



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

# 实用算法设计

## ➤ 算法设计技术

主讲：娄文启

# 1.4 设计算法的步骤



- 问题分析:

- 问题1: 给定一个英语词典, 找出其中的所有变位词集合。例如, " pots" 、 " stop" 和 " tops" 互为变位词。
  - 问题规模? 几万或几十万
  - 变位词具有什么性质?

输入:

```
["eat","tea","race","care","heart","earth","nat","ate","bcd","adc"]
```

输出:

```
[["bcd"],["nat"],["heart","earth"],["adc"],["race","care"],["eat","tea","ate"]]
```

# 1.4 设计算法的步骤



- 问题分析:

- 问题1: 给定一个英语词典, 找出其中的所有变位词集合。例如, " pots" 、 " stop" 和" tops" 互为变位词。

C++ STL库中的unordered\_map: 底层实用哈希表实现, 无序

```
unordered_map<int, string> umv;  
  
umv[1] = "hhh";  
umv[2] = "hahaha";//赋值, 插入  
  
umv.insert( make_pair(1, "haohaohao" ) ); // insert  
umv[2] = "haohaohao";//赋值, 插入  
umv[3];//anyway  
for(const auto& item : umv){  
    cout << item.first << " " << item.second<<endl;  
}
```

输出

```
3  
2 haohaohao  
1 hhh
```

# 1.4 设计算法的步骤



- 问题分析:

- 问题1: 给定一个英语词典, 找出其中的所有变位词集合。例如, " pots" 、 " stop" 和" tops" 互为变位词。

```
vector<vector<string>> groupAnagrams(vector<string>& strs) {  
    unordered_map<string, vector<string> >ump;  
    for(auto &i:strs){  
        string key = i;  
        sort(i.begin(), i.end());  
        ump[i].push_back(key);  
    }  
    vector<vector<string>> res;  
    for(auto &i:ump){  
        res.push_back(i.second);  
    }  
    return res;  
}
```

n: 字符串数量

klog k: 字符串k排序

# 1.4 设计算法的步骤



- 问题分析:

- 问题1: 给定一个英语词典, 找出其中的所有变位词集合。例如, " pots" 、 " stop" 和" tops" 互为变位词。

```
vector<vector<string>> groupAnagrams(vector<string>& strs) {  
    unordered_map<string, vector<string> >ump;  
    for(auto &i:strs){  
        string tmp(26,'0');  
        for(auto &ch:i){  
            tmp[ch-'a']+=1;}  
        ump[tmp].push_back(i);  
    }  
    vector<vector<string>> res;  
    for(auto &i:ump){  
        res.push_back(i.second);}  
    return res;  
};
```

n: 字符串数量

K:遍历 子字符串

# 位图数据类型 (Bitset)



- C++ 标准库中的一个提供了 `std::bitset` 类，用于表示二进制位序列。它提供了一种方便的方式来处理二进制数据，尤其适用于位运算操作。
- `std::bitset` 类型表示一个固定长度的位序列，每个位都只能是 0 或 1。这个固定长度在创建对象时指定，并且不能在运行时更改。类似于整数类型，`std::bitset`

```
#include<bitset>
using namespace std;
int main(){
    cout<<"bitset8: "    <<sizeof(bitset<8>)<<endl;
    cout<<"bitset24: "   <<sizeof(bitset<24>)<<endl;
    cout<<"bitset16: "   <<sizeof(bitset<16>)<<endl;
    cout<<"bitset32: "   <<sizeof(bitset<32>)<<endl;
    cout<<"bitset48: "   <<sizeof(bitset<48>)<<endl;
    cout<<"bitset64: "   <<sizeof(bitset<64>)<<endl;
    cout<<"bitset512: "  <<sizeof(bitset<512>)<<endl;
    cout<<"bitset1024: " <<sizeof(bitset<1024>)<<endl;
    cout<<"bitset2048: " <<sizeof(bitset<2048>)<<endl;
    cout<<"bitset65536: "<<sizeof(bitset<65536>)<<endl;
    cout<<"bitset4000000000: "<<sizeof(bitset<4000000000>)<<endl;
}
```

## 运行结果

```
bitset8: 4
bitset24: 4
bitset16: 4
bitset32: 4
bitset48: 8
bitset64: 8
bitset512: 64
bitset1024: 128
bitset2048: 256
bitset65536: 8192
bitset4000000000: 500000000
```

# 位图数据类型 (Bitset)



- 支持包括位运算、位查询和位设置等多种操作
  - ✓ test(pos) 返回 std::bitset 中位于 pos 位置的值
  - ✓ set(pos) 将 std::bitset 中位于 pos 位置的值设为 1
  - ✓ reset(pos) 将 std::bitset 中位于 pos 位置的值设为 0

```
#include<bitset>
using namespace std;
int main(){
    bitset<32> bitset32;
    bitset32.set(2);
    cout<<bitset32<<endl;
    bitset<8> bitset8("00100010");
    cout<<bitset8<<endl;
    bitset<4> bitset4("0110");
    cout<<bitset4<<" bytes: "<<sizeof(bitset<4>)<<endl;
    bitset<4> bitset4_1(2);
    cout<<bitset4_1<<endl;
}
```

## 运行结果

[illegible]

# 1.5 最佳算法选择的决定因素



## 1) 问题的约束:

- 比如, 可用内存空间, 运行时间的上限约束等。

## 2) 数据的存储方式

- 存储什么信息? (比如, 位图法)
- 选用何种数据结构? (取决于在数据上的常用操作类型 (静态? 动态? )、以及内存空间的限制)
- 比如, 顺序表上进行插入/删除比较慢, 所以, 在顺序表上实现交换2个元素的算法要尽量回避这些速度慢的操作。

## 3) 输入数据的特征: 是否要求有序?

## 4) 输出数据的特征: 是否要求有序? 是否包含重复记录。



问题：给定一个数N，N! 表示N的阶乘，**求N! 的二进制表示中最低为1的位置**

- 可以转换为 N! 二进制表示中末尾0的个数。
- N! 的质因数分解中，一个2的因子将贡献1个0

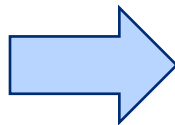
1~N中，每包含一个数 $2^k$ ，将贡献乘积中的k个0

等价于求  $[N/2] + [N/4] + [N/8] + \dots$

```
int lowestOne(int N){
    int Ret = 0;

    for(int i = 1; i <= N; i++) {
        int j = i;
        while(j % 2 == 0) {
            Ret++;
            j = j >> 1;
        }
    }

    return Ret;
}
```



```
int lowestOne(int N)
{
    int Ret = 0;

    while(N)
    {
        N >> 1;
        Ret += N;
    }

    return Ret;
}
```

问题：给定一个数N，N! 表示N的阶乘，**求N! 的二进制表示中最低为1的位置**

- 分析：N!二进制表示中0的个数，等于N减去N的二进制表示中1的数目
- Eg: N = 11011
$$\begin{aligned} & [N/2] + [N/4] + [N/8] + \dots \\ &= 1101 + 110 + 11 + 1 \\ &= (1000 + 100 + 1) + (100 + 10) + (10 + 1) + 1 \\ &= (1000 + 100 + 10 + 1) + (100 + 10 + 1) + (1) \\ &= 1111 + 111 + 1 \\ &= (10000 - 1) + (1000 - 1) + (10 - 1) + (1 - 1) \\ &= 11011 - (N \text{ 的二进制表示中的1的个数}) \end{aligned}$$

- 问题分析:

- 给定一个数 $N$ ,  $N!$  表示 $N$ 的阶乘, 求 $N!$  的十进制末尾0的个数?

- 分析: 考虑质因数分解,  $N! = (2^X) \times (3^Y) \times (5^Z) \dots$  末尾的0的个数为  $M = \min(X, Z)$
    - 能被2整除的数比能被5整除的数多, 有  $X > Z \rightarrow M = Z$

换一个角度考虑 $[1, n]$ 中质因子 $p$ 的个数

$[1, n]$  中  $p$  的倍数有  $n_1 = \left\lceil \frac{n}{p} \right\rceil$  个, 这些数至少贡献出了  $n_1$  个质因子  $p$ 。

$p^2$  的倍数有  $n_2 =$

$\left\lceil \frac{n}{p^2} \right\rceil$  个, 由于这些数已经是  $p$  的倍数了, 为了不重复统计  $p$  的个数, 我们仅考虑额外贡献的质因数个数,

依此类推,  $[1, n]$ 中质因子 $p$ 的个数为

$$\sum_{k=1}^{\infty} \left\lceil \frac{n}{p^k} \right\rceil$$

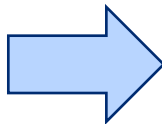
- 问题分析:

- 给定一个数N,  $N!$  表示N的阶乘, 求 $N!$  的十进制末尾0的个数?
  - 分析: 考虑质因数分解,  $N! = (2^X) \times (3^Y) \times (5^Z) \dots$  末尾的0的个数为  $M = \min(X, Z)$
  - 能被2整除的数比能被5整除的数多, 有  $X > Z \rightarrow M = Z$

```
int lowestOne(int N)
{
    int Ret = 0;

    for(int i = 1; i <= N; i++) {
        int j = i;
        while(j % 5 == 0) {
            Ret++;
            j /= 5;
        }
    }

    return Ret;
}
```



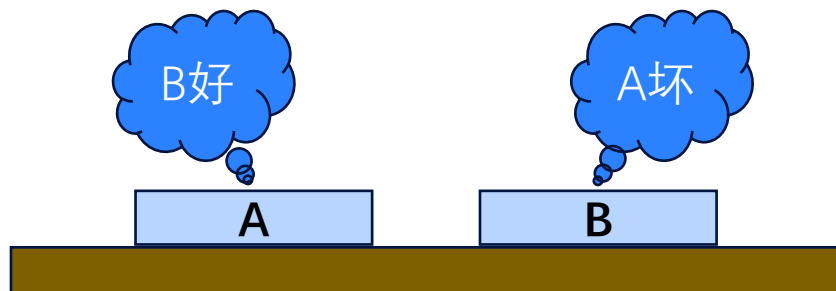
```
int lowestOne(int N)
{
    int Ret = 0;

    while(N)
    {
        Ret += N / 5;
        N /= 5;
    }

    return Ret;
}
```

- 一次测试过程

- 测试方法：将 2 片芯片 (A 和 B) 置于测试台上，互相进行测试，测试报告是“好”或“坏”，只取其一。



## 测试结果分析

A报告	B报告	结论
B是好的	A是好的	A、B都好或都坏
B是好的	A是坏的	至少一片是坏的
B是坏的	A是好的	至少一片是坏的
B是坏的	A是坏的	至少一片是坏的

- 假设：**好芯片的报告一定是正确的，坏芯片的报告是不确定的 (可能会出错)。

- 问题

- **输入**：有  $n$  片芯片，其中好芯片至少比坏芯片多 1 片。
- **问题**：设计一种测试方法，通过测试从  $n$  片芯片中挑出 1 片好芯片。
- **要求**：要求使用最少的测试次数。

- 判定芯片A的好坏

- 问题：给定芯片 A，判定 A 的好坏。
- 方法：用其他  $n-1$  片芯片对 A 测试。
- 例： $n = 7$ ：好芯片数  $\geq 4$ 
  - A 好，6 个报告中至少有 3 个报“好”
  - A 坏，6 个报告中至少有 4 个报“坏”

- **$n$  是奇数：好芯片数  $\geq (n+1)/2$**

- A 好，至少有  $(n-1)/2$  个报“好”
- A 坏，至少有  $(n+1)/2$  个报“坏”

- **$n$  是奇数: 好芯片数  $\geq (n+1)/2$** 
  - A 好, 至少有  $(n-1)/2$  个报 “好”
  - A 坏, 至少有  $(n+1)/2$  个报 “坏”

**结论:** 至少一半报 “好”, A 是好芯片,  
超过一半报 “坏”, A 是坏芯片。

- $n=8$ : 好芯片数  $\geq 5$ 
  - A 好, 7 个报告中至少 4 个报 “好”
  - A 坏, 7 个报告中至少 5 个报 “坏”

- **$n$  是偶数: 好芯片数  $\geq n/2+1$** 
  - A 好, 至少有  $n/2$  个报告 “好”
  - A 坏, 至少有  $n/2+1$  个报告 “坏”。
- **结论:**  $n-1$  份报告中,
  - 至少一半报 “好”, 则 A 为好芯片
  - 超过一半报 “坏”, 则 A 为坏芯片。

- 蛮力算法

- **测试方法：**任取 1 片测试，如果是好芯片，测试结束；如果是坏芯片，抛弃，再从剩下芯片中任取 1 片测试，直到得到 1 片好芯片。

- **时间估计：**

第 1 片坏芯片，最多测试  $n - 2$  次，

第 2 片坏芯片，最多测试  $n - 3$  次，

...

时间复杂度：总计  $\Theta(n^2)$ 。

最坏有  $(n-1)/2$  片坏芯片，  
每次测试复杂度  $\Theta(n)$



- 分治算法设计思想

- 假设  $n$  为偶数，将  $n$  片芯片两两一组做测试淘汰，剩下芯片构成子问题，进入下一轮分组淘汰。

- 淘汰规则：

- “好，好”  $\Rightarrow$  任留 1 片，进入下轮

- 其他情况  $\Rightarrow$  全部抛弃。

- 递归截止条件：  $n \leq 3$

- 3 片芯片，1 次测试可得到好芯片

- 1 或 2 片芯片，不再需要测试

- 分治算法的正确性

- 命题1:** 当  $n$  是偶数时, 在上述淘汰规则下, 经过一轮淘汰, 剩下的好芯片比坏芯片至少多一片

这里说的是芯片的实际情况

- 证: 设  $A$ 、 $B$  都好的芯片为  $i$  组,  $A$  与  $B$  一好一坏为  $j$  组,  $A$  与  $B$  都坏的  $k$  组。淘汰后好芯片  $i$  片, 坏芯片至多  $k$  片。

$$2i + 2j + 2k = n$$

$$2i + j > 2k + j$$

初始芯片总数

初始 好芯片多于坏芯片



$$i > k$$

- $n$  为奇数时的特殊处理

- 输入: 

好	好	好	好	坏	坏	坏
---	---	---	---	---	---	---
- 分组: 

好	好	好	好	坏	坏	坏
---	---	---	---	---	---	---
- 淘汰后: 

好	好	坏	坏
---	---	---	---

处理办法: 当  $n$  为奇数时, 增加一轮对轮空芯片的单独测试。

如果该芯片为好芯片, 算法结束; 如果是坏芯片, 则淘汰该芯片。


## • 伪码描述

算法 Test(n):

```
1.   k = n
2.   while k > 3:
3.       将芯片分成[k/2]组 // 轮空处理
4.       for i = 1 to [k/2]:
5.           if 2 片好:
6.               则任取 1 片留下
7.           else:
8.               2 片同时丢掉
9.       k = 剩下的芯片数
10.  if k = 3:
11.     任取 2 片芯片测试
12.     if 1 好 1 坏:
13.         取没测的芯片
14.     else:
15.         任取 1 片被测芯片
16.  if k = 2 or 1:
17.     任取 1 片
```

## • 时间复杂度分析

- 设输入规模为  $n$
- 每轮淘汰后, 芯片数至少减半
- 测试次数 (含轮空处理) :  $O(n)$

$$W(n) = W(n/2) + O(n)$$
$$W(3) = 1, W(2) = W(1) = 0$$

$$W(n) = O(n)$$

主定理 case3

$$b=2, a=1, \epsilon=0.5$$

$T(n) = aT(n/b) + f(n)$ , 若  $f(n) = \Omega(n^{\log_b a + \epsilon})$ ,  $\epsilon > 0$ ,

且对于某个常数  $c < 1$  和充分大的  $n$  有  $af\left(\frac{n}{b}\right) \leq cf(n)$ ,

那么  $T(n) = \Theta(f(n))$

$c > 0.5$  即可

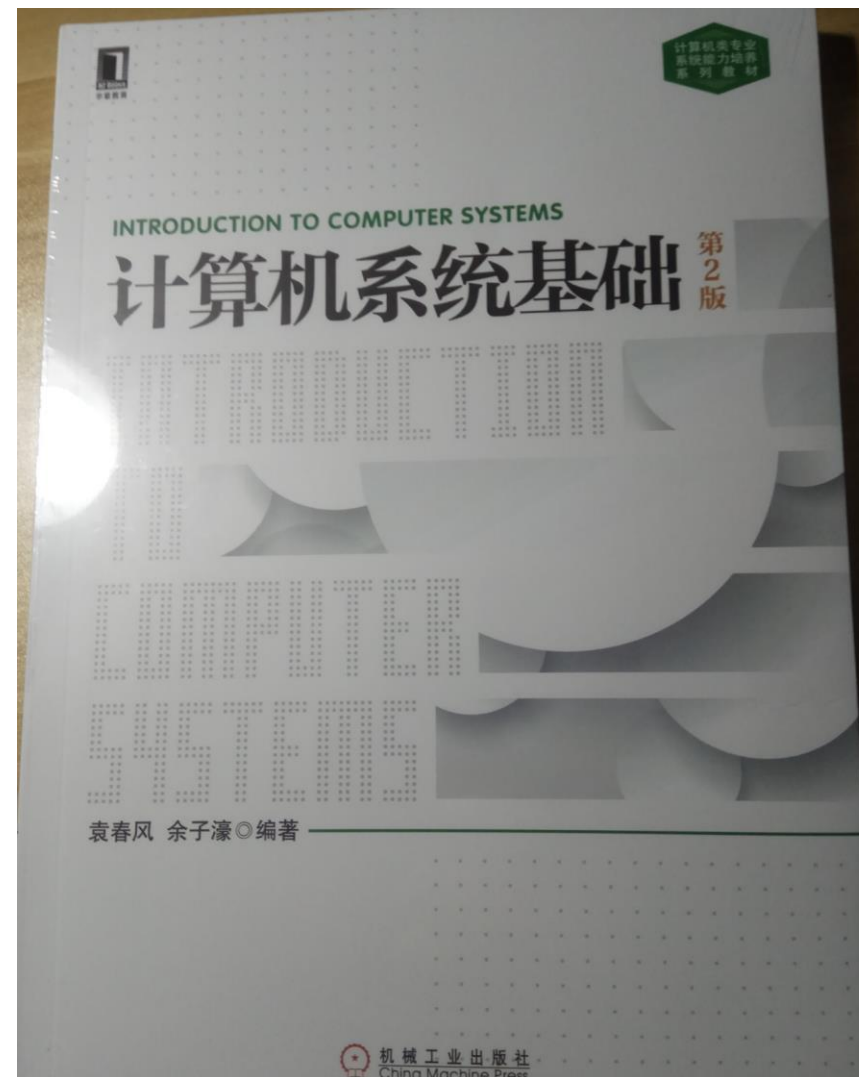
## 1. 程序的翻译

## 2. 程序的运行

# 参考内容



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



## 2.1 程序的编译



- 计算机程序是供计算机执行的编程语言的指令序列或指令集。它是软件的一个组件，还包括文档和其他无形组件。--维基百科
- 算法是所有程序的核心和灵魂，它一般被设计用于以最小的代价、高效的解决特定的问题。
- 算法 == 程序?
- 算法 + 数据结构 = 程序

# 2.1 程序编译



经典的 “hello.c” C-源程序

```
#include <stdio.h>

int main()
{
    printf("hello, world\n");
}
```

功能：输出 “hello,world”

计算机不能直接执行hello.c!

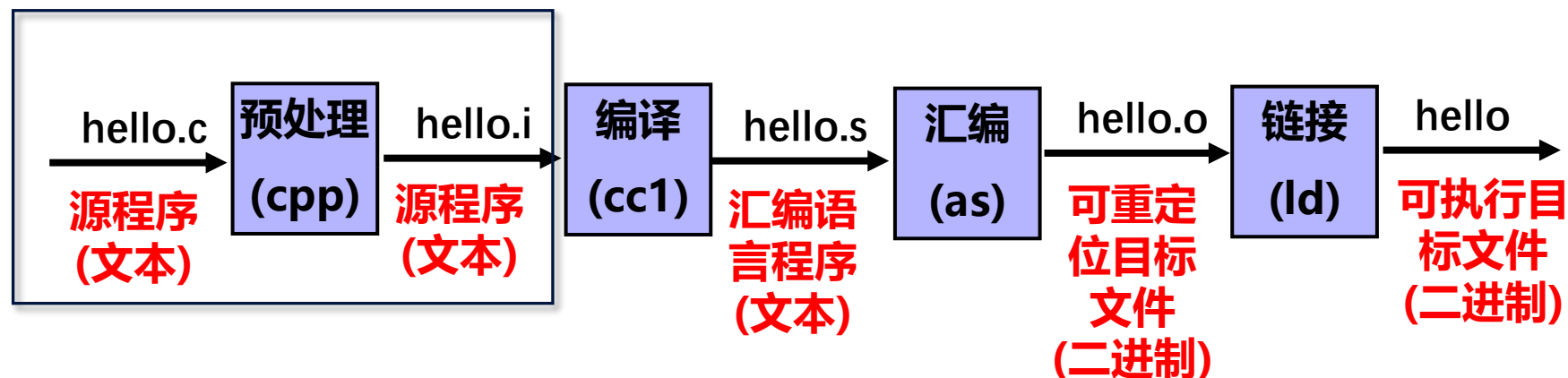
hello.c的ASCII文本表示

```
# i n c l u d e < s p > < s t d i o .
35 105 110 99 108 117 100 101 32 60 115 116 100 105 111 46
h > \n \n i n t < s p > m a i n ( ) \n {
104 62 10 10 105 110 116 32 109 97 105 110 40 41 10 123
\n < s p > < s p > < s p > < s p > p r i n t f ( " h e l
10 32 32 32 32 112 114 105 110 116 102 40 34 104 101 108
l o , < s p > w o r l d \ n " ) ; \n }
108 111 44 32 119 111 114 108 100 92 110 34 41 59 10 125
```

一个程序的执行需要哪些阶段？ 每个阶段做了些什么？



# 程序预处理



◆ 预处理是从源程序变成可执行程序的第一步，C预处理程序为cpp（即C Preprocessor）

- 包括对头文件的包含：例如，对于#include相应.h文件的内容插入到源文件中
- 宏定义的扩展：#define
- 条件编译：“#ifdef”
- 注释的删除

GCC 中的预处理命令是 “gcc -E” 或 “cpp” ，  
例如， “**gcc-E main.c -o main.i**” 或 “**cpp main.c -o main.i**”  
将 main.c转换为预处理后的文件 maini。预处理后的文件是可显示的文本文件。

# 程序预处理

## main.cpp

```
// 条件编译示例
#define DEBUG
#include "test.h"
#include <stdio.h> // 包含标准输入输出头文件
// 宏定义示例
#define SQUARE(x) (x * x)
int N=32;
int M;
int addfun(int a, int b){
    static int cnt=0;
    cnt++;
    printf("cnt value: %d\n", cnt);
    return 2*a+b;
}
```

```
int main() {
    int num = 5;
    // 使用条件编译
#ifdef DEBUG
    printf("Debug mode is enabled.\n");
#endif
    // 使用宏定义
    printf("Square of %d is %d\n", num, SQUARE(num));
    printf("sum of %d and %d is %d\n", num, N, addfun(num, N));
    printf("sum of %d and %d is %d\n", num, M, addfun(num, N));
    printf("sum2 of %d and %d is %d\n", num, N, fun(num, N));
    return 0;
}
```

## test.cpp

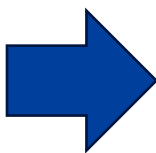
```
int add(int a, int b)
{
    int x = a+b;
    return x;
}
```

# 程序预处理

**g++ -E test.cpp -o test.i**

```
1 # 1 "test.cpp"
2 # 1 "<built-in>"
3 # 1 "<command-line>"
4 # 1 "/usr/include/stdc-predef.h" 1 3 4
5 # 1 "<command-line>" 2
6 # 1 "test.cpp"
7 int add(int a, int b)
8 {
9     int x = a+b;
10    return x;
11 }
```

**g++ -E main.cpp -o main.i**

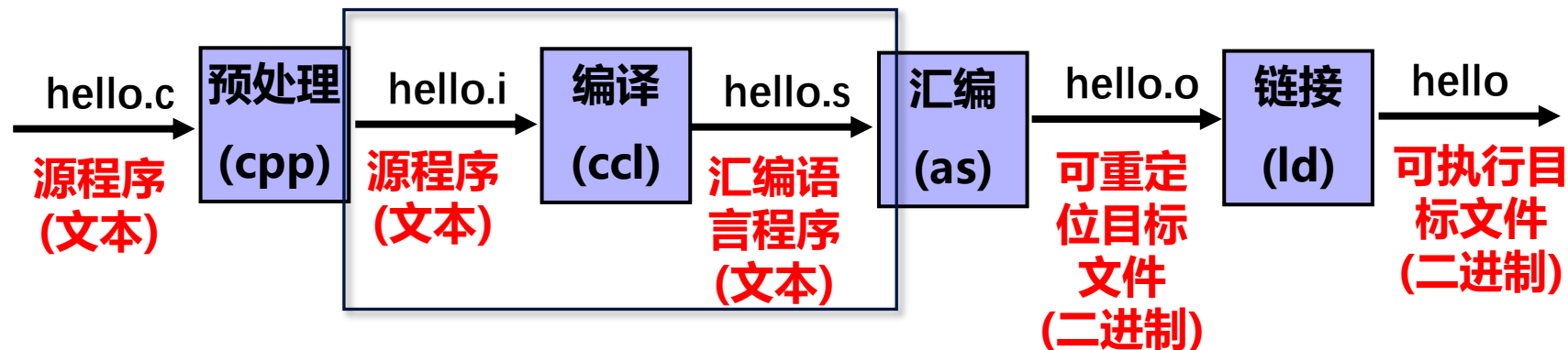


**8KB**

```
634 extern int fscanf (FILE *__restrict __stream,
635     const char *__restrict __format, ...) ;
636
637
638
639
640 extern int scanf (const char *__restrict __format, ...) ;
641
642 extern int sscanf (const char *__restrict __s,
643     const char *__restrict __format, ...) throw ();
644 # 420 "/usr/include/stdio.h" 3 4
645 extern int yfscanf (FILE *__restrict __s, const char *__restrict __format,
646     __gnuc_va_list __arg)
647     __attribute__ ((__format__ (__scanf__, 2, 0))) ;
```

```
915 # 10 "main.cpp"
916 int N=32;
917 int M;
918
919 int addfun(int a, int b){
920     static int cnt=0;
921     cnt++;
922     printf("cnt value: %d\n", cnt);
923     return 2*a+b;
924 }
925
926 int main() {
927     int num = 5;
928     printf("Debug mode is enabled.\n");
929     printf("Square of %d is %d\n", num, (num * num));
930     printf("sum of %d and %d is %d\n",num,N,addfun(num,N));
931     printf("sum of %d and %d is %d\n",num,M,addfun(num,N));
932     printf("sum2 of %d and %d is %d\n",num,N,fun(num,N));
933     return 0;
934 }
```

# 程序的编译



◆ C编译器在进行具体的程序翻译之前，会先对源程序进行**词法分析、语法分析和语义分析**，然后根据分析的结果进行代码优化和存储分配，最终把C语言源程序翻译成汇编程序。

- 编译器通常采用对源程序进行多次扫描的方式进行处理，每次扫描集中完成一项或几项任务，也可以将一项任务分散到几次扫描去完成。
- GCC 可以直接产生机器语言代码，也可以先产生汇编语言代码，然后再通过汇编程序将汇编语言代码转换为机器语言代码。

GCC 中的编译命令是 “gcc -S” 或 “ccl”

例如，可使用命令 “**gcc -S main.i -o main.s**” 或 “**ccl main.i -o main.s**” 对 main.i 进行编译并生成汇编代码文件 main.s，也可以使用命令 “**gcc -S main.cpp -o main.s**”

# 程序的编译

```
g++ -S main.i/ main.cpp -o main.s
```

```
int main() {  
    int num = 5;  
    // 使用条件编译  
#ifdef DEBUG  
    printf("Debug mode is enabled.\n");  
#endif  
    // 使用宏定义  
    printf("Square of %d is %d\n", num,  
    SQUARE(num));  
    printf("sum of %d and %d is  
%d\n", num, N, addfun(num, N));  
    printf("sum of %d and %d is  
%d\n", num, M, addfun(num, N));  
    printf("sum2 of %d and %d is  
%d\n", num, N, fun(num, N));  
    return 0;  
}
```

```
8  N:  
9      .long    32  
10     .globl   M  
11     .bss  
12     .align   4  
13     .type    M, @object  
14     .size    M, 4  
15  M:  
16     .zero    4  
17     .local   _ZZ6addfuniiE3cnt  
18     .comm    _ZZ6addfuniiE3cnt,4,4  
19     .section .rodata
```

```
20  .LC0:  
21      .string "cnt value: %d\n"  
22      .text  
23      .globl   _Z6addfunii  
24      .type    _Z6addfunii, @function
```

## AMD 64位 CPU

main: 标签，表示程序的入口点

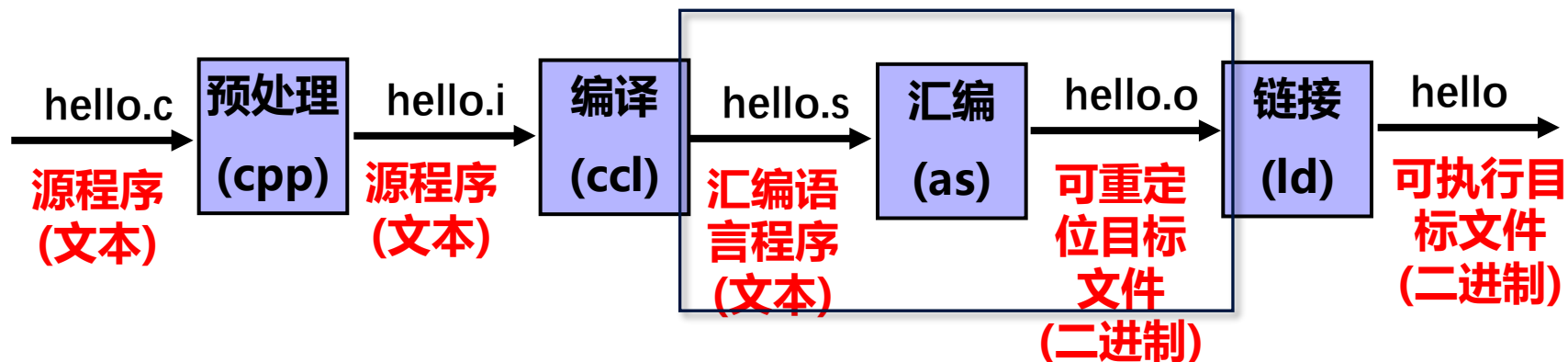
.LFB1:: 一个局部标签，通常用于表示 "Function Begin"。这是编译器生成的标签，用于调试和跟踪函数的起始位置。

pushq %rbp: 将当前的基址指针

movl \$5, -4(%rbp): 将数字 5 存储到相对于 %rbp 偏移 -4 字节的位置，可能是函数的局部变量

```
66  main:  
67  .LFB1:  
68      .cfi_startproc  
69      pushq   %rbp  
70      .cfi_def_cfa_offset 16  
71      .cfi_offset 6, -16  
72      movq    %rsp, %rbp  
73      .cfi_def_cfa_register 6  
74      subq    $16, %rsp  
75      movl    $5, -4(%rbp)  
76      leaq    .LC1(%rip), %rdi  
77      call    puts@PLT  
78      movl    -4(%rbp), %eax  
79      imull   -4(%rbp), %eax  
80      movl    %eax, %edx  
81      movl    -4(%rbp), %eax  
82      movl    %eax, %esi  
83      leaq    .LC2(%rip), %rdi  
84      movl    $0, %eax  
85      call    printf@PLT  
86      movl    N(%rip), %edx  
87      movl    -4(%rbp), %eax
```

# 程序的汇编



- ◆ 汇编器在汇编语言代码转换为**机器语言代码**。因为通常最终的可执行目标文件由多个**不同模块**对应的机器语言目标代码组合而形成，所以，在生成单个模块的机器语言目标代码时，不可能确定**每条指令或每个数据最终的地址**，也即，单个模块的机器语言目标代码需要重新定位。

GCC 中的编译命令是 “gcc -c” 或 “as” 如，**gcc -c main.s/main.cpp -o main.o**  
**as main.s -o main.o**

.o文件是什么，二进制形式无法直观阅读。

类似地似乎还有.obj, .o .dll, .so等

# 程序的汇编

## 可重定位目标文件格式 (.o)

### ELF 头

- ✓ 占16字节，包括字长、字节序（大端/小端）、文件类型 (.o, exec, .so)、机器类型（如 IA-32）、节头表的偏移、节头表的表项大小及表项个数

### .text 节

- ✓ 编译后的代码部分

### .rodata 节

- ✓ 只读数据，如 printf 格式串

### .data 节

- ✓ 已初始化的全局变量

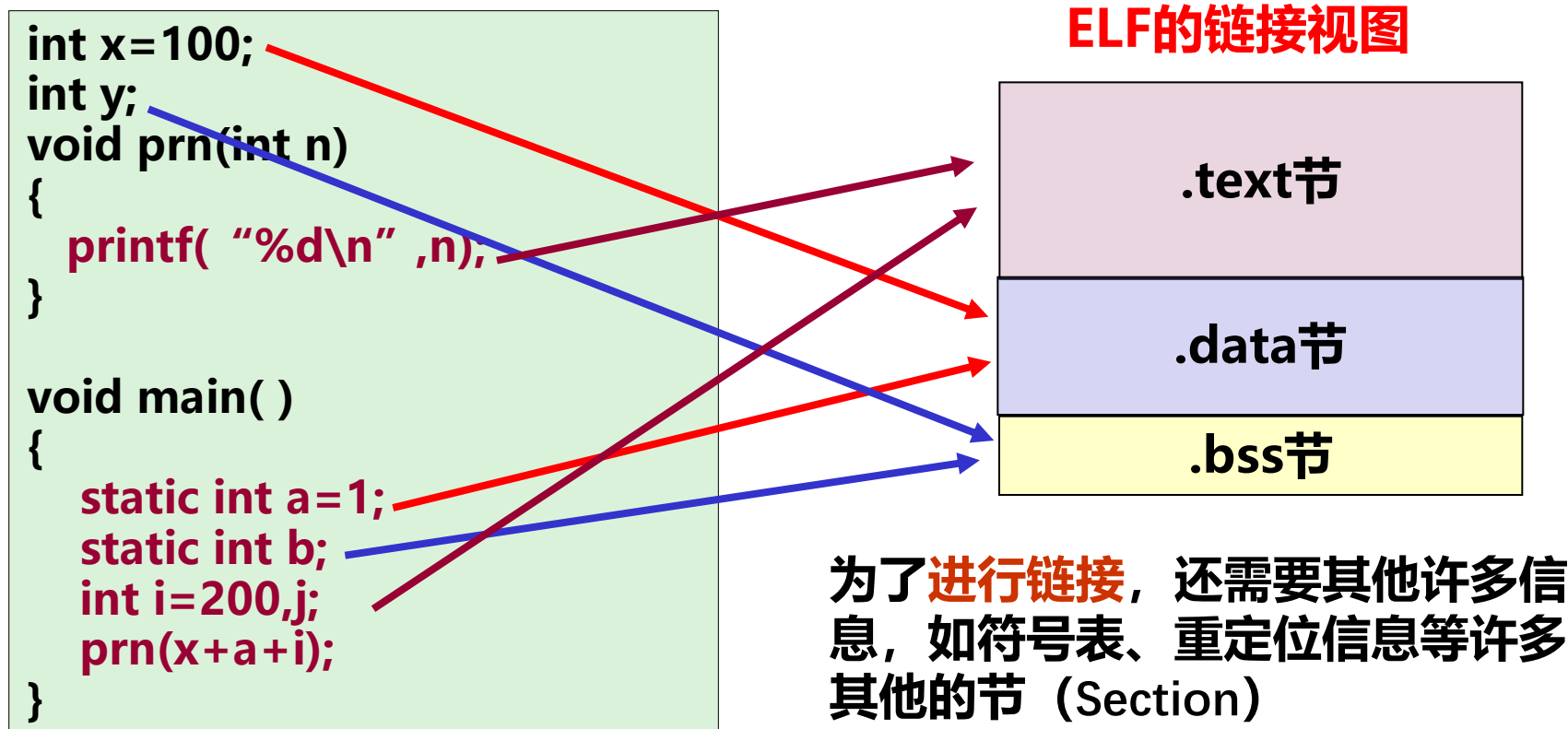
### .bss 节

- ✓ 未初始化全局变量，仅是占位符，不占据任何实际磁盘空间。区分初始化和非初始化是为了空间效率

ELF 头
.text 节
.rodata 节
.data 节
.bss 节
.symtab 节
.rel.txt 节
.rel.data 节
.debug 节
.strtab 节
.line 节
Section header table (节头表)

# 程序的汇编

- 可被链接（合并）生成可执行文件或**共享目标文件(.so)**
- **静态链接库文件**由若干个可重定位目标文件组成
- 包含代码、数据（已初始化.data和未初始化.bss）
- 包含**重定位信息**（指出哪些符号引用处需要重定位）





# 程序的汇编

## 可重定位目标文件

`gcc -c test.cpp -o test.o`

`Objdump -d test.o`

test.o: file format elf64-x86-64

Disassembly of section .text:

**0000000000000000 <\_Z3addii>:**

<b>0:</b>	55	push	%rbp
<b>1:</b>	48 89 e5	mov	%rsp,%rbp
<b>4:</b>	89 7d ec	mov	%edi,-0x14(%rbp)
<b>7:</b>	89 75 e8	mov	%esi,-0x18(%rbp)
<b>a:</b>	8b 55 ec	mov	-0x14(%rbp),%edx
<b>d:</b>	8b 45 e8	mov	-0x18(%rbp),%eax
<b>10:</b>	01 d0	add	%edx,%eax
<b>12:</b>	89 45 fc	mov	%eax,-0x4(%rbp)
<b>15:</b>	8b 45 fc	mov	-0x4(%rbp),%eax
<b>18:</b>	5d	pop	%rbp
<b>19:</b>	c3	retq	

# 程序的汇编

## 三类目标文件

Linux下不区分后缀  
命名只是习惯。  
文件类型可以通过  
file命令查看

- **可重定位**目标文件 (.o), window中的obj
  - 其代码和数据可和其他可重定位文件合并为可执行文件
    - 每个.o 文件由对应的.c文件生成
    - 每个.o文件代码和数据**地址都从0开始**
- **可执行**目标文件 (默认为a.out), window中.exe
  - 包含的代码和数据可以被直接复制到内存并被执行
  - 代码和数据**地址为虚拟地址**空间中的地址
- **共享**的目标文件 (.so)
  - 特殊的可重定位目标文件, 能在装入或运行时被装入到内存并自动被链接, 称为**共享库文件**
  - Windows 中称其为 *Dynamic Link Libraries* (DLLs)

# 符号的定义

## 什么是符号？

每个可重定位目标模块m都有一个符号表，它包含了在m中定义的符号。有三种链接器符号：

- **Global symbols** (模块内部定义的全局符号)
    - 由模块m定义并能被其他模块引用的符号。例如，非static 函数和非static的全局变量（指不带static的全局变量）  
如，main.c 中的全局变量名buf
  - **External symbols** (外部定义的全局符号)
    - 由其他模块定义并被模块m引用的全局符号  
如，main.c 中的函数名swap
  - **Local symbols** (本模块的局部符号)
    - 仅由模块m定义和引用的本地符号。例如，在模块m中定义的带static的函数/全局变量  
如，swap.c 中的static变量名bufp1
- 链接器局部符号不是指程序中的局部变量（分配在栈中的临时性变量），链接器不关心这种局部变量

# 符号表数据结构

**.symtab** 节记录符号表信息，是一个结构数组

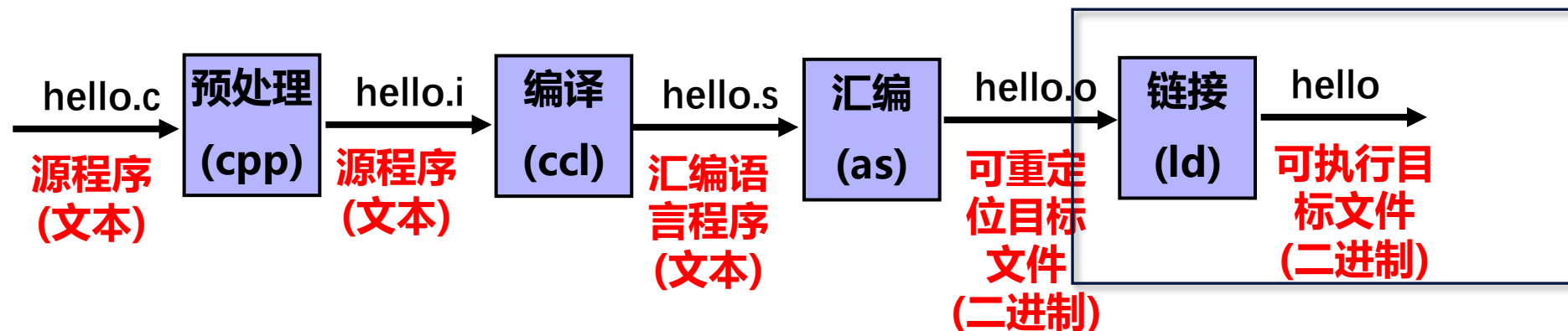
- 符号表 (.symtab) 中每个条目的结构如下：

```
typedef struct {  
    int  name; /*符号对应字符串在strtab节中的偏移量*/  
    int  value; /*在对应节中的偏移量，可执行文件中是虚拟地址*/  
    int  size; /*符号对应目标所占字节数*/  
    char type: 4, /*符号对应目标的类型：数据、函数、源文件、节*/  
        binding: 4; /*符号类别：全局符号、局部符号、弱符号*/  
    char reserved;  
    char section; /*符号对应目标所在的节，或其他情况*/  
} Elf_Symbol;
```

函数名在text节中  
变量名在data节或  
bss节中

函数大小或变量长度

# 程序的链接



将所有关联的可重构文件组合起来以生成可执行文件

- 使用GCC编译器编译并链接生成可执行程序run:
  - `$ gcc -O2 -g -o run main.c test.c`
  - `$ ./run`

- `$ gcc -o run main.c test.o`

-O2: 2级优化 (?) 默认是?

-g: 生成调试信息

-o: 目标文件名

# 程序的链接

## 链接操作的步骤

- 1) 确定标号引用关系（符号解析）
- 2) 合并相关.o文件
- 3) 确定每个标号的地址
- 4) 在指令中填入新地址

链接：符号解析+重定位 -> 代码合体

main.c

```
int buf[2] = {1, 2};  
void swap();  
  
int main()  
{  
    swap();  
    return 0;  
}
```

swap.c

```
extern int buf[];  
int *bufp0 = &buf[0];  
static int *bufp1;  
void swap()  
{  
    int temp;  
    bufp1 = &buf[1];  
    temp = *bufp0;  
    *bufp0 = *bufp1;  
    *bufp1 = temp;  
}
```

你能说出哪些是**符号定义**？哪些是**符号的引用**？

# 程序的链接

链接就是代码合体

ELF 头
<b>程序（段）头表</b>
.init 节
.text 节
.rodata 节
.data 节
.bss 节
.symtab 节
.debug 节
.line 节
.strtab 节

## 可重定位目标文件

系统代码
系统数据

main.o

main()
int buf[2]={1,2}

swap.o

swap()
int *bufp0=&buf[0]
static int *bufp1

## 可执行目标文件

0	Headers	
	系统代码	
	main()	
	swap()	
	更多系统代码	
	系统数据	
	int buf[2]={1,2}	
	int *bufp0=&buf[0]	
	int *bufp1	
	.symtab	
	.debug	

.text

.data

.bss

.text

.data

.text

.data

.text

.data

.bss

# 程序的链接

## test.o

test.o: file format elf64-x86-64

Disassembly of section .text:

**0000000000000000 <\_Z3addii>:**

<b>0:</b>	55	push %rbp
<b>1:</b>	48 89 e5	mov %rsp,%rbp
<b>4:</b>	89 7d ec	mov %edi,-0x14(%rbp)
<b>7:</b>	89 75 e8	mov %esi,-0x18(%rbp)
<b>a:</b>	8b 55 ec	mov -0x14(%rbp),%edx
<b>d:</b>	8b 45 e8	mov -0x18(%rbp),%eax
<b>10:</b>	01 d0	add %edx,%eax
<b>12:</b>	89 45 fc	mov %eax,-0x4(%rbp)
<b>15:</b>	8b 45 fc	mov -0x4(%rbp),%eax
<b>18:</b>	5d	pop %rbp
<b>19:</b>	c3	retq

## main.out中的片段

**000000000000079e <\_Z3addii>:**

<b>79e:</b>	55	push %rbp
<b>79f:</b>	48 89 e5	mov %rsp,%rbp
<b>7a2:</b>	89 7d ec	mov %edi,-0x14(%rbp)
<b>7a5:</b>	89 75 e8	mov %esi,-0x18(%rbp)
<b>7a8:</b>	8b 55 ec	mov -0x14(%rbp),%edx
<b>7ab:</b>	8b 45 e8	mov -0x18(%rbp),%eax
<b>7ae:</b>	01 d0	add %edx,%eax
<b>7b0:</b>	89 45 fc	mov %eax,-0x4(%rbp)
<b>7b3:</b>	8b 45 fc	mov -0x4(%rbp),%eax
<b>7b6:</b>	5d	pop %rbp
<b>7b7:</b>	c3	retq
<b>7b8:</b>	0f 1f 84 00 00 00 00	nopl 0x0(%rax,%rax,1)
<b>7bf:</b>	00	



# 程序的链接（项目编译常报错阶段）

- Step 1. 符号解析（Symbol resolution）

- 程序中有定义和引用的符号（包括变量和函数等）

- void swap() {...} /\* 定义符号swap \*/
- swap(); /\* 引用符号swap \*/
- int \*xp = &x; /\* 定义符号 xp, 引用符号 x \*/

- 编译器将**定义的符号**存放在一个**符号表**（symbol table）中。

- 符号表是一个结构数组
- 每个表项包含符号名、**长度和位置**等信息

- 链接器将每个**符号的引用**都与一个确定的**符号定义**建立关联

- Step 2. 重定位

- 将多个代码段与数据段分别**合并为**一个单独的代码段和数据段
- 计算每个定义的符号在虚拟地址空间中的**绝对地址**
- 将可执行文件中符号引用处的地址**修改为重定位后的地址信息**

add B  
jmp L0

.....

.....

.....

**L0:** sub C

.....

# 符号解析

- 目的：将每个模块中**引用的符号**与某个目标模块中的**定义符号**建立关联。
- 每个**定义符号**在代码段或数据段中都被分配了存储空间，将**引用符号**与**定义符号**建立关联后，就可在重定位时将**引用符号的地址重定位为相关联的定义符号的地址**。
- **本地符号**在本模块内定义并引用，因此，其解析较简单，只要与本模块内唯一的定义符号关联即可。
- **全局符号**（外部定义的、内部定义的）的解析涉及多个模块，故较复杂

“符号的定义” 其实质是什么？

指被分配了存储空间。为函数名即指其代码所在区；为变量名即指其所占的静态数据区。

所有定义符号的值就是其目标所在的首地址

```
add B
jmp L0
.....
L0: sub 23
.....
B: .....
```

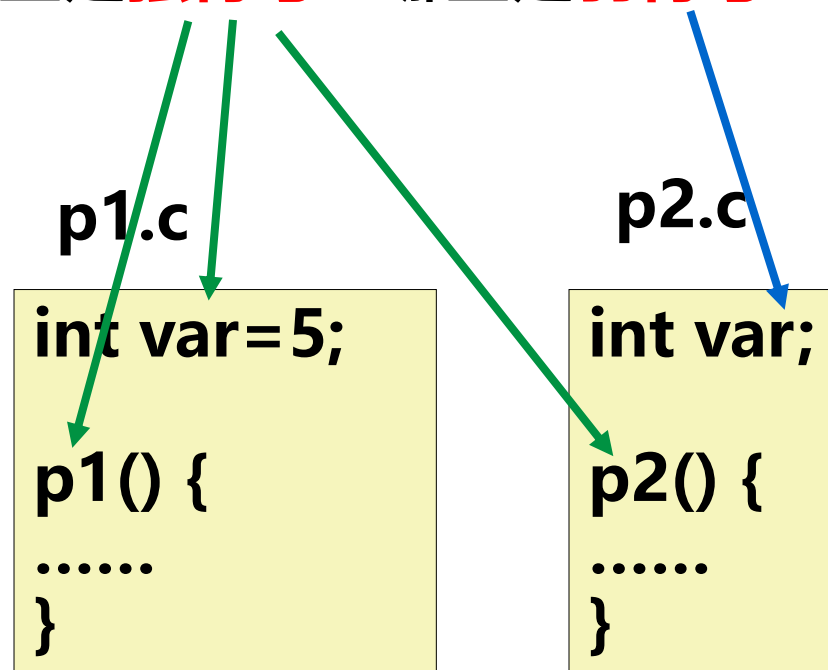
确定L0的地址，  
再在jmp指令中  
填入L0的地址

符号解析也称**符号绑定**

# 全局符号的符号解析

- 全局符号的强/弱特性
  - 函数名和已初始化的全局变量名是**强符号**
  - 未初始化的全局变量名是**弱符号**

以下符号哪些是**强符号**？ 哪些是**弱符号**？



# 全局符号的符号解析

以下符号哪些是**强符号**？ 哪些是**弱符号**？

main.c

```
int buf[2] = {1, 2};  
void swap();  
  
int main()  
{  
    swap();  
    return 0;  
}
```

此处为引用

swap.c

```
extern int buf[];  
  
int *bufp0 = &buf[0];  
static int *bufp1;  
  
void swap()  
{  
    int temp;  
  
    bufp1 = &buf[1];  
    temp = *bufp0;  
    *bufp0 = *bufp1;  
    *bufp1 = temp;  
}
```

局部变量

本地局部符号

# 链接器对符号的解析规则

符号解析时只能有一个确定的定义（即每个符号仅占一处存储空间）

- 多重定义符号的处理规则

Rule 1: 强符号不能多次定义

- 强符号只能被定义一次，否则链接错误

Rule 2: 若一个符号被定义为一次强符号和多次弱符号，则按强定义为准

- 对弱符号的引用被解析为其强定义符号

Rule 3: 若有多个弱符号定义，则任选其中一个

- 使用命令 `gcc -fno-common` 链接时，会告诉链接器在遇到多个弱定义的全局符号时输出一条警告信息。

# 多重定义符号的解析举例

以下程序会发生链接出错吗？

```
int x=10;  
int p1(void);  
int main()  
{  
    x=p1();  
    return x;  
}
```

main.c

```
int x=20;  
int p1()  
{  
    return x;  
}
```

p1.c

main只有一次强定义

p1有一次强定义，一次弱定义

x有两次强定义，所以，链接器将输出一条出错信息

# 多重定义符号的解析举例

以下程序会发生链接出错吗？

```
# include <stdio.h>
int y=100;
int z;
void p1(void);
int main()
{
    z=1000;
    p1();
    printf( "y=%d, z=%d\n" , y, z);
    return 0;
}
```

main.c

y一次强定义，一次弱定义  
z两次弱定义  
p1一次强定义，一次弱定义  
main一次强定义

```
int y;
int z;
void p1( )
{
    y=200;
    z=2000;
}
```

p1.c

问题：打印结果是什么？

y=200, z=2000

该例说明：在两个不同模块定义相同变量名，很可能发生意想不到的结果！

# 为什么要链接？

## 链接带来的好处1：模块化

- (1) 一个程序可以分成很多源程序文件
- (2) 可构建公共函数库，如数学库，标准C库等

## 链接带来的好处2：效率高

- (1) 时间上，可分开编译

只需重新编译被修改的源程序文件，然后重新链接

- (2) 空间上，无需包含共享库所有代码

源文件中无需包含共享库函数的源码，只要直接调用即可  
可执行文件和运行时的内存中只需包含所调用函数的代码  
而不需要包含整个共享库



# 为什么要链接？

1. 大家共同开发项目，通过.h沟通接口。
2. printf, cout直接使用，不关心实现，链接到libstdc++.so.6。  
STL标准库。CUBLAS(.so)。
3. CUTLASS模板库(优点，仅编译需要代码)，深度学习算子。

```
1 // example.cpp
2 #include <iostream>
3
4 void helloWorld() {
5     std::cout << "Hello, World!" << std::endl;
6 }
7
```

g++ -shared -fPIC -o libexample.so example.cpp

```
// main.cpp
void helloWorld();

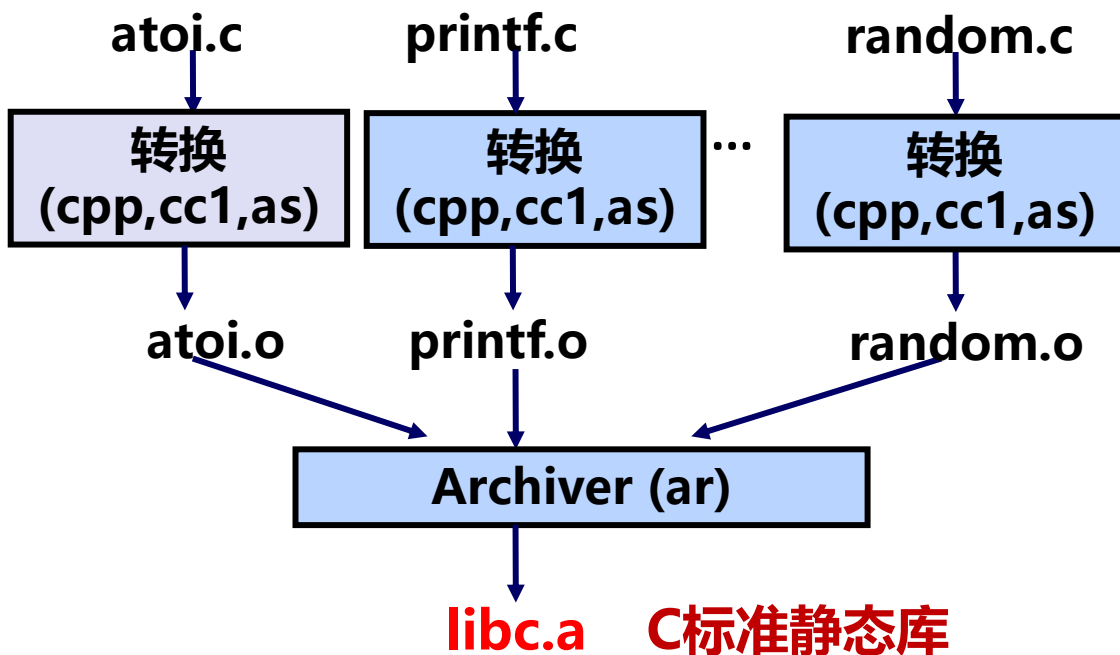
int main() {
    helloWorld();
    return 0;
}
```

g++ -o main main.cpp -L/path/to/library -lexample

一个编译动态链接库的例子。动态库？静态库？不编到库，多个.o？都需要链接！！！！

# 静态库

- **静态库** (.a archive files)
  - 增强了链接器功能，使其能通过查找一个或多个库文件中的符号来解析符号
  - 在构建可执行文件时只需指定库文件名，链接器会自动到库中寻找那些应用程序用到的目标模块，并且**只把用到的模块从库中拷贝出来**
  - 在gcc命令行中无需明显指定C标准库libc.a(默认库)



```
$ ar rcs libc.a \
  atoi.o printf.o ... random.o
```

# 创建一个静态库

举例：将myproc1.o和myproc2.o打包生成mylib.a

## myproc1.c

```
# include <stdio.h>
void myfunc1() {
    printf("This is myfunc1!\n");
}
```

## myproc2.c

```
# include <stdio.h>
void myfunc2() {
    printf("This is myfunc2!\n");
}
```

**\$ gcc -c myproc1.c myproc2.c**

**\$ ar rcs mylib.a myproc1.o myproc2.o**

## main.c

```
void myfunc1(viod);
int main()
{
    myfunc1();
    return 0;
}
```

**\$ gcc -c main.c**      **libc.a**无需明显指出!

**\$ gcc -static -o myproc main.o ./mylib.a**

**调用关系: main→myfunc1→printf**

# 静态库

- 静态库有一些缺点：
  - 库函数（如printf）被包含在每个运行进程的代码段中，对于并发运行上百个进程的系统，造成极大的主存资源浪费
  - 库函数（如printf）被合并到可执行目标中，磁盘上存放着数千个可执行文件，造成磁盘空间的极大浪费
  - 程序员需关注是否有函数库的新版本出现，并须定期下载、重新编译和链接，更新困难、使用不便
- 解决方案: Shared Libraries（共享库）
  - 是一个目标文件，包含有代码和数据
  - 从程序中分离出来，磁盘和内存中都只有一个备份
  - 可以动态地在装入时或运行时被加载并链接
  - Window称其为动态链接库（Dynamic Link Libraries, .dll文件）
  - Linux称其为动态共享对象（Dynamic Shared Objects, .so文件）

# 动态链接库

动态链接可以按以下两种方式进行：

- 在第一次加载并运行时进行 (load-time linking).
  - Linux通常由动态链接器(ld-linux.so)自动处理
  - 标准C库 (libc.so) 通常按这种方式动态被链接
- 在已经开始运行后进行(run-time linking).
  - 在Linux中，通过调用 dlopen()等接口来实现
    - 分发软件包、构建高性能Web服务器等

在内存中只有一个备份，被所有进程共享（调用），节省内存空间

一个共享库目标文件被所有程序共享链接，节省磁盘空间

共享库升级时，被自动加载到内存和程序动态链接，使用方便

共享库可分模块、独立、用不同编程语言进行开发，效率高

第三方开发的共享库可作为程序插件，使程序功能易于扩展

# 创建动态库

## myproc1.c

```
# include <stdio.h>
void myfunc1()
{
    printf("%s","This is myfunc1!\n");
}
```

## PIC: Position Independent Code

### 位置无关代码

- 1) 保证共享库代码的位置可以是不确定的
- 2) 即使共享库代码的长度发生变化, 也不会影响调用它的程序

## myproc2.c

```
# include <stdio.h>
void myfunc2()
{
    printf("%s","This is myfunc2\n");
}
```

gcc -c myproc1.c myproc2.c 位置无关的共享代码库文件  
gcc -shared -fPIC -o mylib.so myproc1.o myproc2.o

# 位置无关代码

- 动态链接用到一个重要概念：
  - 位置无关代码（Position-Independent Code, PIC
  - GCC选项-fPIC指示生成PIC代码
- 共享库代码是一种PIC
  - 共享库代码的位置可以是不确定的
  - 即使共享库代码的长度发生变化，也不影响调用它的程序
- 引入PIC的目的
  - 链接器无需修改代码即可将共享库加载到任意地址运行
- 所有引用情况
  - (1) 模块内的过程调用、跳转，采用PC相对偏移寻址
  - (2) 模块内数据访问，如模块内的全局变量和静态变量
  - (3) 模块外的过程调用、跳转
  - (4) 模块外的数据访问，如外部变量的访问

# 加载时动态链接

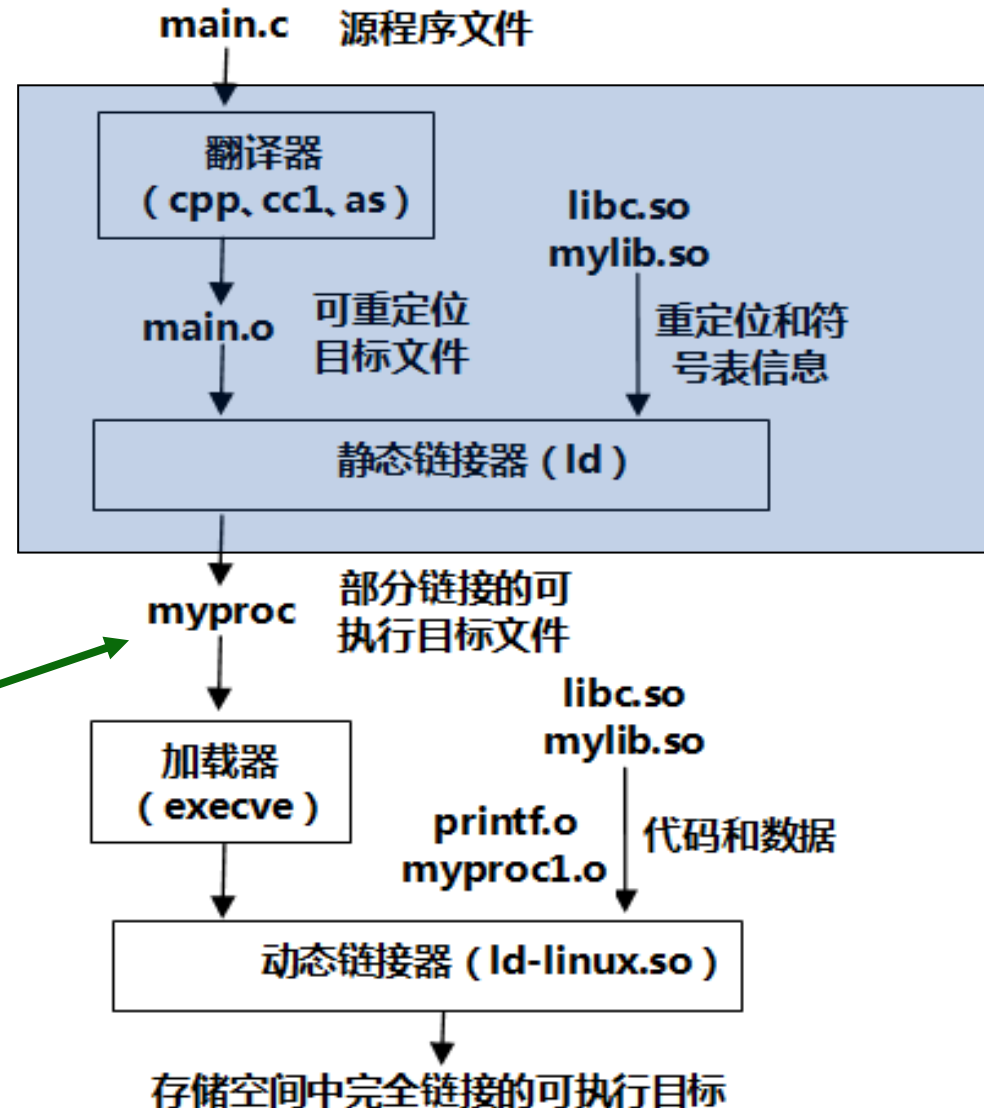
```
gcc -c main.c
```

```
gcc -o myproc main.o ./mylib.so
```

调用关系: `main`→`myfunc1`→`printf`

```
void myfunc1(viod);  
int main() {  
    myfunc1();  
    return 0;  
}
```

加载 `myproc` 时, 加载器发现在其程序头表中有 `.interp` 段, 其中包含了动态链接器路径名 `ld-linux.so`, 因而加载器根据指定路径加载并启动动态链接器运行。动态链接器完成相应的重定位工作后, 再把控制权交给 `myproc`, 启动其第一条指令执行。



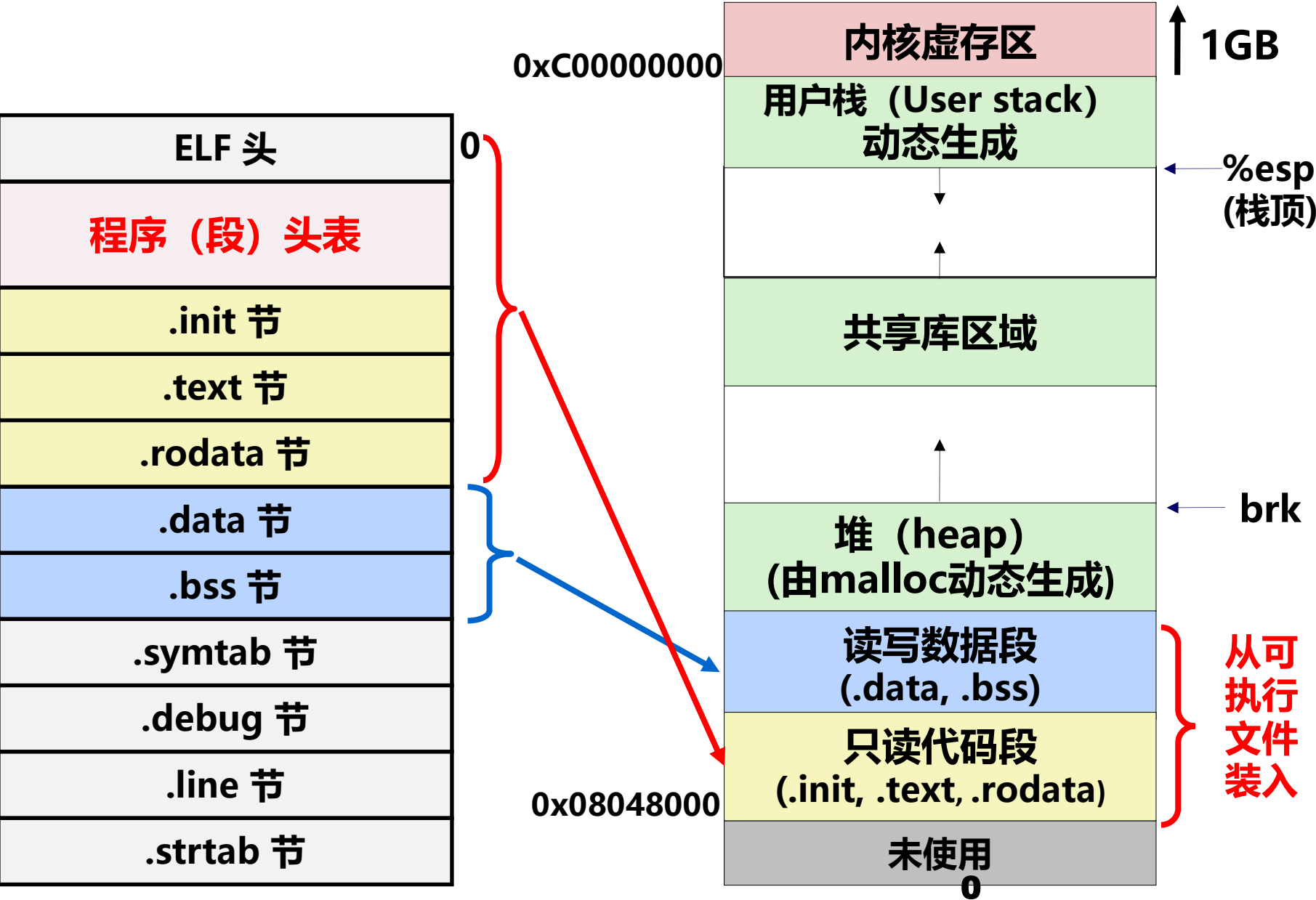


# 程序的编译和运行

- 运行PE文件后，发生了什么？(从OS角度去看)
  - Step1: 创建一个进程。
    - 进程的最关键特征：拥有独立的虚拟地址空间，即拥有一个虚拟空间VM到物理内存的映射关系
  - Step2: 装载相应的PE文件并执行
    - 读取可执行文件头，并且建立虚拟空间VM与可执行文件的映射关系
    - 将CPU的指令寄存器设置成可执行文件的入口地址，启动运行
- 从操作系统OS的角度去看： PE文件的装载和执行
- 程序vs.进程： 菜谱 vs. 炒菜的过程
- 虚拟内存和内存
  - 如何将计算机上有限的物理内存分配给多个程序使用？

利用 虚拟内存

# 可执行文件到虚拟地址空间的映射

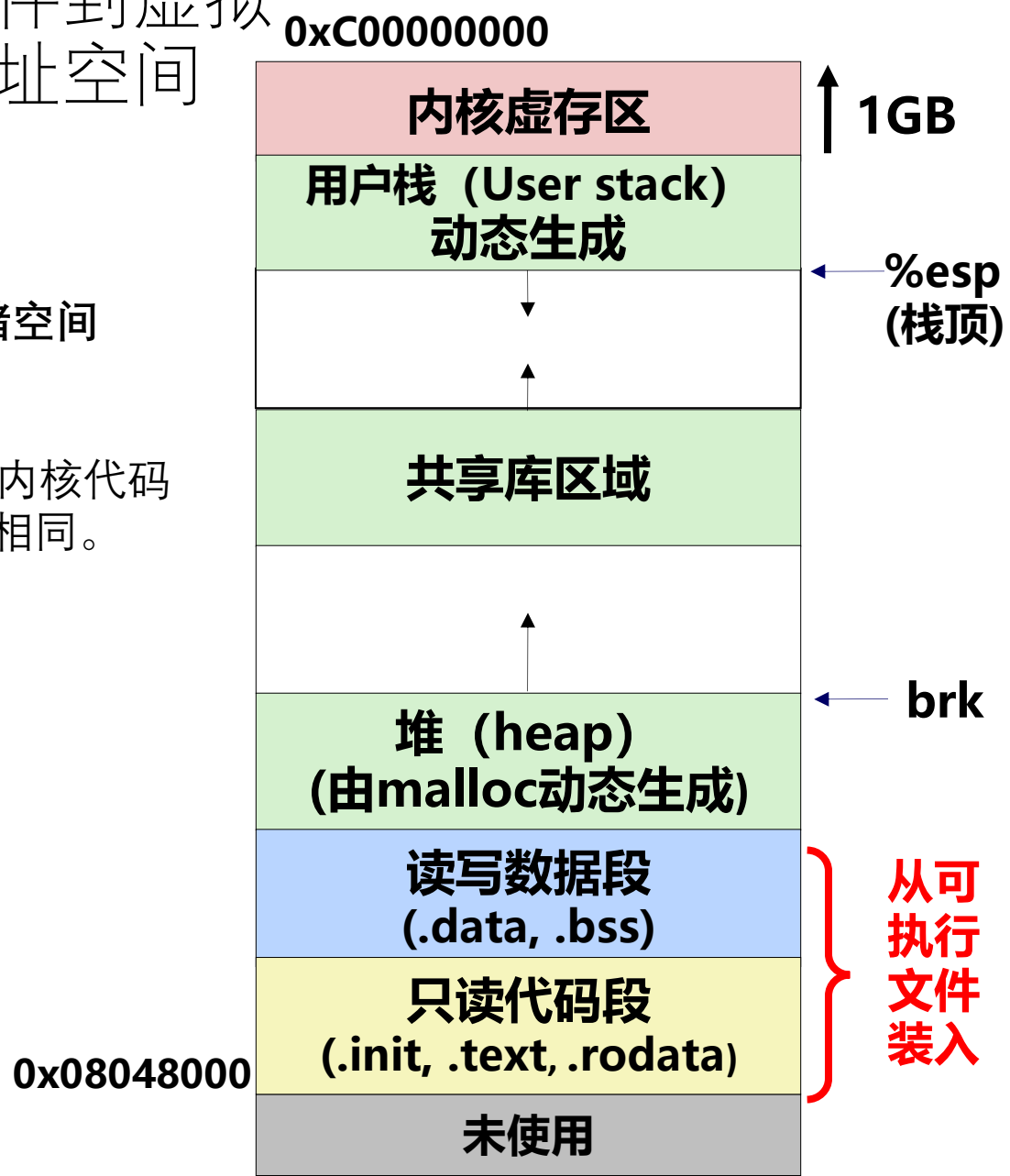


# 虚拟地址空间，前面已经看了可执行文件到虚拟地址空间的映射，我们详细看看虚拟地址空间（理论）

虚拟地址空间分为两大部分: **内核虚拟存储空间**和**用户虚拟存储空间**

分别简称为**内核空间(kernel space)**和**用户空间(user space)**.  
内核空间在0xc0000000以上的高端地址上，用来存放操作系统内核代码和数据等，其中内核代码和数据区在每个进程的地址空间中都相同。  
用户程序没有权限访问内核区。

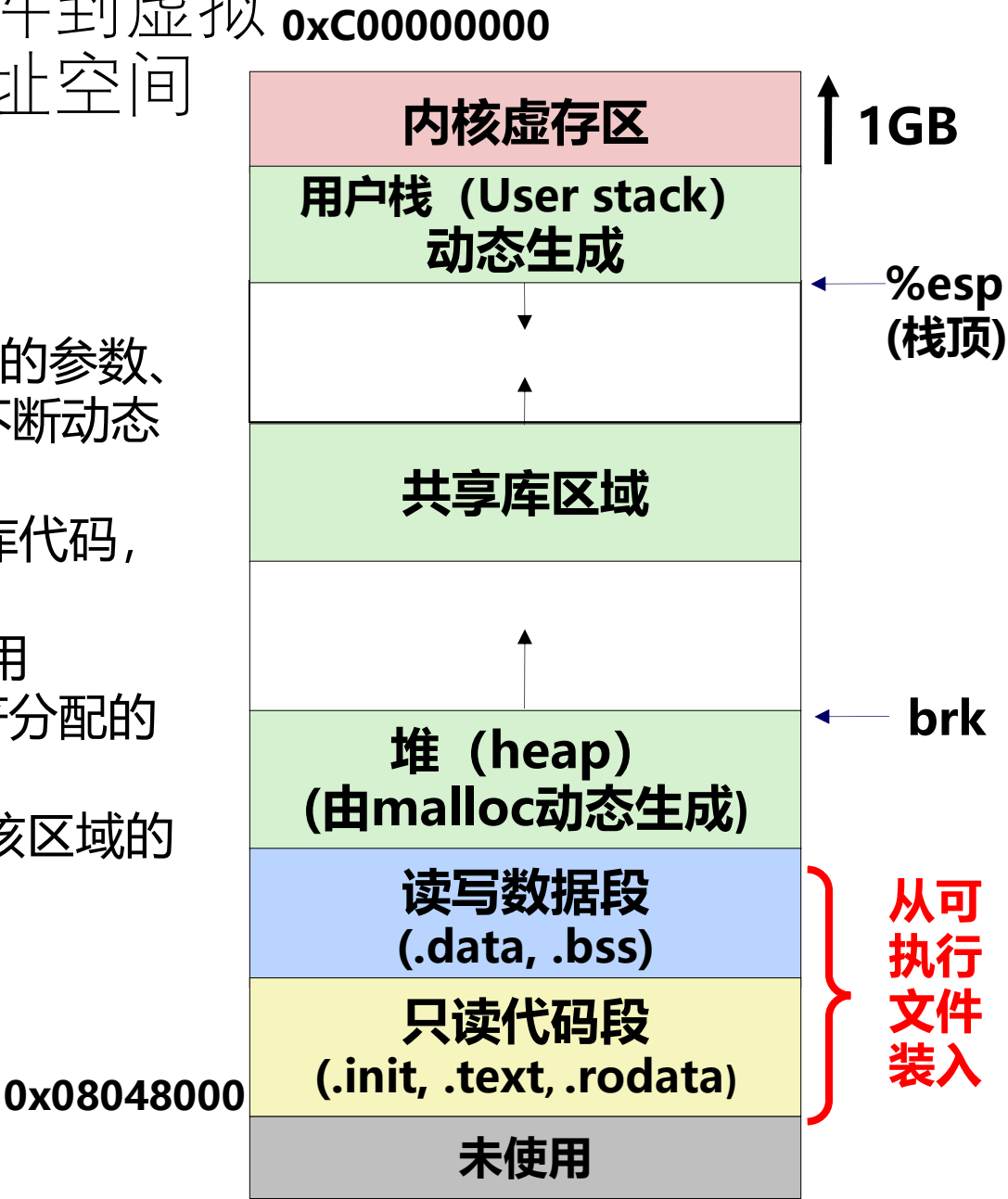
用户空间用来存放进程的代码和数据等，分为以下几个区域：



虚拟地址空间，前面已经看了可执行文件到虚拟地址空间的映射，我们详细看看虚拟地址空间（理论）

- ① 用户栈（userstack）。用来存放程序运行时过程调用的参数、返回地址、过程局部变量等，随着程序的执行，该区会不断动态地从高地址向低地址增长或向反方向减退。
- ② 共享库（shared library）。用来存 公共的共享函数库代码，如hello中的 printf（） 函数等。
- ③ 堆（heap）。用于动态申请存储区，例如，C语言中用 malloe（） 函数分配的存储区，或C++中用 new 操作符分配的存储区。
- ④ 可读写数据区。存放进程中的静态全局变量，堆区从该区域的结尾处开始向高地址增长。
- ⑤ 是读数握和代码区。存放进程中的代码和只读数据，如hello 进程中的程序代码租字符串 “hello, world”。

每个区域都有相应的起始位置

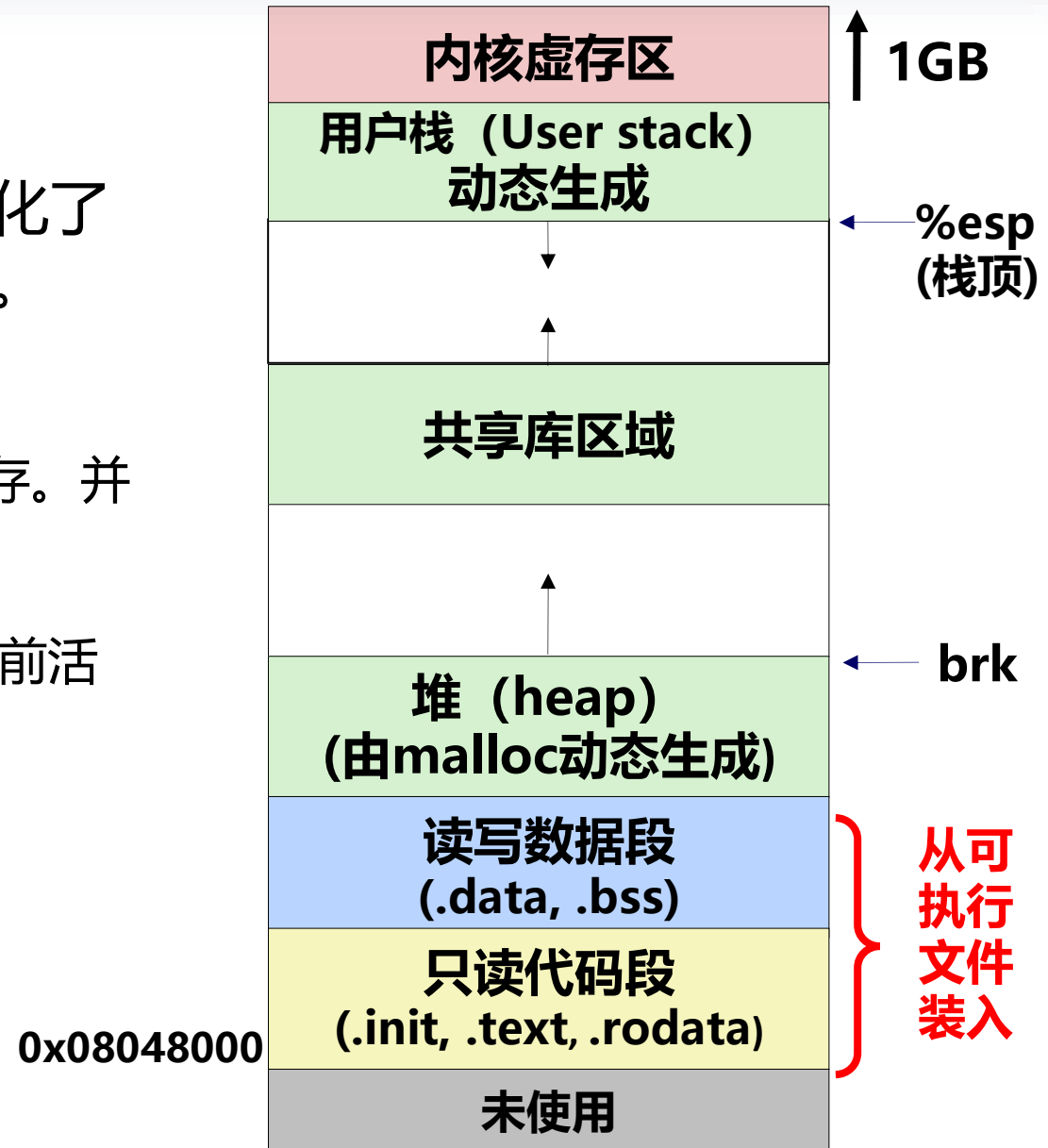


# 虚拟地址空间的好处

所有进程的虚拟地址空间大小和结构一致，这简化了链接器的设计和实现，也简化了程序的加载过程。  
虚拟地址空间是内存和硬盘存储器的抽象。

虚存机制带来了一个假象，使得每个程序都在独立使用主存。并且空间极大，这在三个好处：

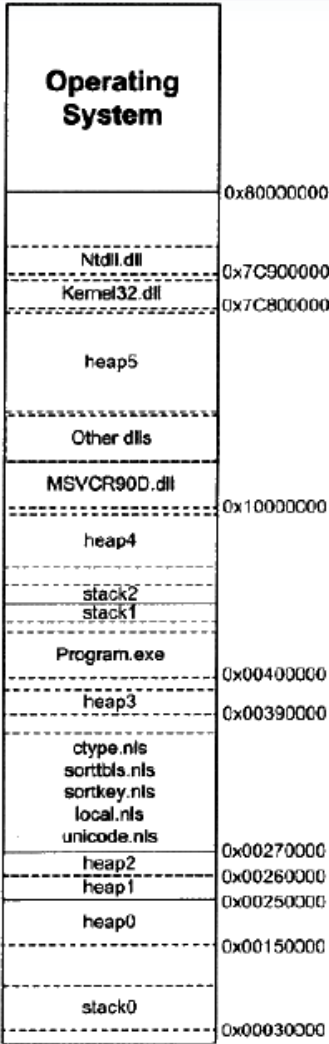
- ① 每个进程具有一致的虚拟地址空间，简化存储管理；
- ② 主存看成是硬盘在储器的一个缓存，在主存中仅保存当前活动的程序段和数据
- ③ 每个进程的虚拟地址空间是私有、独立的。



# 虚拟地址空间（实际 ASLR内存随机化保护）

User-mode ASLR is usually what is meant by the term ASLR. It being enabled implies this protection to be available on the user space mapping of every process. Effectively, ASLR being enabled implies that the absolute memory map of user-mode processes will vary every time they're run.

Tunable value	Interpretation of this value in <code>/proc/sys/kernel/randomize_va_space</code>
0	(User mode) ASLR turned OFF; or can be turned off by passing the kernel parameter <code>noexec</code> at boot.
1	(User mode) ASLR is ON: <code>mmap (2)</code> based allocations, the stack, and the vDSO page is randomized. It also implies that shared library load locations and shared memory segments are randomized.
2	(User mode) ASLR is ON: all of the preceding (value 1) <i>plus</i> the heap location is randomized (since 2.6.25); this is the OS value by default.



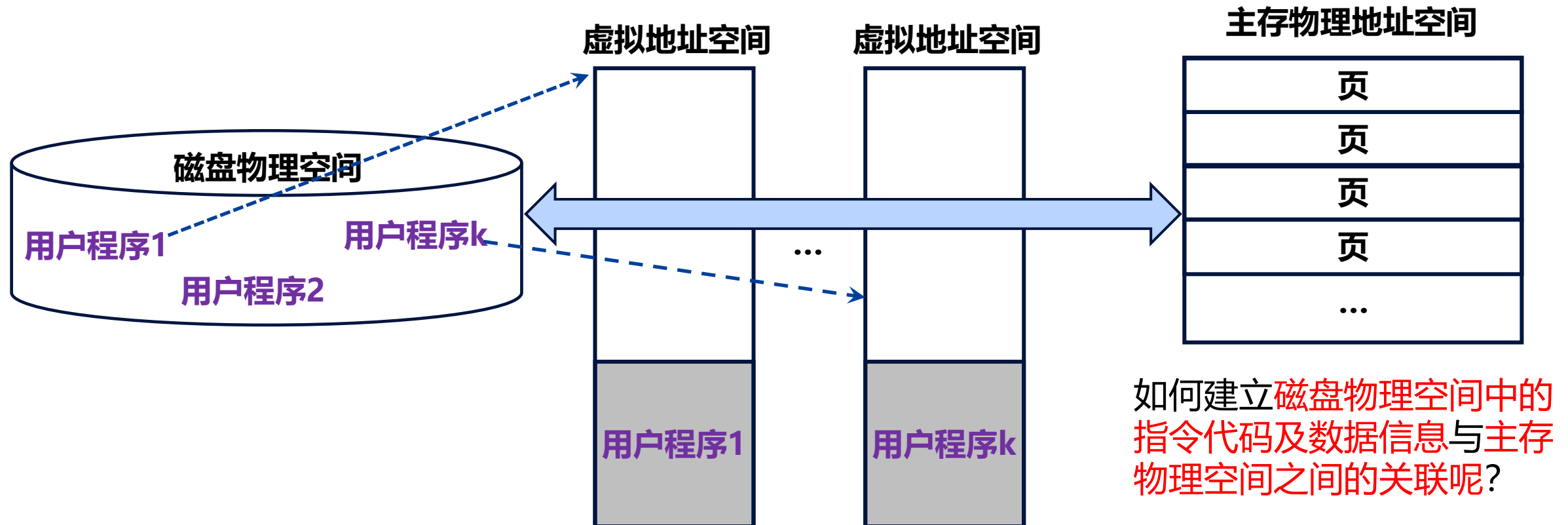
实际使用ASLR技术

Windows Process  
Virtual Space

# 虚拟地址空间到物理内存的映射

每个用户程序都有各自**独立的虚拟地址空间**，用户程序以**可执行文件**为式存在磁盘上

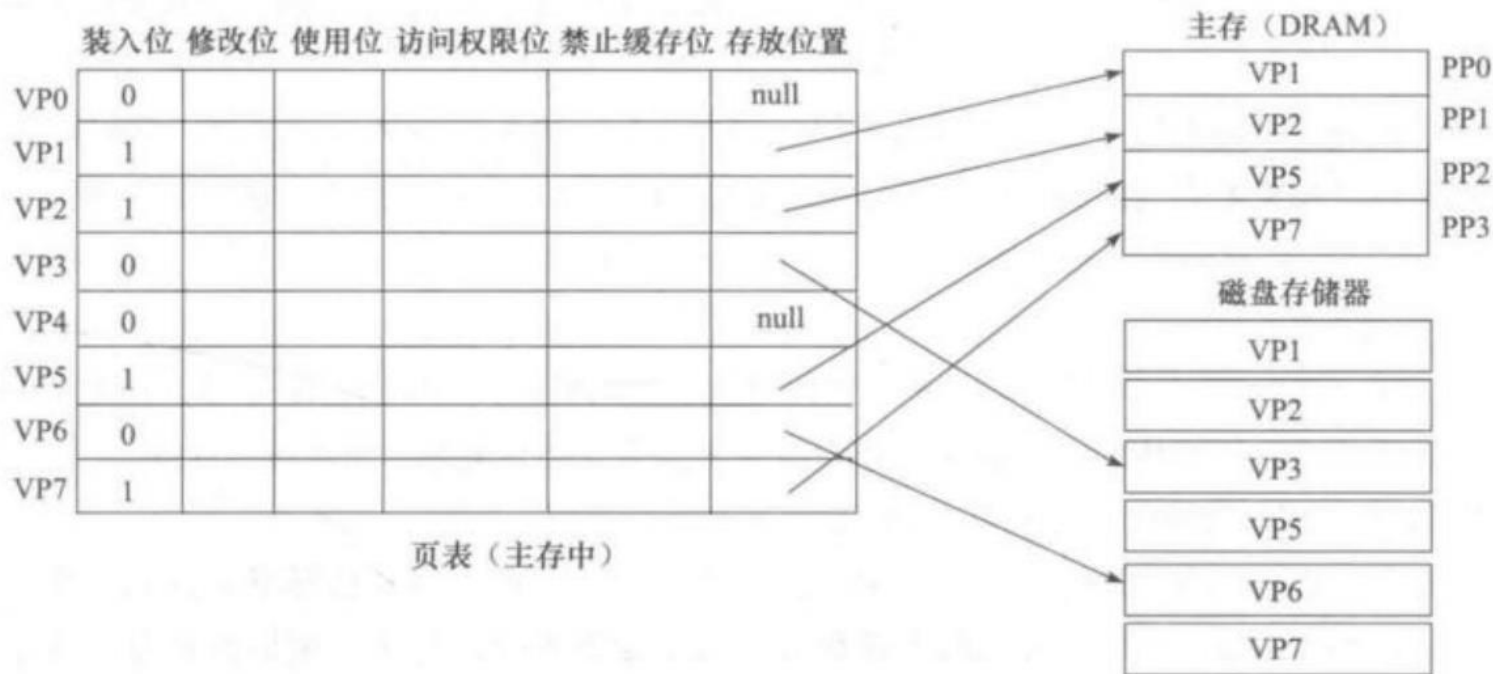
假定某一时刻用户**程序1**、**用户程序2**和**用户程序k**都需要被运行



# 页表

虚拟地址空间被划分为大小相等的页面，成为虚页（VP）；主存中的页被称为物理页（PP）

进程中的每个虚拟页在页表中都有一个对应的表项，称为页表项。页表项内容包括诸多标识位

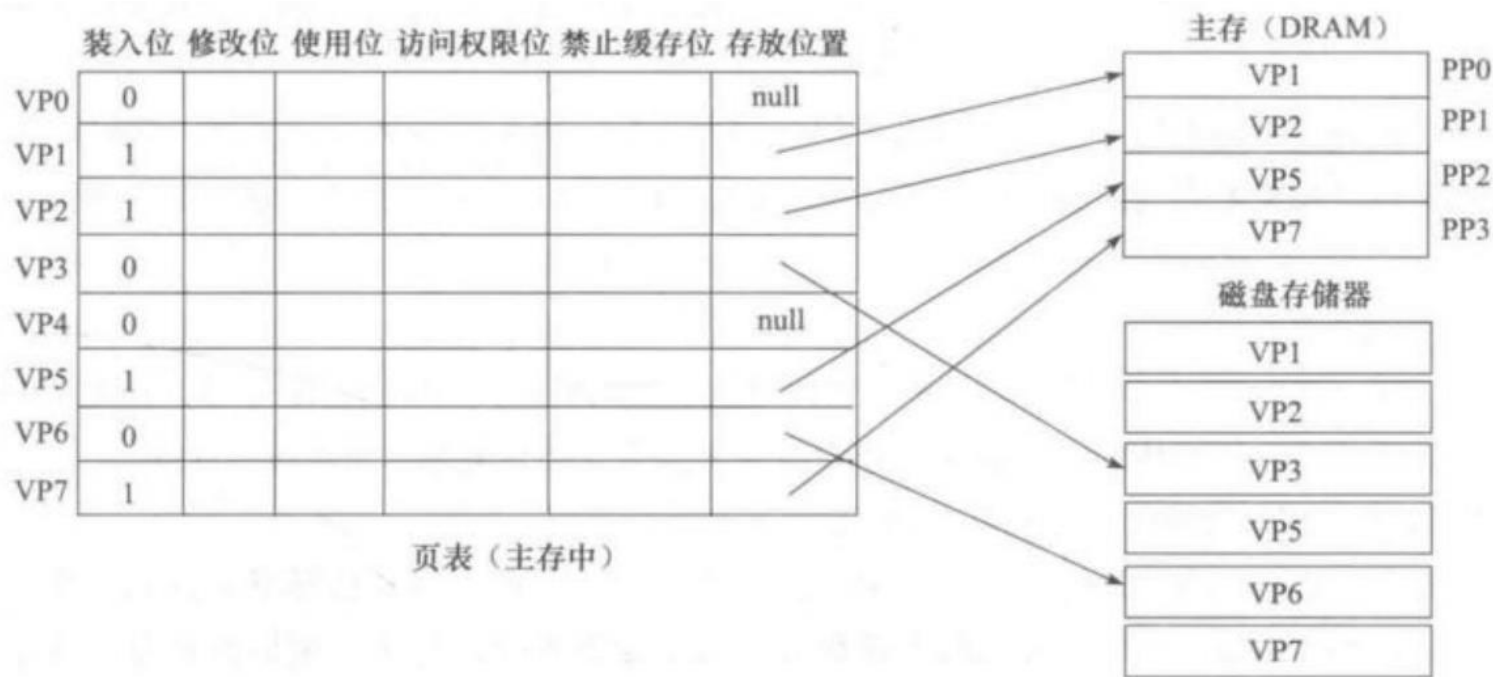


装入位也称为有效位。表示该虚拟页是否调入主存，若为“0”，是一个“**缓存页**”，此时，存放位置字段存放主存物理页号；若为“0”、则表示没有被调入主存。此时，若存放位置字段为 null，则说明是一个“**未分配页**”，否则是一个“**未缓存页**”，其存放位置字段给出该虚拟页在磁盘上的起始地址

其中有4个缓存页：VP1、VP2、VP5 和 VP7；  
有两个未分配页：VP0和 VP4；  
有两个未缓存页：VP3和VP6。



# 页表



new的数据在虚拟地址空间的堆区，但必须在Mem和Swp中有物理映射。

内存泄漏？

为什么长代码不会爆内存？

为什么new的空间会爆内存？

内存+Swap+磁盘。

mmap的匿名映射与文件映射。

复杂的继承关系最容导致内存泄漏？

找不到内存泄漏位置怎么办？

重启，客户将连接不到服务器。

Mem [||||| 30.1G/126G]  
Swp [|| 1.16G/256G]

# 函数调用

- 实参和形参之间，按值、按位置进行传递
  - 无论形参是普通数据类型，还是指针类型
- 函数栈的增长方向为：高地址 -> 低地址



## 3.2 递归程序的编程实现

- 递归的例1： 求解 $n!$ 
    - $0!=1$
    - $1!=1=1*0!$
    - $2!=2*1=2*1!$
    - $3!=3*2*1=3*2!$
    - .....
    - $n!=n*(n-1)! \ (n>0)$  (将未知问题转化为已知问题)
- 总结： 
$$n! = \begin{cases} 1 & (n=0 \text{时}) \\ n*(n-1)! & (n>0 \text{时}) \end{cases}$$

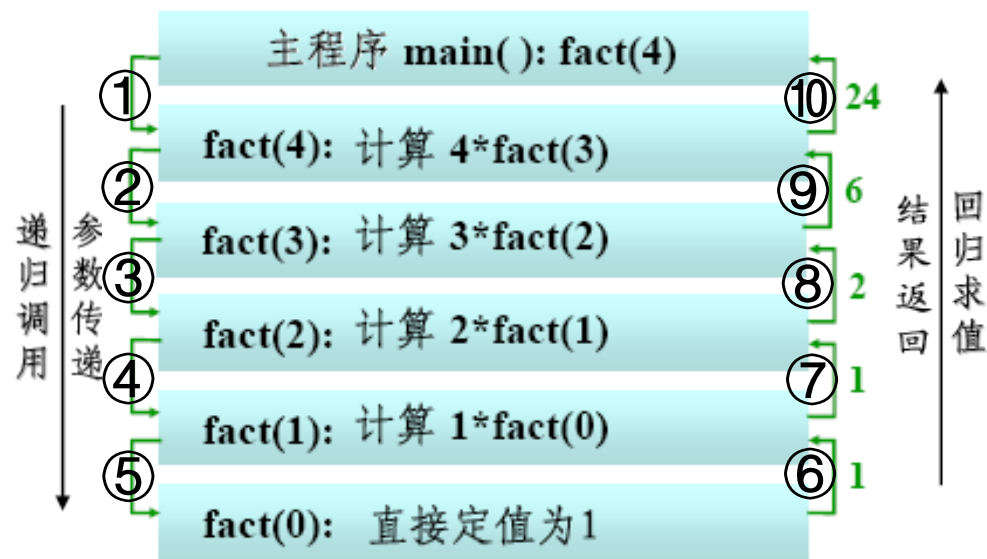
```
long fact( long n)
```

```
{
```

```
    if (n==0) return 1; //递归结束条件
```

```
    else return n*fact(n-1);// 递归的规则
```

```
}
```



- 问题分析:

- 问题2: 给定一个最多包含40亿个随机排列的32 bits非负整数的顺序文件, 找出一个不在文件中的32 bits整数 (在文件中必然缺失一个这样的数——为什么? )。顺序文件, 表示的是文件写入方式是顺序写入

- 问题规模? 40亿 (4e9) 个INT型
- 找到一个、多个、所有不在文件中的值?
- 在具有足够内存的情况下, 如何解决该问题?
- 如果有几个外部的“临时”文件可用, 但是仅有几百字节的内存, 又该如何解决?



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

谢谢！

中国科学技术大学