广东工业大学计算机学院

第十章

目标程序运行时的存储组织

引言

- “运行时的存储区组织”，是指目标程序运行时的数据空间的管理和组织。
- 为了使目标程序能够运行，编译程序要从操作系统中得到一块存储区，以使目标程序能够在其上运行。该存储区需容纳：

(1) 生成的目标代码

(2) 目标代码运行时的数据空间

存储区的内容

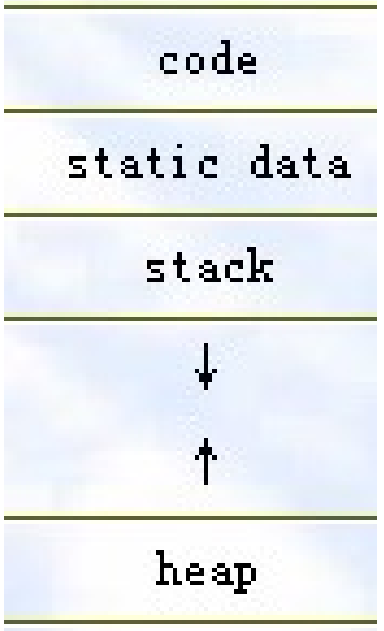
- **数据空间**应包括：
- (1) 用户定义的**各种类型的数据对象**（变量和常数）所需的存储空间；
- (2) 作为保留中间结果和传递参数的**临时工作单元**；
- (3) 调用过程时所需的**连接单元**；
- (4) 组织输入/输出所需的**缓冲区**.

- 目标代码所占用空间的大小在编译时能确定。

- 有些数据对象所占用的空间也能**在编译时确定**，其地址可以编译进目标代码中。
- 而有些数据对象**具有可变体积和待分配性质**，**无法**在编译时确定存储空间的位置。(如书P229)

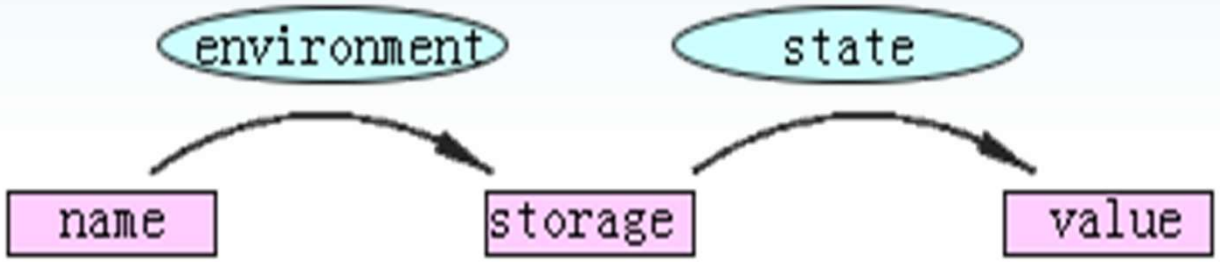
存储区的划分

- 因此运行时的存储区常常划分成：目标代码区、静态数据区、栈区和堆区等，如下图所示就是一种典型划分：
- 代码(code)区：存放目标代码，在编译时能确定其大小；
- 静态数据区(static data)：存放编译时能确定所占用空间的数据；
- 堆栈区(stack and heap)（或叫动态数据区）用于可变数据以及管理过程活动的控制信息。
- 保留地址区、共享库和分别编译模块区（第3版的，第2版没有提到）



分配数据空间的本质

- **数据空间分配的本质：将程序中的每个名字与一个存储位置关联起来，该存储位置用以容纳名字的值。**关联（Binding）



- **源文件中的名字N ↔ 运行时的存储位置S**
- **运行时的存储位置S ↔ 运行时的值V**

存储管理复杂程度的相关因素

决定存储管理复杂程度的因素—— 源语言本身

1.允许的数据类型的多少

2.语言中允许的数据项是：{静态确定 / 动态确定}

3.程序结构： 决定名字的作用域的规则和结构

(1)段结构(如FORTRAN语言)

(2)过程定义不嵌套,只允许过程递归调用(如C语言)

(3)分程序结构(如ALGOL语言)

分程序嵌套

过程定义嵌套

存储分配方案策略：

- 静态存储分配 (不介绍)
- 动态存储分配——栈式

堆式(不介绍)

本章的学习目标

- 1. 全面*了解*目标程序运行时存储区的整体布局；
- 2. *了解*每种存储区的组织方式和管理方法；
- 3. 重点掌握栈式动态存储分配的组织方式，以及运行时进栈退栈的活动实现方法。

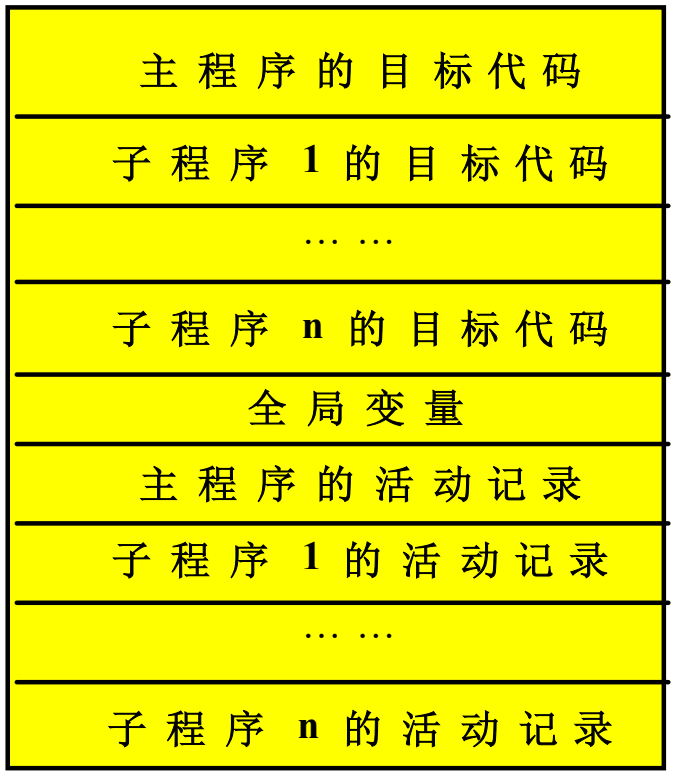
本章内容

- 10.1 数据空间的三种不同使用方法和管理方法
- 10.2 栈式存储分配的实现
- 10.3 参数传递
- 10.4 过程调用、过程进入和过程返回

数据空间的使用方法和管理方法

- 数据空间的使用和管理方法分成三种：静态存储分配、栈式动态存储分配和堆式动态存储分配。
- 1 静态存储分配
- 在编译时能确定目标程序运行中所需的全部数据空间的大小。包括：
- 安排好目标程序运行时的全部数据空间；
- 确定每个数据对象的存储位置。

- 例如：像FORTRAN这样的语言，其程序是段结构的，整个程序所需数据空间的总量通常在编译时完全确定，从而每个数据名的地址就可静态进行分配。

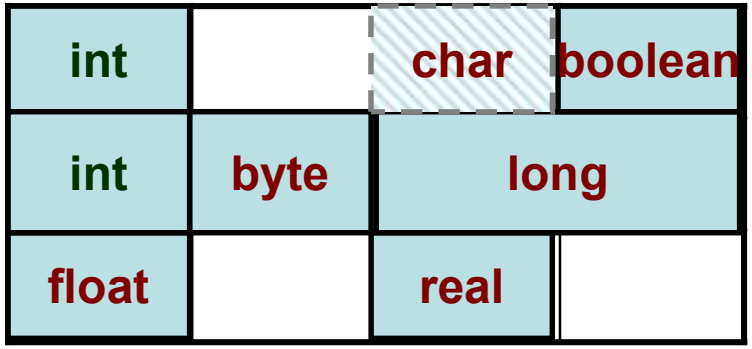


数据空间——动态存储分配

- 动态存储管理技术有两种方式：栈式(stack)和堆式(heap)。
- (1) 栈式动态存储分配
- 此分配策略式将整个程序的数据空间设计为一个栈。它适用于Pascal、C、ALGOL之类的语言实现。
- 每当调用一个过程(函数)时，就在栈顶分配该过程(函数)所需的空间，调用完毕后释放此空间。
- 过程(函数)所需的空间分为两部分：
- ① 生存期为本过程(函数)调用活动中数据对象。如：局部变量、参数单元、临时变量等。
- ② 用于管理过程活动的记录信息所占的空间。如：程序计数器(返回地址)、寄存器的值等。

数据空间——堆式动态存储分配

- **堆式动态存储分配：用户程序可以自由的申请数据空间和退还数据空间（书P233）。**
- **经过一段时间的运行后，程序运行空间被划分成许多块，甚至会比较“凌乱”。**
- **例如右图所示一个堆空间：有些空间被占用，有些空闲。**
- **问题：现在需要为double变量分配空间，它需要两个空间，该如何办?**
- **Answer：使用“垃圾回收技术”(GC)，回收无用单元，或者适当移动已被占用的空间。**



本章内容

- 10.1 数据空间的三种不同使用方法和管理方法
- 10.2 栈式存储分配的实现
- 10.3 参数传递
- 10.4 过程调用、过程进入和过程返回

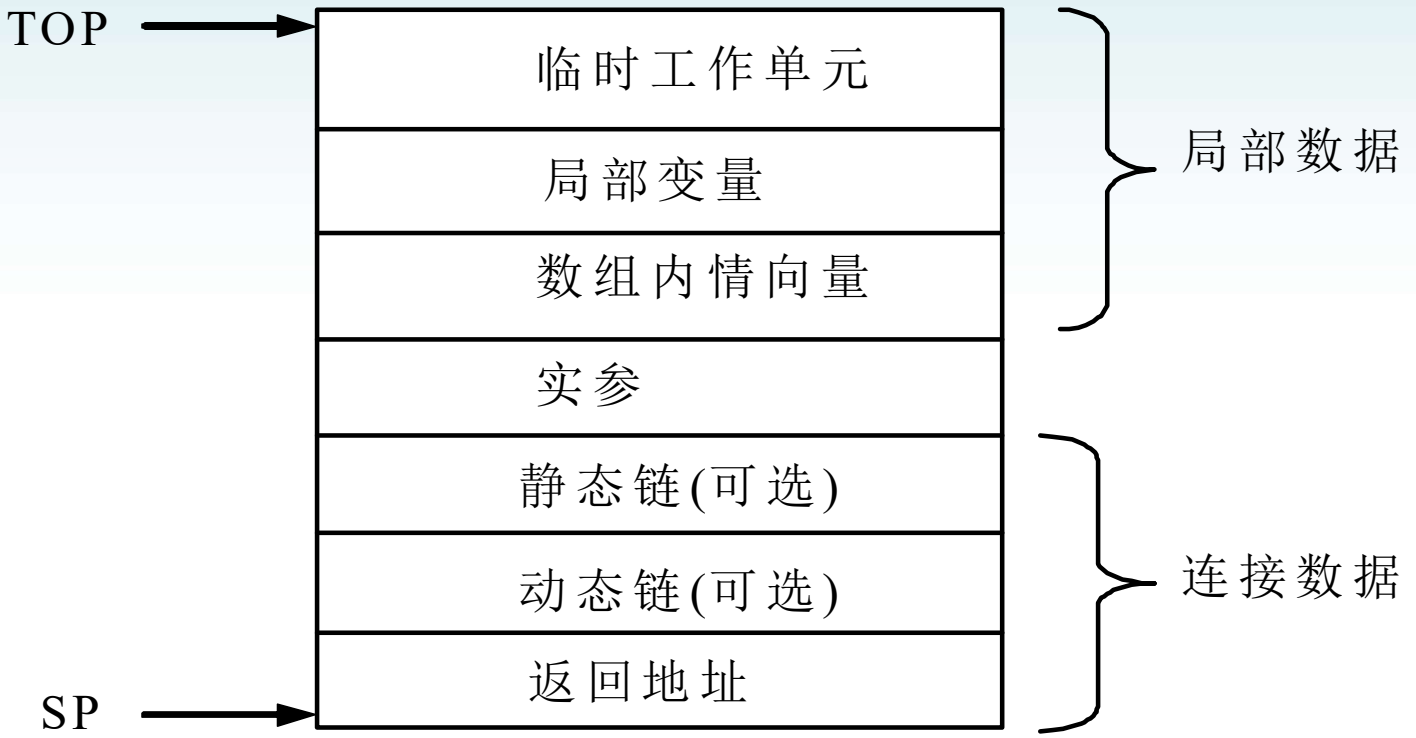
栈式存储分配的实现

- **栈式存储分配策略**：目标程序运行时每当进入一个过程(函数)，就在栈顶为该过程**分配**所需的**数据空间**；
- 执行完毕返回时，此数据空间也即**释放**。

- 过程的**活动记录AR(Activation Record)**：AR是一段**连续的存储区（新书：在运行栈上的栈帧）**，用以存放过程(函数)的一次执行所需要的**动态信息**。

- 每当进入一个过程(函数)，编译程序就在栈顶压入一个**AR**。

栈式存储分配的实现



活动记录的内容

一般这个段要记录：

- ① 临时工作单元：比如计算表达式过程中需存放中间结果用的临时值单元。

② 局部变量：一个过程的局部变量。

④ 存取链（或称静态链）：用以存取非局部变量，这些变量存放于其它过程的活动记录中。并不是所有语言需要该信息。

⑤ 控制链(或称动态链)：指向调用该过程的那个过程的活动记录，这也不是所有语言都需要的。

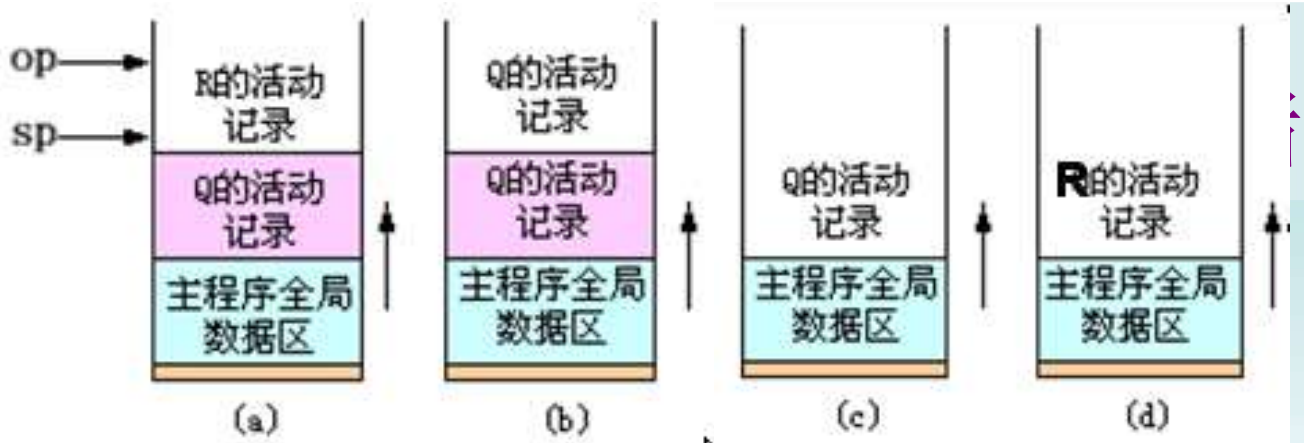
⑥ 实参：也称形式单元，由调用过程向该被调过程提供实参的值（或地址）。当然在实际编译程序中，也常常使用机器寄存器传递实参。

⑦ 返回地址：保存该被调过程返回后的地址。

活动记录的三个联系单元

- 每个过程的AR有（过程与过程之间的联系）
- 3个联系单元：
  - SL： 存储链，指向定义该过程的直接外过程 （或主程序）运行时最新数据段的基地址。（9.2.2.2节）
  - DL： 动态链，指向调用该过程前正在运行过 程的数据段基地址。即指向调用本过程的那个过程的活动记录的地址（老SP）（9.2.2.2节）
  - RA： 返回地址，记录调用该过程时目标程序的断点，保存该过程返回后的地址。

分配的实现



- **一种最简单的程序设计语言结构：没有分程序结构，过程定义不嵌套，但允许过程递归调用。例如：**

```
program main;      //主程序头
    全局变量或数组的说明;
    proc R;                  //过程R的头
    …                        //过程R的体
    end (R);                 //过程R的尾
    proc Q;                  //过程Q的头
      R( );                  //过程Q的体
      Q( );
    end (Q);                 //过程Q的尾
    主程序执行语句           //主程序体
    end.(main)               //主程序尾
```

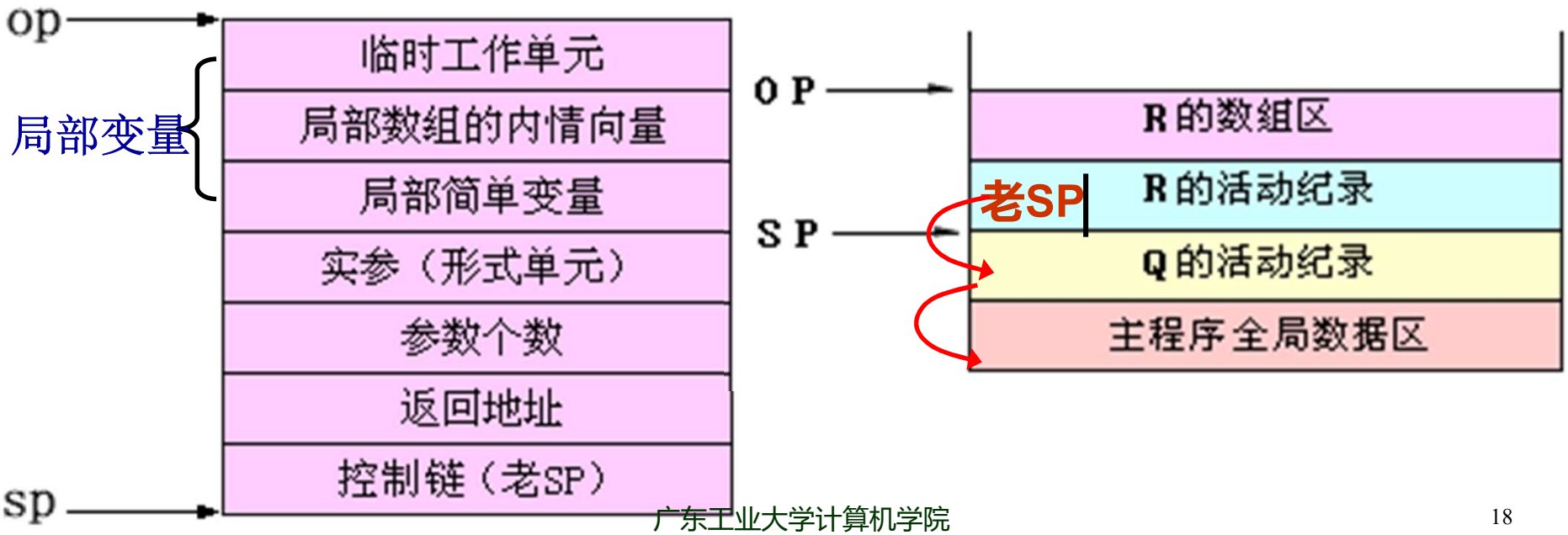
SP：指向现行过程活动记录的起点

TOP：始终指向已占用的栈顶单元。

```
//主程序语句
begin
……
Q( );
R(.);
```

含有可变数组的情况

- 若含有可变数组，则其过程活动记录AR的内容如图所示。
- 控制链(老SP)：调用新过程的那个旧过程的最新活动记录的起点。
- 在右图中，R的AR中的老SP即是Q的AR的起点(SP)。



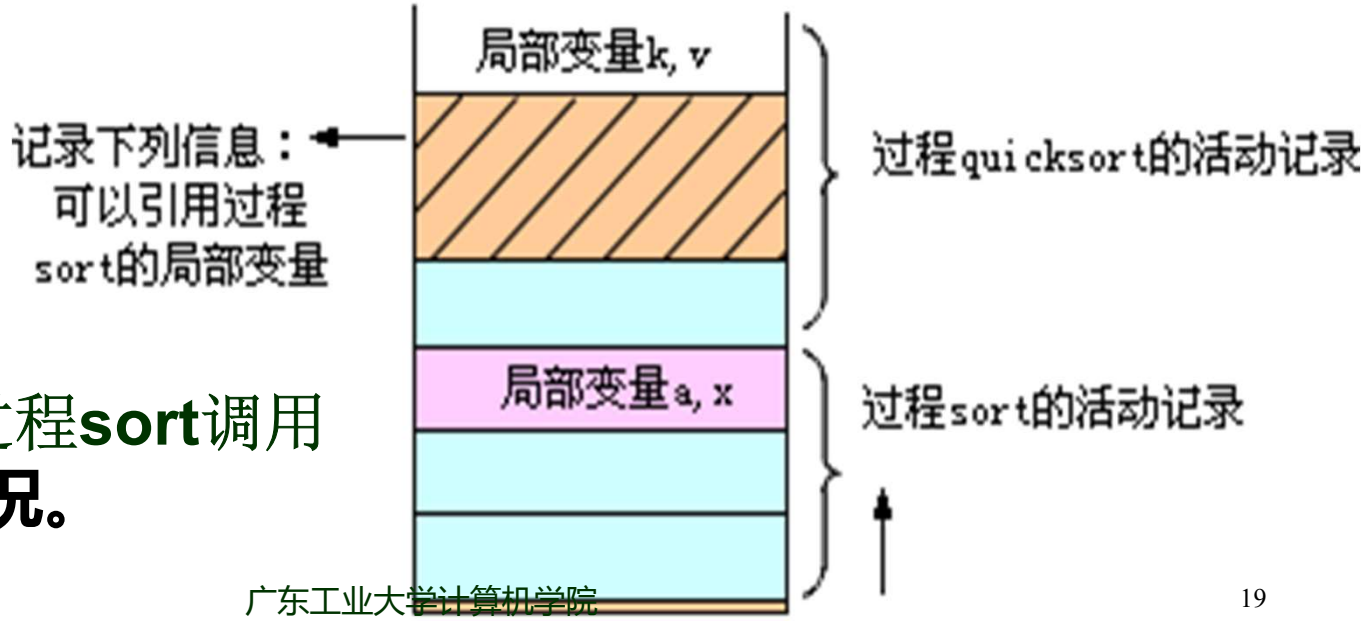
2 嵌套过程语言的栈式实现

- 允许过程嵌套定义，一个过程可以引用包围在它的外层过程所定义的标识（如变量，数组或过程等）。(PASCAL语言)

- 所以需要在它的AR中应增设一些内容，以解决对非局部变量的引用问题。

- 例如：存在PASCAL程序中过程定义的嵌套情况如下(书P236)：

sort
  readarray
  exchange
  quicksort
    partition



在某一时刻，过程sort调用quicksort的情况。

嵌套过程语言的栈式实现方式

解决对非局部变量的引用问题的方法

1. 用存储链（如PL/0的SL）。

   引入一个称为存储链的指针，该指针为活动记录的一个域，指向直接外层的最新活动记录的地址。

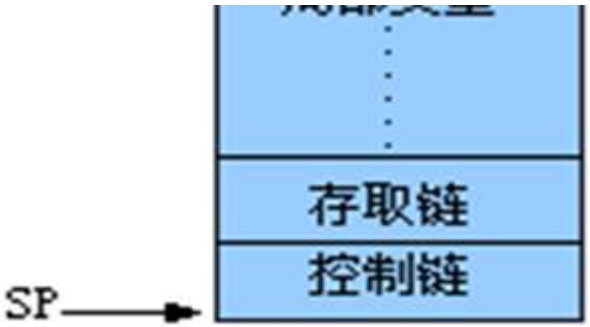
2. 用DISPLAY表。

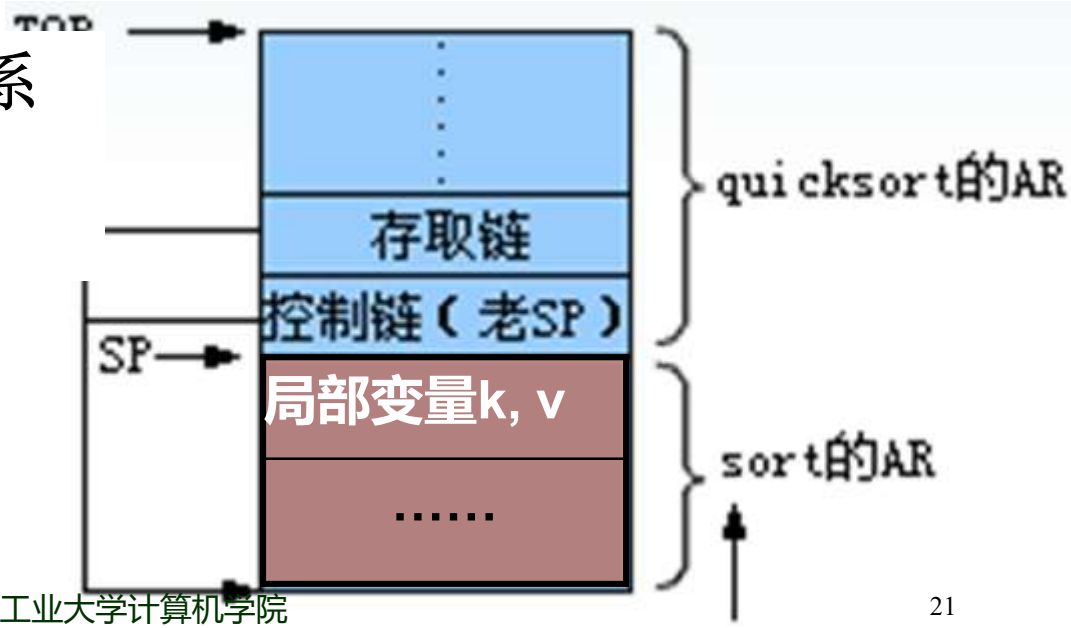
实现对非局部量的存取的方法

- **如何在内层过程中对非局部变量(即外层过程的变量)作访问?**
- **基本思想：跟踪每个外层过程的最新活动记录AR的位置。**
- **跟踪办法之一：在过程活动记录中增设存取链，指向包含该过程的直接外层过程的最新活动记录的起始位置。**

存取链表示静态的过程嵌套关系

控制链表示动态的调用关系

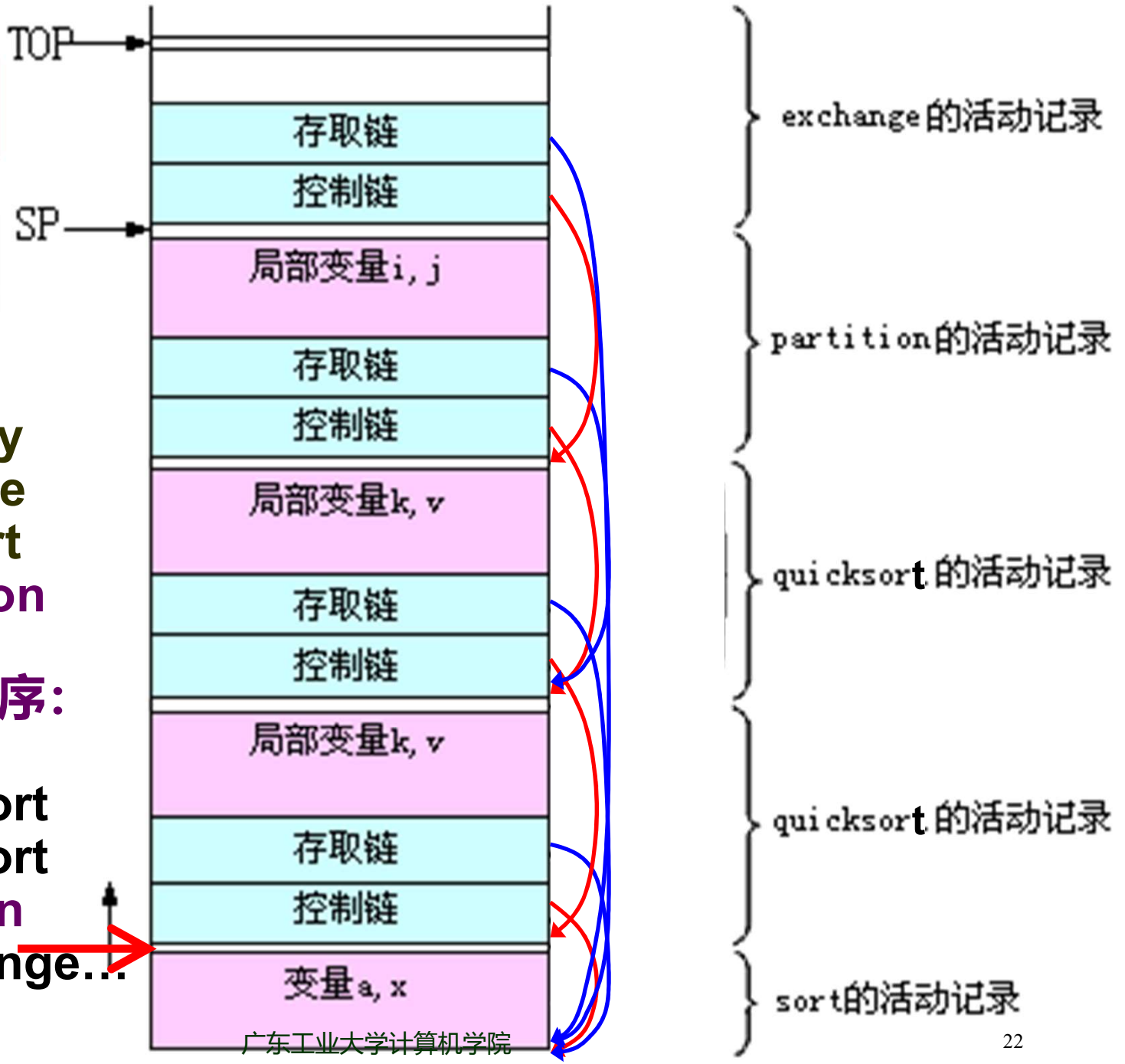




举例

- **嵌套层次：**
  sort
    readarray
    exchange
    quicksort
      partition

- **某次执行顺序：**
  sort
    →quicksort
    →quicksort
    →partition
      →exchange…

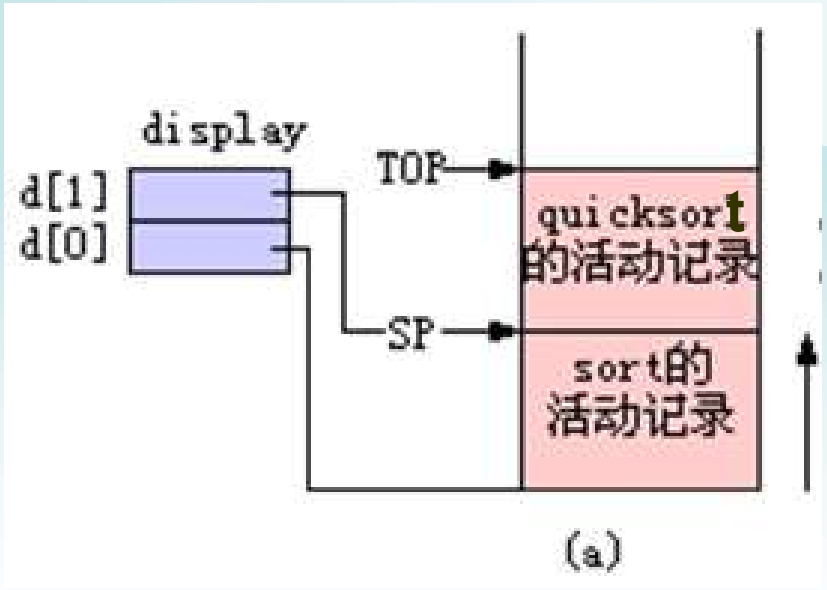


跟踪办法之二：嵌套层次显示表

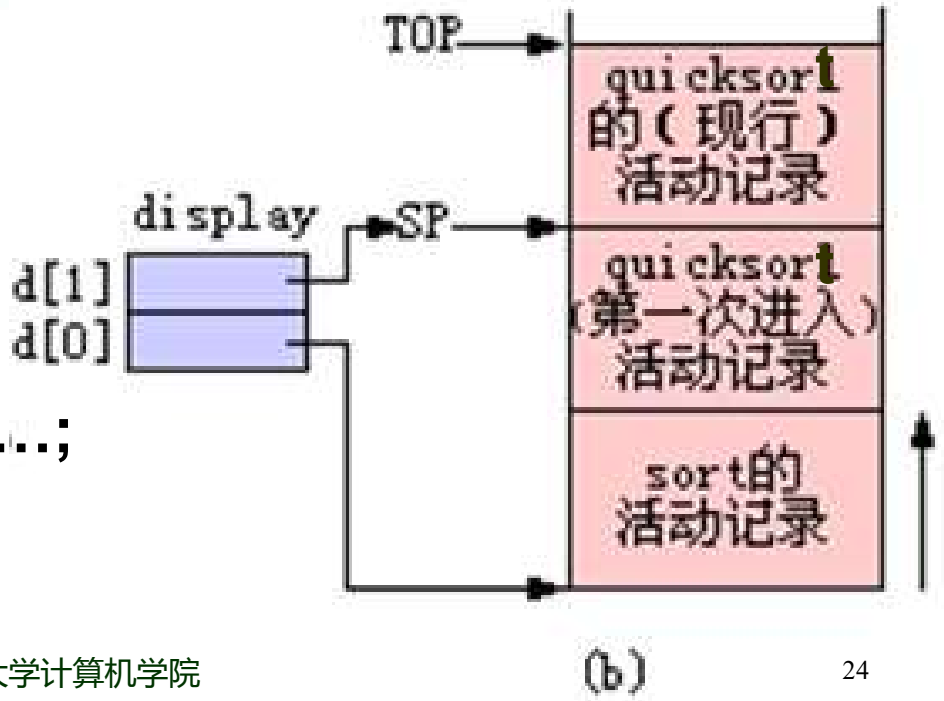
- **跟踪办法之二：每进入一个过程后，在建立活动记录的同时，建立一张嵌套层次显示表(display)。**
- **display表是一个指针数组d，自顶向下每个单元依次存放现行层，直接外层，……直至最外层(0层，主程序层)等每一层过程的最新AR的地址。**
- **d[i]指向嵌套层次i的AR，如果要访问第i层的局部变量a，则可以由d[i]找到相应的AR地址，再在该AR中寻找到$x[SP^i]$。**

嵌套层次显示表举例

- 假设过程定义的嵌套情况如下：
- sort
  - readarray
  - exchange
  - quicksort
    - partition

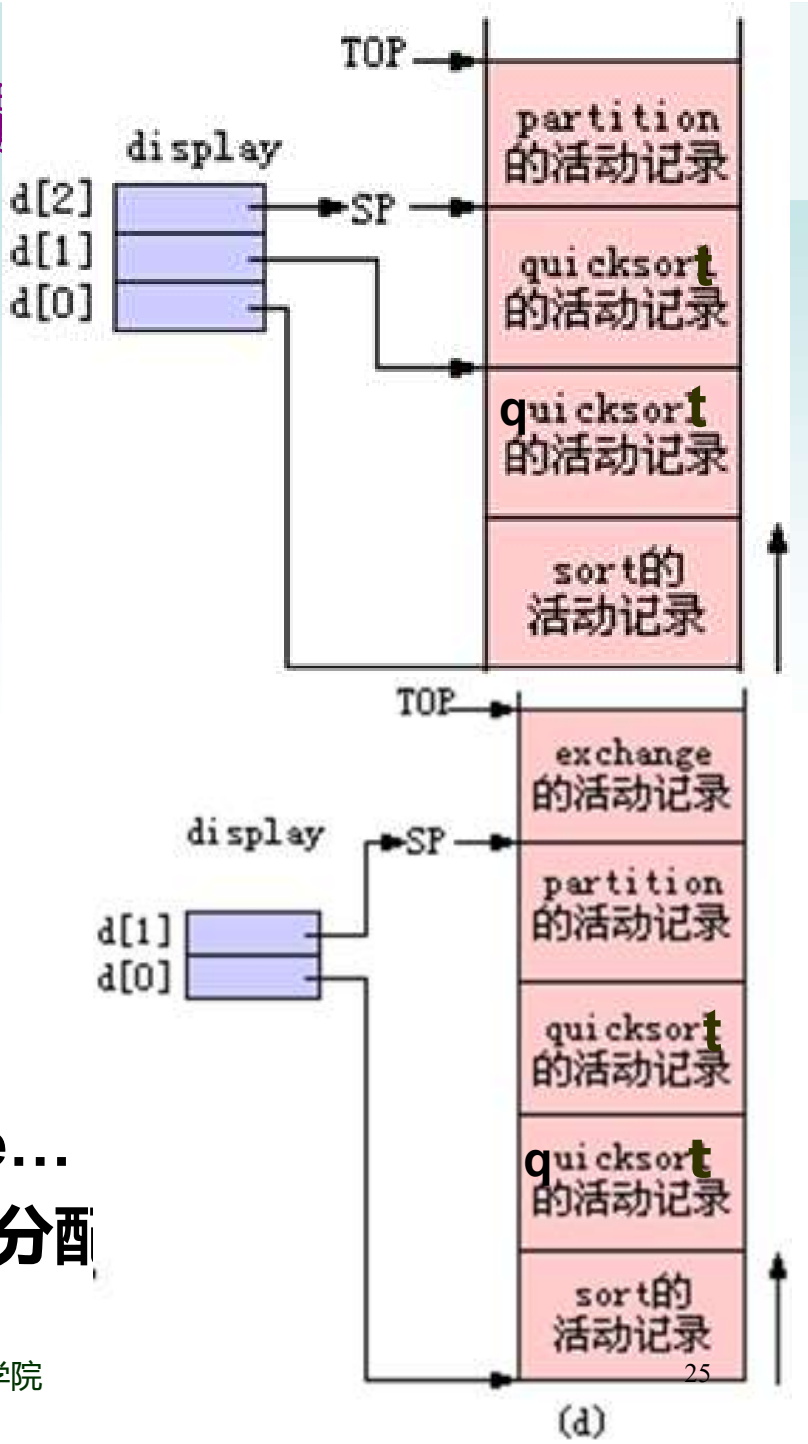


- 假定有以下调用情况：
- (a) sort→quicksort…;
- (b) sort→quicksort→ quicksort…;



嵌套层次显示表举例(续

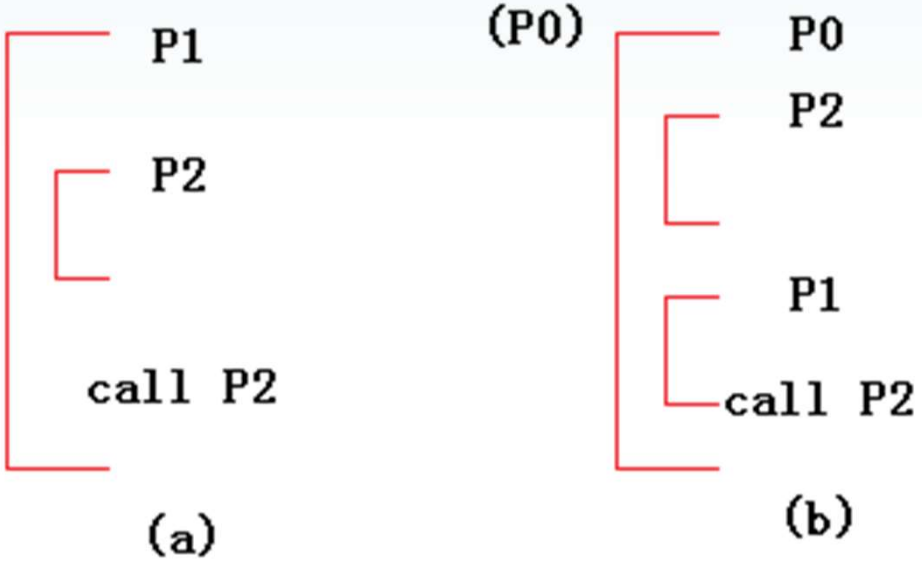
- 假设有过程定义的嵌套情况如下：
- sort
  - readarray
  - exchange
  - quicksort
    - partition
- 假定有以下调用情况：
- (c) sort→quicksort→
  quicksort→partition…;
- (d) sort→quicksort→
  quicksort→partition
  →exchange…
- 另外，display可以作为单独的表分配存储，也可以作为AR的一部分。



如何建立display

- 当过程P1调用过程P2时，为建立P2的display，必须知道P2的直接外层过程(记为P0)的display。

- P1调用P2有以下两种情况：
- (a) P2定义在P1内部，此时 P0 = P1
- (b) P1与与P2并列，定义在另一个过程P0的内部。

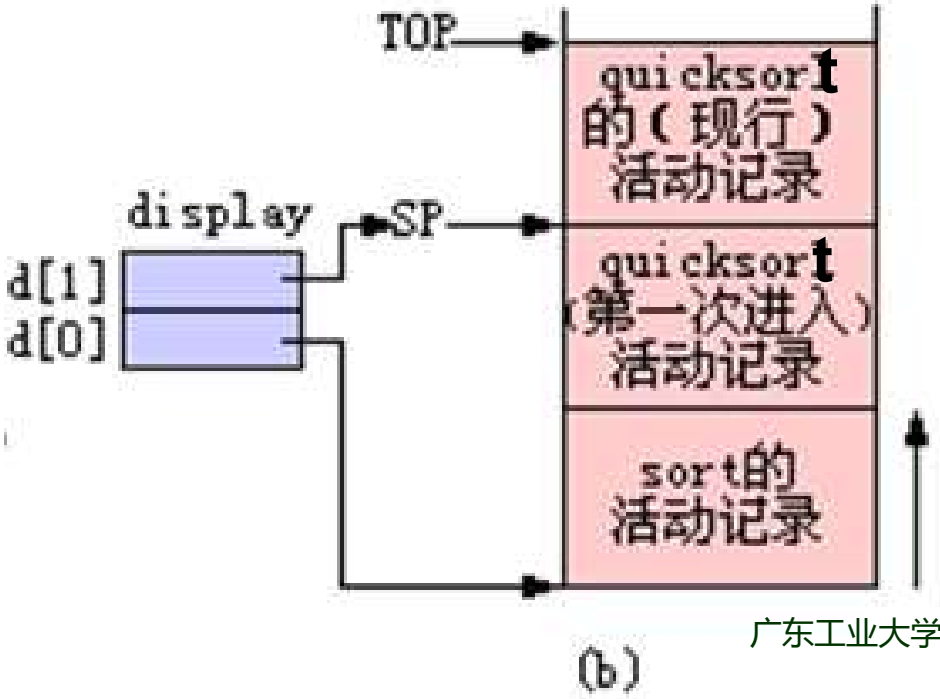


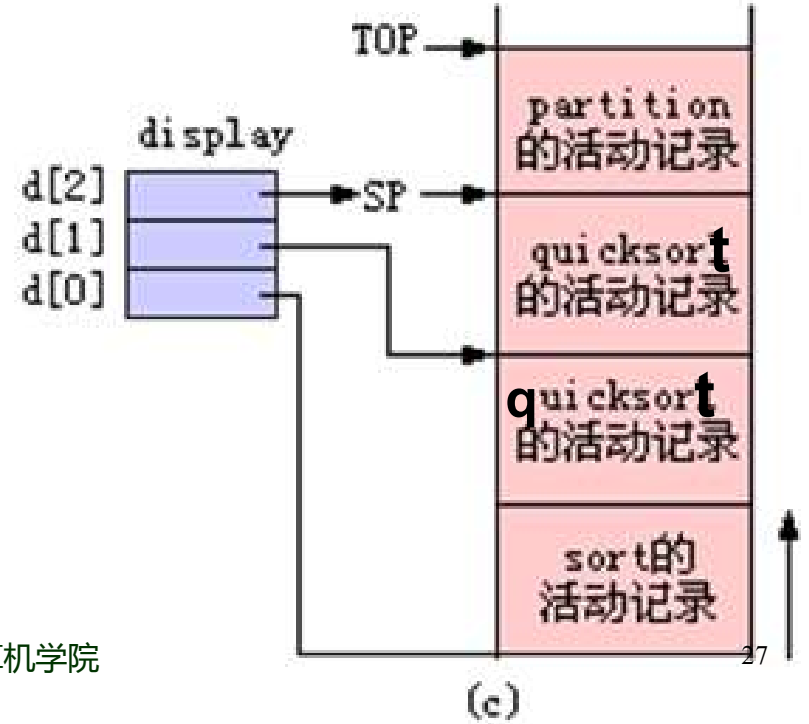
- 要建立P2的display，则必须把P0的display地址作为连接数据之一传给P2。

如何建立di

- sort
  - readarray
  - exchange
  - quicksort
    - partition

- 设P1调用P2，则P2的display 表包括
- (1) 从P1的display取*l*个元素(*l*为P2的嵌套层数)
- (2) 加上P2的SP作为最后一个元素。

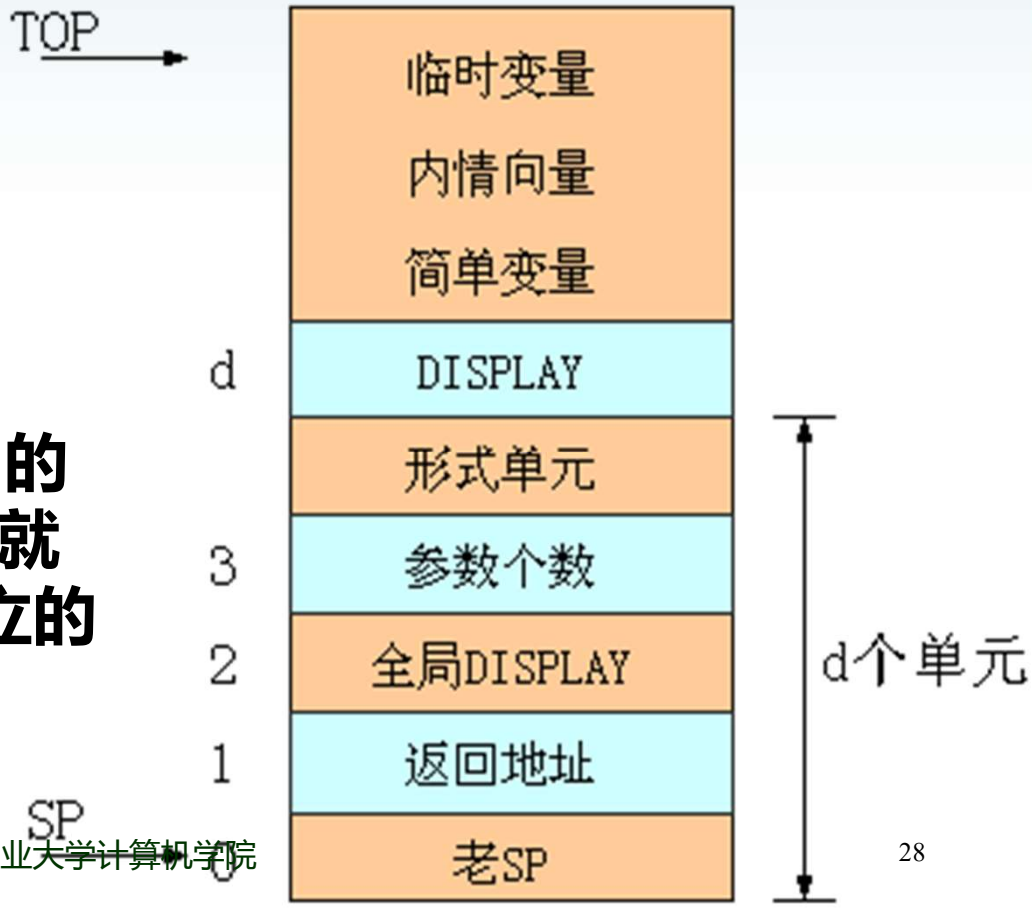




把display作为活动记录的一部分

- 为了能在P2中记录P0的display地址，必须在P1调用P2时把P1的display地址作为连接数据之一(称为“全局display地址”)传送给P2。
- 于是连接数据变为三项:
- (1) 老SP值;
- (2) 返回地址;
- (3) 全局display地址。

- 注意：0层过程（主程序）的display只含一项，这一项就是主程序开始工作时所建立的第一个SP值。



3. 分程序结构的存储管理

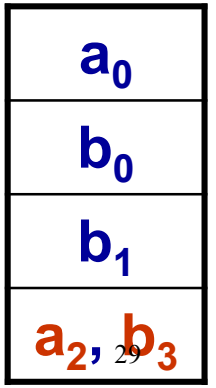
```
main( )
{
    int a = 0;
    int b = 0;
    {
        int b = 1;
        {
            int a = 2;
            printf("%d %d\n", a, b);
        }
        {
            int b = 3;
            printf("%d %d\n", a, b);
        }
        printf("%d %d\n", a, b);
    }
    printf("%d %d\n", a, b);
}
```

B0 B1 B2 B3

- 首先以C语言为例，看看什么是分程序：
- 为所有声明的变量分配存储如图所示。

声明	作用域
int a = 0;	B0 – B2
int b = 0;	B0 – B1
int b = 1;	B1 – B3
int a = 2;	B2
int b = 3;	B3

各变量名字的存储分配：



a2和b3处于同一层，
可共享同一处存储空间

分程序结构的栈式存储分配实现

- 分程序结构可以用**栈式**存储分配实现。

- **办法之一：**把分程序看成一种“**无参过程**”，看成是在该分程序入口处调用，分程序出口处返回。分程序在哪里定义就在哪里被调用。

- 在该方法中，每个**过程**被当作是**0**层分程序。而**过程体**分程序(假定是一个分程序)当作是它所管辖的第**1**层分程序。

- **办法之二：**一次性地为一个完整的过程全分配存储，即把一个过程体中的所有分程序所需的存储一次分配好。

分程序存储管理的改进办法

- 把分程序看成无参过程的办法效率很低。原因如下：
- (1) 每当进入一个分程序时，都建立相应的连接数据和display表,这是不必要的。
- 改进办法：不把分程序看作“无参过程”，每个分程序共享包围它的那个最近过程的display。
- 即在分程序B1中，可以直接使用partition的display表：
- 查到变量i所在的位置。

```
• procedure quicksort(…)
  var i: integer;
  begin
     function partition(…): …
     begin
        ……
        begin
B1      …i…
        call exchange
        end
     end
end
```

分程序存储管理的改进办法

- 把分程序看成**无参过程**的办法效率很低。原因如下：
- (2) 当从内层分程序向外层转移时，可能同时要结束若干个分程序。

- **例如**从第5层跳到第1层，按照**过程处理**办法，意味着必须一层一层地通过“返回” 来恢复所要到达的那个分程序的数据区，但不能直接到达。

- **改进办法**：对每个过程和分程序都建立有自己的栈顶指示器TOP，每个TOP的值保存在各自的活动记录中。

分程序的存储管理举例

- 假设存在一个过程A，它是一个ALGOL程序
- procedure A(m, n);
- integer m, n;
- B1: begin
- real z; array B[m: n];
- B2: begin
- real d, e;
- L3;
- end;
- B4: begin
- array C[1: m];
- B5: begin
- real e;
- L6;
- end;
- end;
- L8;
- end

各个TOP值如何变化？

分程序的存储管理举例(a)

- 假设存在一个过程A，它是一个ALGOL程序：

```
procedure A(m, n);
    integer m, n;
B1: begin
        real z;   array B[m: n];
    B2: begin
            real d, e;
            L3;
        end;

    B4: begin
            array C[1: m];
        B5: begin
                real e;
                L6;
            end;
        end;
        L8;
    end
```

已调用过程A，但尚未开始执行过程体(即仅仅到达B1处)时：

A的TOP值确定，因为在编译时已经能确定整个AR的大小。

分程序的存储管理举例(b)

- 假设存在一个过程A，它是一个ALGOL程序：

```
procedure A(m, n);
        integer m, n;
B1: begin
        real z;   array B[m: n];
    B2: begin
            real d, e;
            L3;
        end;

    B4: begin
            array C[1: m];
        B5: begin
                real e;
                L6;
            end;
        end;
        L8;
    end
```

已进入B1，但尚未分配数组空间时栈的情况。

这时有

B1的TOP = A的TOP

分程序的存储管理举例(c)

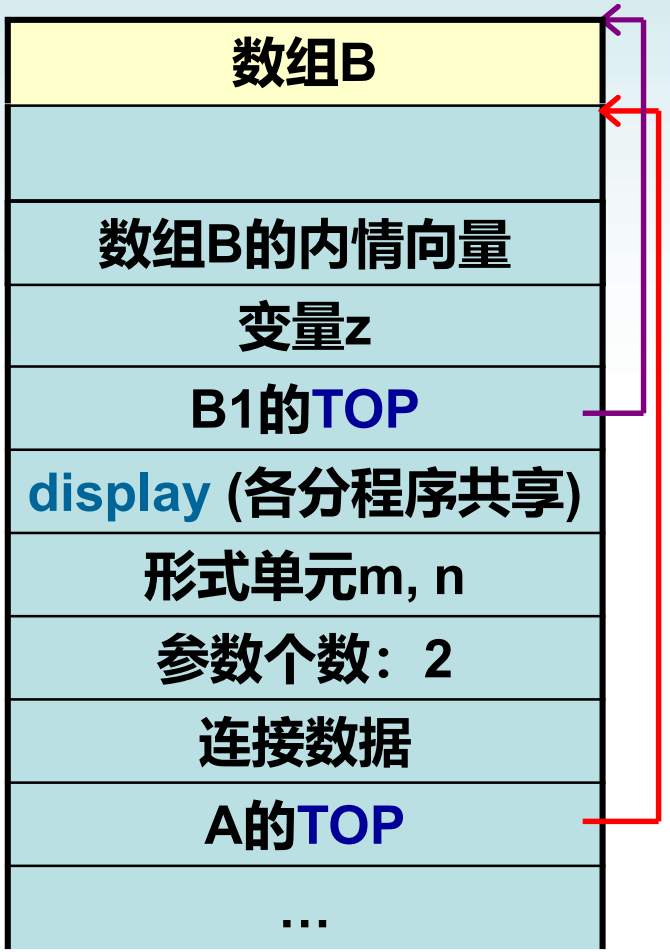
- 假设存在一个过程A，它是一个ALGOL程序：

```
procedure A(m, n);
      integer m, n;
B1: begin
        real z;   array B[m: n];
      B2: begin
            real d, e;
            L3;
          end;

      B4: begin
            array C[1: m];
          B5: begin
                real e;
                L6;
              end;
          end;
          L8;
      end
```

总结：分程序的的TOP永远指向栈顶，方便分程序退出时的跨层跳跃。

已进入B1，并分配数组空间时栈的情况。这时B1的TOP值升至新栈顶

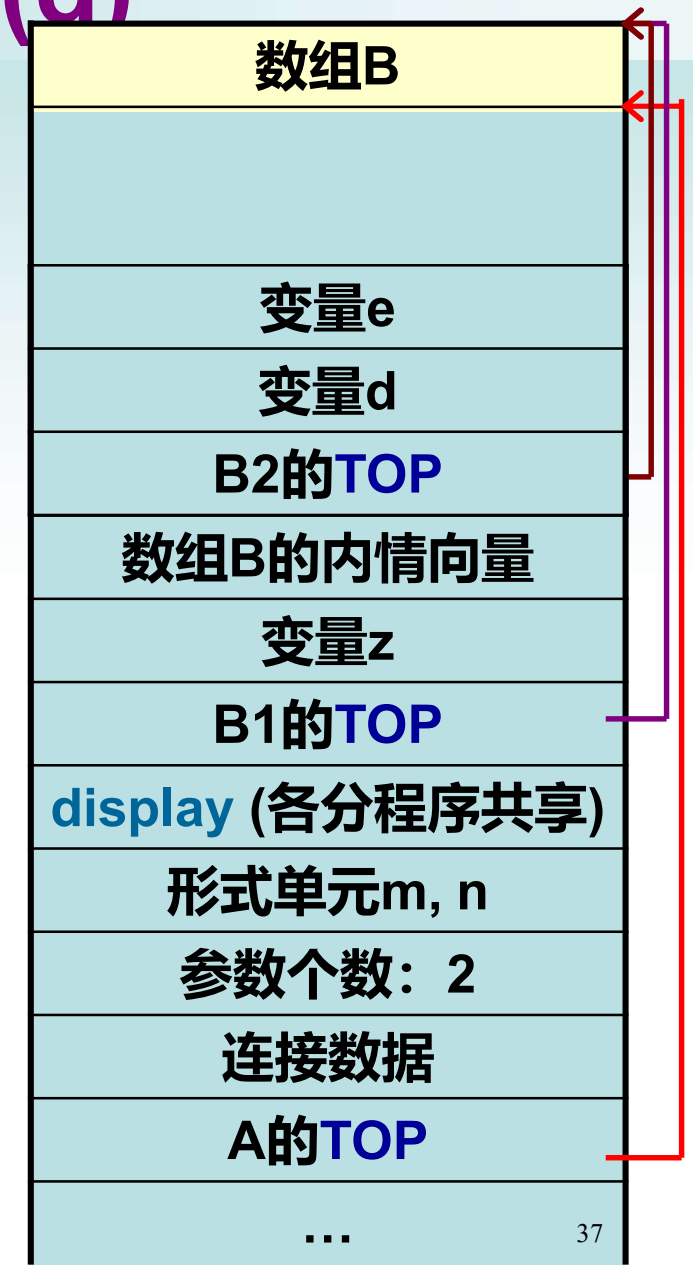


分程序的存储管理举例(d)

- 假设存在一个过程A，它是一个ALGOL程序：
- procedure A(m, n);
- integer m, n;
- B1: begin
- real z; array B[m: n];
- B2: begin
- real d, e;
- L3;
- end;
- B4: begin
- array C[1: m];
- B5: begin
- real e;
- L6;
- end;
- end;
- L8;
- end

已进入B2时栈的情况

这时B2的TOP = B1的TOP



分程序的存储管理举例(e)

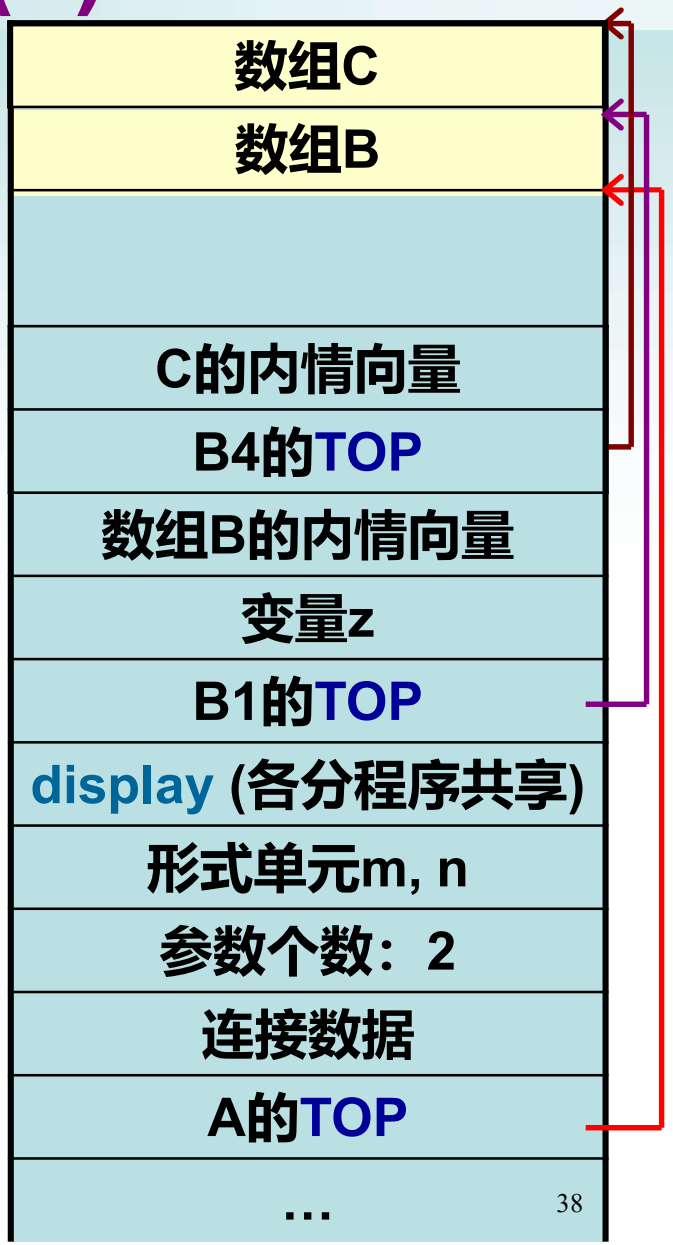
- 假设存在一个过程A，它是一个ALGOL程序：

```
procedure A(m, n);
        integer m, n;
B1: begin
        real z;   array B[m: n];
    B2: begin
            real d, e;
            L3;
        end;

    B4: begin
            array C[1: m];
        B5: begin
                real e;
                L6;
            end;
        end;
        L8;
    end
```

这是已退出B2，并且进入B4分配数组空间时，栈的情况。

这时B4的TOP指向新栈顶



分程序的存储管理举例(f)

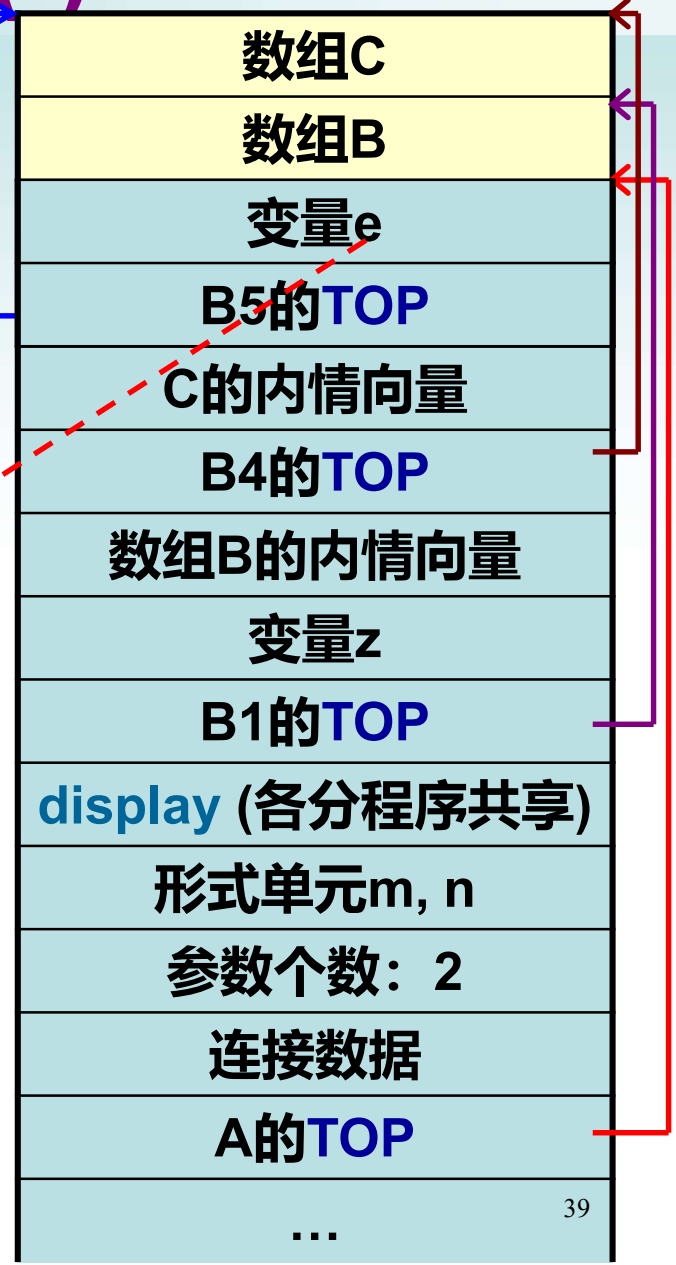
- 假设存在一个过程A，它是一个ALGOL程序：

```
procedure A(m, n);
    integer m, n;
B1: begin
        real z;   array B[m: n];
    B2: begin
            real d, e;
            L3;
        end;

    B4: begin
            array C[1: m];
        B5: begin
                real e;
                L6;
            end;
        end;
        L8;
    end
```

这是进入B5时，栈的情况。

这时B5的TOP指向新栈顶



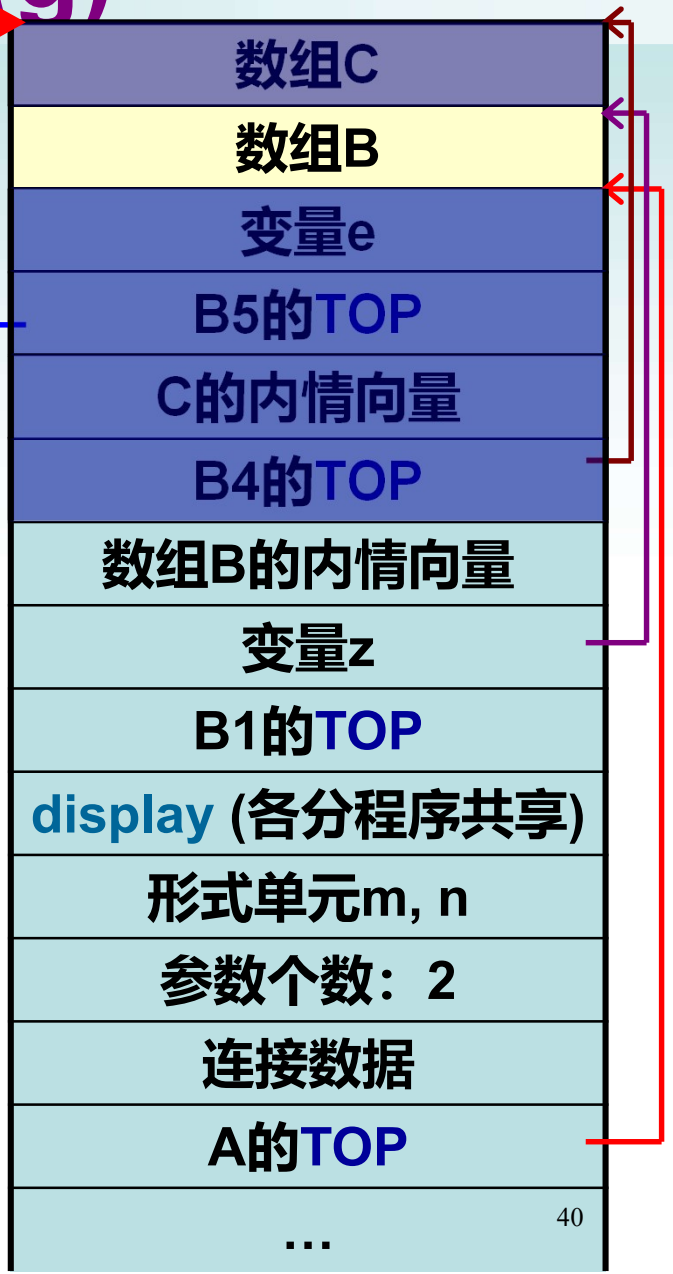
分程序的存储管理举例(g)

- 假设存在一个过程A，它是一个ALGOL程序：

```
procedure A(m, n);
        integer m, n;
B1: begin
        real z;   array B[m: n];
    B2: begin
            real d, e;
            L3;
        end;

    B4: begin
            array C[1: m];
        B5: begin
                real e;
                L6;
            end;
        end;
        L8;
    end
```

这是从B5直接跳转到B1时栈的情况。



本章内容

- 10.1 数据空间的三种不同使用方法和管理方法
- 10.2 栈式存储分配的实现
- 10.3 参数传递
- 10.4 过程调用、过程进入和过程返回

1. 传值

- **传值，即call by value，也称值调用。这是最简单的参数传递方法，其实现步骤如下：**
- **① 在被调过程的活动记录中开辟形参的存储空间，即形参(形式单元)。**
- **② 调用过程计算实参的值，并将它们的右值（r value）放在形式单元的空间中。**
- **③ 被调用过程执行时，就像使用局部变量一样使用这些形式单元。**
- **值调用的重要特点：对形参的任何运行不影响调用过程的活动记录AR中实参的值。**

传值举例

```
(1)program  reference(input,output);
(2)var  a,b:integer;
(3)procedure  swap( {var} x,y:integer);
(4)         var  temp:integer;
(5)         begin
(6)               temp:=x;
(7)                    x:=y;
(8)                   y:=temp
(9)         end;
(10)begin
(11)       a:=1;  b:=2;
(12)       swap(a,b);
(13)       writeln('a=',a);writeln('b=',b)
(14)end.
```

2. 传地址

- 当参数通过引用传递时，也称作传地址，或引用调用。调用过程传给被调过程的是**指针**——指向实参存储位置。分以下情况：
- 1. 如实参是一个名字或是具有**左值**的表达式，则**左值**本身传递过去。
- 2. 如实参是一表达式，比如是a + b或2，而没有**左值**，则表达式先求值，并存入某一位置，然后该位置的地址传递过去。
- 在被调用过程中，对形式参数的任何引用和赋值都通过**指针**处理成**间接访问**。

传地址举例

- **在一个值调用过程中使用指针的C程序：**

```
(1)  swap(x, y)                    /* 定义swap过程 */
(2)  int *x, *y;
(3)  {   int temp;                 //swap的过程体
(4)      temp=   x;    x=   y;    y=temp;
(5)  }
```

```
(6)  main()                        //主程序
(7)  {   int a = 1, b = 2;
(8)      swap(&a, &b);    //调用swap,其实在参数是a和b的地址
(9)      printf("a is now%d,b is now%d\n", a, b);
(10) }
```

传名

- **传名：将被调用的过程体复制到调用处，并将每个形参“文字地”替换成实参。**

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;
 }
```

传名时**Print(A)**的结果是多少？

传名

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： 经过P(A+B, A, A)函数调用，将实参的名传过去，X就是A + B，Y就是A，Z也是A
- Y:=Y+1；变为： A :=A+1 = 3
- Z:=Z+X；变为： A :=A+A + B = 3 + 3 + 3 = 9
- 所以传名的结果：9

作业

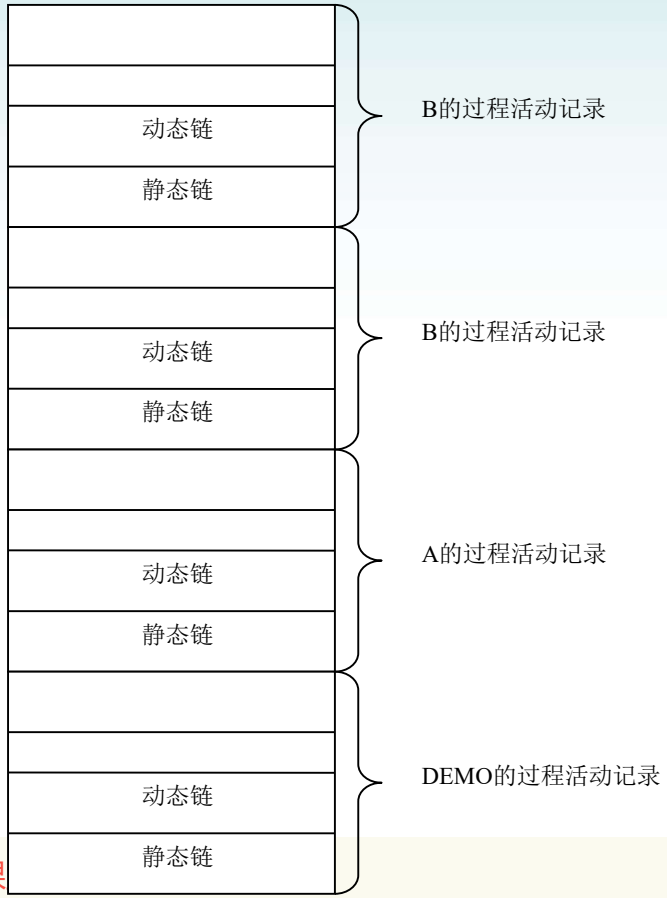
- **写上日期**
- **及时交作业**

主观题 10分

设置

（6分）类PASCAL程序结构（嵌套过程）如下，该语言的编译程序采用栈式动态分配策略管理目标程序存储空间。若过程调用情况为Demo->A->B->B,画出程序运行到第二个B过程的时刻，栈内静态链、动态链的指示情况。

```
Program Demo;
      Procedure A
        Procedure B
          Begin(*B*)
            … …
            If d then B else A;
          End(*B*)
        Begin(*A*)
          B;
        End(*A*)
      Begin(*Demo*)
        A
      End(*Demo *)
```



正常使用主观题需2.0以上版本雨课

作答

主观题 10分

设置

（10分） 下面是一个PASCAL语言源程序，其中函数gcd(m,n)的功能是返回m和n的最大公约数，当第二次递归调用进入gcd时，试将运行栈的内容补充完整。

```
PROGRAM gmn( );
 VAR m,n,g:integer;
 FUNCTION gcd(m,n:integer):integer;
  BEGIN
IF n=0 THEN
  g:=m
ELSE
  g:=gcd(n,m MOD n)
END;
BEGIN
  m:=24;
  n:=16;
  g:=gcd(m,n)
END
```

17		参数 n 值
16		参数 m 值
15		参数个数
14	RA	
13		DL
12		SL
11		参数 n 值
10		参数 m 值
9		参数个数
8	RA	
7		DL
6		SL
5	g	
4	n	
3	m	
2	RA	
1	0	DL

正常使用主观题需2.

作答

- **附录**

PL/0编译程序的目标代码解释执行时的存储分配

- PL/0编译程序调用解释程序时，即对存放在CODE中的目标代码CODE[0]开始进行解释执行，其过程中会使用到栈式数据区S(一维整型数组)。
- 解释程序还定义了4个寄存器：
- 1. 栈顶寄存器(指针)T：由于每个过程当它被运行时，给它分配的数据空间(下边称数据段)可分成两部分。
  - 静态部分：包括变量存放区和三个联系单元(SL、DL和RA)。
  - 动态部分：作为临时工作单元和累加器用。需要时随时分配，用完后立即释放。
- 2. 基址寄存器(指针)B：指向每个过程被调用时，在数据区S中给它分配的数据段起始地址，也称基地址。
- 3. 程序地址寄存器 P：指向下一条要执行的指令
- 4. 指令寄存器 I：存放正在执行的指令

三个联系单元SL、DL和RA

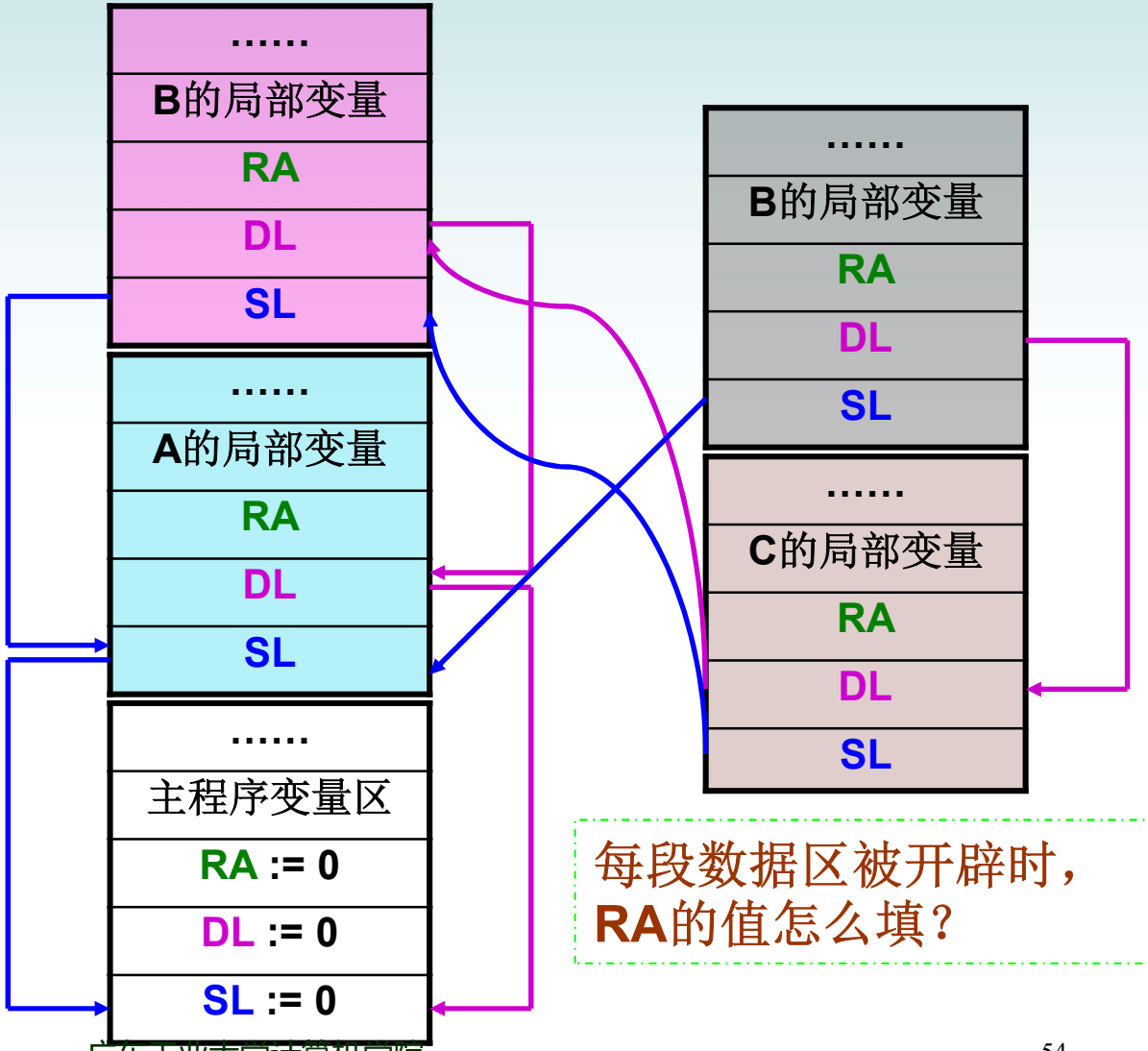
- 当程序进行某个过程P调用时，需要给它分配数据段，也就是对P所需的数据段进栈；过程运行结束后释放数据段，即P所需的数据段退栈。

- 每个过程被调用时，在栈顶分配三个联系单元，这三个单元存放的内容分别为：
- (1) SL(静态链)：它是指向定义过程P的直接外过程(或主程序)运行时最新数据段的基地址。
- (2) DL(动态链)：它是指向调用过程P前正在运行过程Q的数据段基地址。
- (3) RA(返回地址)：记录调用过程P时目标程序的断点，即当时的程序地址寄存器P的值。也就是调用过程指令的下一条指令的地址。

SL、DL和RA应用举例

- …
  procedure A:
  …
- **procedure B:**
- …
  procedure C:
  程序体C
  …
  call B:
  …
- 程序体B
  …
  call C:
  …
- 程序体A
  …
  call B:
  …
- 主程序
  …
  call A:
  …

过程B是由过程A定义的，它的名字在A层的名字表中。
当在C过程中调用B过程时，层次差为2。



每段数据区被开辟时，RA的值怎么填？