤ 6.4.6 通过display访问变量

### □ 6.5 堆式存储分配

#### ➤ 6.5.1 定长块管理

#### ➤ 6.5.2 保留元数据

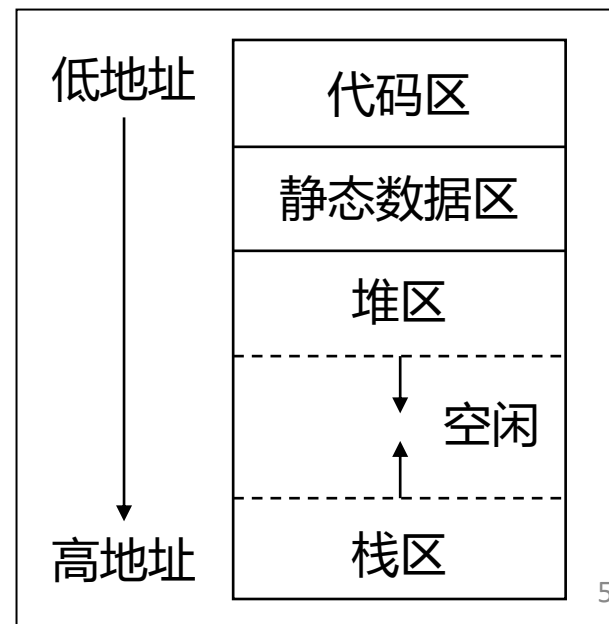
#### ➤ 6.5.3 变长块管理

#### ➤ 6.5.4 存储回收

## 运行时存储空间访问

### 不同区域的访问方式有明显区别

- 对**代码区**，一般是转移指令跳转到某个位置继续执行，可以通过建立标签（label），通过无条件转移指令或条件转移指令实现跳转。
- 对**静态数据区**，可以为每个数据指定一个名字（如程序员写的名字），通过“offset 变量名”可以得到该变量地址。
- 对**栈区动态数据**，一般通过push 和pop 指令进行操作，也可以通过EBP 和ESP 加上一个偏移量访问一个数据。
- 对**堆区动态数据**，一般动态申请分配空间。  
但对堆区的管理，需要知道堆区的起始地址。



## 第六章 运行时存储空间组织

### □ 6.1 目标程序运行时的活动

- 6.1.1 运行时存储空间访问
- 6.1.2 栈帧结构
- 6.1.3 存储空间分配策略

### □ 6.2 过程调用规范

- 6.2.1 高级程序参数传递
- 6.2.2 std call
- 6.2.3 C调用规范
- 6.2.4 x64调用规范
- 6.2.5 寄存器保护
- 6.2.6 地址计算
- 6.2.7 ARM规范

### □ 6.3 运行时库

- 6.3.1 使用C运行时库输入输出
- 6.3.2 编译器生成输入输出代码

### ➤ 6.3.3 幂运算

### ➤ 6.3.4 跨文件调用

### ➤ 6.3.5 封装库

### □ 6.4 嵌套过程栈帧结构

### ➤ 6.4.1 静态链

### ➤ 6.4.2 静态链构建

### ➤ 6.4.3 外层变量访问

### ➤ 6.4.4 嵌套层次显示表

### ➤ 6.4.5 Display表构建

### ➤ 6.4.6 通过display访问变量

### □ 6.5 堆式存储分配

### ➤ 6.5.1 定长块管理

### ➤ 6.5.2 保留元数据

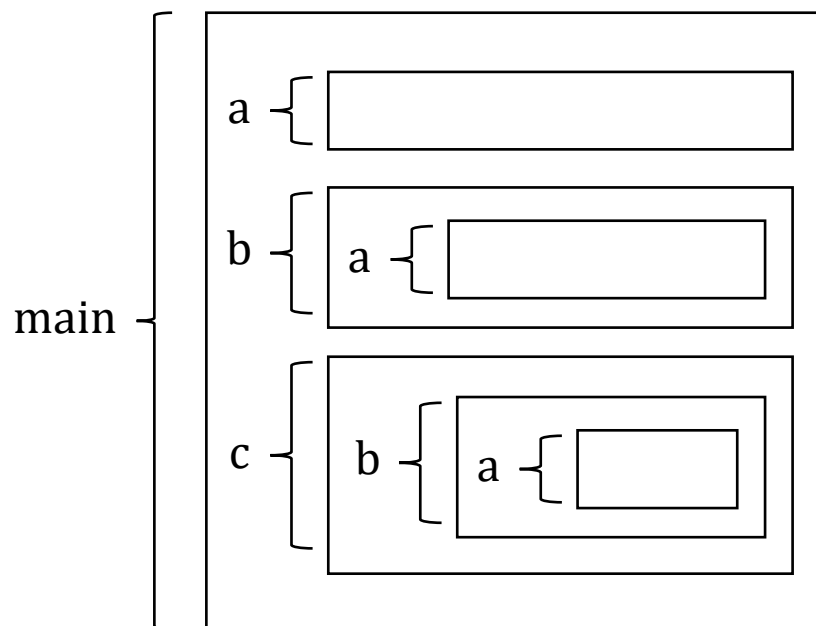
### ➤ 6.5.3 变长块管理

### ➤ 6.5.4 存储回收

## 栈帧结构

- 一个过程的**活动**指该过程的一次执行。
- 关于过程P的一个**活动的生存期**, 指从执行该过程体第一步操作到最后一步操作之间的操作序列。
  - 如果a和b都是过程的活动, 那么它们的**生存期**或者是不重叠的, 或者是**嵌套**的

```
1 void a()  
2 {  
3     ...  
4 }  
5 void b()  
6 {  
7     a();  
8 }  
9 void c()  
10 {  
11     b();  
12 }  
13 int main()  
14 {  
15     a();  
16     b();  
17     c();  
18     return 0;  
19 }
```

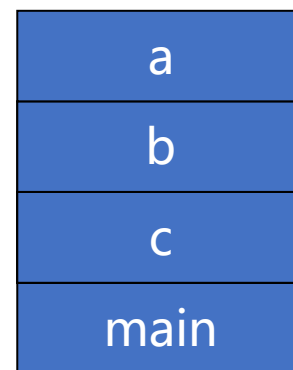
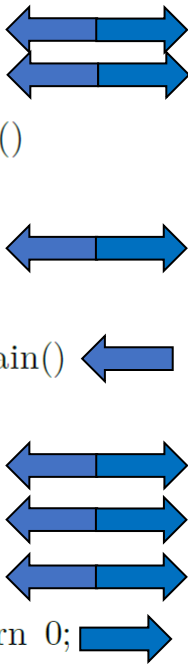


## 栈帧结构

□ **活动记录**：为了管理过程在一次执行中所需要的信息，使用一个**连续的存储块**，这样的连续存储块称为**活动记录**（**Activation Record**），也称为**栈帧结构**（**Stack Frame**）。

- 如Pascal和C语言，当过程调用时，产生活动记录，并压入栈
- 过程返回时，弹出栈。

```
1 void a()  
2 {  
3   ...  
4 }  
5 void b()  
6 {  
7   a();  
8 }  
9 void c()  
10 {  
11   b();  
12 }  
13 int main()  
14 {  
15   a();  
16   b();  
17   c();  
18   return 0;  
19 }
```





# 栈帧结构

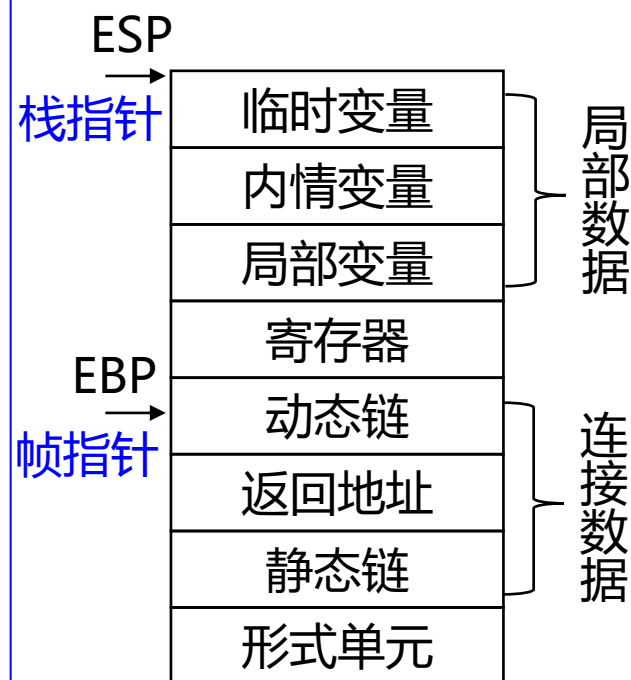
## □ 活动记录的内容

### ➤ 连接数据

- 返回地址
- 动态链：指向该过程前的最新活动记录的指针，运行时，使运行栈上各数据区按动态建立的次序结成链，链头是栈顶起始位置；
- 静态链：指向静态直接外层最新活动记录的指针，用来访问非局部数据。

➤ 形式单元：存放相应的实在参数的地址或值。

➤ 局部数据区：局部变量、内情向量、临时变量。



## 第六章 运行时存储空间组织

### □ 6.1 目标程序运行时的活动

- 6.1.1 运行时存储空间访问
- 6.1.2 栈帧结构
- 6.1.3 存储空间分配策略

### □ 6.2 过程调用规范

- 6.2.1 高级程序参数传递
- 6.2.2 std call
- 6.2.3 C调用规范
- 6.2.4 x64调用规范
- 6.2.5 寄存器保护
- 6.2.6 地址计算
- 6.2.7 ARM规范

### □ 6.3 运行时库

- 6.3.1 使用C运行时库输入输出
- 6.3.2 编译器生成输入输出代码

### ➤ 6.3.3 幂运算

### ➤ 6.3.4 跨文件调用

### ➤ 6.3.5 封装库

### □ 6.4 嵌套过程栈帧结构

### ➤ 6.4.1 静态链

### ➤ 6.4.2 静态链构建

### ➤ 6.4.3 外层变量访问

### ➤ 6.4.4 嵌套层次显示表

### ➤ 6.4.5 Display表构建

### ➤ 6.4.6 通过display访问变量

### □ 6.5 堆式存储分配

### ➤ 6.5.1 定长块管理

### ➤ 6.5.2 保留元数据

### ➤ 6.5.3 变长块管理

### ➤ 6.5.4 存储回收

## 存储空间分配策略

- ❑ **静态分配策略**: 在编译时对所有数据对象分配固定的存储单元, 且在运行时始终保持不变。
- ❑ **栈式动态分配策略**: 在运行时把存储器作为一个栈进行管理, 每当调用一个过程时, 它所需要的存储空间就动态地分配于栈顶; 一旦退出, 它所占用的空间就予以释放。
- ❑ **堆式动态分配策略**: 在运行时把存储器组织成堆结构, 以方便用户关于存储空间的申请与回收; 用户申请时从堆中分配一块空间, 释放时退回给堆。

## 第六章 运行时存储空间组织

### □ 6.1 目标程序运行时的活动

- 6.1.1 运行时存储空间访问
- 6.1.2 栈帧结构
- 6.1.3 存储空间分配策略

### □ 6.2 过程调用规范

- 6.2.1 高级程序参数传递
- 6.2.2 std call
- 6.2.3 C调用规范
- 6.2.4 x64调用规范
- 6.2.5 寄存器保护
- 6.2.6 地址计算
- 6.2.7 ARM规范

### □ 6.3 运行时库

- 6.3.1 使用C运行时库输入输出
- 6.3.2 编译器生成输入输出代码

### ➤ 6.3.3 幂运算

### ➤ 6.3.4 跨文件调用

### ➤ 6.3.5 封装库

### □ 6.4 嵌套过程栈帧结构

### ➤ 6.4.1 静态链

### ➤ 6.4.2 静态链构建

### ➤ 6.4.3 外层变量访问

### ➤ 6.4.4 嵌套层次显示表

### ➤ 6.4.5 Display表构建

### ➤ 6.4.6 通过display访问变量

### □ 6.5 堆式存储分配

### ➤ 6.5.1 定长块管理

### ➤ 6.5.2 保留元数据

### ➤ 6.5.3 变长块管理

### ➤ 6.5.4 存储回收

# 高级程序参数传递

## □ 参数

- **形式参数**, 简称**形参** (Fortran中称为**哑元**) , 即函数的自变量, 其初值来源于函数的调用, 函数被调用之前并不分配内存;
- **实在参数**, 简称**实参**, 其本质是一个变量, 已经占用内存空间, 函数调用时实参赋值给形参。

## □ 实参如何传递给形参?

- 传值 (call by value)
- 传地址 (call by address )
- 引用 (call by reference )
- 传名字 (call by name)

## □ 函数工作完毕返回时, 如何把函数值送回?

- 编译器可以把函数值保留在某个寄存器中。

# 高级程序参数传递

## □ 传值

- 调用段把实参的值计算出来，存放到一个被调用段可以拿的到的地方；
- 被调用段开始工作时，把这些值抄进自己的形式单元中；
- 使用形参时，就像使用局部变量一样使用这些形式单元。

```
1  void sum(int x, int y) {  
2      return x + y;  
3  }  
4  int main() {  
5      int a = 0, b = 1;  
6      printf("a + b = %d\n", sum(a, b));  
7      return 0;  
8  }
```

## 高级程序参数传递

### □ 传地址：把实在参数的地址作为值传递给相应的形式参数

- 在过程段中每个形参都有一个相应的单元，称为**形式单元**，用来存放相应的实参地址；
- 当调用一个过程时，**调用段**必须把**实参地址**传递到**被调用段**可以拿得到的地方
  - 如果实参是一个**变量**，则直接传递它的地址；
  - 如果实参是**常数**或**表达式**，则先计算它的值，并存放到一个临时单元，然后传递这个临时单元的地址。

### □ 程序控制转入被调用段后

- 被调用段首先把**实参地址**抄进自己相应的**形式单元**；
- 过程体对形式参数的任何**引用**或**赋值**，都被处理成**对形式单元的间接访问**；
- 被调用段工作完毕返回时，形式单元所指的实参单元就**持有**了所期望的值。

## 高级程序参数传递

□ 引用：把实在参数的地址传递给相应的形式参数

- 以地址的方式传递参数;
- 传递以后，形参和实参都是同一个对象，只是它们名字不同。

```
1 void Swap(int &x, int &y)
2 {
3     int a = x;
4     x = y;
5     y = a;
6 }
7 int main()
8 {
9     int a = 0, b = 1;
10    Swap(a, b);
11    printf("a=%d, b=%d\n", a, b);
12    return 0;
13 }
```

```
1 void Swap(int *x, int *y)
2 {
3     int a = *x;
4     *x = *y;
5     *y = a;
6 }
7 int main()
8 {
9     int a = 0, b = 1;
10    Swap(&a, &b);
11    printf("a=%d, b=%d\n", a, b);
12    return 0;
13 }
```



## 高级程序参数传递

□ **传结果 (call by result)** : 与传地址类似但不完全等价

- 每个形参对应**两个单元**, 第一个单元存放实参地址, 第二个单元存放实参值;
- 过程体中对形式参数的任何**引用或赋值**, 都看成是对**第二个单元**的直接访问;
- 过程工作返回前, 把第二个单元的内容存放到**第一个单元所指的实参单元**中。

```
1 void A(out int x) {  
2     x = 100;  
3     ...  
4 }  
5 void B() {  
6     int a;  
7     A(out a);  
8     Console.WriteLine("a={0}", a);  
9 }
```

## 高级程序参数传递

□ **传名字**：是Algol60定义的一种特殊的形-实参数结合方式

- 过程调用段作用相当于把被调用段的**过程体抄到调用出现的地方**；
- 把其中任一出现的形参都替换成相应的实参（文字替换）；
- 如果在替换时发现过程体中的局部名和实参中的名字相同，则必须用不同的标识符来表示这些局部名；
- 为了表现实在参数的整体性，必要时在替换前先把它用括号括起来。

## 第六章 运行时存储空间组织

### □ 6.1 目标程序运行时的活动

- 6.1.1 运行时存储空间访问
- 6.1.2 栈帧结构
- 6.1.3 存储空间分配策略

### □ 6.2 过程调用规范

- 6.2.1 高级程序参数传递
- 6.2.2 std call
- 6.2.3 C调用规范
- 6.2.4 x64调用规范
- 6.2.5 寄存器保护
- 6.2.6 地址计算
- 6.2.7 ARM规范

### □ 6.3 运行时库

- 6.3.1 使用C运行时库输入输出
- 6.3.2 编译器生成输入输出代码

### ➤ 6.3.3 幂运算

### ➤ 6.3.4 跨文件调用

### ➤ 6.3.5 封装库

### □ 6.4 嵌套过程栈帧结构

### ➤ 6.4.1 静态链

### ➤ 6.4.2 静态链构建

### ➤ 6.4.3 外层变量访问

### ➤ 6.4.4 嵌套层次显示表

### ➤ 6.4.5 Display表构建

### ➤ 6.4.6 通过display访问变量

### □ 6.5 堆式存储分配

### ➤ 6.5.1 定长块管理

### ➤ 6.5.2 保留元数据

### ➤ 6.5.3 变长块管理

### ➤ 6.5.4 存储回收

# std call

```
void sum(int x, int y)
{
    return x + y;
}
```

```
sum proc
    push ebp
    mov ebp, esp      ; 堆栈帧的基址
    mov eax, [ebp + 8] ; 第一个参数
    add eax, [ebp + 12] ; 第二个参数
    pop ebp
    ret 8              ; 清除栈帧
sum endp
```

```
main proc
    push dword ptr 1
    push dword ptr 0
    call sum
    ret
main endp
end main
```

## std call

- 参数反序进栈;
- EAX寄存器传递返回值;
- 由被调用过程清理栈区。



# C调用规范

```
void sum(int x, int y)
{
    return x + y;
}
```

```
sum proc
    push ebp
    mov ebp, esp
    mov eax, [ebp + 8]
    add eax, [ebp + 12]
    pop ebp
    ret
```

```
sum endp
main proc
    push 1
    push 0
    call sum
    add esp, 8
    ret
main endp
end main
```

## □ cdecl(c declaration)

- 参数反序进栈;
- EAX寄存器传递返回值;
- 由调用过程清理栈区。



## x64调用规范

### □ fast call

- fast call 是微软提出的一种过程调用规范，它用寄存器而不是内存传递参数，以提升参数传递速度；
- 但在当时寄存器数量非常少的情况下，由于过程调用要保护现和恢复现场，使得该规范并不fast，被放弃。

### □ x64调用规范恢复了fast call

- 使用寄存器传参，被调方清理栈空间，返回值存入RAX；
- MSVC前4个参数依次使用RCX、RDX、R8、R9进行传递，超过4个参数通过栈帧传参；
- GCC前6个参数依次使用RDI、RSI、RDX、RCX、R8、R9，从第7个参数开始反序入栈；
- 仍然会申请参数预留空间，大部分编译器在进入被调过程后，会首先将寄存器的值写入到栈帧中，后面使用形参的值，还是通过访问栈帧获得。

# x64调用规范

```
1 // Type your code here, or load an example.
2 void Fun(int a, int b, int c, int d, int e, int f, int g)
3 {
4     int x = a + b;
5 }
6 int main()
7 {
8     Fun(1, 2, 3, 4, 5, 6, 7);
9     return 0;
10 }
```

x86-64 gcc 13.2

▼

Compiler options...

A ▼

Output...

Filter...

Libraries

Overrides

+ Add new.

1 Fun(int, int, int, int, int, int, int):

push rbp

mov rbp, rsp

mov DWORD PTR [rbp-20], edi

mov DWORD PTR [rbp-24], esi

mov DWORD PTR [rbp-28], edx

mov DWORD PTR [rbp-32], ecx

mov DWORD PTR [rbp-36], r8d

mov DWORD PTR [rbp-40], r9d

mov edx, DWORD PTR [rbp-20]

mov eax, DWORD PTR [rbp-24]

add eax, edx

mov DWORD PTR [rbp-4], eax

nop

pop rbp

ret

17 main:

push rbp

mov rbp, rsp

push 7

mov r9d, 6

mov r8d, 5

mov ecx, 4

mov edx, 3

mov esi, 2

mov edi, 1

call Fun(int, int, int, int, int, int, int)

add rsp, 8

mov eax, 0

leave

ret

# x64调用规范

```
1 // Type your code here, or load an example.
2 void Fun(int a, int b, int c, int d, int e, int f, int g)
3 {
4     int x = a + b;
5 }
6 int main()
7 {
8     Fun(1, 2, 3, 4, 5, 6, 7);
9     return 0;
10 }
```

MSVC x64

```
void Fun(int a, int b, int c, int d, int e, int f, int g)
{
    00007FF67F1C1F80 mov     dword ptr [rsp+20h], r9d
    00007FF67F1C1F85 mov     dword ptr [rsp+18h], r8d
    00007FF67F1C1F8A mov     dword ptr [rsp+10h], edx
    00007FF67F1C1F8E mov     dword ptr [rsp+8], ecx
    00007FF67F1C1F92 push    rbp
    00007FF67F1C1F93 push    rdi
    00007FF67F1C1F94 sub     rsp, 108h
    00007FF67F1C1F9B lea     rbp, [rsp+20h]
    00007FF67F1C1FA0 lea     rcx, [__1F66B38C_CPlusTest@cpp (07FF67F1D302Bh)]
    00007FF67F1C1FA7 call    __CheckForDebuggerJustMyCode (07FF67F1C13A2h)

    int x = a + b;
    00007FF67F1C1FAC mov     eax, dword ptr [b]
    00007FF67F1C1FB2 mov     ecx, dword ptr [a]
    00007FF67F1C1FB8 add     ecx, eax
    00007FF67F1C1FBA mov     eax, ecx
    00007FF67F1C1FBC mov     dword ptr [x], eax
}
```

```
Fun(1, 2, 3, 4, 5, 6, 7);
00007FF75586214B mov     dword ptr [rsp+30h], 7
00007FF755862153 mov     dword ptr [rsp+28h], 6
00007FF75586215B mov     dword ptr [rsp+20h], 5
00007FF755862163 mov     r9d, 4
00007FF755862169 mov     r8d, 3
00007FF75586216F mov     edx, 2
00007FF755862174 mov     ecx, 1
00007FF755862179 call    Fun (07FF755861172h)

return 0;
```



## 第六章 运行时存储空间组织

### □ 6.1 目标程序运行时的活动

- 6.1.1 运行时存储空间访问
- 6.1.2 栈帧结构
- 6.1.3 存储空间分配策略

### □ 6.2 过程调用规范

- 6.2.1 高级程序参数传递
- 6.2.2 std call
- 6.2.3 C调用规范
- 6.2.4 x64调用规范
- 6.2.5 寄存器保护
- 6.2.6 地址计算
- 6.2.7 ARM规范

### □ 6.3 运行时库

- 6.3.1 使用C运行时库输入输出
- 6.3.2 编译器生成输入输出代码

### ➤ 6.3.3 幂运算

### ➤ 6.3.4 跨文件调用

### ➤ 6.3.5 封装库

### □ 6.4 嵌套过程栈帧结构

### ➤ 6.4.1 静态链

### ➤ 6.4.2 静态链构建

### ➤ 6.4.3 外层变量访问

### ➤ 6.4.4 嵌套层次显示表

### ➤ 6.4.5 Display表构建

### ➤ 6.4.6 通过display访问变量

### □ 6.5 堆式存储分配

### ➤ 6.5.1 定长块管理

### ➤ 6.5.2 保留元数据

### ➤ 6.5.3 变长块管理

### ➤ 6.5.4 存储回收

## 寄存器保护

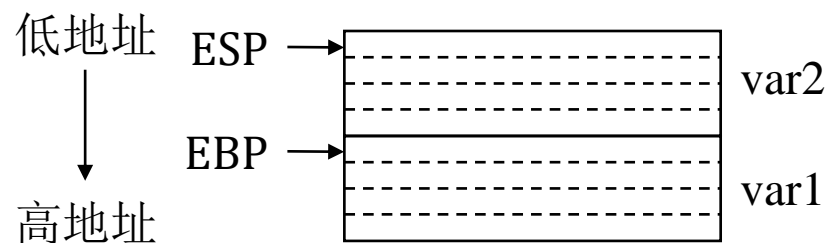
### □ 寄存器保护约定

- 在前述内容中，我们一直采用被调方保护寄存器的策略；
- GCC规定EAX、ECX和EDX这三个寄存器为调用方保护，而EBX、ESI和EDI这三个寄存器为被调方保护；
- GCC这样规定的原因，可能是跨过程转移的需要，条件转移指令需要两个寄存器，返回值需要EAX寄存器；
- 很多编译器采用了GCC标准，比如下一节介绍的C运行时库函数printf和scanf，都由主调过程保护EAX、ECX和EDX。

## 地址计算

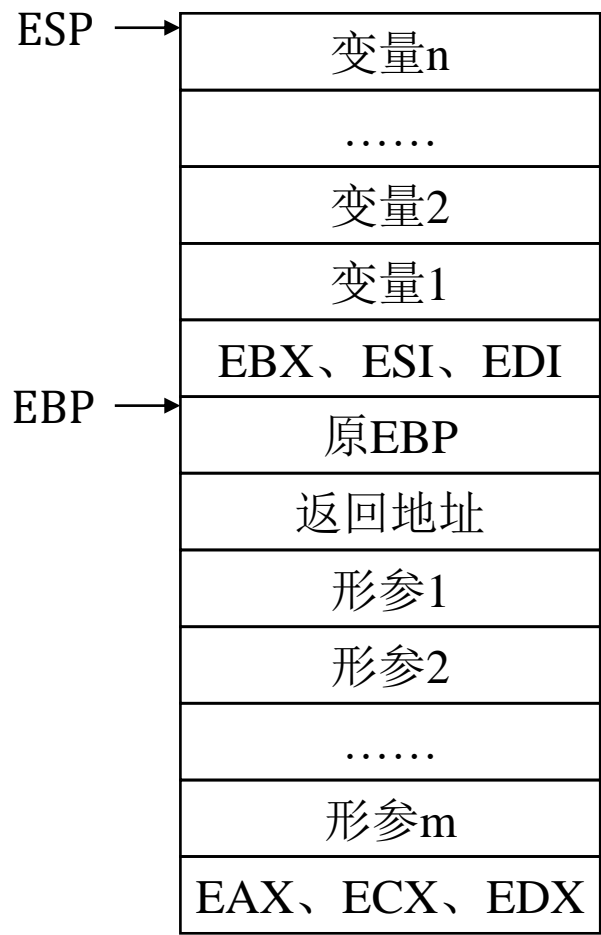
- 过程的变量在栈帧中无法体现名字，一般通过**帧指针EBP**加一个**偏移量按地址**进行存取操作
- 称ESP总是指向栈顶元素，但栈顶元素可能占多个字节，实际**ESP总是指向栈顶字位**，这个字位也是栈顶元素的起始字位。

```
push var1  
mov ebp, esp  
push var2
```



# 地址计算

- 形参变量:  $[EBP + \delta_i^f]$ , 其中  $\delta_i^f = \delta_i^f + 8$
- 普通变量:  $[EBP - \delta_i^v]$ , 其中  $\delta_i^v = \delta_i^v + w_i^v + 12$



名字	类别	.....	字宽	偏移量
形参1	形参	.....	$w_1^f$	$\delta_1^f$
形参2	形参	.....	$w_2^f$	$\delta_2^f$
.....	.....	.....	.....	.....
形参m	形参	.....	$w_m^f$	$\delta_m^f$
变量1	变量	.....	$w_1^v$	$\delta_1^v$
变量2	变量	.....	$w_2^v$	$\delta_2^v$
.....	.....	.....	.....	.....
变量n	变量	.....	$w_n^v$	$\delta_n^v$

从0开始

从0开始

## 第六章 运行时存储空间组织

### □ 6.1 目标程序运行时的活动

- 6.1.1 运行时存储空间访问
- 6.1.2 栈帧结构
- 6.1.3 存储空间分配策略

### □ 6.2 过程调用规范

- 6.2.1 高级程序参数传递
- 6.2.2 std call
- 6.2.3 C调用规范
- 6.2.4 x64调用规范
- 6.2.5 寄存器保护
- 6.2.6 地址计算
- 6.2.7 ARM规范

### □ 6.3 运行时库

- 6.3.1 使用C运行时库输入输出
- 6.3.2 编译器生成输入输出代码

### ➤ 6.3.3 幂运算

### ➤ 6.3.4 跨文件调用

### ➤ 6.3.5 封装库

### □ 6.4 嵌套过程栈帧结构

### ➤ 6.4.1 静态链

### ➤ 6.4.2 静态链构建

### ➤ 6.4.3 外层变量访问

### ➤ 6.4.4 嵌套层次显示表

### ➤ 6.4.5 Display表构建

### ➤ 6.4.6 通过display访问变量

### □ 6.5 堆式存储分配

### ➤ 6.5.1 定长块管理

### ➤ 6.5.2 保留元数据

### ➤ 6.5.3 变长块管理

### ➤ 6.5.4 存储回收

# ARM规范

## □ ARM规范

- 参数少于4个时，按从左到右的顺序依次放在R0、R1、R2、R3中，这4个寄存器被调方使用时，不需恢复原来的内容。
- 参数多于4个时，前4个放在R0、R1、R2、R3中，剩余的反序入栈，即从右到左入栈，第5个最后入栈。
- 被调方使用R3之后的寄存器时，需要返回前恢复原来的值。
- 被调方可以使用堆栈，但一定要保证堆栈指针（R13）在进入时和退出时相等。
- R14用于保存返回地址，使用前一定要备份。
- 返回值为32位时，通过R0返回；返回值为64位时，R0放低32位，R1放高32位。

## 第六章 运行时存储空间组织

### □ 6.1 目标程序运行时的活动

- 6.1.1 运行时存储空间访问
- 6.1.2 栈帧结构
- 6.1.3 存储空间分配策略

### □ 6.2 过程调用规范

- 6.2.1 高级程序参数传递
- 6.2.2 std call
- 6.2.3 C调用规范
- 6.2.4 x64调用规范
- 6.2.5 寄存器保护
- 6.2.6 地址计算
- 6.2.7 ARM规范

### □ 6.3 运行时库

- 6.3.1 使用C运行时库输入输出
- 6.3.2 编译器生成输入输出代码

#### ➤ 6.3.3 幂运算

#### ➤ 6.3.4 跨文件调用

#### ➤ 6.3.5 封装库

### □ 6.4 嵌套过程栈帧结构

#### ➤ 6.4.1 静态链

#### ➤ 6.4.2 静态链构建

#### ➤ 6.4.3 外层变量访问

#### ➤ 6.4.4 嵌套层次显示表

#### ➤ 6.4.5 Display表构建

#### ➤ 6.4.6 通过display访问变量

### □ 6.5 堆式存储分配

#### ➤ 6.5.1 定长块管理

#### ➤ 6.5.2 保留元数据


#### ➤ 6.5.3 变长块管理

#### ➤ 6.5.4 存储回收

## 使用C运行时库输入输出

### □ printf 和 scanf

```
printf("Input a number: ");  
scanf("%d", &nInputValue);  
printf("eax = %d, ebx = %d, ecx = %d", eax, ebx, ecx);
```

 Microsoft Visual Studio 调试控制台

```
Input a number: 100  
eax = 77, ebx = 100, ecx = 142897635
```

D:\Source\MASM\Test\Debug\Test.exe (进程 19848) 已退出, 代码为 0。  
要在调试停止时自动关闭控制台, 请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。  
按任意键关闭此窗口. . .



## 使用C运行时库输入输出的汇编代码

```
.386
.model flat, stdcall
.stack 4096

include msvcrt.inc
includelib msvcrt.lib

ExitProcess PROTO, dwExitCode: dword

.data
    sPrompt db "Input a number: ", 0
    sOutput db "eax = %d, ebx = %d, ecx = %d", 0dh, 0ah, 0
    nInputValue dword ?
    sInputFormat db '%d', 0
```

```
.code
```

```
main proc
```

```
push offset sPrompt  
call crt_printf  
add esp, 4
```

```
push offset nInputValue  
push offset sInputFormat  
call crt_scanf  
add esp, 8
```

```
mov eax, 77
```

```
mov ebx, nInputValue
```

```
push ecx  
push ebx  
push eax  
push offset sOutput  
call crt_printf  
add esp, 16
```

```
push 0h
```

```
call ExitProcess
```

```
ret
```

```
main endp
```

```
end main
```

## 默认实参提升

### □ 变长实参特殊事项

- 如printf之类的变长实参函数, 可变实参列表中的每个实参都要经过称为 默认实参提升的额外转换。
- 如把一个float传参给printf, 压入栈的应该用double的8字节, 而不是float的4字节。

### □ 默认实参提升包括:

- `std::nullptr_t`转换到`void*`(C++11起)
- `float`转换到`double`
- `bool`、`char`、`short`及无作用域枚举转换到`int`或更宽的整数类型

## 第六章 运行时存储空间组织

### □ 6.1 目标程序运行时的活动

- 6.1.1 运行时存储空间访问
- 6.1.2 栈帧结构
- 6.1.3 存储空间分配策略

### □ 6.2 过程调用规范

- 6.2.1 高级程序参数传递
- 6.2.2 std call
- 6.2.3 C调用规范
- 6.2.4 x64调用规范
- 6.2.5 寄存器保护
- 6.2.6 地址计算
- 6.2.7 ARM规范

### □ 6.3 运行时库

- 6.3.1 使用C运行时库输入输出
- 6.3.2 编译器生成输入输出代码

### ➤ 6.3.3 幂运算

### ➤ 6.3.4 跨文件调用

### ➤ 6.3.5 封装库

### □ 6.4 嵌套过程栈帧结构

### ➤ 6.4.1 静态链

### ➤ 6.4.2 静态链构建

### ➤ 6.4.3 外层变量访问

### ➤ 6.4.4 嵌套层次显示表

### ➤ 6.4.5 Display表构建

### ➤ 6.4.6 通过display访问变量

### □ 6.5 堆式存储分配

### ➤ 6.5.1 定长块管理

### ➤ 6.5.2 保留元数据

### ➤ 6.5.3 变长块管理

### ➤ 6.5.4 存储回收

## 编译器生成输入输出代码

### ❑ 编译器不可能将输入输出格式字符串放到静态数据区

- 输入字符串本身是常量或者局部变量，在静态代码区设置大量此类数据，会占用大量静态空间、产生命名冲突等，这样的设计不是一个好的设计。
- 有时候格式字符串是动态计算出来的，需要在运行时才能得到，在这种场景下编译时为其创建静态变量也是不可能的。

```
void Fun()  
{  
    scanf("%d+%d", &a, &b);  
    printf("=%d\r\n", a + b);  
}
```

```
Fun proc  
    push ebp  
    mov ebp, esp  
    push ebx  
    push esi  
    push edi
```

```
sub esp, 8          ; 局部变量a、b的空间
sub esp, 8          ; 格式字符串"%d+%d\n"的空间, 6字节对齐到8字节
mov [ebp - 28], byte ptr 25h
mov [ebp - 27], byte ptr 64h
mov [ebp - 26], byte ptr 2bh
mov [ebp - 25], byte ptr 25h
mov [ebp - 24], byte ptr 64h
mov [ebp - 23], byte ptr 0h
mov ebx, ebp
sub ebx, 20         ; b的地址
mov ecx, ebp
sub ecx, 16         ; a的地址
mov edx, ebp
sub edx, 28         ; 格式字符串的地址
push eax            ; 保护寄存器EAX
push ecx            ; 保护寄存器ECX
push edx            ; 保护寄存器EDX
push ebx            ; 参数b地址
push ecx            ; 参数a地址
push edx            ; 格式字符串地址
call crt_scanf      ; 调用scanf
add esp, 12         ; 清理形参空间
```

```
pop edx          ; 恢复寄存器EDX
pop ecx          ; 恢复寄存器ECX
pop eax          ; 恢复寄存器EAX
add esp, 8       ; 释放格式字符串的空间
sub esp, 8       ; 格式字符串“=%d\r\n\0”的空间
mov [ebp - 28], byte ptr 3dh
mov [ebp - 27], byte ptr 25h
mov [ebp - 26], byte ptr 64h
mov [ebp - 25], byte ptr 0dh
mov [ebp - 24], byte ptr 0ah
mov [ebp - 23], byte ptr 0h
mov esi, [ecx]   ; ECX中存放着a的地址
add esi, [ebx]   ; EBX中存放着b的地址, 现在ESI中是a+b
mov edi, ebp
sub edi, 28      ; 格式字符串地址
push eax
push ecx
push edx
push esi        ; 压入a+b的和
push edi        ; 格式字符串地址
call crt_printf ; 调用printf
add esp, 8      ; 清理形参空间
```

```
    pop edx
    pop ecx
    pop eax
    add esp, 8      ; 释放格式字符串空间
    add esp, 8      ; 释放局部变量a、b空间
    pop edi
    pop esi
    pop ebx
    pop ebp
    ret
Fun endp
```



## 第六章 运行时存储空间组织

### □ 6.1 目标程序运行时的活动

- 6.1.1 运行时存储空间访问
- 6.1.2 栈帧结构
- 6.1.3 存储空间分配策略

### □ 6.2 过程调用规范

- 6.2.1 高级程序参数传递
- 6.2.2 std call
- 6.2.3 C调用规范
- 6.2.4 x64调用规范
- 6.2.5 寄存器保护
- 6.2.6 地址计算
- 6.2.7 ARM规范

### □ 6.3 运行时库

- 6.3.1 使用C运行时库输入输出
- 6.3.2 编译器生成输入输出代码

### ➤ 6.3.3 幂运算

- 6.3.4 跨文件调用
- 6.3.5 封装库

### □ 6.4 嵌套过程栈帧结构

- 6.4.1 静态链
- 6.4.2 静态链构建
- 6.4.3 外层变量访问
- 6.4.4 嵌套层次显示表
- 6.4.5 Display表构建
- 6.4.6 通过display访问变量

### □ 6.5 堆式存储分配

- 6.5.1 定长块管理
- 6.5.2 保留元数据
- 6.5.3 变长块管理
- 6.5.4 存储回收

## 超越函数指令

### □ 超越函数指令 (Transcendental Instructions)

- **FYL2X**: 计算  $ST(1) \times \log_2(ST(0))$ , 并将其存入  $ST(1)$ , 再弹出  $ST(0)$ 。
- **FYL2XP1**: 计算  $ST(1) \times \log_2(ST(0) + 1)$ , 并将其存入  $ST(1)$ , 再弹出  $ST(0)$ 。
- **F2XM1**: 把  $ST(0)$  替换为  $2^{ST(0)} - 1$ , 要求运算前  $ST(0)$  必须在  $[-1, 1]$  范围内。
- **FSCALE**: 计算  $ST(0) \times 2^{ST(1)}$ , 存入  $ST(0)$ , 要求  $ST(1)$  必须是  $[-32768, 32767]$  范围内的整数。
- **FSIN**: 将  $ST(0)$  替换为  $\sin(ST(0))$ , 单位为弧度。
- **FCOS**: 将  $ST(0)$  替换为  $\cos(ST(0))$ , 单位为弧度。
- **FSINCOS**: 先计算  $\sin(ST(0))$  和  $\cos(ST(0))$ , 然后将  $ST(0)$  替换为  $\sin(ST(0))$ , 再压入  $\cos(ST(0))$ , 即余弦在栈顶正弦在  $ST(1)$ , 单位为弧度。
- **FPTAN**: 将  $ST(0)$  替换为  $\tan(ST(0))$ , 再压入 1。
- **FPATAN**: 将  $ST(1)$  替换为  $\arctan \frac{ST(1)}{ST(0)}$ , 再弹出  $ST(0)$ 。

## 幂运算原理

### □ 幂运算原理 $a^b$

- 任意底 $a$ 变换为底2:  $a^b = 2^{\log_2(a^b)} = 2^{b\log_2 a}$
- 由于指令F2XM1要求参数在 $[-1,1]$ 范围内, 而FSCALE需要指数参数是整数, 因此需要把得到的2的幂指数分成整数和小数两个部分。
- 记 $x = [b\log_2 a]$ 为整数部分,  $y = b\log_2 a - x$ 为小数部分, 则有:  $a^b = 2^{x+y} = 2^y \times 2^x$
- 通过FIST指令可以实现取整运算, 从而得到 $x$ , 进一步得到 $y$ 。
- 再通过F2XM1指令对 $y$ 操作, 再加上1得到 $2^y$ 。
- $x$ 是整数, 使用FSCALE就可以得到最终结果。

## 幂运算实现

PowerEE proc NEAR32

push ebp	; 保留EBP
mov ebp, esp	; 帧指针
push ebx	
push esi	
push edi	
sub esp, 4	; 临时变量空间
fstcw [ebp - 16]	; 将控制字保存到主存的临时变量空间
mov ax, [ebp - 16]	; 将其备份到AX以便后续恢复原值时使用
mov bx, ax	; 放到BX用于计算
or bx, 1100000000000b	; 设置为截断模式
mov [ebp - 16], bx	; 存入主存空间
fldcw [ebp - 16]	; 控制字取到FPU

```
fld real8 ptr [ebp + 16] ; 指数入ST(0)
fld real8 ptr [ebp + 8] ; 底数入ST(0), 指数压入ST(1)
fyl2x ; ST(1) * log2[ST(0)], 存在ST(0)
fist dword ptr [ebp - 16] ; 整数部分[b*log a]存入临时变量
fild dword ptr [ebp - 16] ; ST(0)是[b*log a], ST(1)是b*log a
fsub ; 相减并出栈, 因此栈顶是小数部分b*log a - [b*log a]
f2xm1 ;  $2^{ST(0)} - 1 = 2^{(b \cdot \log a - [b \cdot \log a])} - 1$ 
fld1 ; 常数1取到栈顶
fadd ;  $2^{(b \cdot \log a - [b \cdot \log a])}$ 
fild dword ptr [ebp - 16] ; 整数部分取回来
fxch ; ST(0)和ST(1)交换
fscale ; 计算ST(0) *  $2^{ST(1)}$ , 即栈顶为 $a^b$ 
mov ebx, [ebp + 24] ; 取出结果Result8的地址
fstp qword ptr [ebx] ; 结果写入这个地址
mov [ebp - 16], ax ; 控制字写入临时变量
fldcw [ebp - 16] ; 控制字恢复
add esp, 4 ; 释放临时变量空间
pop edi
pop esi
pop ebx
pop ebp
ret
```

PowerEE endp

## 第六章 运行时存储空间组织

### □ 6.1 目标程序运行时的活动

- 6.1.1 运行时存储空间访问
- 6.1.2 栈帧结构
- 6.1.3 存储空间分配策略

### □ 6.2 过程调用规范

- 6.2.1 高级程序参数传递
- 6.2.2 std call
- 6.2.3 C调用规范
- 6.2.4 x64调用规范
- 6.2.5 寄存器保护
- 6.2.6 地址计算
- 6.2.7 ARM规范

### □ 6.3 运行时库

- 6.3.1 使用C运行时库输入输出
- 6.3.2 编译器生成输入输出代码

### ➤ 6.3.3 幂运算

### ➤ 6.3.4 跨文件调用

### ➤ 6.3.5 封装库

### □ 6.4 嵌套过程栈帧结构

### ➤ 6.4.1 静态链

### ➤ 6.4.2 静态链构建

### ➤ 6.4.3 外层变量访问

### ➤ 6.4.4 嵌套层次显示表

### ➤ 6.4.5 Display表构建

### ➤ 6.4.6 通过display访问变量

### □ 6.5 堆式存储分配

### ➤ 6.5.1 定长块管理

### ➤ 6.5.2 保留元数据

### ➤ 6.5.3 变长块管理

### ➤ 6.5.4 存储回收

## 跨文件调用

### □ 跨文件调用及传递浮点数参数

- 这个PowerEE之类的过程，可以放入一个独立文件中，如Power.asm，也可以改为其它后缀如Power.inc，然后在需要的地方include进去使用。
- 需要注意的是这个Power.inc文件（或.asm）没有.386、.model等信息，可以有.data和.code也可以没有，但最后一定不能有end。
- 另外浮点数做参数，需要把浮点数按字节存放，然后看成一个整体在栈内操作

### □ 以下代码仅用做调用示例，不规范之处包括：

- 数据在静态区分配。
- 没有按照规范在过程开始保护寄存器EBP、EBX、ESI和EDI，在过程结束前恢复寄存器。
- 没有在调用过程PowerEE和printf前保护寄存器EAX、ECX、EDX，在调用完成后恢复寄存器。特别是调用printf过程，本代码只是因为恰好没有使用这3个寄存器，才得以正确执行。

```
.386
.model flat, C
.stack 4096
include msvcrt.inc
includelib msvcrt.lib
include Power.inc
ExitProcess PROTO, dwExitCode: dword
.data
    nInputValue1 real8 2.5
    nInputValue2 real8 3.4
    Result8 real8 ?
    sOutput db "answer = %lf", 0dh, 0ah, 0
```



```
.code
main proc
    push offset Result8           ; 结果的地址入栈
    fld nInputValue2             ; 指数取到ST(0)
    sub esp, 8                   ; 指数存储空间
    fstp qword ptr [esp]         ; ST(0)存入这个指数存储空间
    fld nInputValue1             ; 底数取到ST(0)
    sub esp, 8                   ; 底数存储空间
    fstp qword ptr [esp]         ; ST(0)存入这个底数存储空间
    call PowerEE                 ; 调用
    add esp, 20                  ; 清理参数空间
    fld Result8                  ; 结果取到ST(0)
    sub esp, 8                   ; 结果存储空间
    fstp qword ptr [esp]         ; ST(0)存入这个存储空间
    push offset sOutput          ; 输出格式字符串地址
    call crt_printf              ; printf
    add esp, 12
    push 0h
    call ExitProcess

main endp
end main
```

## 第六章 运行时存储空间组织

### □ 6.1 目标程序运行时的活动

- 6.1.1 运行时存储空间访问
- 6.1.2 栈帧结构
- 6.1.3 存储空间分配策略

### □ 6.2 过程调用规范

- 6.2.1 高级程序参数传递
- 6.2.2 std call
- 6.2.3 C调用规范
- 6.2.4 x64调用规范
- 6.2.5 寄存器保护
- 6.2.6 地址计算
- 6.2.7 ARM规范

### □ 6.3 运行时库

- 6.3.1 使用C运行时库输入输出
- 6.3.2 编译器生成输入输出代码

### ➤ 6.3.3 幂运算

### ➤ 6.3.4 跨文件调用

### ➤ 6.3.5 封装库

### □ 6.4 嵌套过程栈帧结构

### ➤ 6.4.1 静态链

### ➤ 6.4.2 静态链构建

### ➤ 6.4.3 外层变量访问

### ➤ 6.4.4 嵌套层次显示表

### ➤ 6.4.5 Display表构建

### ➤ 6.4.6 通过display访问变量

### □ 6.5 堆式存储分配

### ➤ 6.5.1 定长块管理

### ➤ 6.5.2 保留元数据

### ➤ 6.5.3 变长块管理

### ➤ 6.5.4 存储回收

## 封装库

### □ 封装库

- 上述调用不同文件中的过程调用，采用的是include整个文件的方式，对编译器编译的用户程序，应该采用这种方式把用户编写的不同文件整合为一个整体。
- 但对于幂计算过程这种非用户编写的过程，是我们编译器提供给用户的功能，我们可以称之为系统过程，这种过程应该封装成库文件（分动态库.dll和静态库.lib），如同我们调用C运行时库的printf和scanf一样，.inc文件只向用户暴露过程接口，这就是库的封装。
- 封装库时，如使用Visual Studio的IDE，可以创建一个静态库项目，然后删掉所有的文件和文件夹，添加一个.asm文件。该文件包含.386和.model伪指令，然后每个过程的定义写在.code段，并且在.code之前用“public 过程名”声明。该文件最后有end，但是end后没有入口过程名。
- 调用时，需要在调用前用proto声明该过程的名字、参数等信息，然后如同调用外部库过程一样调用。

## 第六章 运行时存储空间组织

### □ 6.1 目标程序运行时的活动

- 6.1.1 运行时存储空间访问
- 6.1.2 栈帧结构
- 6.1.3 存储空间分配策略

### □ 6.2 过程调用规范

- 6.2.1 高级程序参数传递
- 6.2.2 std call
- 6.2.3 C调用规范
- 6.2.4 x64调用规范
- 6.2.5 寄存器保护
- 6.2.6 地址计算
- 6.2.7 ARM规范

### □ 6.3 运行时库

- 6.3.1 使用C运行时库输入输出
- 6.3.2 编译器生成输入输出代码

### ➤ 6.3.3 幂运算

### ➤ 6.3.4 跨文件调用

### ➤ 6.3.5 封装库

### □ 6.4 嵌套过程栈帧结构

#### ➤ 6.4.1 静态链

### ➤ 6.4.2 静态链构建

### ➤ 6.4.3 外层变量访问

### ➤ 6.4.4 嵌套层次显示表

### ➤ 6.4.5 Display表构建

### ➤ 6.4.6 通过display访问变量

### □ 6.5 堆式存储分配

### ➤ 6.5.1 定长块管理

### ➤ 6.5.2 保留元数据

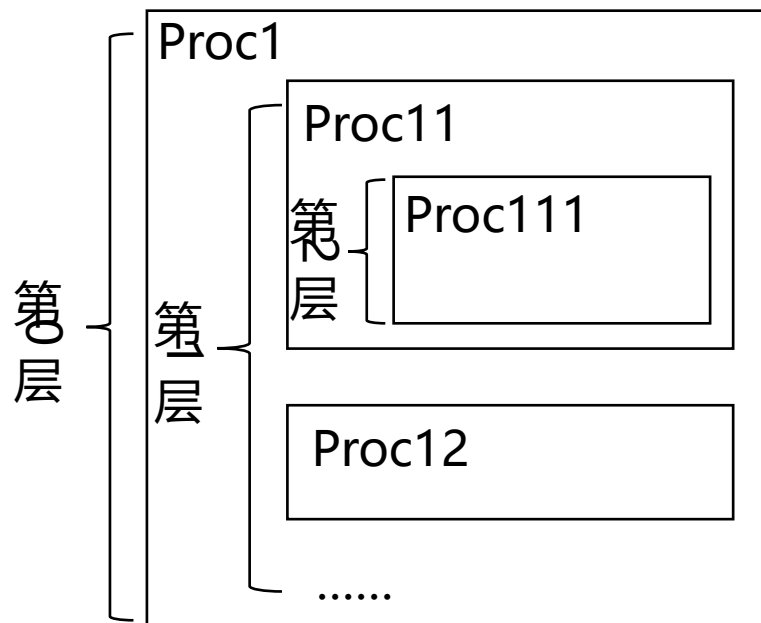
### ➤ 6.5.3 变长块管理

### ➤ 6.5.4 存储回收



## 静态链

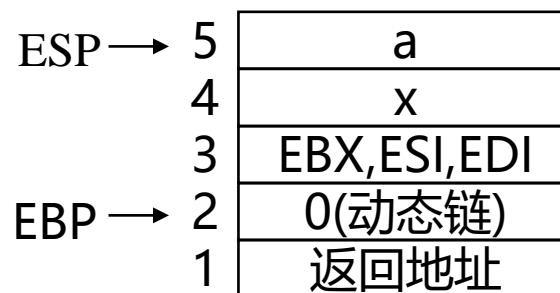
```
1  program Proc1;
2      var a, x: integer;
3      procedure Proc11 (b: integer);
4          var i: integer;
5          procedure Proc111 (u, v: integer);
6              var c, d: integer;
7              begin
8                  ... if u = 1 then Proc111(u+1, v); ...
9                  v := (a + c) * (b - d); ...
10             end {Proc111}
11         begin
12             ... Proc111(1, x); ...
13         end {Proc11}
14     procedure Proc12;
15         var c, i: integer;
16         begin
17             a := 1; Proc11(c); ...
18         end {Proc12}
19     begin
20         a := 0;
21         Proc12;
22         ...
23     end {Proc1}
```





# 运行Proc1

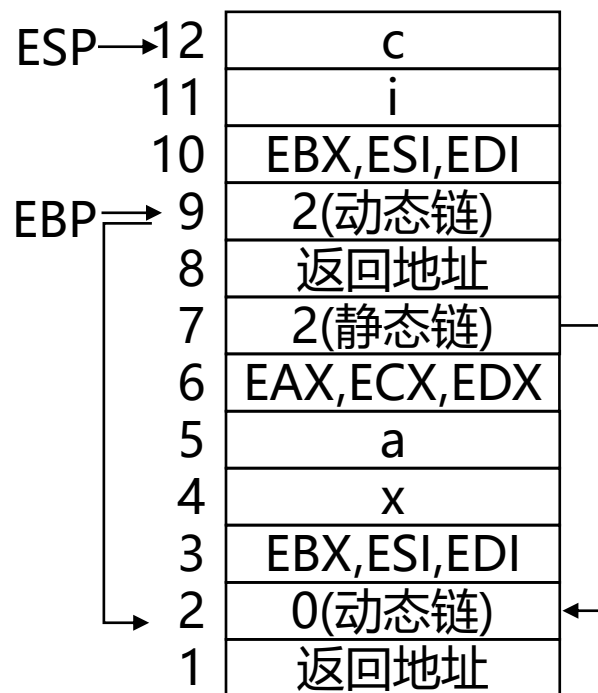
```
1  program Proc1;
2    var a, x: integer;
3    procedure Proc11 (b: integer);
4      var i: integer;
5      procedure Proc111 (u, v: integer);
6        var c, d: integer;
7        begin
8          ... if u = 1 then Proc111(u+1, v); ...
9          v := (a + c) * (b - d); ...
10       end {Proc111}
11     begin
12       ... Proc111(1, x); ...
13     end {Proc11}
14   procedure Proc12;
15     var c, i: integer;
16     begin
17       a := 1; Proc11(c); ...
18     end {Proc12}
19   begin
20     a := 0;
21     Proc12;
22     ...
23   end {Proc1}
```





## 调用Proc12

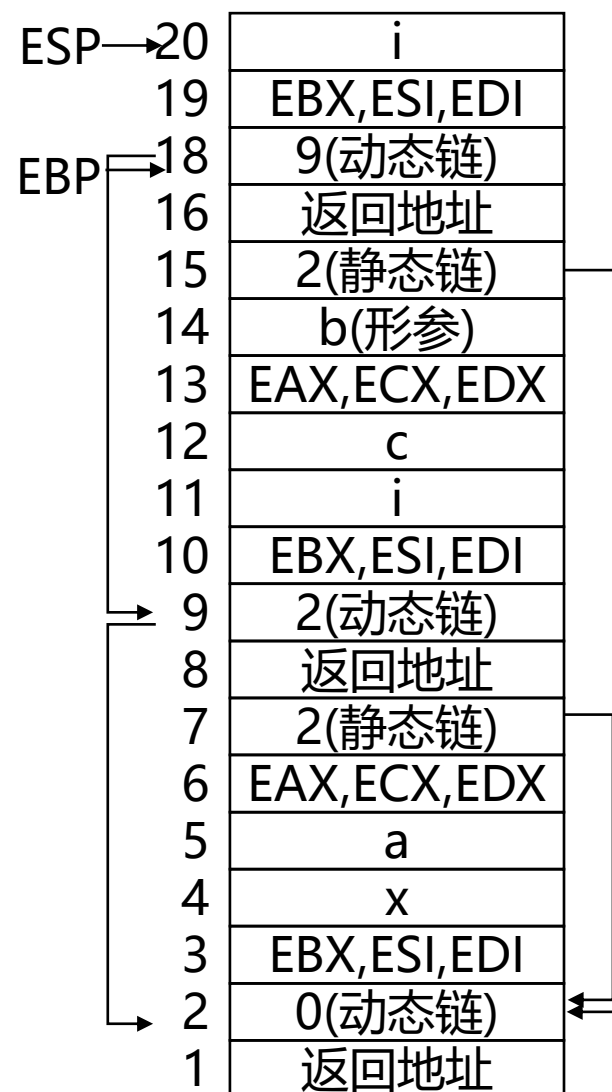
```
1  program Proc1;
2    var a, x: integer;
3    procedure Proc11 (b: integer);
4      var i: integer;
5      procedure Proc111 (u, v: integer);
6        var c, d: integer;
7        begin
8          ... if u = 1 then Proc111(u+1, v); ...
9          v := (a + c) * (b - d); ...
10       end {Proc111}
11     begin
12       ... Proc111(1, x); ...
13     end {Proc11}
14   procedure Proc12;
15     var c, i: integer;
16     begin
17       a := 1; Proc11(c); ...
18     end {Proc12}
19   begin
20     a := 0;
21     Proc12;
22     ...
23   end {Proc1}
```





# 调用Proc11

```
1  program Proc1;  
2    var a, x: integer;  
3    procedure Proc11 (b: integer);  
4      var i: integer;  
5      procedure Proc111 (u, v: integer);  
6        var c, d: integer;  
7        begin  
8          ... if u = 1 then Proc111(u+1, v); ...  
9          v := (a + c) * (b - d); ...  
10       end {Proc111}  
11     begin  
12       ... Proc111(1, x); ...  
13     end {Proc11}  
14   procedure Proc12;  
15     var c, i: integer;  
16     begin  
17       a := 1; Proc11(c); ...  
18     end {Proc12}  
19   begin  
20     a := 0;  
21     Proc12;  
22     ...  
23   end {Proc1}
```

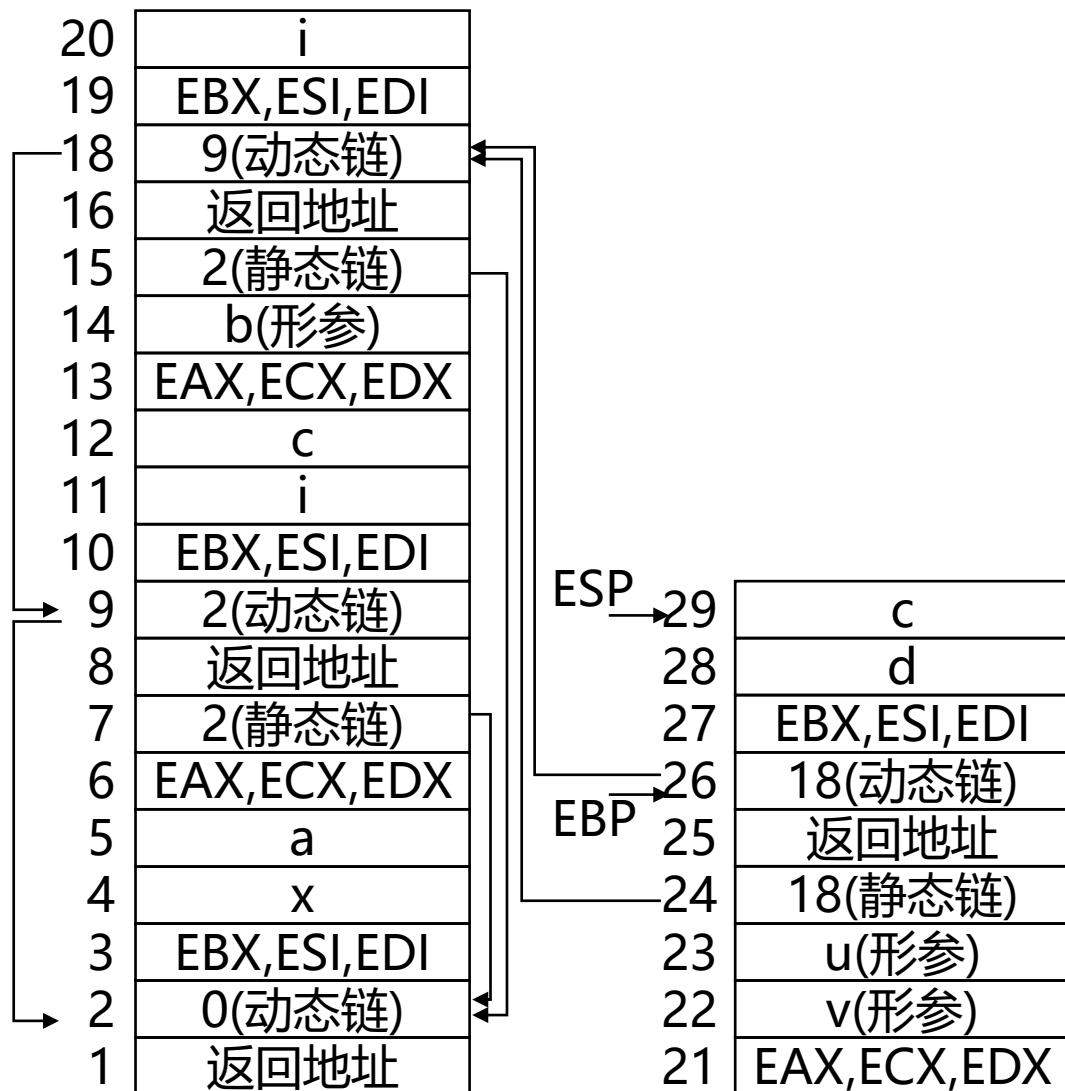






# 调用Proc111

```
1  program Proc1;  
2    var a, x: integer;  
3    procedure Proc11 (b: integer);  
4      var i: integer;  
5      procedure Proc111 (u, v: integer);  
6        var c, d: integer;  
7        begin  
8          ... if u = 1 then Proc111(u+1, v); ...  
9          v := (a + c) * (b - d); ...  
10       end {Proc111}  
11     begin  
12       ... Proc111(1, x); ...  
13     end {Proc11}  
14   procedure Proc12;  
15     var c, i: integer;  
16     begin  
17       a := 1; Proc11(c); ...  
18     end {Proc12}  
19   begin  
20     a := 0;  
21     Proc12;  
22     ...  
23   end {Proc1}
```



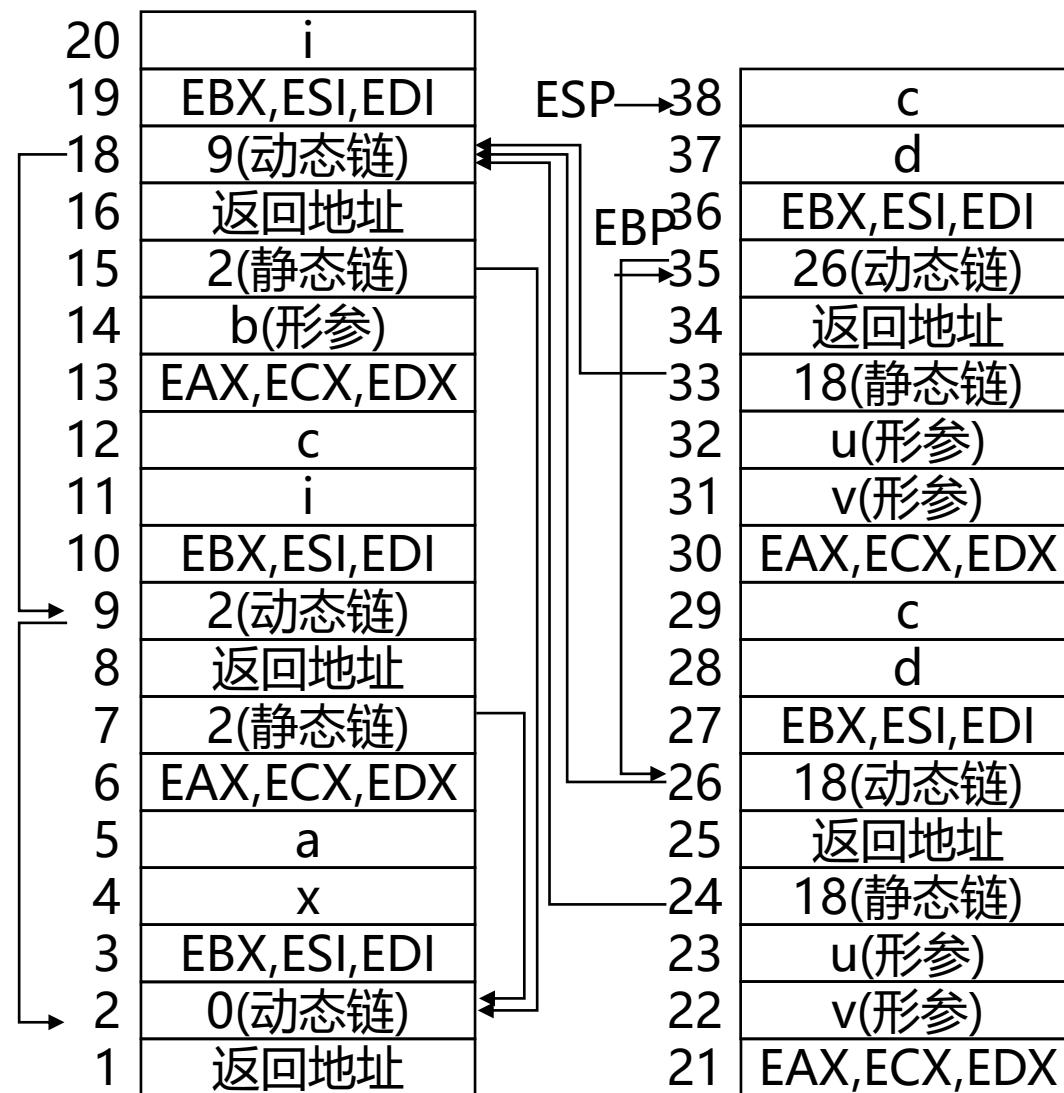


## 再次调用Proc111

```

1  program Proc1;
2    var a, x: integer;
3    procedure Proc11 (b: integer);
4      var i: integer;
5      procedure Proc111 (u, v: integer);
6        var c, d: integer;
7        begin
8          ... if u = 1 then Proc111(u+1, v); ...
9          v := (a + c) * (b - d); ...
10         end {Proc111}
11       begin
12         ... Proc111(1, x); ...
13       end {Proc11}
14     procedure Proc12;
15       var c, i: integer;
16       begin
17         a := 1; Proc11(c); ...
18       end {Proc12}
19     begin
20       a := 0;
21       Proc12;
22       ...
23   end {Proc1}

```



## 第六章 运行时存储空间组织

### □ 6.1 目标程序运行时的活动

- 6.1.1 运行时存储空间访问
- 6.1.2 栈帧结构
- 6.1.3 存储空间分配策略

### □ 6.2 过程调用规范

- 6.2.1 高级程序参数传递
- 6.2.2 std call
- 6.2.3 C调用规范
- 6.2.4 x64调用规范
- 6.2.5 寄存器保护
- 6.2.6 地址计算
- 6.2.7 ARM规范

### □ 6.3 运行时库

- 6.3.1 使用C运行时库输入输出
- 6.3.2 编译器生成输入输出代码

### ➤ 6.3.3 幂运算

### ➤ 6.3.4 跨文件调用

### ➤ 6.3.5 封装库

### □ 6.4 嵌套过程栈帧结构

### ➤ 6.4.1 静态链

### ➤ 6.4.2 静态链构建

### ➤ 6.4.3 外层变量访问

### ➤ 6.4.4 嵌套层次显示表

### ➤ 6.4.5 Display表构建

### ➤ 6.4.6 通过display访问变量

### □ 6.5 堆式存储分配

### ➤ 6.5.1 定长块管理

### ➤ 6.5.2 保留元数据

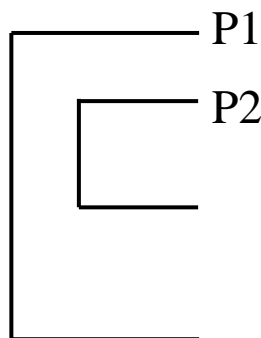
### ➤ 6.5.3 变长块管理

### ➤ 6.5.4 存储回收

## 6.4.2 静态链构建

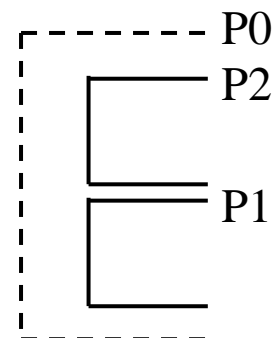
### □ P1 调用过程P2 时:

- 要么被调过程P2 是主调过程P1 的子过程;
- 要么被调过程P2 是和主调过程P1 平级的过程。



```

1  P1 proc
2      ...
3      ; 此处为压入实参结束位置
4      push ebp ; P2的静态链
5      call P2  ; 调用P2
6      ...
7  P1 endp
  
```



```

1  P1 proc
2      ...
3      ; 此处为压入实参结束位置
4      mov eax, [ebp + 8] ; 取出EBP+8处数据
5      push eax ; P2的静态链
6      call P2  ; 调用P2
7      ...
8  P1 endp
  
```

## 第六章 运行时存储空间组织

### □ 6.1 目标程序运行时的活动

- 6.1.1 运行时存储空间访问
- 6.1.2 栈帧结构
- 6.1.3 存储空间分配策略

### □ 6.2 过程调用规范

- 6.2.1 高级程序参数传递
- 6.2.2 std call
- 6.2.3 C调用规范
- 6.2.4 x64调用规范
- 6.2.5 寄存器保护
- 6.2.6 地址计算
- 6.2.7 ARM规范

### □ 6.3 运行时库

- 6.3.1 使用C运行时库输入输出
- 6.3.2 编译器生成输入输出代码

### ➤ 6.3.3 幂运算

### ➤ 6.3.4 跨文件调用

### ➤ 6.3.5 封装库

### □ 6.4 嵌套过程栈帧结构

### ➤ 6.4.1 静态链

### ➤ 6.4.2 静态链构建

### ➤ 6.4.3 外层变量访问

### ➤ 6.4.4 嵌套层次显示表

### ➤ 6.4.5 Display表构建

### ➤ 6.4.6 通过display访问变量

### □ 6.5 堆式存储分配

### ➤ 6.5.1 定长块管理

### ➤ 6.5.2 保留元数据

### ➤ 6.5.3 变长块管理

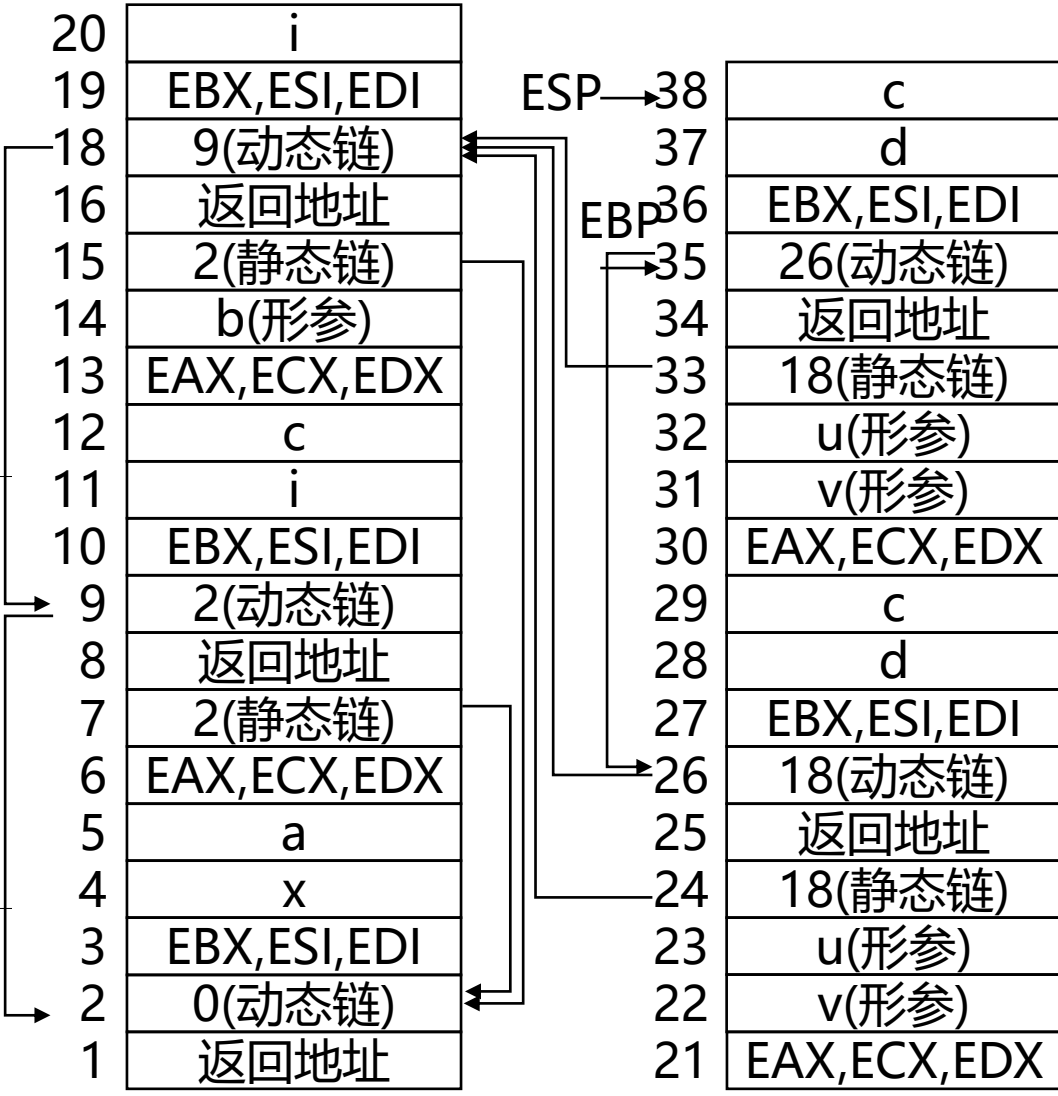
### ➤ 6.5.4 存储回收

# 6.4.3 外层变量访问

Proc111 访问Proc1中变量a:

- $\delta_a^v = 4, w_a^v = 4;$
- $\hat{\delta}_a^v = 4 + 4 + 12 = 20。$

```
1 Proc111 proc
2   ...
3   mov ebx, [ebp + 8] ; 取出Proc111静态链内容
4   mov ebx, [ebx + 8] ; 取出Proc11静态链内容
5   mov ebx, [ebx - 20] ; 取出a的值
6   ...
7 Proc111 endp
```



## 6.4.3 外层变量访问

---

### 算法 6.1 加载变量的目标代码生成

---

输入: 当前过程层次  $l_c$ , 变量声明所在过程层次  $l_p$ , 变量相对于 EBP 的偏移量为  $\hat{\delta}$ , 变量加载的目的寄存器  $R$

输出: 生成加载变量到寄存器  $R$  的代码

```
1 if  $l_c = l_p$  then
2   | gen("mov R, [ebp  $\pm$   $\hat{\delta}$ ]");
3 else if  $l_c > l_p$  then
4   | gen("mov R, [ebp + 8]");
5   | for  $i = 1 : l_c - l_p - 1$  do
6   |   | gen("mov R, [R + 8]");
7   | end
8   | gen("mov R, [R  $\pm$   $\hat{\delta}$ ]");
9 end
```

---

## 第六章 运行时存储空间组织

### □ 6.1 目标程序运行时的活动

- 6.1.1 运行时存储空间访问
- 6.1.2 栈帧结构
- 6.1.3 存储空间分配策略

### □ 6.2 过程调用规范

- 6.2.1 高级程序参数传递
- 6.2.2 std call
- 6.2.3 C调用规范
- 6.2.4 x64调用规范
- 6.2.5 寄存器保护
- 6.2.6 地址计算
- 6.2.7 ARM规范

### □ 6.3 运行时库

- 6.3.1 使用C运行时库输入输出
- 6.3.2 编译器生成输入输出代码

### ➤ 6.3.3 幂运算

### ➤ 6.3.4 跨文件调用

### ➤ 6.3.5 封装库

### □ 6.4 嵌套过程栈帧结构

### ➤ 6.4.1 静态链

### ➤ 6.4.2 静态链构建

### ➤ 6.4.3 外层变量访问

### ➤ 6.4.4 嵌套层次显示表

### ➤ 6.4.5 Display表构建

### ➤ 6.4.6 通过display访问变量

### □ 6.5 堆式存储分配

### ➤ 6.5.1 定长块管理

### ➤ 6.5.2 保留元数据

### ➤ 6.5.3 变长块管理

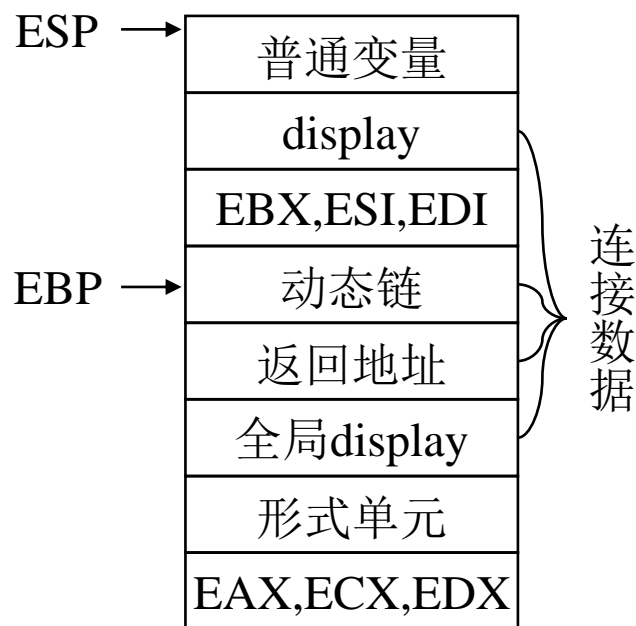
### ➤ 6.5.4 存储回收



## 6.4.4 嵌套层次显示表

□ 嵌套层次显示表 (display) : 引入指针数组, 指向本过程的所有外层。

- Display是一个小栈, 自栈顶向下依次指向当前层、直接外层、直接外层的直接外层、...、直至最外层 (0 层) 。
- 需要主调过程将直接外层的display 地址, 作为一个参数传递给被调过程, 称为全局嵌套层次显示表 (全局display)





# 运行Proc1

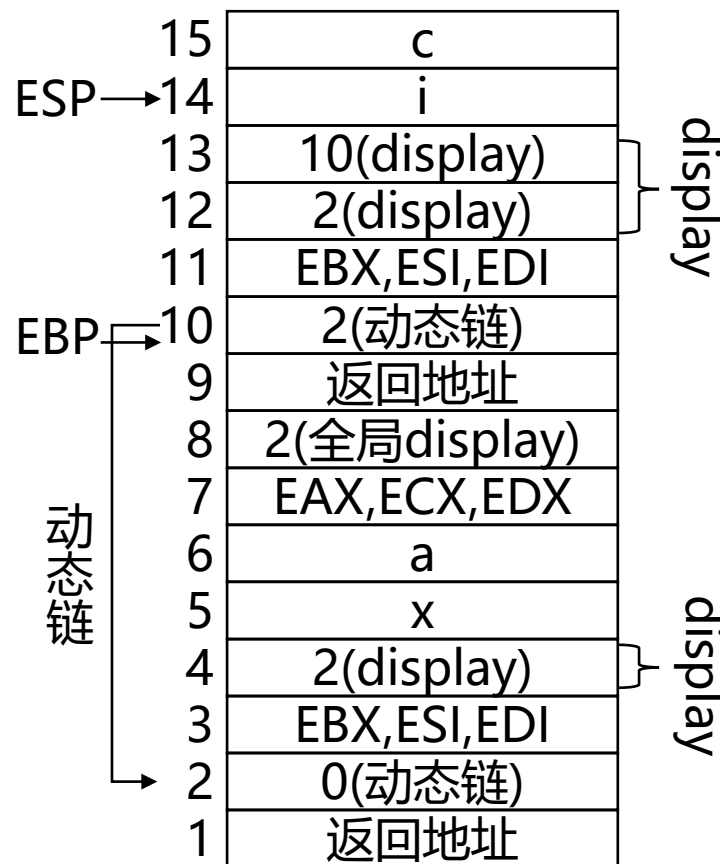
```
1  program Proc1;
2    var a, x: integer;
3    procedure Proc11 (b: integer);
4      var i: integer;
5      procedure Proc111 (u, v: integer);
6        var c, d: integer;
7        begin
8          ... if u = 1 then Proc111(u+1, v); ...
9          v := (a + c) * (b - d); ...
10       end {Proc111}
11     begin
12       ... Proc111(1, x); ...
13     end {Proc11}
14   procedure Proc12;
15     var c, i: integer;
16     begin
17       a := 1; Proc11(c); ...
18     end {Proc12}
19   begin
20     a := 0;
21     Proc12;
22     ...
23   end {Proc1}
```





# 调用Proc12

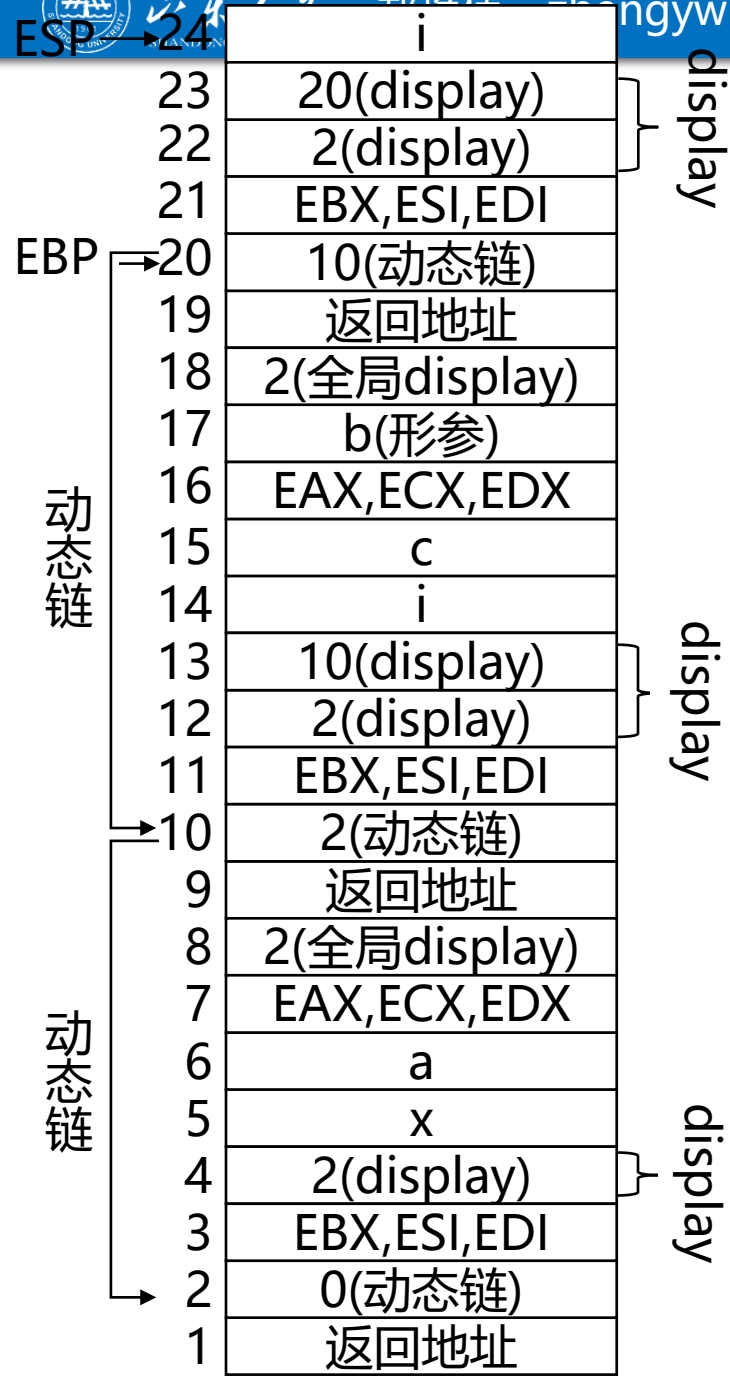
```
1  program Proc1;  
2    var a, x: integer;  
3    procedure Proc11 (b: integer);  
4      var i: integer;  
5      procedure Proc111 (u, v: integer);  
6        var c, d: integer;  
7        begin  
8          ... if u = 1 then Proc111(u+1, v); ...  
9          v := (a + c) * (b - d); ...  
10       end {Proc111}  
11     begin  
12       ... Proc111(1, x); ...  
13     end {Proc11}  
14   procedure Proc12;  
15     var c, i: integer;  
16     begin  
17       a := 1; Proc11(c); ...  
18     end {Proc12}  
19   begin  
20     a := 0;  
21     Proc12;  
22     ...  
23   end {Proc1}
```



```

1  program Proc1;
2      var a, x: integer;
3      procedure Proc11 (b: integer);
4          var i: integer;
5          procedure Proc111 (u, v: integer);
6              var c, d: integer;
7              begin
8                  ... if u = 1 then Proc111(u+1, v); ...
9                  v := (a + c) * (b - d); ...
10             end {Proc111}
11         begin
12             ... Proc111(1, x); ...
13         end {Proc11}
14     procedure Proc12;
15         var c, i: integer;
16         begin
17             a := 1; Proc11(c); ...
18         end {Proc12}
19     begin
20         a := 0;
21         Proc12;
22         ...
23     end {Proc1}
    
```

调用Proc11





# 调用 Proc111

```

1
2
3
4   var i: integer;
5   procedure Proc111 (u, v: integer);
6       var c, d: integer;
7       begin
8           ... if u = 1 then Proc111(u+1, v);
9           v := (a + c) * (b - d);
10        end {Proc111};
11    begin
12        ... Proc111(1, x); ...
13    end {Proc11};
14    procedure Proc12;
15        var c, i: integer;
16        begin
17            a := 1; Proc11(c); ...
18        end {Proc12};
19    begin
20        a := 0;
21        Proc12;
22        ...
23    end {Proc1}

```

);

动态链

24	i
23	20(display)
22	2(display)
21	EBX,ESI,EDI
20	10(动态链)
19	返回地址
18	2(全局display)
17	b(形参)
16	EAX,ECX,EDX
15	c
14	i
13	10(display)
12	2(display)
11	EBX,ESI,EDI
10	2(动态链)
9	返回地址
8	2(全局display)
7	EAX,ECX,EDX
6	a
5	x
4	2(display)
3	EBX,ESI,EDI
2	0(动态链)
1	返回地址

display

display

display

动态链

ESP

EBP

36	c
35	d
34	30(display)
33	20(display)
32	2(display)
31	EBX,ESI,EDI
30	20(动态链)
29	返回地址
28	20(全局display)
27	u(形参)
26	v(形参)
25	EAX,ECX,EDX

display

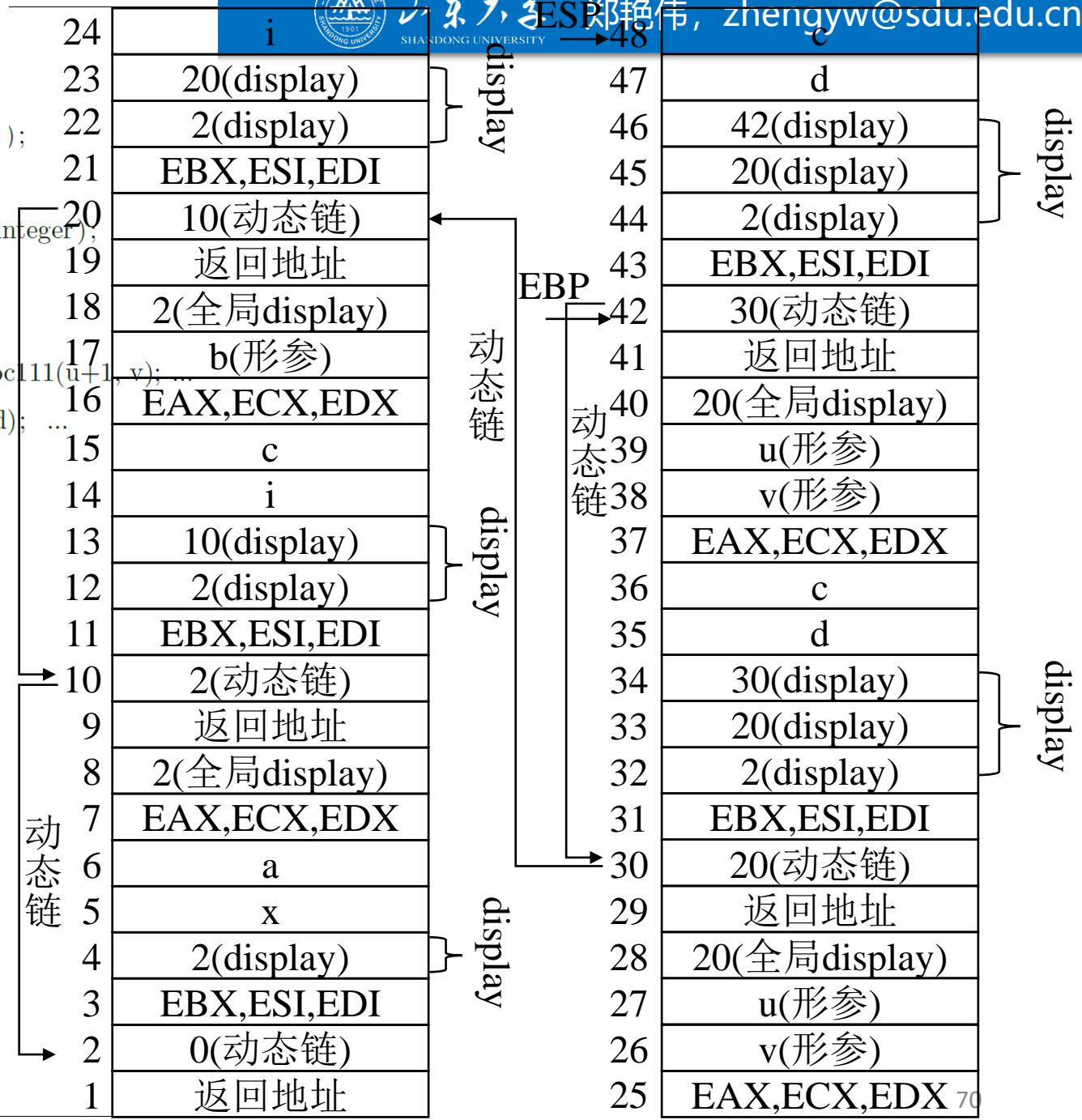


# 再次调用 Proc111

```

1  var i: integer;
2
3  );
4
5  procedure Proc111 (u, v: integer);
6      var c, d: integer;
7      begin
8          ... if u = 1 then Proc111(u+1, v); ...
9          v := (a + c) * (b - d); ...
10         end {Proc111}
11     begin
12         ... Proc111(1, x); ...
13     end {Proc11}
14 procedure Proc12;
15     var c, i: integer;
16     begin
17         a := 1; Proc11(c); ...
18     end {Proc12}
19 begin
20     a := 0;
21     Proc12;
22     ...
23 end {Proc1}

```



## 第六章 运行时存储空间组织

### □ 6.1 目标程序运行时的活动

- 6.1.1 运行时存储空间访问
- 6.1.2 栈帧结构
- 6.1.3 存储空间分配策略

### □ 6.2 过程调用规范

- 6.2.1 高级程序参数传递
- 6.2.2 std call
- 6.2.3 C调用规范
- 6.2.4 x64调用规范
- 6.2.5 寄存器保护
- 6.2.6 地址计算
- 6.2.7 ARM规范

### □ 6.3 运行时库

- 6.3.1 使用C运行时库输入输出
- 6.3.2 编译器生成输入输出代码

### ➤ 6.3.3 幂运算

### ➤ 6.3.4 跨文件调用

### ➤ 6.3.5 封装库

### □ 6.4 嵌套过程栈帧结构

### ➤ 6.4.1 静态链

### ➤ 6.4.2 静态链构建

### ➤ 6.4.3 外层变量访问

### ➤ 6.4.4 嵌套层次显示表

### ➤ 6.4.5 Display表构建

### ➤ 6.4.6 通过display访问变量

### □ 6.5 堆式存储分配

### ➤ 6.5.1 定长块管理

### ➤ 6.5.2 保留元数据

### ➤ 6.5.3 变长块管理

### ➤ 6.5.4 存储回收

## 6.4.5 Display表构建

```
1  P2 proc
2      push ebp    ; 保存动态链
3      mov ebp, esp ; 新EBP指向动态链位置
4      push ebx, esi, edi ; 保护寄存器, 为节省纸面空间写入了一行
5      mov eax, [ebp + 8] ; 取出全局display
6      mov ebx, [eax - 16] ; 第1个display
7      push ebx
8      mov ebx, [eax - 20] ; 第2个display
9      push ebx
10     ...          ; P2处于第n层, 就生成n-1个复制display的代码
11     push ebp     ; 本层display
12     ...
13 P2 endp
```



## 第六章 运行时存储空间组织

### □ 6.1 目标程序运行时的活动

- 6.1.1 运行时存储空间访问
- 6.1.2 栈帧结构
- 6.1.3 存储空间分配策略

### □ 6.2 过程调用规范

- 6.2.1 高级程序参数传递
- 6.2.2 std call
- 6.2.3 C调用规范
- 6.2.4 x64调用规范
- 6.2.5 寄存器保护
- 6.2.6 地址计算
- 6.2.7 ARM规范

### □ 6.3 运行时库

- 6.3.1 使用C运行时库输入输出
- 6.3.2 编译器生成输入输出代码

### ➤ 6.3.3 幂运算

### ➤ 6.3.4 跨文件调用

### ➤ 6.3.5 封装库

### □ 6.4 嵌套过程栈帧结构

### ➤ 6.4.1 静态链

### ➤ 6.4.2 静态链构建

### ➤ 6.4.3 外层变量访问

### ➤ 6.4.4 嵌套层次显示表

### ➤ 6.4.5 Display表构建

### ➤ 6.4.6 通过display访问变量

### □ 6.5 堆式存储分配

### ➤ 6.5.1 定长块管理

### ➤ 6.5.2 保留元数据

### ➤ 6.5.3 变长块管理

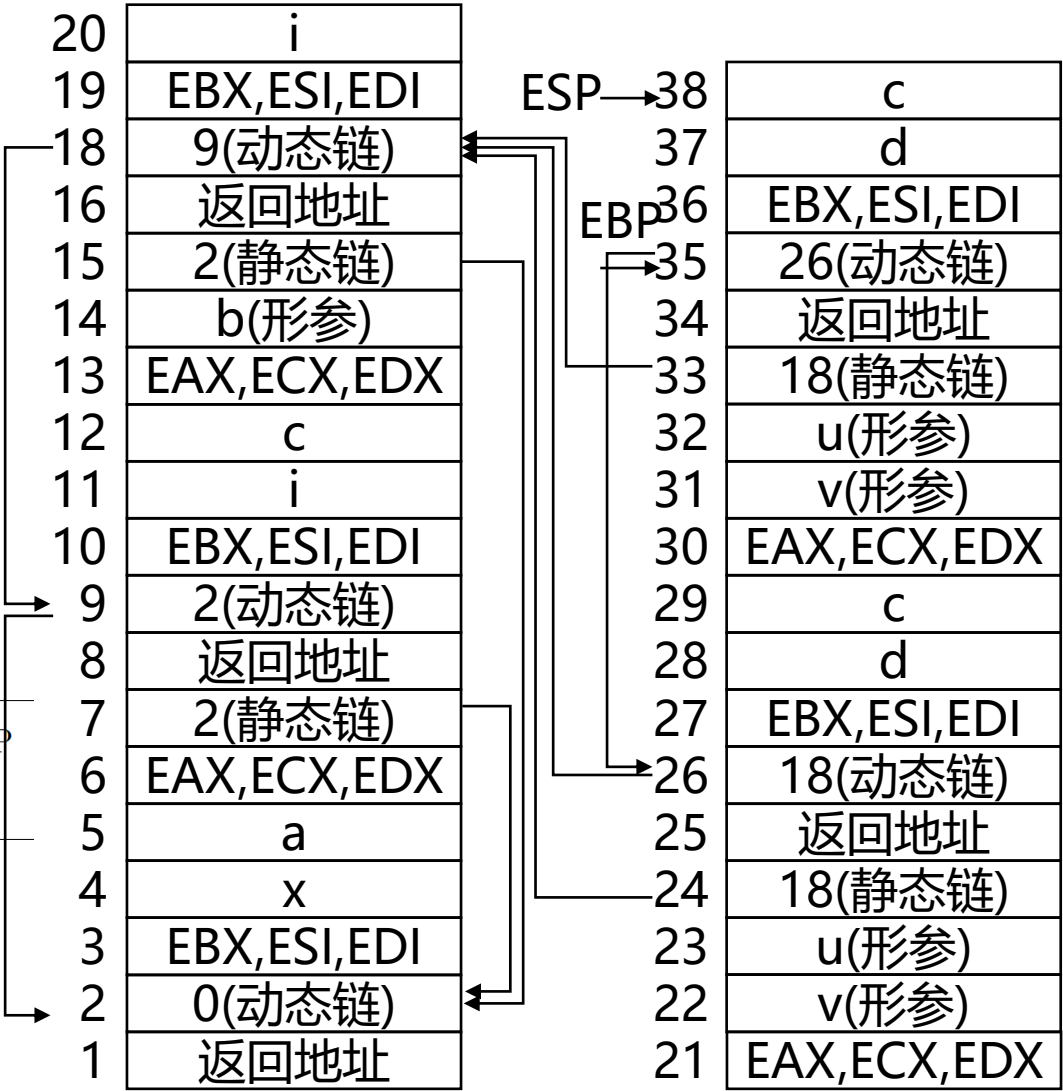
### ➤ 6.5.4 存储回收

# 6.4.6 通过display访问变量

Proc111 访问Proc1中变量a:

- 父过程层次为 $l_c$ , 则父过程display位置:  $dis(l_c) = 12 + 4(l_c + 1);$
- 本例Proc1:  $dis(l_c) = 12 + 4(0 + 1) = 16。$

```
1 mov ebx, [ebp - 16] ; 取到Proc1的动态链地址EBP
2 mov eax, [ebx - 24] ; 取到变量a的值
```



## 第六章 运行时存储空间组织

### □ 6.1 目标程序运行时的活动

- 6.1.1 运行时存储空间访问
- 6.1.2 栈帧结构
- 6.1.3 存储空间分配策略

### □ 6.2 过程调用规范

- 6.2.1 高级程序参数传递
- 6.2.2 std call
- 6.2.3 C调用规范
- 6.2.4 x64调用规范
- 6.2.5 寄存器保护
- 6.2.6 地址计算
- 6.2.7 ARM规范

### □ 6.3 运行时库

- 6.3.1 使用C运行时库输入输出
- 6.3.2 编译器生成输入输出代码

### ➤ 6.3.3 幂运算

### ➤ 6.3.4 跨文件调用

### ➤ 6.3.5 封装库

### □ 6.4 嵌套过程栈帧结构

### ➤ 6.4.1 静态链

### ➤ 6.4.2 静态链构建

### ➤ 6.4.3 外层变量访问

### ➤ 6.4.4 嵌套层次显示表

### ➤ 6.4.5 Display表构建

### ➤ 6.4.6 通过display访问变量

### □ 6.5 堆式存储分配

### ➤ 6.5.1 定长块管理

### ➤ 6.5.2 保留元数据

### ➤ 6.5.3 变长块管理

### ➤ 6.5.4 存储回收

## 运行时存储空间访问

### □ 堆区地址的获取

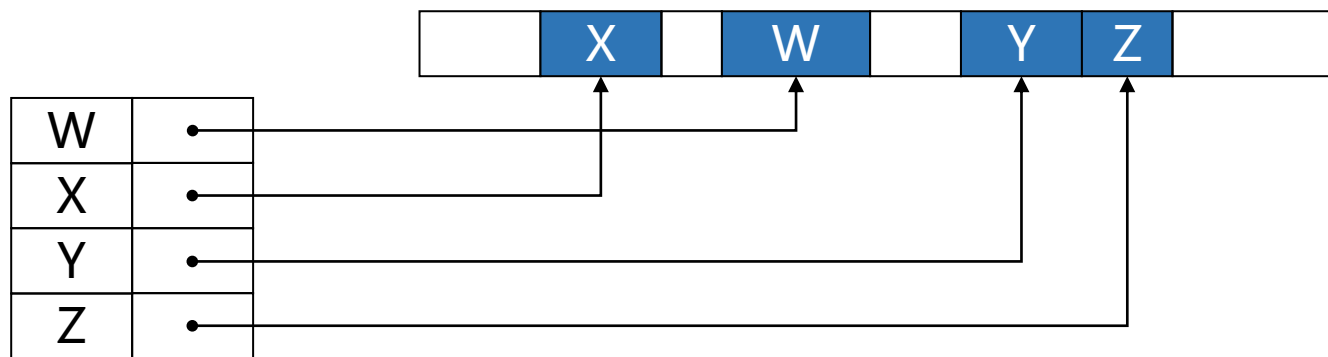
- 注意32位以后的OS，很多将堆管理放入系统内核，各区也不再连续，此方法不适用。

```
1  .386
2  .model flat, stdcall
3  .stack 4096 ; 栈长度
4  .data
5      available dword ? ; 定义变量available存储堆地址
6  .code
7      main proc
8          mov ebx, esp ; 在对栈做任何操作之前调用，因此是栈的起始地址
9          sub ebx, 4096 ; 减去栈长度，即堆起始地址
10         mov available, ebx ; 保存到变量
11         ...
12     main endp
13     end main
```

## 堆式存储分配

□ 当运行程序请求一块体积为 $N$ 的空间时，**应该分配哪一块？**

- 从比 $N$ 稍大的一个空闲块中取出 $N$ 个单元，以便使大的空闲块派更大的用场；
- 先碰上哪块比 $N$ 大，就从其中分出 $N$ 个单元。



□ 运行一段时间后，空间会零碎不堪

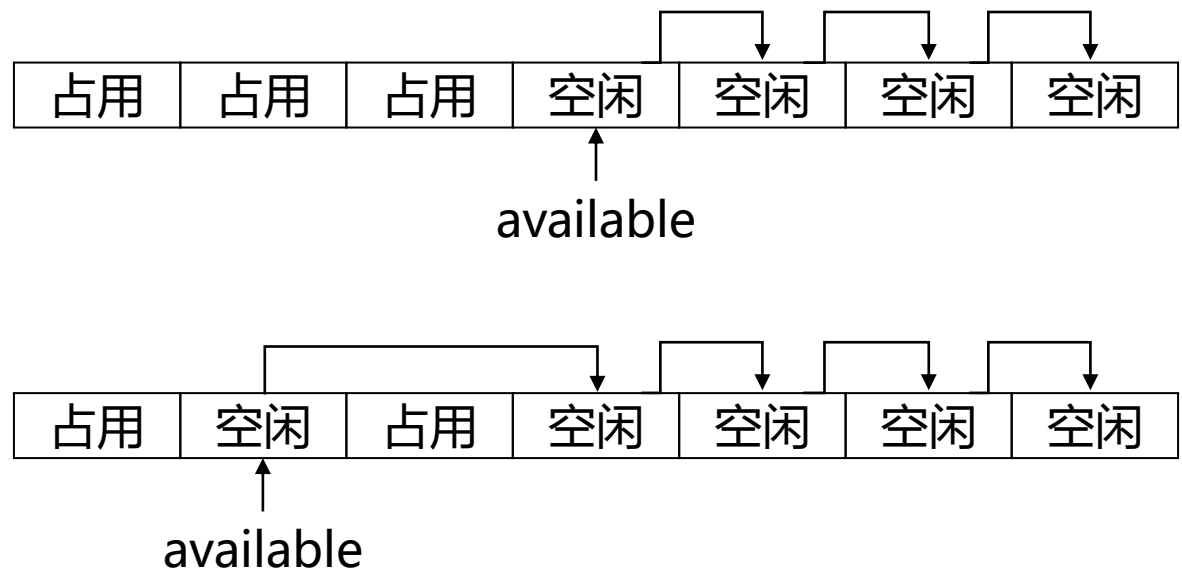
- 没有任何一块比 $N$ 大，但总和比 $N$ 大的多。

□ 总和也比 $N$ 小，如何收回空间？

- 垃圾回收的机制如何设计。

# 堆式动态分配的实现

## □ 定长块管理



## 堆式动态分配的实现

### □ 变长块管理

- 初始化时，堆区为一整块；
- 用户申请时，从一个整块里分割出满足需要的一小块；
- 用户释放时，如果释放块能和现有空闲块合并，则合并，不能则成链。

### □ 变长块分配策略

- **首次满足法**：只要在空闲块链表中找到满足需要的一块，就分配；如果该块比申请的块大不了多少，就整块分配出去。
- **最优满足法**：将空闲链中不小于申请块，且最接近申请块的空闲块分配给用户；需要将链表块从小到大排序，可能会产生很多小碎片块。
- **最差满足法**：将空闲块中不小于申请块，且最大的空闲的一部分分配给用户；此时链表块从大到小排序，分配时不需要查找，最终结点趋于均匀。

## 隐式存储回收

- **隐式存储回收**：垃圾回收子程序与用户程序并行工作，需要知道分配给用户程序的存储块何时不再使用。
- 第一个阶段为**标记阶段**，对已分配的块跟踪程序中各指针的访问路径，如果某个块被访问过，就给这个块加个标记。
  - 第二个阶段为**回收阶段**，所有未加标记的存储块回收到一起，并插入空闲块链表中，然后消除在存储块中所加的全部标记。

块长度
访问计数标记
指针
用户使用空间

存储块格式





山东大学  
SHANDONG UNIVERSITY

## 第六章 运行时存储空间组织

*The End*

谢谢

授 课 教 师 : 郑艳伟  
手 机 : 18614002860 (微信同号)  
邮 箱 : zhengyw@sdu.edu.cn