第9章 运行时的存储组织

重点： 符号表的内容、组织，过程调用实现，
静态存储分配、动态存储分配的基本方法。

难点： 参数传递，过程说明语句代码结构，
过程调用语句的代码结构，
过程调用语句的语法制导定义，
栈式存储分配。

第9章 运行时的存储组织

9.1 与存储组织有关的源语言概念与特征

9.2 存储组织

9.3 静态存储分配

9.4 栈式存储分配

9.5 栈中非局部数据的访问

9.6 堆管理

9.7 本章小结

9.1 与存储组织有关的源语言概念与特征

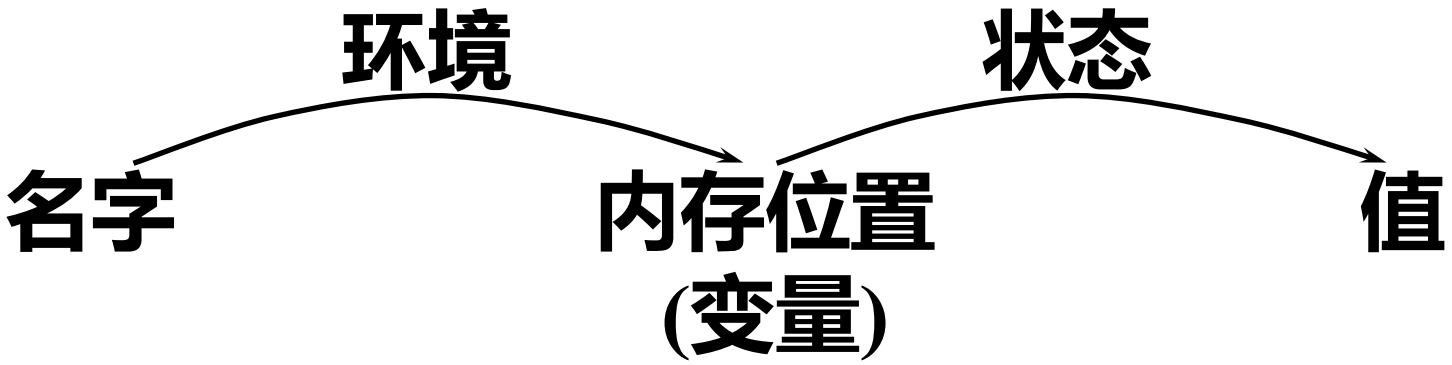
- 给定一个源程序，编译程序必须根据源语言的特征为源程序中的许多问题做出决策，包括何时、怎样为名字分配内存地址。
  - **静态策略**：在编译时即可做出决定的策略
  - **动态策略**：直到程序执行时才能做出决定的策略

9.1.1 名字及其绑定

- “名字”、“变量”和“标识符” 的区别与联系
  - 名字和变量分别表示编译时的名字和运行时该名字所代表的内存位置。
  - 标识符则是一个字符串，用于指示数据对象、过程、类或对象的入口。
  - 所有标识符都是名字，但并不是所有的名字都是标识符，名字还可以是表达式。例如，名字x.y可能表示x所代表结构的域y。
  - 同一标识符可以被声明多次，但每个声明都引入一个新的变量。即使每个标识符只被声明一次，局部于某个递归过程的标识符在不同的运行时刻也将指向不同的内存位置。

名字的绑定

- 从名字到值的两步映射
  - 环境把名字映射到左值，而状态把左值映射到右值
  - 赋值改变状态，但不改变环境。



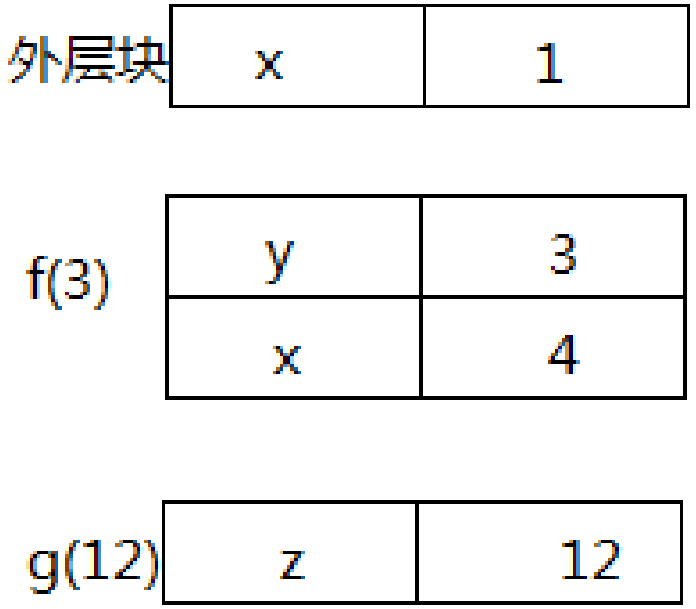
9.1.2 声明的作用域

- x的声明的作用域是程序中的这样一段区域，在该区域中，x的引用均指向x的这一声明。
- 对于某种程序设计语言，如果只通过考察其程序就可以确定某个声明的作用域，则称该语言使用**静态作用域**；否则称该语言使用**动态作用域**。
- C++、Java和C#等还提供了对**作用域的显式控制**，其方法是使用public、private和protected这样的关键字。

9.1.2 声明的作用域

- **静态作用域和动态作用域**
  - **静态作用域规则：g(int z)中的x = 1**
  - **动态作用域规则：g(int z)中的x = 4**

```
1 int x = 1;
2 int g(int z) { return x + z; }
3 int f(int y)
4 {
5         int x = y + 1;
6         return g(y*x);
7 }
8 f(3);
```



1. 静态作用域

- 在具有程序块结构的语言中，变量声明的静态作用域规则如下：
  - 如果名字x的声明D属于程序块B，则D的作用域是B的所有语句，只有满足如下条件的程序块B'除外：B'嵌套在B中(可以是任意的嵌套深度)，且B'中具有同一名字x的一个新的声明。

1. 静态作用域

```
(1)  int main()
(2)  {
(3)     int a=0;
(4)     int b=0;
(5)     {
(6)        int b=1;
(7)        {
(8)           int a=2;
(9)           printf("%d  %d\n", a, b );
(10)       }
(11)       {
(12)          int b=3;
(13)          printf("%d  %d\n", a, b);
(14)       }
(15)       printf("%d  %d\n", a, b);
(16)    }
(17)    printf("%d  %d\n", a, b);
(18) }
```

考虑如下声明的作用域

(3) int a = 0;
(4) int b = 0;
(6) int b = 1;
(8) int a = 2;
(12) int b = 3;

2. 显式访问控制

- 类和结构为其成员引入了一种新的作用域
  - 如果p是某个带有域(成员)x的类的对象，则p.x中x的引用将指向该类定义中的域x。
  - 与程序块结构类似的是，类D中成员x的声明的作用域将会扩展到D的任何子类D'，除非D'中具有同一名字x的一个局部声明。

2. 显式访问控制

- 通过使用像public、 private和protected这样的关键字， C++或Java类的面向对象语言提供了一种对超类中成员名字的显式访问控制。
- 这些关键字通过限制访问来支持封装。
- 私有名的作用域只包含与该类及其友类相关联的方法声明和定义
- 保护名只对其子类是可访问的
- 公用名从类的外部也是可以访问的。

3. 动态作用域

- 动态作用域规则相对于时间，而静态作用域规则相对于空间
  - 静态作用域规则要求我们找出某个引用所指向的声明，条件是该声明处在包围该引用的“空间上最近的”单元(程序块)中。
  - 动态作用域也是要求我们找出某个引用所指向的声明，但条件是该声明处在包围该引用的“时间上最近的”单元(过程活动)中。

9.1.3 过程及其活动

- 将“过程、函数和方法”统称为“过程”
- **过程定义**是把一个标识符和一组语句联系起来。该标识符是**过程名**，这组语句是**过程体**。
- 当过程名出现在可执行语句中时，称相应的过程在该点被调用。**过程调用**就是执行被调用过程的过程体。

9.1.3 过程及其活动

- 出现在过程定义中的某些标识符具有特殊的意义，称为该过程的形式参数，简称为**形参**。
- 调用过程时，表达式作为实在参数(或**实参**)传递给被调用的过程，以替换出现在过程体中的对应形式参数。

9.1.3 过程及其活动

- 过程体的每次执行叫做该过程的一个**活动**。
- 过程p的一个活动的生存期是从过程体执行的第一步到最后一步，包括执行被p调用的过程的时间，以及再由这样的过程调用其它过程所花的时间，等等。

9.1.3 过程及其活动

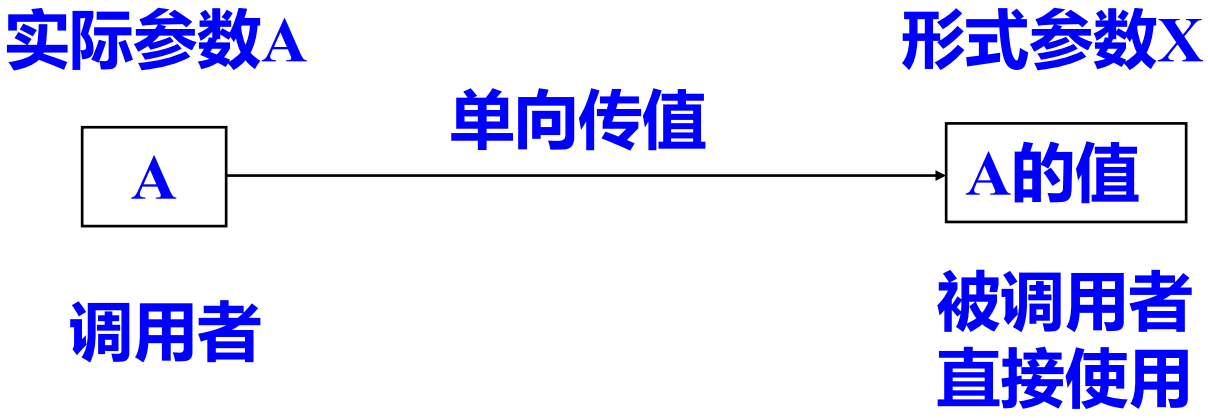
- 如果*a*和*b*是过程的活动，那么它们的生存期或者不交迭，或者嵌套。也就是说，如果在a结束之前b就开始了，那么b必须在a结束之前结束。
- 如果同一个过程的一次新的活动可以在前一次活动结束前开始，则称这样的过程是递归的。递归过程p也可以间接地调用自己。
- 如果某个过程是递归的，则在某一时刻可能有它的几个活动同时活跃，这时必须合理组织这些同时活跃着的活动的内存空间。

9.1.4 参数传递方式

- 当一个过程调用另一个过程时，它们之间交换信息的方法通常是通过非局部名字和被调用过程的参数来实现的。
  - 传值
  - 传地址
  - 传值结果
  - 传名
- 其主要区别在于实参所代表的究竟是左值、右值还是实参的正文本身

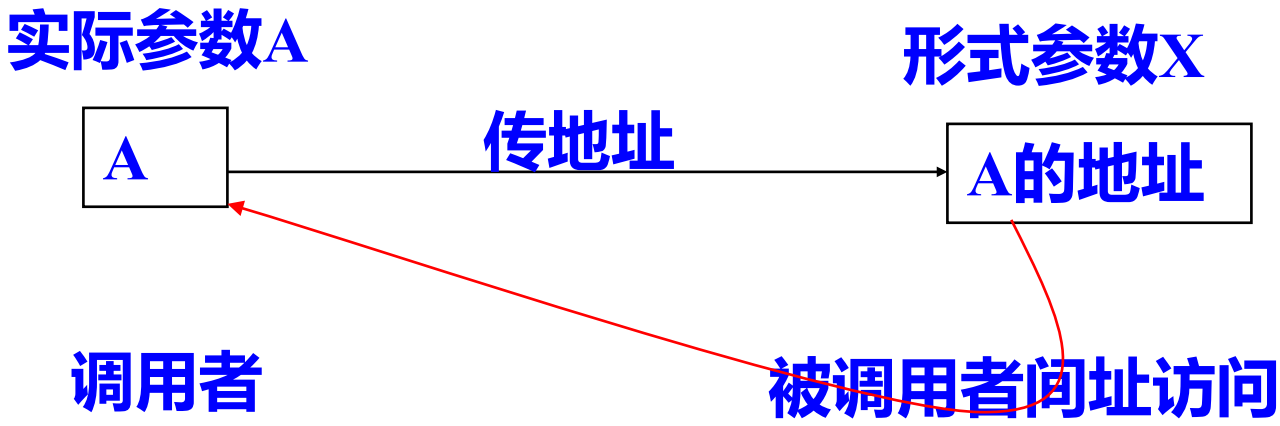
1. 传值

- 计算实参并将其右值传递给被调用过程
- 传值方式可以如下实现：
  - 被调用过程为每个形参开辟一个称为形式单元的存储单元，用于存放相应实参的值。
  - 调用过程计算实参，并把右值放入相应的形式单元中。



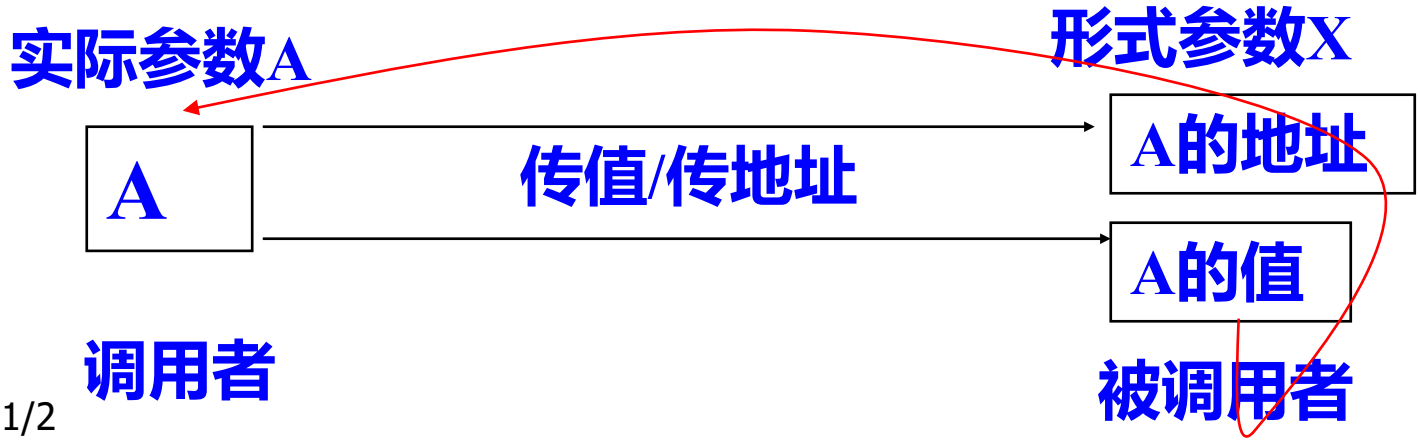
2. 传地址

- 调用过程将实参的地址传递给被调用过程
- 传地址方式可以如下实现：
  - 如果实参是一个具有左值的名字或表达式，则传递该左值本身
  - 如果实参是a+b或2这样的没有左值的表达式，则调用过程首先计算实参的值并将其存入一个新的存储单元，然后将这个新单元的地址传递给被调用过程



3. 传值结果

- 传值结果就是将传值和传地址这两种方式结合起来
- 传值结果方式可以如下实现：
  - 实参的右值和左值同时传给被调用过程。
  - 在被调用过程中，像传值方式那样使用实参的右值。
  - 当控制返回调用过程时，根据传递来的实参的左值，将形参当前的值复制到实参存储单元。



4. 传名

- 用实参表达式对形参进行文字替换，类似于宏替换。
- 传名方式可以如下实现：
  - 在调用过程中设置计算实参左值或右值的形实转换子程序。
  - 被调用过程为每个形参开辟一个存储单元，用于存放该实参的形实转换子程序的入口地址。
  - 被调过程执行时，每当要向形参赋值或取该形参的值时，就调用相应于该形参的形实转换子程序，以获得相应的实参地址或值。

主程序

A:=2; B:=3;

P(A+B, A, A);

print A

子程序

P(X,Y,Z);

{Y:=Y+1;

Z:=Z+X}

传名

A:=A+1=3

A:=A+A+B=3+3+3

9

临时单元:
T:A+B=5

传值结果:

X=T=5,Y=Z=A=2

Y:=Y+1=3

Z:=Z+X=2+5=7

Y → A=3

Z → A=7

7

传地址:

X=T=5,Y=Z=A=2

A:=A+1=2+1

A:=A+T=3+5

8

传值:

2

9.2 存储组织

- 9.2.1 运行时内存的划分
- 9.2.2 活动记录
- 9.2.3 局部数据的组织
- 9.2.4 全局存储分配策略

9.2.1 运行时内存的划分

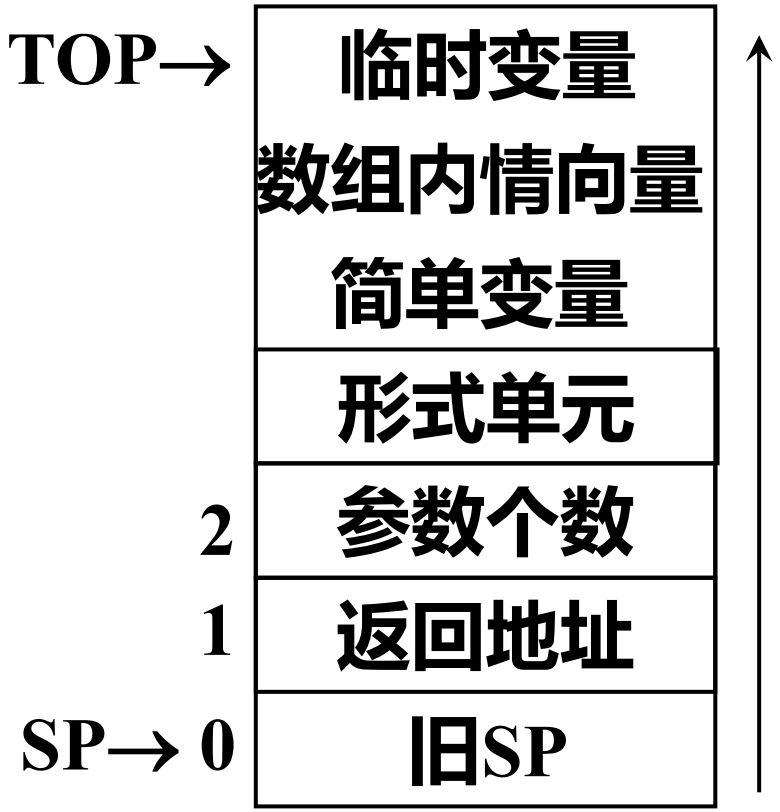
- 为运行目标程序，编译器需要先获得一块存储区，并划分成块

9.2.2 活动记录

- 过程的每个活动所需要的信息用一块连续的存储区来管理，这块存储区叫做**活动记录**
- 采用**栈式存储分配**机制
- 活动记录：运行时，每当进入一个过程就有一个相应的活动记录压入栈顶。活动记录一般含有连接数据、形式单元、局部变量、局部数组的内情向量和临时工作单元等。
- 数组的内情向量，包括数组的类型、维数、各维的上下界及数组首地址等

每个过程的活动记录内容

——非嵌套语言(如C)



对任何局部变量X的引用可表示为变址访问:

dx[SP]

dx:变量X相对于活动记录起点的地址，在编译时可确定。

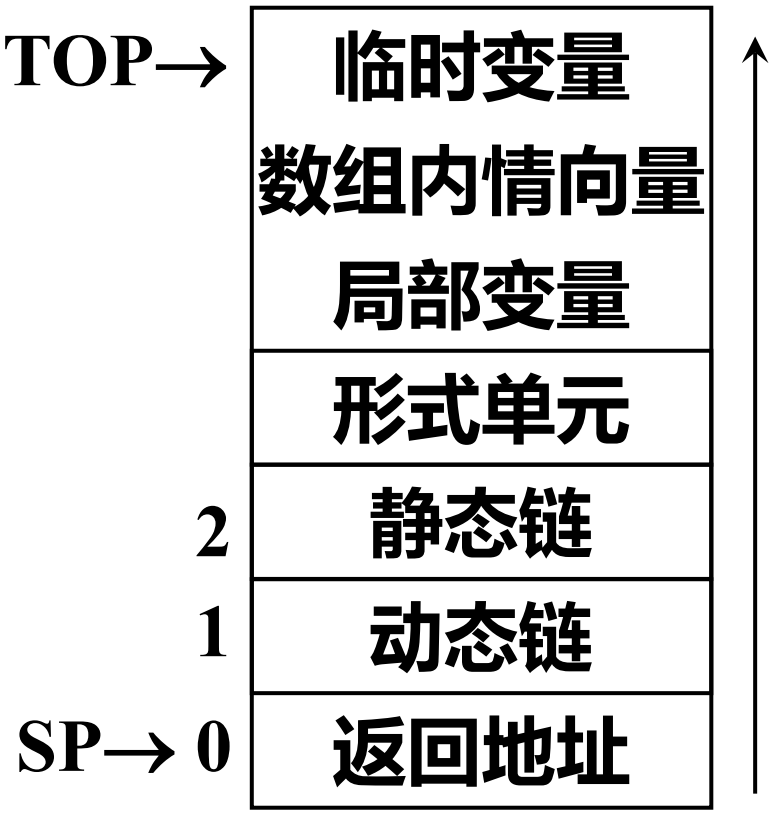
指针**sp**指向现行过程的活动记录在栈中的起始位置，**top**指向栈顶

每个过程的活动记录内容

——嵌套语言(如Pascal)

- 连接数据
  - 返回地址
  - 动态链：指向调用者的活动记录
  - 静态链：访问存放在其他活动记录中的非局部数据

每个过程的活动记录内容



- 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、临时工作单元(如存放对表达式求值的结果)。

9.2.3 局部数据的组织

- 局部数据域在编译过程中处理声明语句时进行分配
- 变量所需的存储空间可以根据其类型而静态确定。
- 一个过程所声明的局部变量，按这些变量声明时出现的次序，在局部数据域中依次分配空间。
- 局部数据的地址可以用相对于某个位置的地址来表示。
- 数据对象的存储安排还有对齐的问题，整数必须放在内存中特定的位置

9.2.3 局部数据的组织

在SPARC/Solaris工作站上下面两个结构的size分别是24和16，为什么不一样?

```
typedef struct  _a{
        char c1;
        long  i;
        char c2;
        double f;
}a;
```

```
typedef struct  _b{
        char c1;
        char c2;
        long i;
        double f;
}b;
```

9.2.3 局部数据的组织

在SPARC/Solaris工作站上下面两个结构的size分别是24和16，为什么不一样?

```
typedef struct  _a{
        char c1;    0
        long  i;    4
        char c2;    8
        double f;   16
}a;
```

```
typedef struct  _b{
        char c1;    0
        char c2; 1
        long i;  4
        double f;8
}b;
```

整数对齐到4，双精度对齐到8

9.2.3 局部数据的组织

在X86/Linux机器的结果和SPARC/Solaris工作站不一样，是20和16。

```
typedef struct  _a{
        char c1;    0
        long  i;    4
        char c2;    8
        double f;   12
}a;
```

```
typedef struct  _b{
        char c1;    0
        char c2; 1
        long i;  4
        double f;8
}b;
```

双精度对齐到4

9.2.4 全局存储分配策略

- **静态存储分配策略(FORTRAN)**

  如果在编译时能确定数据空间的大小，则可采用静态分配方法：在编译时刻为每个数据项目确定出在运行时刻的存储空间中的位置。

- **动态存储分配策略(PASCAL)**

  如果在编译时不能确定运行时数据空间的大小，则必须采用动态分配方法。允许递归过程及动态申请、释放内存。

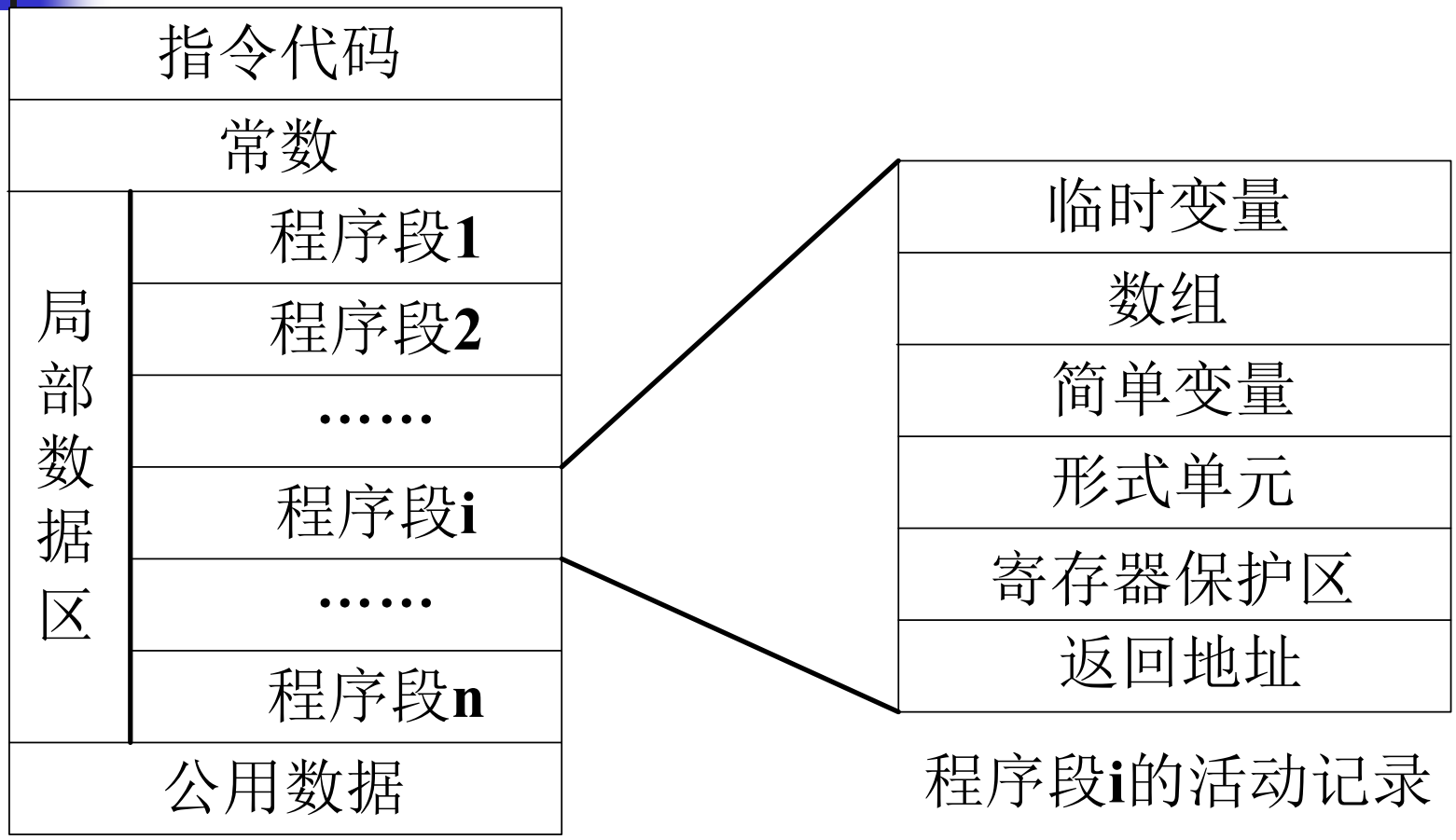
9.3 静态存储分配策略

- 如果在编译时就能确定一个程序在运行时所需要的存储空间的大小
- 在编译时就能安排好目标程序运行时的全部数据空间
- 确定每个数据项的地址

9.3 静态存储分配策略

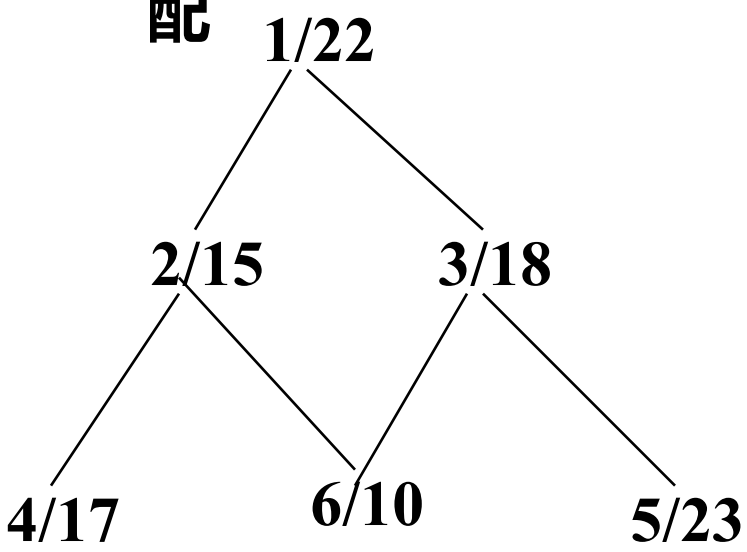
- 必须满足如下条件：
  - 数据对象的长度和它在内存中的位置在编译时必须是已知的；
  - 不允许过程的递归调用；
  - 数据结构不能动态建立
- 在编译时，编译器知道所有数据对象的偏移位置

某分段式程序运行时的内存划分



静态存储分配策略-组织局部数据区的方法

- 顺序分配算法（基于调用图）
  - 不考虑调用关系，按照程序段出现的先后顺序逐段分配



程序段号/所需数据空间

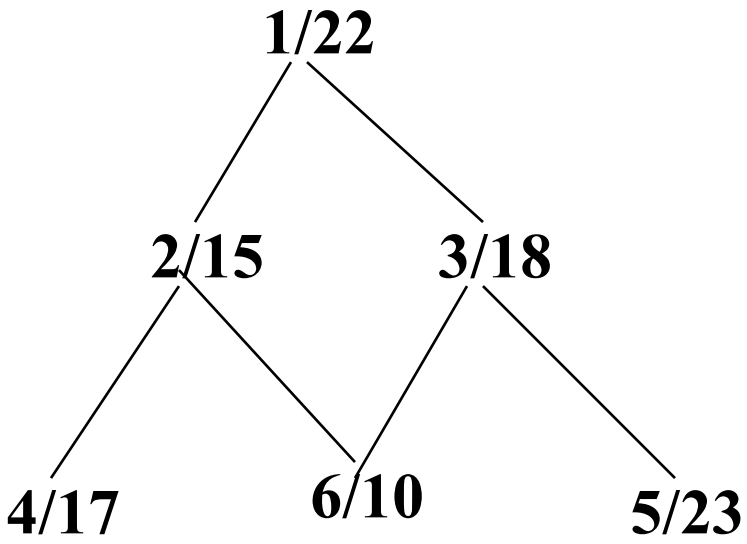
程序段	区域
1	0~21
2	22~36
3	37~54
4	55~71
5	72~94
6	95~104

共需要105个存储单元

能用更少的空间么?

分层分配算法

- 允许程序段之间的覆盖（覆盖可能性分析）



程序段	区域
6	0~9
5	0~22
4	0~16
3	23~40
2	23~37
1	41~62

共需要63个存储单元

9.4 栈式存储分配策略

- 过程的每次调用都将触发一个活动
- 过程的每个活动所需要的数据被组织成活动记录，必须保存每个活动相应的数据区内容
- 引入一个运行栈
  - 让过程的每次执行和过程的活动记录相对应。
  - 每调用一次过程，就把该过程的活动记录压入栈中，返回时弹出。

9.4.1 调用序列和返回序列

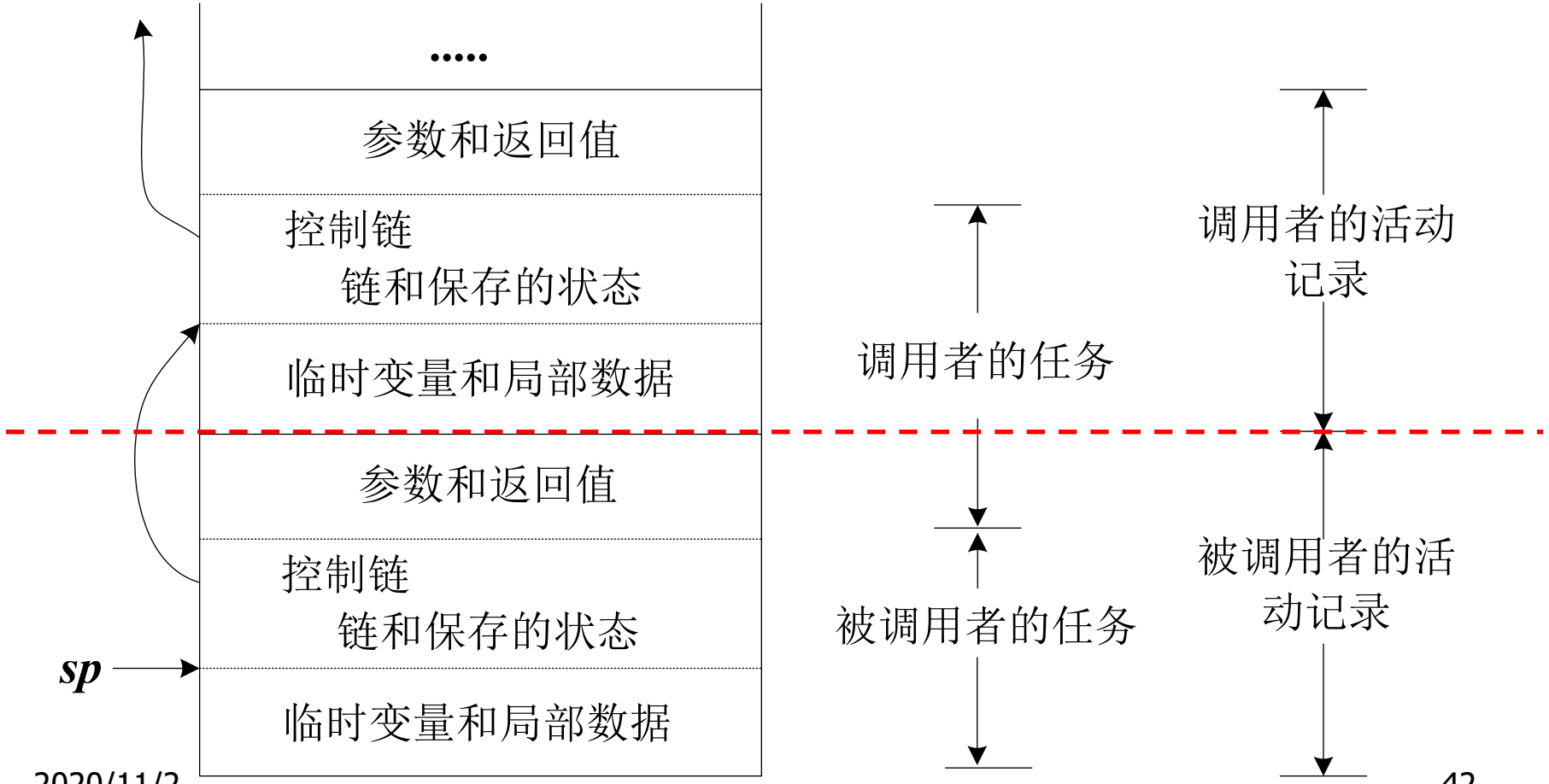
- 过程调用和过程返回都需要执行一些代码来管理活动记录栈，保存或恢复机器状态等
- 过程调用和返回是通过在目标代码中生成调用序列和返回序列来实现的
  - 调用序列负责分配活动记录，并将相关信息填入活动记录中
  - 返回序列负责恢复机器状态，使调用过程能够继续执行
- 调用序列和返回序列常常都分成两部分，分别处于调用过程和被调用过程中

9.4.1 调用序列和返回序列

- 即使是同一种语言，过程调用序列、返回序列和活动记录中各域的排放次序，也会因实现而异
- 设计这些序列和活动记录的一些原则：
  - 把参数域和可能有的返回值域放在紧靠调用者活动记录的地方
  - 用同样的代码来执行各个活动的状态保存和恢复

9.4.1 调用序列和返回序列

调用者和被调用者之间的任务划分



9.4.1 调用序列和返回序列

一种可能的调用序列：

- 调用者计算实参
- 调用者把返回地址和*sp*的旧值存入被调用者的活动记录中，重新设置*sp*的新值(将*sp*移过调用者的局部数据域和临时变量域，以及被调用者的参数域和状态域)
- 被调用者保存寄存器值和其它机器状态信息
- 被调用者初始化其局部数据，并开始执行。

9.4.1 调用序列和返回序列

一种可能的返回序列：

- 被调用者将返回值放入临近调用者的活动记录的地方。
- 利用状态域中的信息，被调用者恢复*sp*和其它寄存器，并且按调用者代码中的返回地址返回。
- 尽管*sp*的值减小了，调用者仍然可以将返回值复制到自己的活动记录中

9.4.1 调用序列和返回序列

过程的参数个数可变的情况

- 如C语言的printf
- 编译器产生将这些参数逆序进栈的代码
- 被调用函数能准确地知道第一个参数的位置
- 被调用函数根据第一个参数到栈中取第二、第三个参数等等

9.4.2 C语言的过程调用和返回

- **过程调用的三地址码:**

param x_1

param x_2

…

param x_n

call p，n

对于par和call产生的目标代码

1) 每个param x_i(i=1,2,…n)可直接翻译成如下指令:

(i+3)[top]:= x_i (传值)

(i+3)[top]:=addr(x_i) (传地址)

2) call p，n 被翻译成如下指令:

1[top]:=sp (保护现行sp)

3[top]:=n (传递参数个数)

JSR p (转子指令)

3) 进入过程p后，首先应执行下述指令：

sp:=top+1　　(定义新的sp)

1[sp]:=返回地址　(保护返回地址)

top:=top+L　　(新top)

L：过程P的活动记录所需单元数，在编译时可确定。

4) 过程返回时，应执行下列指令:

top:=sp-1　　(恢复调用前top)

sp:=0[sp]　　(恢复调用前SP)

X:=2[top]　　(把返回地址取到X中)

UJ 0[X]　　(按X返回)

UJ为无条件转移指令，
即按X中的返回地址实行变址转移

9.4.3 栈中的可变长数据

活动记录的长度在编译时不能确定的情况

- 局部数组的大小要等到过程激活时才能确定
- 在活动记录中为这样的数组分别存放数组指针的单元
- 运行时，这些指针指向分配在栈顶的存储空间

9.4.3 栈中的可变长数据

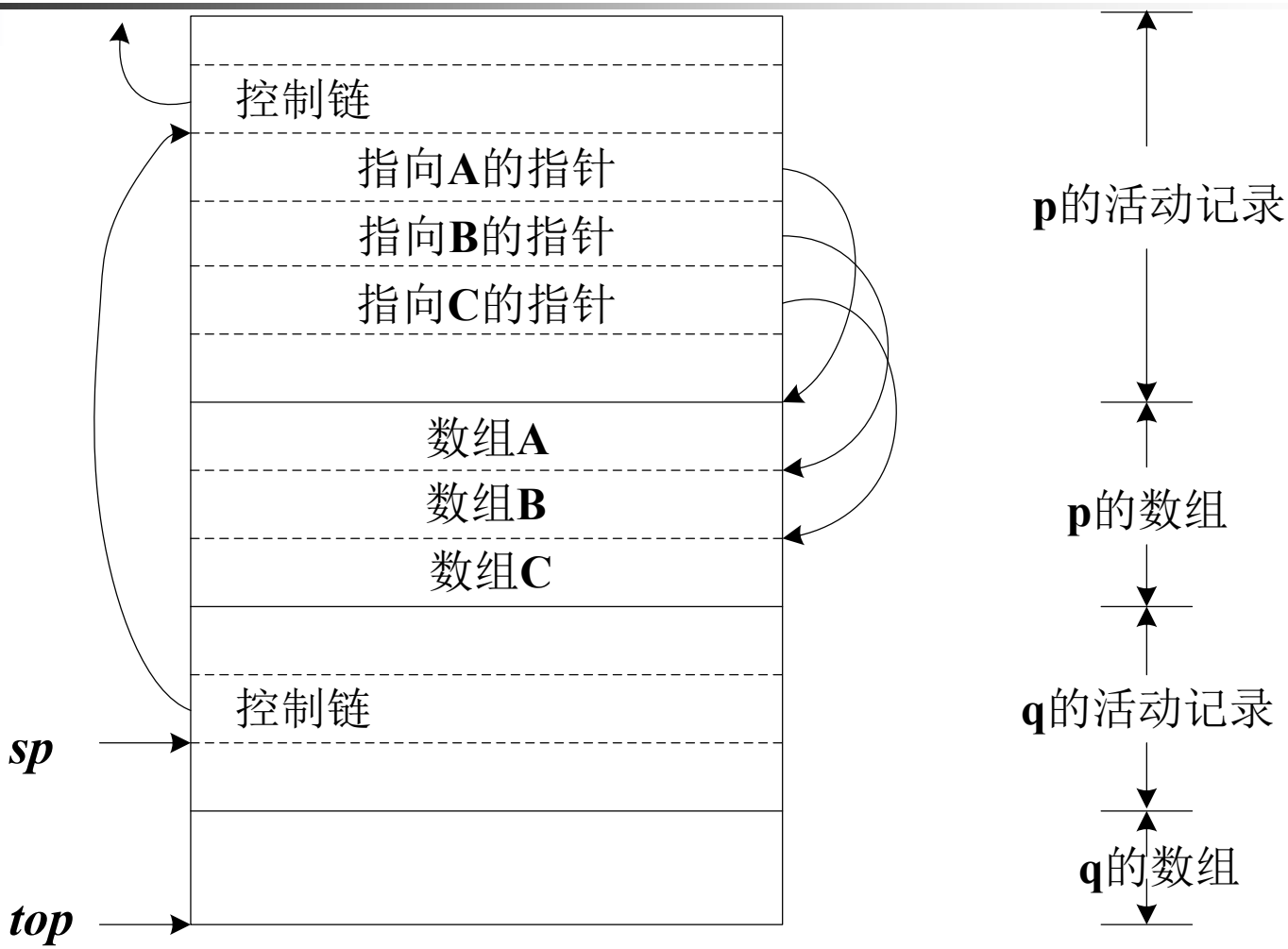


图9.13 访问动态分配的数组

栈式存储分配策略的局限

栈式分配策略在下列情况下行不通：

- 过程活动停止后，局部名字的值还必须维持
- 被调用者的活动比调用者的活动的生存期更长，此时活动树不能正确描绘程序的控制流

9.6 堆管理

- 对于允许程序为变量在运行时**动态申请和释放存储空间**的语言，采用堆式分配是最有效的解决方案.
  - 活动结束时必须保持局部名字的值。
  - 被调用者的活动比调用者的活动的生存期长。

9.6 堆管理

- 堆式分配的基本思想是, 为运行的程序划出适当大的空间(称为堆Heap), 每当程序申请空间时, 就从堆的空闲区找出一块空间分配给程序, 每当释放时则回收之.
- 内存管理器
  - 分配和回收堆区空间的子系统
  - 是应用程序和操作系统之间的一个接口

9.6.1 内存管理器

- 内存管理器的基本任务
  - 空间分配。每当程序为某个变量或对象申请一块内存空间时，内存管理器就产生一块连续的具有被请求大小的堆空间。
  - 空间回收。内存管理器把回收的空间返还到空闲空间的缓冲池中，用于满足其它的分配请求。

9.6.1 内存管理器

- 如果下面两个条件成立的话，内存管理将会变得相对简单一些
  - 所有的分配请求都要求相同大小的块；
  - 存储空间按照某种可以预见的方式来释放，如先分配者先释放。

9.6.1 内存管理器

- 内存管理器的设计目标
  - **空间效率**。使程序所需堆区空间的总量达到最小，以便在某个固定大小的虚拟地址空间中运行更大的程序。方法：**减少内存碎片的数量**。
  - **程序效率**。充分利用内存子系统，以便使程序运行得更快。方法：**利用程序的“局部性”**，即程序在访问内存时具有非随机性聚集的特性。
  - **低开销**。尽可能地提高这些操作的执行效率，也就是说要尽可能地降低它们的开销。

9.6.2 内存体系

- 要想做好内存管理和编译器优化，首先要充分了解内存的工作机理。
- 内存访问时间的巨大差异来源于硬件技术的局限。

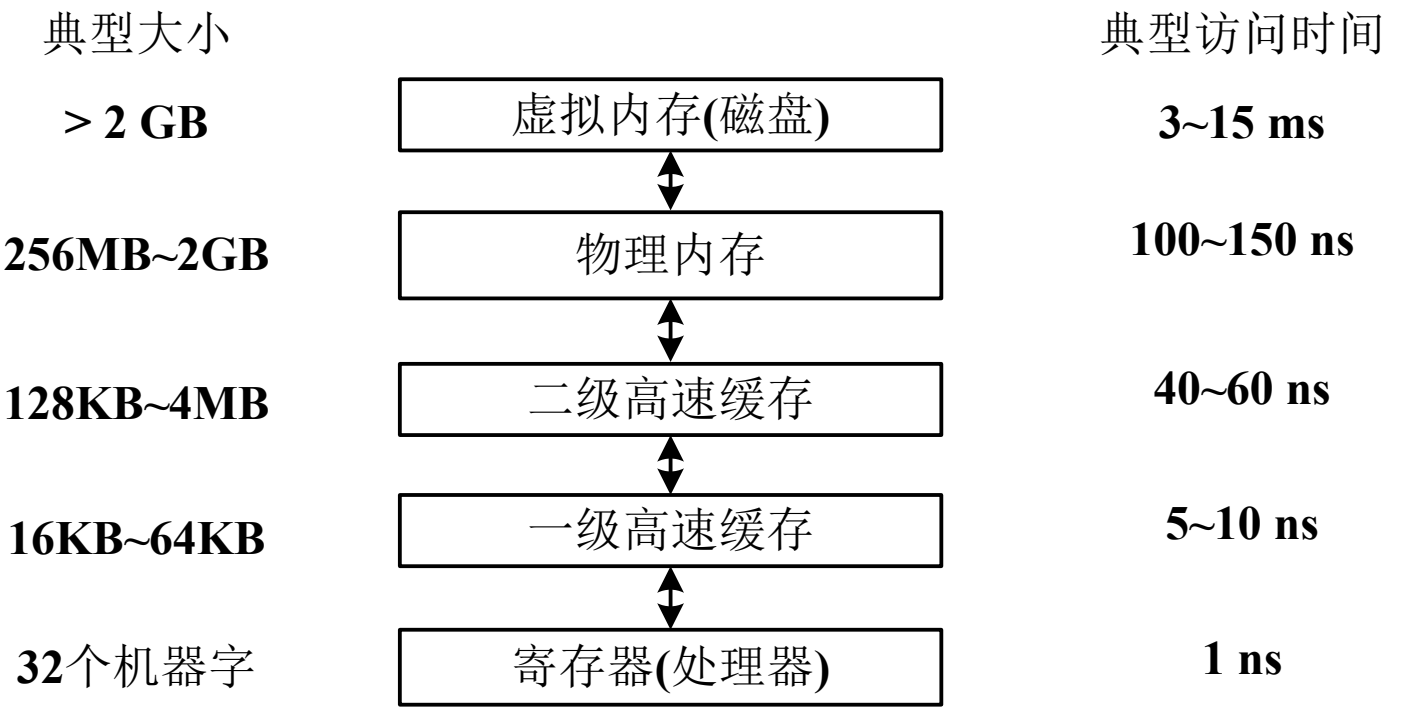


图9.21 典型的内存体系

9.6.3 程序中的局部性

- 程序的局部性
  - 程序中的大部分运行时间都花费在较少的一部分代码中，而且只是涉及到一小部分数据。
  - 时间局部性：如果某个程序访问的内存位置有可能在很短的时间内被再次访问
  - 空间局部性：如果被访问过的内存位置的邻近位置有可能在很短的时间内被再次访问

9.6.3 程序中的局部性

- 程序的局部性使得我们可以充分利用内存层次结构，即将最常用的指令和数据放在快而小的内存中，而将其余部分放在慢而大的内存中，这将显著降低程序的平均内存访问时间
- 很多程序在对指令和数据的访问方式上既表现出时间局部性，又表现出空间局部性

9.6.3 程序中的局部性

- 虽然将最近使用过的数据放在最快的内存层中在普通程序中可以发挥很好的作用，但是在某些数据密集型的程序中的作用却并不明显，如**循环遍历大数组**的程序。
- 将最近使用过的指令放在高速缓存中的策略一般都很有效。

9.6.3 程序中的局部性

- 空间局部性：让编译器将可能会连续执行的指令连续存放
  - 执行一条新指令时，其下一条指令也很有可能被执行
  - 属于同一个循环或同一个函数的指令也极可能被一起执行

9.6.3 程序中的局部性

- 通过改变数据布局或计算顺序也可以改进程序中数据访问的时间局部性和空间局部性
  - 例当某些程序反复访问**大量数据**而每次访问只完成**少量计算**时，它们的性能就不会很好。
  - 我们可以每次将一部分数据从内存层次中的较慢层加载到较快层，并趁它们处于较快层时执行**所有针对这些数据的运算**，这必将大大提高程序的性能。

9.6.4 降低碎片量的堆区空间管理策略

- 空闲块又被称为孔洞
- 如果对孔洞的使用不加管理的话，空闲的内存空间最终就会变成若干碎片，即大量不连续且很小的孔洞。
- 此时就会出现这样的情况：尽管总的空闲空间足够大，却找不到一个足够大的孔洞来满足某个即将到来的分配请求。

9.6.4 降低碎片量的堆区空间管理策略

- 1. 堆区空间分配策略
- 2. 空闲空间管理策略

堆区空间分配策略

- 最佳适应策略
  - 总是将请求的内存分配在满足要求的**最小可用孔洞**中
  - 倾向于将大孔洞预留起来以便用来满足后续的更大请求，这是令程序产生**最少碎片**的一种很好的堆分配策略

堆区空间分配策略

- 首次适应策略
  - 将请求的内存分配在第一个满足要求的孔洞中
  - 这种策略在分配空间时所花费的时间较少，但在总体性能上要低于最佳适应策略

堆区空间分配策略

- 如果将空闲块按大小不同放入不同的桶中，则可以更有效地实现最佳适应策略
- 桶机制更容易按最佳适应策略找到所需的空闲块：
  - 如果被请求的空闲块尺寸对应有一个专用桶，则从该桶中任意取出一个空闲块即可
  - 如果被请求的空闲块尺寸没有对应的专用桶，则找一个允许存放所需尺寸空闲块的桶。在桶中使用首次适应策略或者最佳适应策略寻找满足要求的空闲块
  - 如果目标桶是空的，或者桶中没有满足要求的空闲块，则需要在稍大的桶中进行搜索

空闲空间管理策略

- 如果通过手工方式释放某个对象所占用的内存块，则内存管理器必须将其设置为空闲的，以便它可以被再次分配。
- 为了减少碎片的产生，如果回收的内存块在堆中的相邻块也是空闲的，则需要将它们合并成更大的空闲块。

空闲空间管理策略

- 可以使用下面的两种数据结构来接合相邻的空闲块
  - 边界标签
  - 双向链接的空闲块列表

空闲空间管理策略

- 边界标签
  - 在每个内存块(已用/空闲)的高低两端均设置一个free/used标签位，用来标识该块是已用(used)还是空闲(free)，在与free/used位相邻的位置上则存放该块的字节总数。

空闲空间管理策略

- 双向链接的空闲块列表
  - 各个空闲块还由一个双向链表链接起来。链表的指针就保存在这些空闲块中，如紧挨某一端边界标签的位置上。
  - 于是我们不需要额外的空间来存放该空闲块链表，当然这会给空闲块的大小设置一个下界，即空闲块必须保存两个边界标签和两个指针，即使要保存的对象只有一个字节也得这样。

本章小结

- 存储组织与管理是编译系统的重要组成部分，用来实现目标程序的存储组织与分配。根据程序设计语言的要求，有静态管理策略和动态管理策略
- 名字的绑定要体现名字的声明和作用域约定
- 过程调用中参数的传递方式分为传值、传地址、传值结果和传名4种
- 典型的运行时内存空间包括目标代码、静态数据区和动态数据区

本章小结

- 对编译时就能确定其运行时所需要的存储空间的对象实行静态存储分配，对编译时无法确定过程何时运行、运行时所需要存储空间的对象实行动态存储分配
- 静态存储管理方式支持较高的运行效率，利用适当的策略可以提高内存的利用率，但是无法支持动态数据和“过程”的递归调用

本章小结

- 栈式动态存储管理策略针对过程的每一次调用在栈顶创建一个栈单元用来存放这次过程调用产生的活动的数据区，访问链和display表可以用来实现相关数据的访问。这种方式支持过程的递归调用
- 堆管理解决活动结束后数据仍需有效的问题，但需要使用内存管理器实现对孔洞的管理
- 对内存的层次体系的有效利用有助于提高目标代码的运行效率