



中国科学技术大学
University of Science and Technology of China

运行时存储空间的组织与管理-III

《编译原理(H)》

张昱

0551-63603804, yuzhang@ustc.edu.cn

中国科学技术大学
计算机科学与技术学院



6.3 非局部名字访问

- 静态数据区、堆
- 静态作用域：无过程嵌套的（C）、有过程嵌套的（Pascal）
- 动态作用域（Lisp、JavaScript）



非局部数据的存储

□ 静态数据区

- 全局变量、静态局部变量

□ 堆

- C: malloc、free

glibc 的 [ptmalloc](#), [Doug Lea's dlmalloc](#)

高效的并发内存分配器 [jemalloc](#),

[TBBmalloc](#), [TCMalloc \(gperftools\)](#)

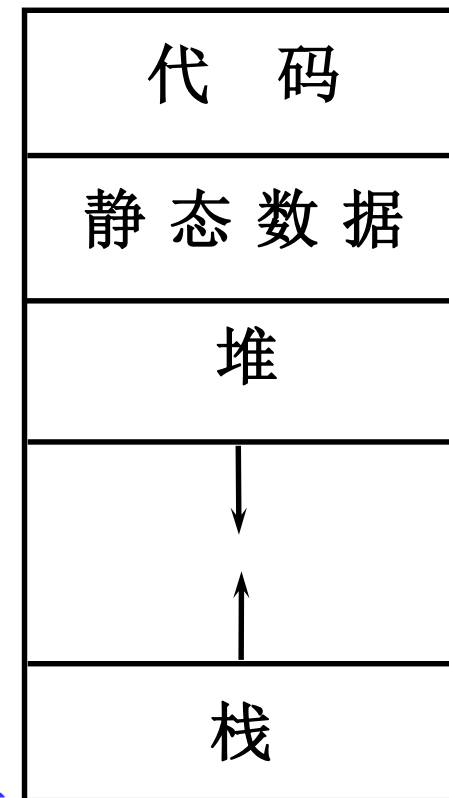
- Java: new、Garbage Collection

[Richard Jones's the Garbage Collection Page](#)

- JavaScript等动态类型绑定的语言

变量的空间采用**隐式**的堆分配

低地址



高地址



□ 无过程嵌套定义时，如C语言

- 非静态的局部变量的访问：位于栈顶的活动记录，通过基址指针 *base_sp* 来访问
- 过程体中的非局部引用、静态局部变量：直接用静态确定的地址（位于静态数据区中的数据）
- 过程可以作为参数来传递，也可以作为结果来返回
- 无须访问链

□ 有过程嵌套定义时

如Ada、JavaScript、Pascal、Python、Fortran 2003+

- 需要构建访问链，并通过访问链访问在外围过程中声明的非局部名字



有过程嵌套定义的静态作用域

- 非局部名字访问：访问链
- 过程作为参数产生的问题和解决
- 过程作为返回值产生的问题



过程嵌套定义的程序举例

图6.14

sort

readarray

exchange

quicksort

partition



图6.14

■ **过程(定义)的嵌套深度**

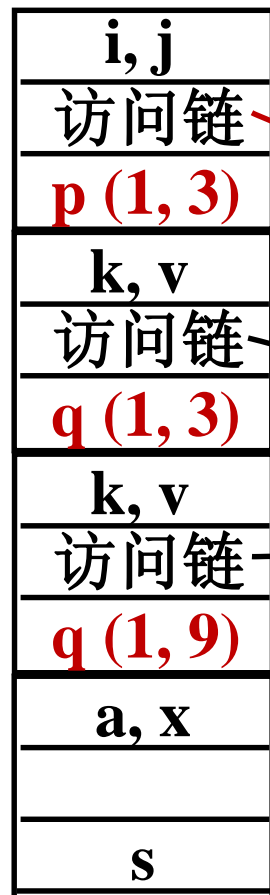
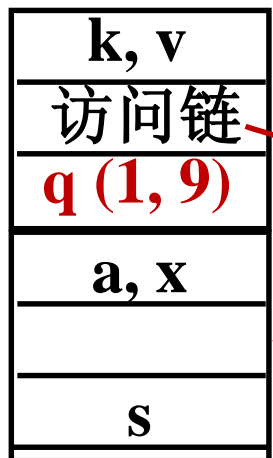
sort	1
readarray	2
exchange	2
quicksort	2
partition	3

■ **变量的嵌套深度**: 以它的声明所在的过程的嵌套深度作为该名字的嵌套深度

■ 用来寻找非局部名字的存储单元

sort	1
<u>readarray</u>	2
exchange	2
quicksort	2
partition	3

访问链反映过程之间的嵌套定义关系
控制链反映过程之间的调用关系





针对访问链的两个关键问题

□ 通过访问链访问非局部引用

假定过程 p 的嵌套深度为 n_p ，它引用嵌套深度为 n_a 的变量 a ， $n_a \leq n_p$ ，如何访问 a 的存储单元

□ 访问链的建立

假定嵌套深度为 n_p 的过程 p 调用嵌套深度为 n_x 的过程 x ，分别考虑 (1) $n_p < n_x$ ，(2) $n_p \geq n_x$ 的情况

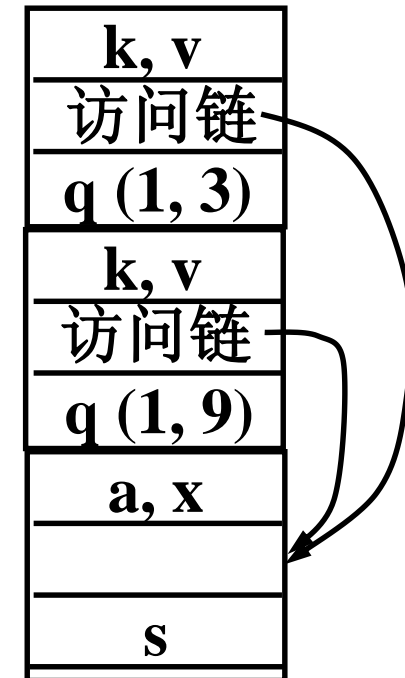


通过访问链访问非局部引用

假定过程 p 的嵌套深度为 n_p ，它引用嵌套深度为 n_a 的变量 a ， $n_a \leq n_p$ ，如何访问 a 的存储单元

- 从栈顶的活动记录开始，追踪访问链 $n_p - n_a$ 次
- 到达 a 的声明所在过程的活动记录
- 访问链的追踪用间接操作就可完成

sort	1
readarray	2
exchange	2
quicksort	2
partition	3





非局部名字引用的表示

过程 p 对变量 a 访问时， a 的地址由下面的二元组表示：

$(n_p - n_a, a$ 在活动记录中的偏移)



访问链的建立

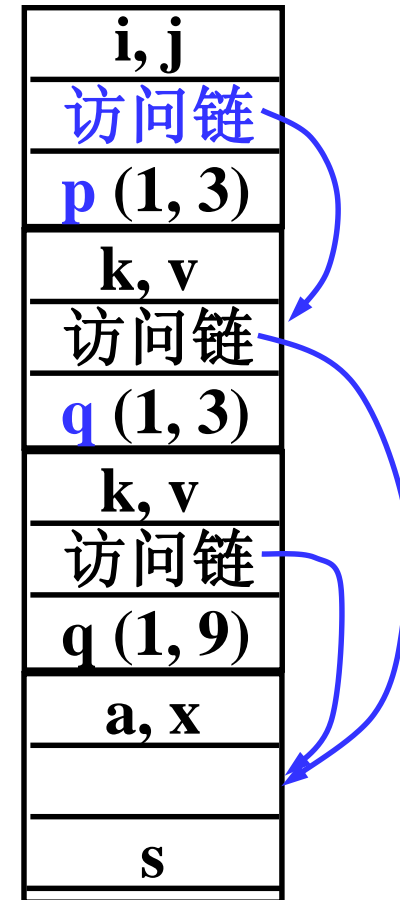
假定嵌套深度为 n_p 的过程 p 调用嵌套深度为 n_x 的过程 x

1. $n_p < n_x$ 的情况 这时 x 肯定就声明在 p 中

■ 被调用过程的访问链须指向调用过程的活
动记录的访问链

■ sort调用quicksort、quicksort调用partition

sort	1
readarray	2
exchange	2
quicksort	2
partition	3





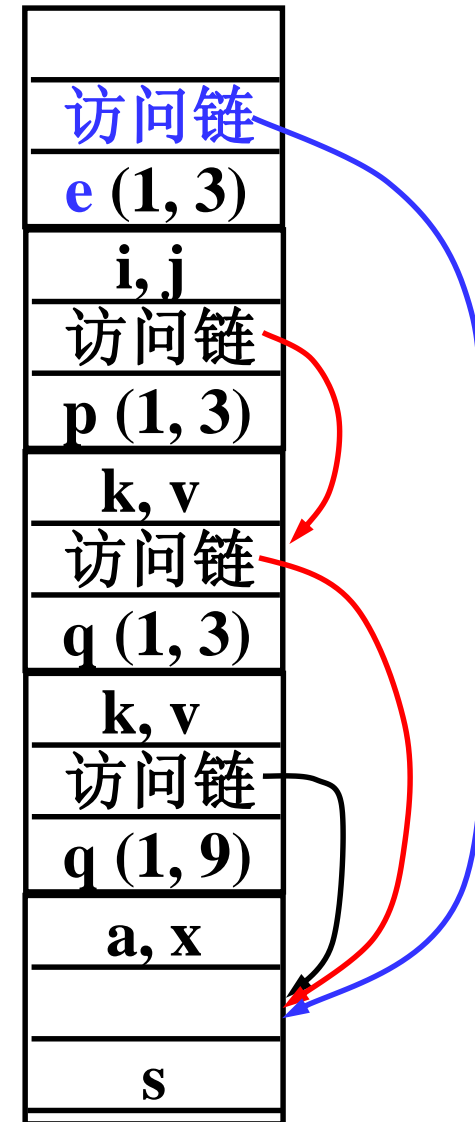
访问链的建立

2. $n_p \geq n_x$ 的情况

p 和 x 有公共的外围过程

- 追踪访问链 $n_p - n_x + 1$ 次, 到达静态包围 x 和 p 且离它们最近的那个过程的最新活动记录
- 所到达的活动记录就是 x 的活动记录中的访问链应该指向的那个活动记录
- partition调用exchange

sort	1
readarray	2
exchange	2
quicksort	2
partition	3





有过程嵌套定义的静态作用域

- 非局部名字访问：访问链
- 过程作为参数产生的问题和解决
- 过程作为返回值产生的问题



过程作为参数

```

program param(input, output); (过程作为参数)
  procedure b(function h(n: integer): integer);
    begin writeln(h(2)) end {b};
  procedure c;
    var m: integer;
    function f(n: integer): integer;
      begin f := m+n end {f};
    begin m := 0; b(f) end {c};
begin
  c
end.

```

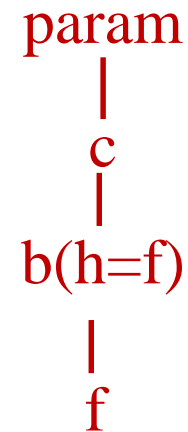
要先理解每个过程和函数的类型

b: (integer → integer) → void
c: void → void
f: integer → integer

静态的嵌套定义关系

param	1
b	2
c[m]	2
f	3

动态的调用关系





过程作为参数

```

program param(input, output); (过程作为参数)
  procedure b(function h(n: integer): integer);
    begin writeln(h(2)) end {b};
  procedure c;
    var m: integer;
    function f(n: integer): integer;
      begin f := m+n end {f};
    begin m := 0; b(f) end {c};
begin
  c
end.

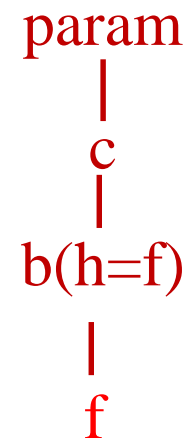
```

作为参数传递时，怎样在 f 被
激活时建立它的访问链，以便访
问非局部名字 m?

静态的嵌套定义关系

param	1
b	2
c[m]	2
f	3

调用关系





过程作为参数

```
program param(input, output); (过程作为参数)
```

```
  procedure b(function h(n: integer): integer);
```

```
    begin writeln(h(2)) end {b};
```

```
  procedure c;
```

b: (integer→integer)→void

```
    var m: integer;
```

```
  function f(n: integer): integer;
```

```
    begin f := m+n end {f};
```

```
  begin m := 0; b(f) end {c};
```

```
begin
```

```
  c
```

```
end.
```

param

b

c[m]

f

param

|

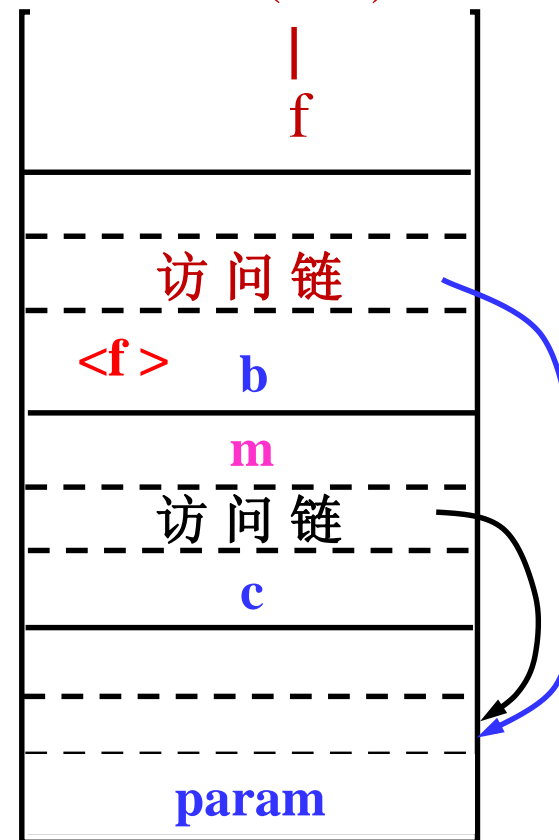
c

|

b(h=f)

|

f



从b的访问链难以建立
f的访问链



过程作为参数

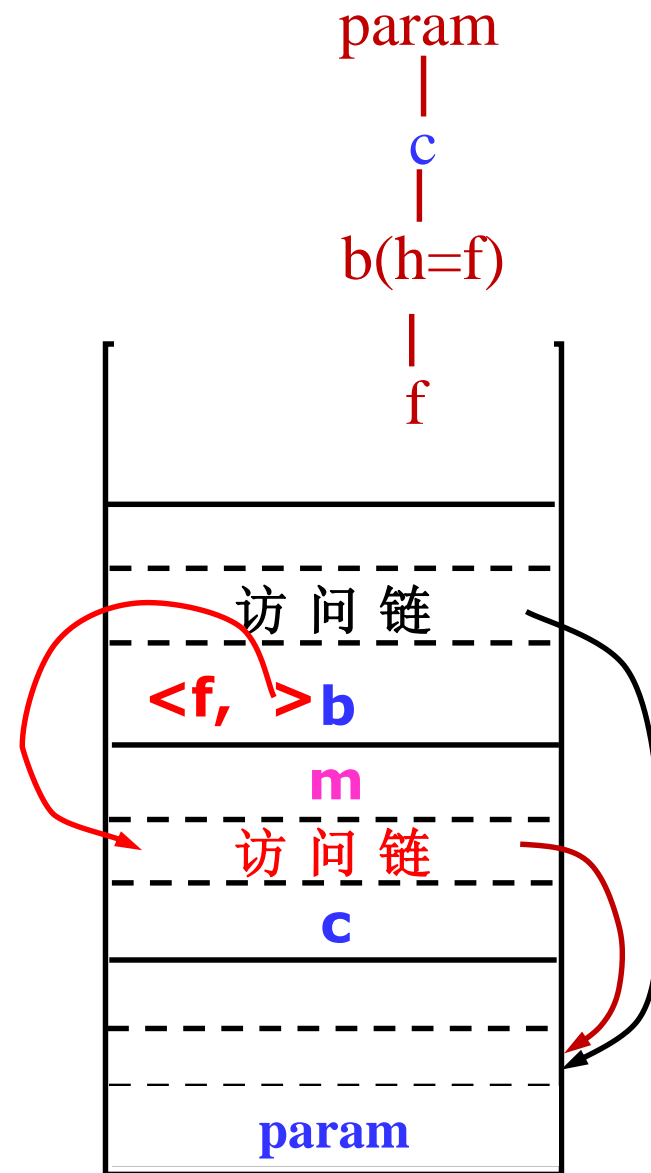
```

program param(input, output); (过程作为参数)
  procedure b(function h(n: integer): integer);
    begin writeln(h(2)) end {b};
  procedure c;
    var m: integer;
    function f(n: integer): integer;
      begin f := m+n end {f};
      begin m := 0; b(f) end {c};
begin
  c
end.

```

f作为参数传递时，它的起始地址连同它的访问链一起传递

param
b
c[m]
f





过程作为参数

```

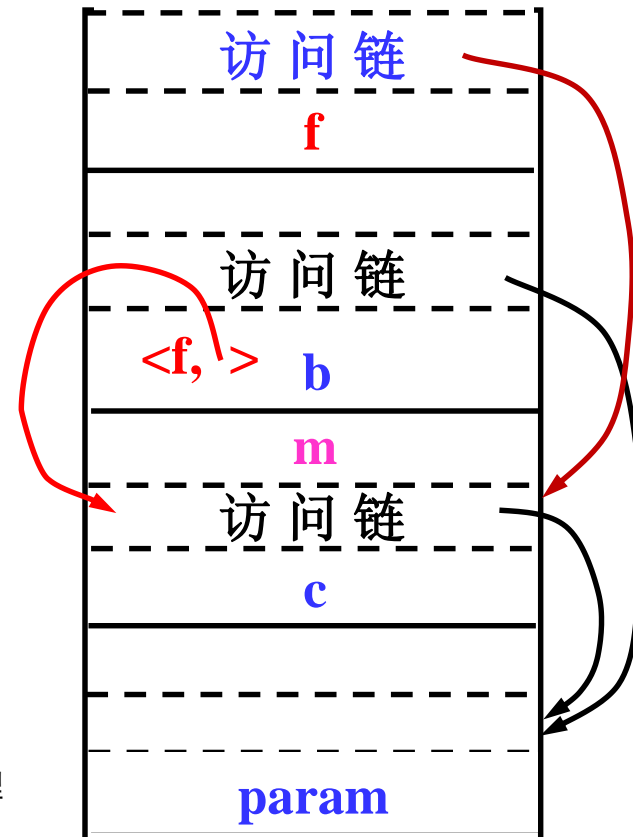
program param(input, output); (过程作为参数)
  procedure b(function h(n: integer): integer);
    begin writeln(h(2)) end {b};
  procedure c;
    var m: integer;
    function f(n: integer): integer;
      begin f := m+n end {f};
      begin m := 0; b(f) end {c};
  begin
    c
  end.

```

b调用f时，用传递过来的访问链来建立f的访问链

param
b
c[m]
f

param
|
c
|
b(h=f)
|
f





有过程嵌套定义的静态作用域

- 非局部名字访问：访问链
- 过程作为参数产生的问题和解决
- 过程作为返回值产生的问题



过程作为返回值

program ret (input, output); (过程作为返回值)

var **f**: function (integer): integer;

function **a**: function (integer): integer;

var **m**: integer;

function **addm** (n: integer): integer;

begin return **m**+n end;

begin m:= 0; return **addm** end;

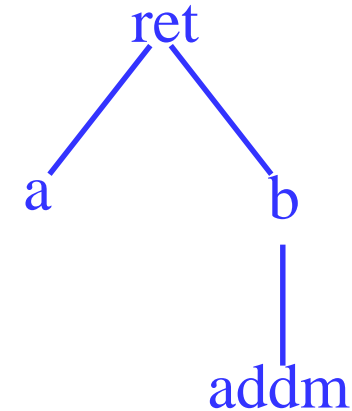
procedure **b** (g: function (integer): integer);

begin writeln (**g**(2)) end;

begin 这里是对a的调用

f := **a**; **b**(f)

end.



要理解每个过程和函数的类型

a: integer → integer ? ~~✗~~

addm: integer → integer

b: (integer → integer) → void

a: void → (integer → integer)



过程作为返回值

~~program ret (input, output); (过程作为返回值)~~

var f: function (integer): integer;

function **a**: function (integer): integer;

var **m**: integer;

function addm (n: integer): integer;

begin return **m**+n end;

begin m:= 0; return addm end;

procedure b (g: function (integer): integer);

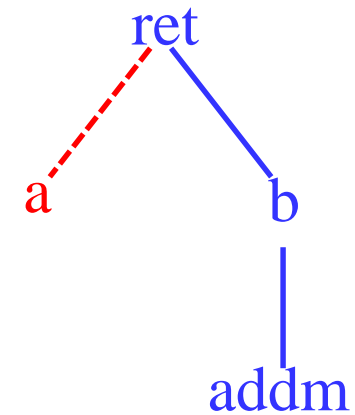
begin writeln (g(2)) end;

begin

f := **a**; b(f)

end.

执行addm时，a的活
动记录已不存在，取不
到m的值





- 不能嵌套定义
- 当前激活的函数要访问的数据分成两种情况
 - 非静态局部变量（包括形式参数）：分配在活动记录**栈顶的那个活动记录**中
 - 外部变量（包括定义在其它源文件之中的外部变量）和静态的局部变量：都分配在**静态数据区**
 - C语言允许函数（的指针）作为返回值



□ 采用静态作用域的语言

- 无嵌套过程：如 C++、Java、C#
- 嵌套过程：如 Python、JavaScript、Ruby

□ 闭包（closure）

- 解决过程作为返回值时要面对的问题

```
function add(x) {  
    return function(y) {  
        return x + y;  
    }  
}  
var add3 = add(3); // add3是闭包对象，包含函数(上述划线处)及其  
                // 声明时的环境{x=3}  
alert(add3(4));
```



□ 闭包 (closure)

- 解决过程作为返回值时要面对的问题
- 过程作为参数时也存在相似的问题

```
function do10times(fn)
  for i = 0,9 do
    fn(i)
  end
end
sum = 0
function addsum(i)
  sum = sum + i
end
do10times(addsum)
print(sum)
```



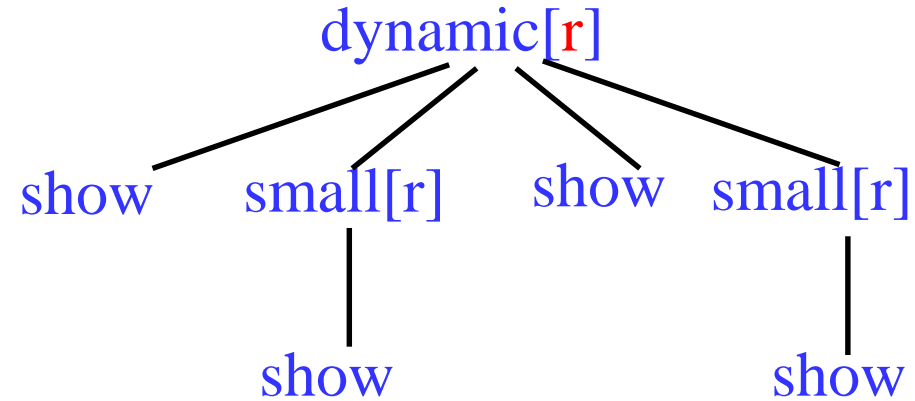
动态作用域

- 基于运行时的调用关系来确定非局部名字引用的存储单元
- 过程调用时，仅为被调用过程的局部名字建立新的绑定（在活动记录中）
- 实现动态作用域的方法
 - 深访问、浅访问



示例：基于静态作用域时

```
program dynamic(input, output);  
  var r: real;  
  procedure show;  
    begin write(r: 5: 3) end;  
  procedure small;  
    var r: real;  
    begin r := 0.125; show end;  
begin  
  r := 0.25;  
  show; small; writeln;  
  show; small; writeln  
end.
```



show在dynamic中定义，
show中访问的r是
dynamic中定义的

执行后输出：

0.250 0.250

0.250 0.250



示例：基于动态作用域时

```
program dynamic(input, output);
```

```
  var r: real;
```

```
  procedure show;
```

```
    begin write(r: 5: 3) end;
```

```
  procedure small;
```

```
    var r: real;
```

```
    begin r := 0.125; show end;
```

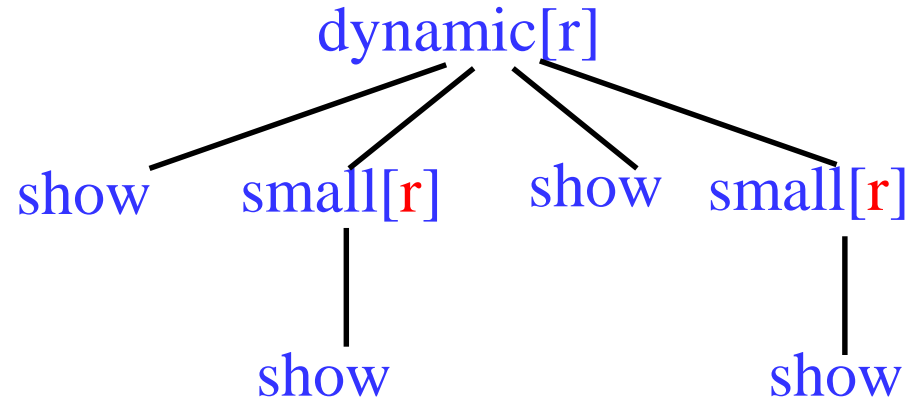
```
begin
```

```
  r := 0.25;
```

```
  show; small; writeln;
```

```
  show; small; writeln
```

```
end.
```



- **dynamic**中调用的**show**所访问的**r**是**dynamic**中定义的
- **small**中调用的**show**所访问的**r**是**small**中定义的

执行后输出:

0.250 **0.125**

0.250 **0.125**



□ 使用动态作用域的语言

- Pascal、Emacs Lisp、Common Lisp(兼有静态作用域)、Perl (兼有静态作用域)、Shell语言 (bash, dash, PowerShell)

□ 其他

- 宏展开

[https://en.wikipedia.org/wiki/Scope_\(computer_science\)](https://en.wikipedia.org/wiki/Scope_(computer_science))



□ 深访问

- 用控制链搜索运行栈，寻找包含该非局部名字的第一个活动记录

□ 浅访问

- 为每个名字在**静态分配的存储空间**中保存它的当前值
- 当过程 p 的新活动出现时， p 的局部名字 n 使用在静态数据区分配给 n 的存储单元。 n 的先前值保存在 p 的**活动记录**中，当 p 的活动结束时再恢复



基于浅访问实现动态作用域

program dynamic(input, output);

var r: real;

procedure show;

begin write(r: 5: 3) end;

procedure **small**;

var r: real;

begin r := 0.125; **show** end;

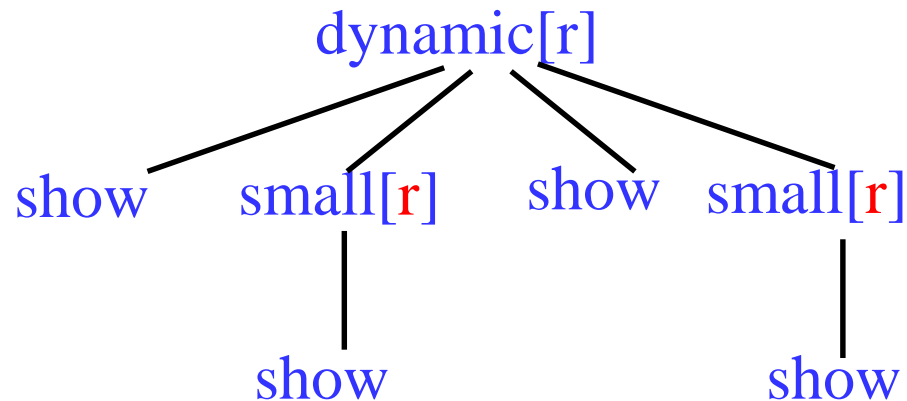
begin(蓝色表示已执行部分)

r := 0.25;

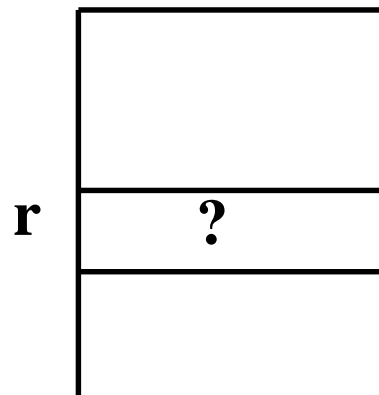
show; small; writeln;

show; small; writeln

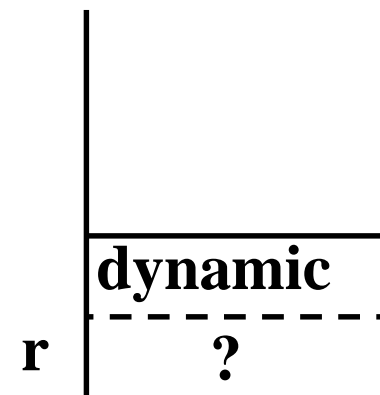
end.



静态区
使用值的地方



栈区
暂存值的地方





基于浅访问实现动态作用域

program dynamic(input, output);

var r: real;

procedure show;

begin write(r: 5: 3) end;

procedure **small**;

var r: real;

begin r := 0.125; **show** end;

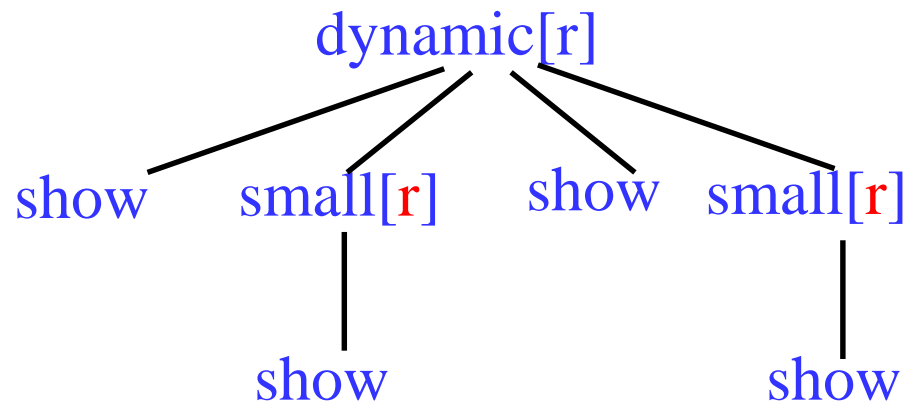
begin(蓝色表示已执行部分)

r := 0.25;

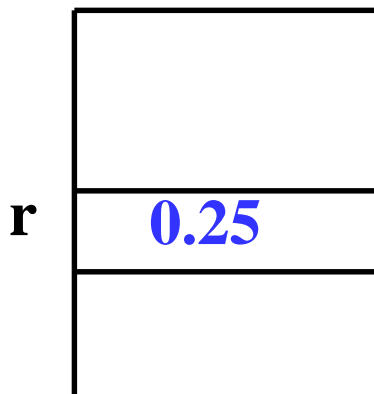
show; small; writeln;

show; small; writeln

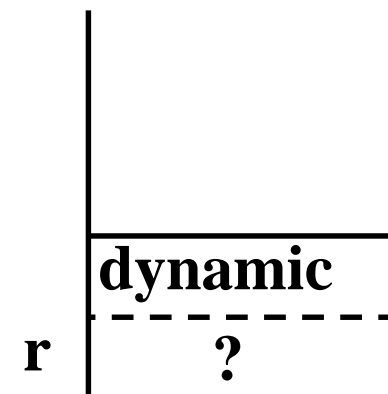
end.



静态区
使用值的地方



栈区
暂存值的地方





基于浅访问实现动态作用域

program dynamic(input, output);

var r: real;

procedure show;

begin write(r: 5: 3) end;

procedure **small;**

var r: real;

begin r := 0.125; **show** end;

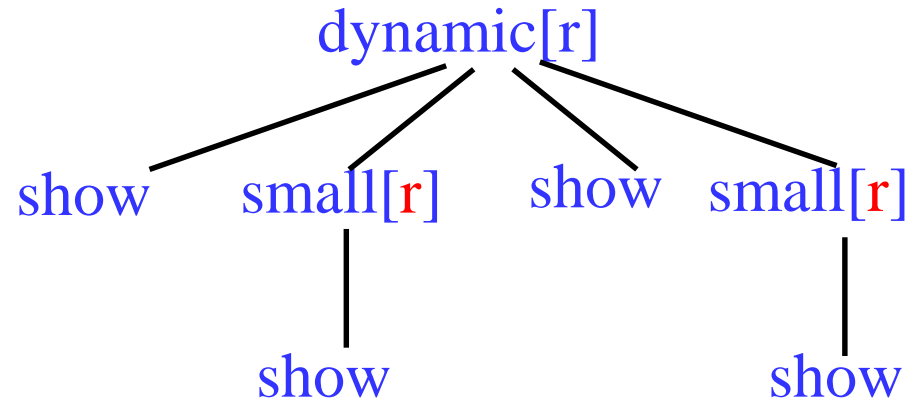
begin(蓝色表示已执行部分)

r := 0.25;

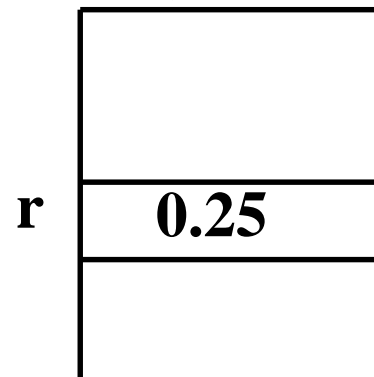
show; small; writeln;

show; small; writeln

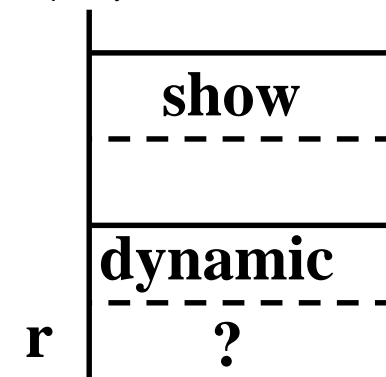
end.



静态区
使用值的地方



栈区
暂存值的地方





基于浅访问实现动态作用域

program dynamic(input, output);

var r: real;

procedure show;

begin write(r: 5: 3) end;

procedure **small**;

var r: real;

begin r := 0.125; **show** end;

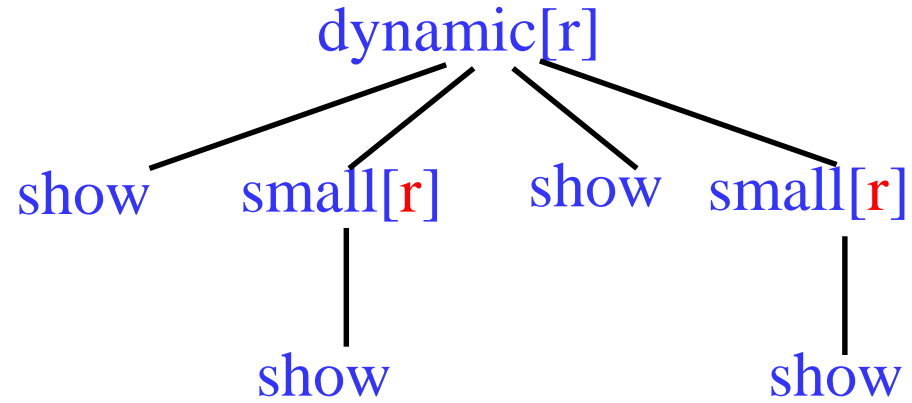
begin(蓝色表示已执行部分)

r := 0.25;

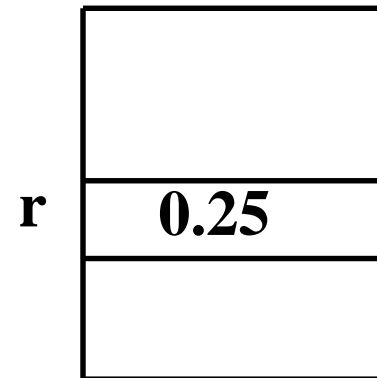
show; small; writeln;

show; small; writeln

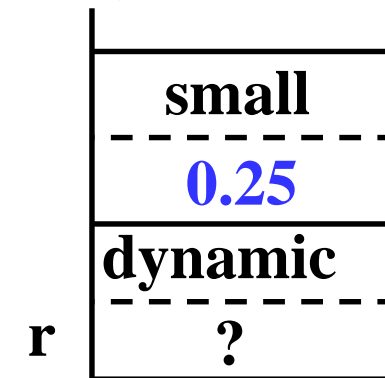
end.



静态区
使用值的地方



栈区
暂存值的地方





基于浅访问实现动态作用域

program dynamic(input, output);

var r: real;

procedure show;

begin write(r: 5: 3) end;

procedure **small;**

var r: real;

begin r := 0.125; show end;

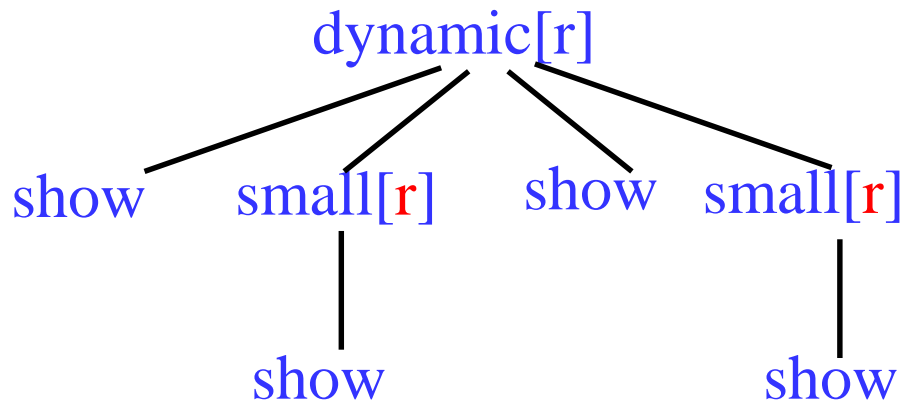
begin(蓝色表示已执行部分)

r := 0.25;

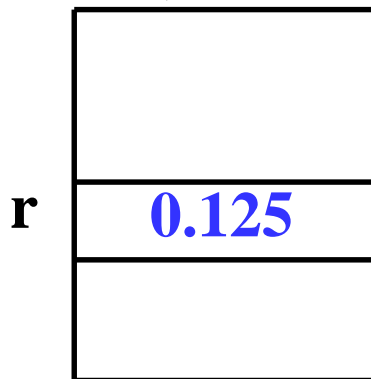
show; small; writeln;

show; small; writeln

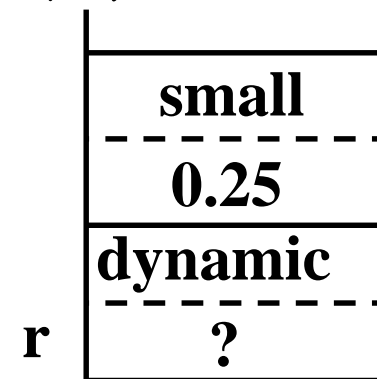
end.



静态区
使用值的地方



栈区
暂存值的地方





基于浅访问实现动态作用域

program dynamic(input, output);

var r: real;

procedure show;

begin write(r: 5: 3) end;

procedure **small;**

var r: real;

begin r := 0.125; **show** end;

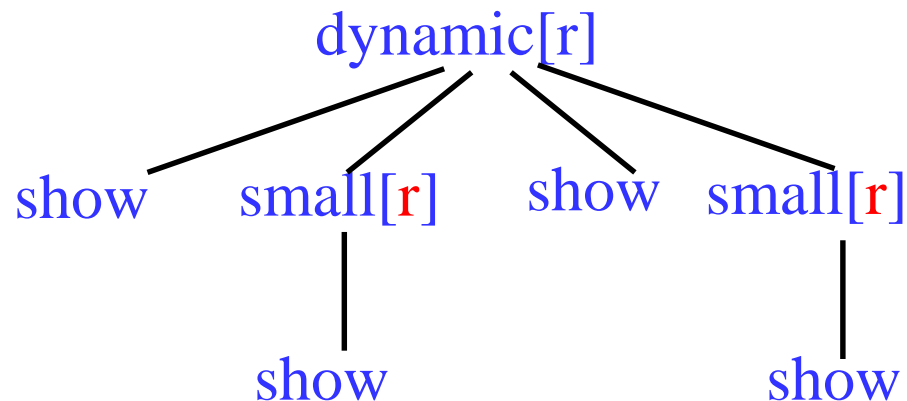
begin(蓝色表示已执行部分)

r := 0.25;

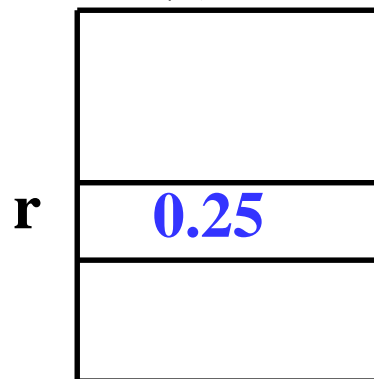
show; small; writeln;

show; small; writeln

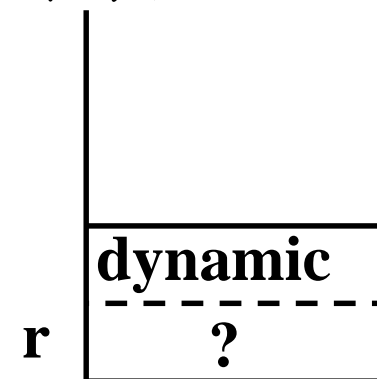
end.



静态区
使用值的地方



栈区
暂存值的地方





6.4 参数传递

- 值调用
- 引用调用
- 换名调用



值调用(call by value)

□ 特点

- 实参的右值传给被调用过程

□ 值调用的可能实现方法

- 把形参当作所在过程的局部名看待，形参的存储单元在该过程的活动记录中
- 调用过程计算实参，并把其右值放入被调用过程形参的存储单元中

□ 语言示例

■ C、C++、Pascal

- 数组名或对象名作为参数传递时，本质是传递它们的地址或引用

■ Go均是值传递

- 对于slice、map、channel等，实际只传递底层数据结构的引用



引用调用(call by reference)

□ 特点

- 实参的左值传给被调用过程

□ 引用调用的可能实现方法

- 调用过程计算实参，把实参的左值放入被调用过程形参的存储单元
- 在被调用过程的目标代码中，任何对形参的引用都是通过传给该过程的指针间接引用实参

□ 语言示例

- Pascal中的var参数、C++中的引用类型参数（如 `int &`）
- Python均按对象引用传递：
 - 可变对象（列表、字典等）：对它们的修改直接影响到原始对象；
 - 不可变对象（如整数、字符串、元组等）：修改这些对象会生成新对象
 - 利用copy模块的浅复制 `copy()` 或深复制 `deepcopy()` 函数可对可变对象进行复制



换名调用(call by name)

□ 特点

- 用实参表达式对形参进行正文替换,然后再执行

```
procedure swap(var x, y: integer);
```

```
var temp: integer;
```

```
begin
```

```
    temp := x;
```

```
    x := y;
```

```
    y := temp
```

```
end
```

例如: 调用swap(i, a[i])

替换结果: temp := i;
 i := a[i];
 a[i] := temp

**交换两个数据的程序
并非总是正确**



换名调用(call by name)

□ 特点

- 用实参表达式对形参进行正文替换,然后再执行

□ 使用及实现

- 宏展开或内联展开,实参建议用括号包围以保持其完整性
- 展开前被调用过程的局部名可通过换名,以和调用过程的名字保持区别

□ 语言示例

- C/C++: `inline`关键字, Rust: `#[inline]`属性
- Java: 编译器或虚拟机的内联优化; Go: 编译器的内联优化



6.5 其他

- 堆：分配与回收
- 计算机内存分层
- 局部性：时间、空间



□ 堆：存放生存期不确定的数据

■ C: malloc、free

glibc 的 [ptmalloc](#), [Doug Lea's dlmalloc](#);

高效的并发内存分配器 [jemalloc](#), [TBBmalloc](#), [TCMalloc \(gperftools\)](#)

■ C++: new、delete

“[Foundations of the C++ Concurrency Memory Model](#)”(PLDI2008)

■ Java: new、Garbage Collection

[Richard Jones's the Garbage Collection Page](#)

■ Go: new、Garbage Collection

[The Go Memory Model](#)

■ Rust: ownership; new、drop/dealloc

<https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html>



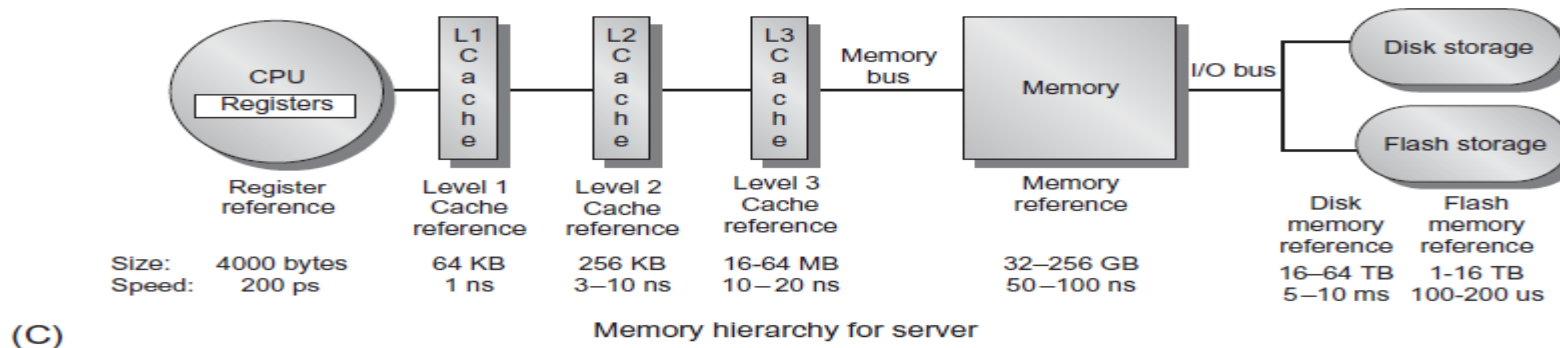
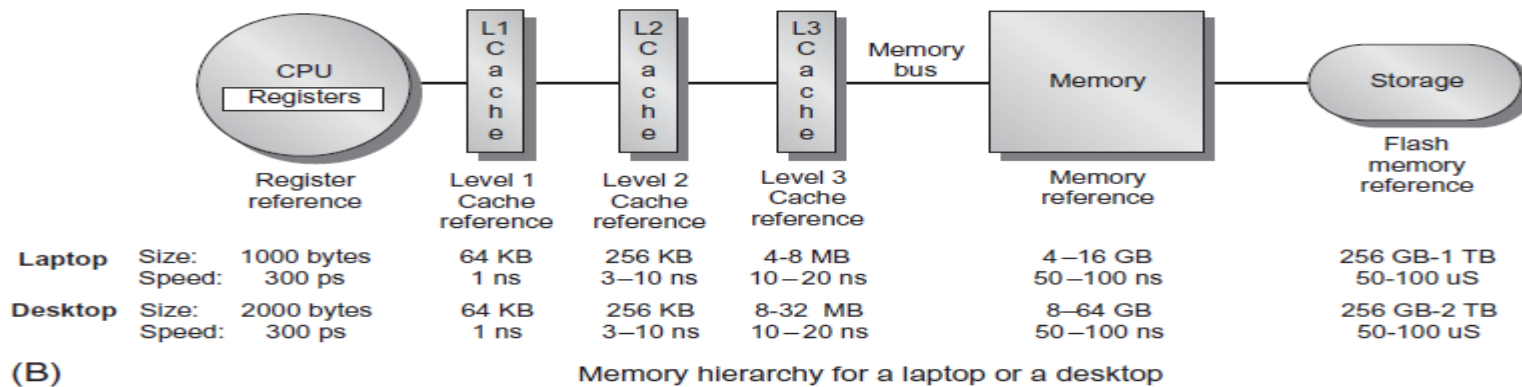
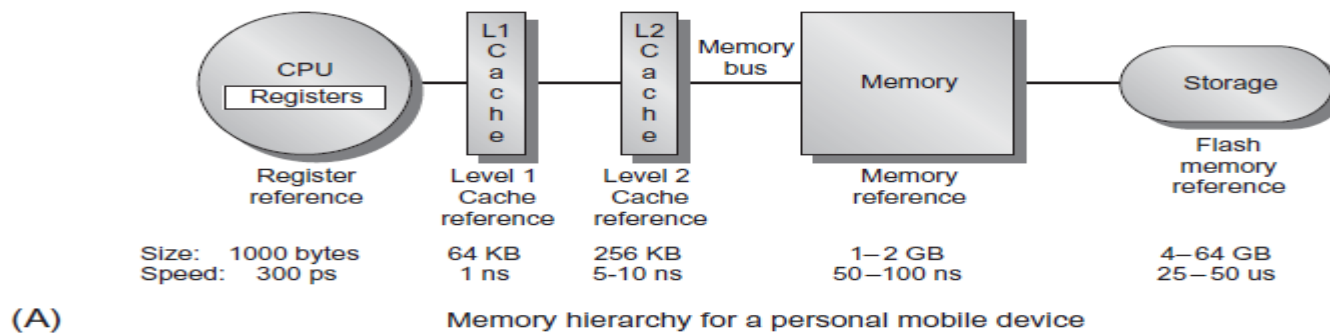
- **内存管理器,也称内存分配器(allocator)**
 - 维护的基本信息: 堆中空闲空间、...
 - 重点要实现的函数: 分配、回收
- **内存管理器应具有下列性质**
 - **空间有效性:** 极小化程序需要的堆空间总量
 - **程序有效性:** 较好地利用内存子系统, 使得程序能运行得快一些
 - **低开销:** 分配和回收操作所花时间在整個程序执行时间中的比例尽量小



计算机内存分层

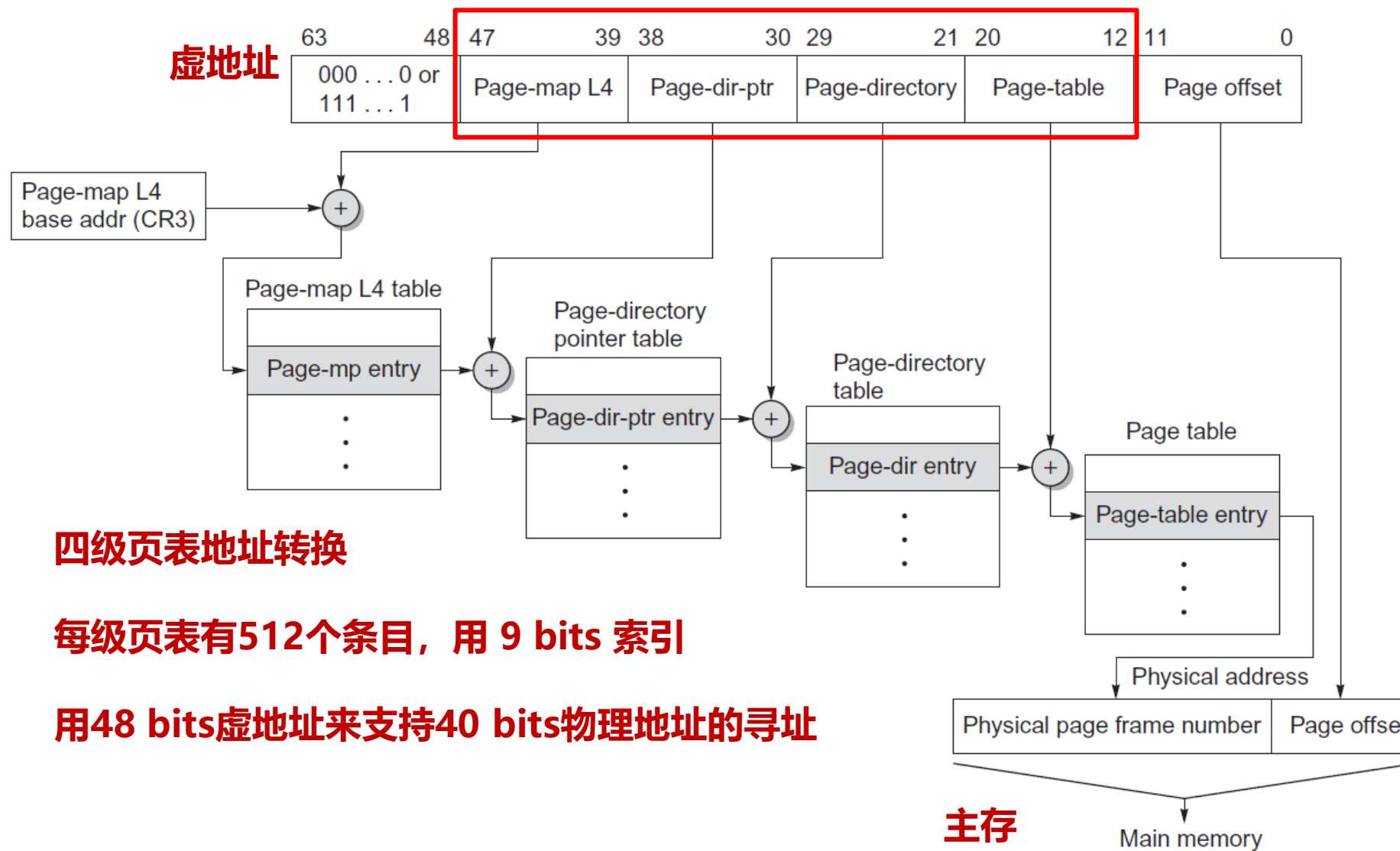
<https://item.jd.com/12553439.html> 第6版

Computer Architecture, Sixth Edition: A Quantitative Approach, 2019



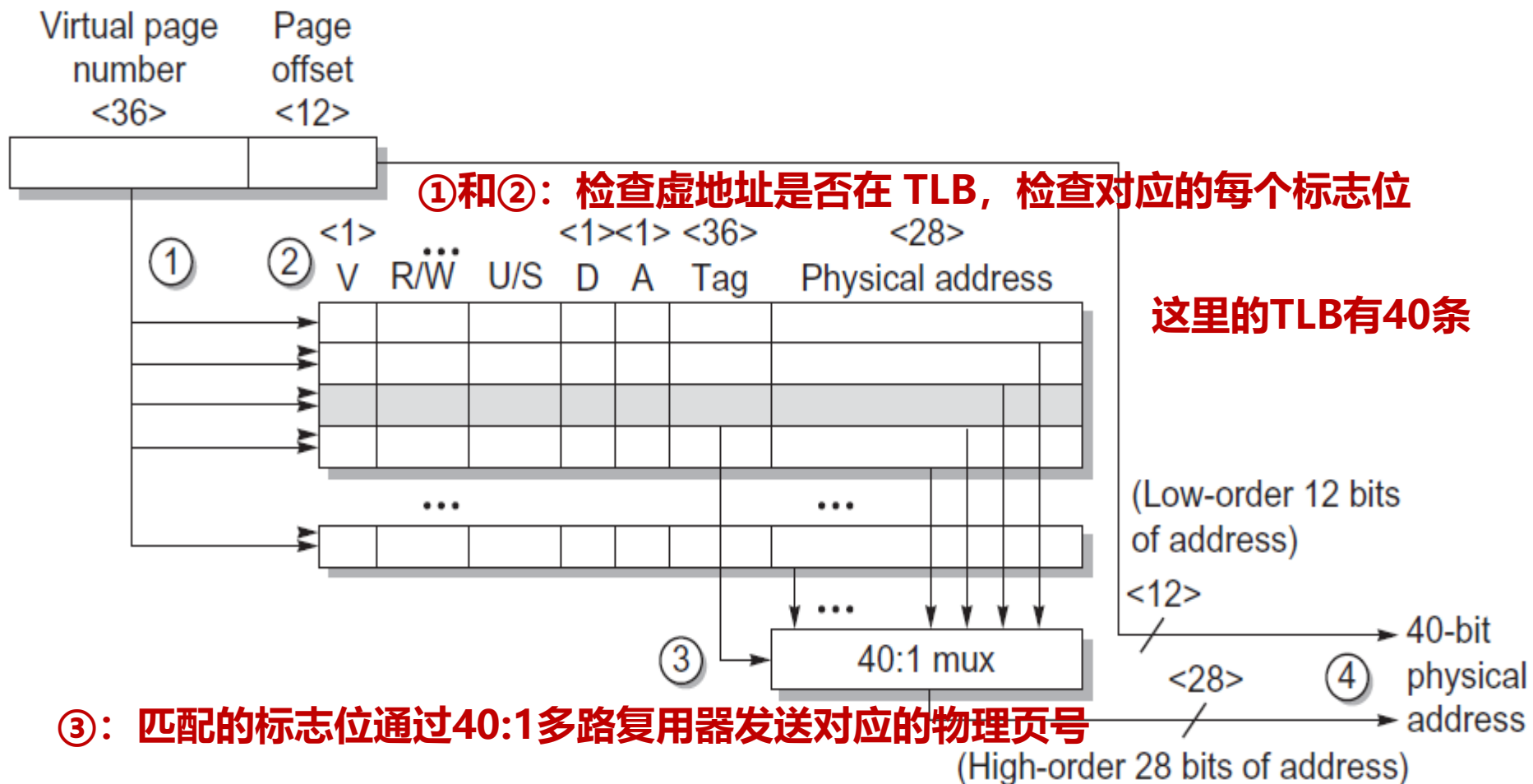


AMD Opteron 皓龙虚地址映射





□ 快表 (TLB, Translation look aside buffer)





Intel Core i7

峰值内存带宽：25 GB/s；使用48位虚拟地址、36位物理地址；两级 TLB

Characteristic	Instruction TLB	Data DLB	Second-level TLB
Size	128	64	512
Associativity	4-way	4-way	4-way
Replacement	Pseudo-LRU	Pseudo-LRU	Pseudo-LRU
Access latency	1 cycle	1 cycle	6 cycles
Miss	7 cycles	7 cycles	Hundreds of cycles to access page table

TLB
结构

Characteristic	L1	L2	L3
Size	32 KB I/32 KB D	256 KB	2 MB per core
Associativity	4-way I/8-way D	8-way	16-way
Access latency	4 cycles, pipelined	10 cycles	35 cycles
Replacement scheme	Pseudo-LRU	Pseudo-LRU	Pseudo-LRU but with an ordered selection algorithm

三级
cache
结构



程序局部性(locality)

大多数程序的大部分时间在执行一小部分代码，并且仅涉及一小部分数据

□ 时间局部性(temporal locality)

程序访问的内存单元在很短的时间内可能再次被程序访问

□ 空间局部性(spatial locality)

毗邻被访问单元的内存单元在很短的时间内可能被访问

□ 有效利用缓存

- Cache容量有限、最近使用的指令保存在cache中

- **改变数据布局或计算次序**=>改进程序局部性



举例: 改变计算次序

```
void copyij (int src[2048][2048],  
            int dst[2048][2048])  
{  
    int i,j;  
    for (i = 0; i < 2048; i++)  
        for (j = 0; j < 2048; j++)  
            dst[i][j] = src[i][j];  
}
```

```
void copyji (int src[2048][2048],  
            int dst[2048][2048])  
{  
    int i,j;  
    for (j = 0; j < 2048; j++)  
        for (i = 0; i < 2048; i++)  
            dst[i][j] = src[i][j];  
}
```

上述两个函数功能、算法一样，但执行时间一样吗？

```
int a[2048][2048] = {1, 1};  
int b[2048][2048];  
int main( )  
{  
    copyij(a, b);  
    // copyji(a, b);  
}
```

```
$ time ./copyij  
real 0m0.046s  
user 0m0.037s  
sys 0m0.008s
```

```
$ time ./copyji  
real 0m0.404s  
user 0m0.384s  
sys 0m0.020s
```



举例: 改变计算次序

```
int a[2048][2048] = {1, 1};  
int b[2048][2048];  
int main( )  
{  
    copyij(a, b);  
    // copyji(a, b);  
}
```



```
int a[2048][2048] = {1, 1};  
int main( )  
{  
    int b[2048][2048];  
    copyij(a, b);  
    // copyji(a, b);  
}
```

上述功能一样, 但执行会有什么变化呢?

Segmentation fault (core dumped) ☹️

Why ?

操作系统、编译器: 进程地址空间布局: 栈大小有限, 如为8MB

$2048 * 2048 * 4 = 16MB$



举例: 改变数据布局

例: 一个结构体大数组分拆成若干个数组

```
struct student {          int num[10000];  
int num;                 char name[10000][20];  
char name[20];           ...    ...  
...    ...  
}  
struct student st[10000];
```

- 若是顺序处理每个结构体的多个域, 左边的数据局部性较好
- 若是先顺序处理每个结构的num域, 再处理每个结构的name域, ..., 则右边的数据局部性较好
- 最好是按左边编程, 由编译器决定是否需要将数据按右边布局