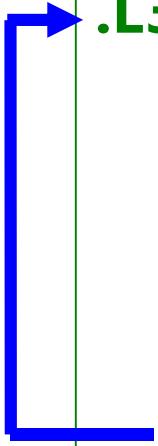


# 回顾：程序的机器级表示与执行

```
int sum(int a[ ], unsigned len)
{
    int i, sum = 0;
    for (i = 0; i <= len-1; i++)
        sum += a[i];
    return sum;
}
```



```
sum:
    ...
.L3:
    ...
    movl -4(%ebp), %eax
    movl 12(%ebp), %edx
    subl $1, %edx
    cmpl %edx, %eax
    jbe .L3
    ...
```

程序的正常执行顺序有哪两种？

(1) 按顺序取下一条指令执行

(2) 通过CALL/RET/Jcc/JMP等指令跳转到转移目标地址处执行

CPU所执行的指令的地址序列称为CPU的控制流，通过上述两种方式得到的控制流为正常控制流。

# 重定位后

程序始终按正常  
控制流执行吗?

08048380 <main>:

```
8048380: 55          push %ebp
8048381: 89 e5       mov  %esp,%ebp
8048383: 83 e4 f0    and  $0xffffffff0,%esp
8048386: e8 09 00 00 00 call 8048394 <swap>
804838b: b8 00 00 00 00 mov  $0x0,%eax
```

```
8048390: c9
8048391: c3
8048392: 90
8048393: 90
```

假定每个函数  
要求4字节边界  
对齐,故填充两  
条nop指令

R[eip]=0x804838b

- 1) R[esp] ← R[esp]-4
- 2) M[R[esp]] ← R[eip]
- 3) R[eip] ← R[eip]+0x9

08048394 <swap>:

```
8048394: 55          push %ebp
8048395: 89 e5       mov  %esp,%ebp
8048397: 83 ec 10    sub  $0x10,%esp
804839a: c7 05 00 97 04 08 24 mov  $0x8049624,0x8049700
80483a1: 96 04 08
80483a4: a1 28 96 04 08    mov  0x8049628,%eax
80483a9: 8b 00       mov  (%eax),%eax
80483ab: 89 45 fc    mov  %eax,-0x4(%ebp)
80483ae: a1 28 96 04 08    mov  0x8049628,%eax
80483b3: 8b 15 00 97 04 08 mov  0x8049700,%edx
80493b9: 8b 12       mov  (%edx),%edx
80493bb: 89 10       mov  %edx,(%eax)
80493bd: a1 00 97 04 08    mov  0x8049700,%eax
80493c2: 8b 55 fc    mov  -0x4(%ebp),%edx
80493c5: 89 10       mov  %edx,(%eax)
80493c7: c9          leave
80493c8: c3          ret
```

[BACK](#)

# 异常控制流

---

- CPU会因为遇到**内部异常**或**外部中断**等原因而打断程序的正常控制流，转去执行操作系统提供的针对这些特殊事件的处理程序。
- 由于某些特殊情况**引起用户程序的正常执行被打断**所形成的意外控制流称为**异常控制流**（Exceptional Control of Flow, ECF）。
- 异常控制流的形成原因：
  - 内部异常（缺页、越权、越级、整除0、溢出等）
  - 外部中断（Ctrl-C、打印缺纸、DMA结束等）
  - 进程的上下文切换（发生在操作系统层）
  - 一个进程直接发送信号给另一个进程（发生在应用软件层）

} 发生在  
硬件层

本章主要介绍发生在OS层和硬件层的异常控制流

# “程序” 和 “进程”

---

**程序 (program)** 指按某种方式组合形成的代码和数据集合，代码即是机器指令序列，因而程序是一种**静态**概念。

**进程 (process)** 指程序的一次运行过程。更确切说，进程是具有独立功能的一个程序关于某个数据集合的一次运行活动，因而进程具有**动态**含义。同一个程序处理不同的数据就是不同的进程

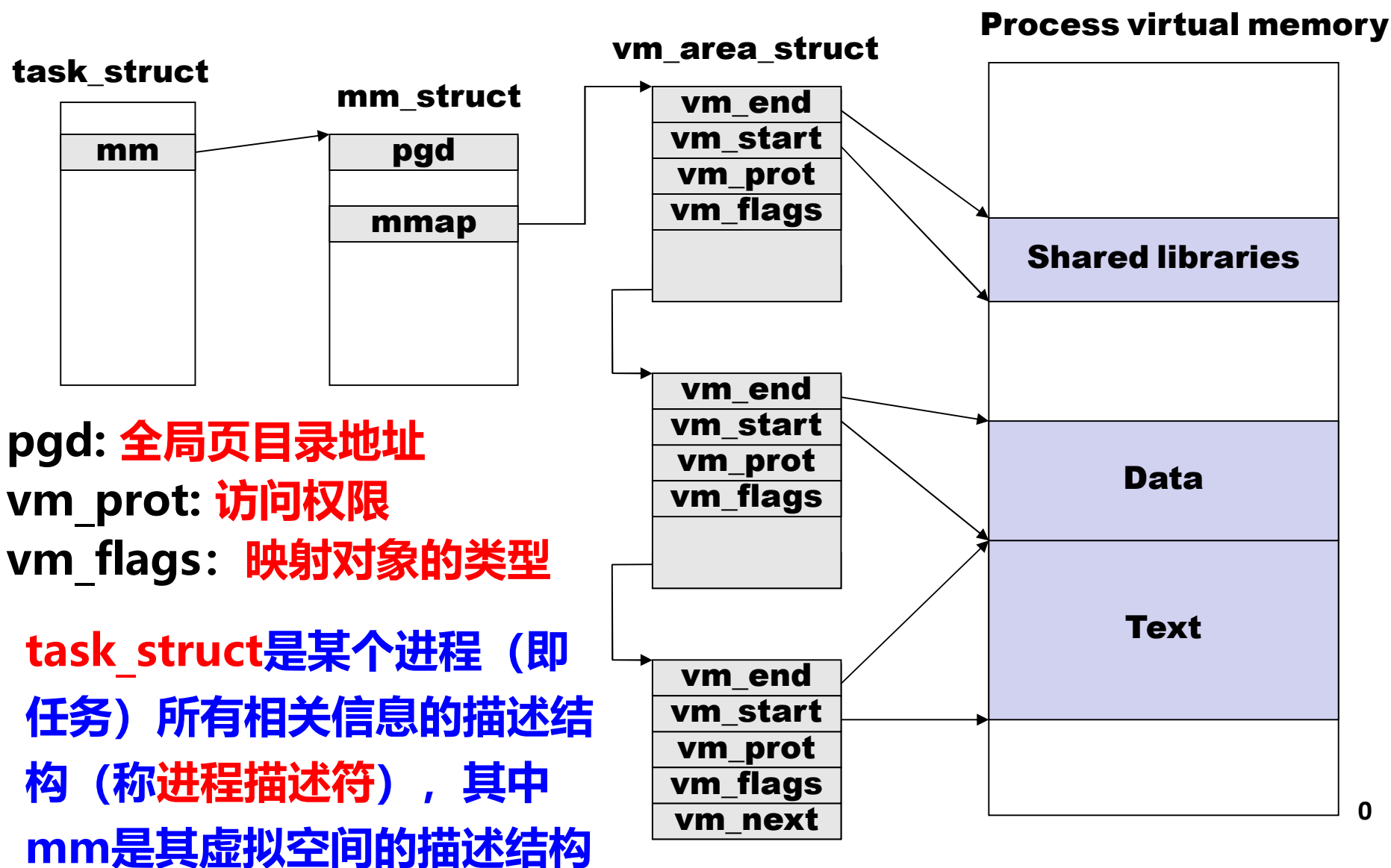
- 进程是OS对CPU执行的程序的运行过程的一种抽象。进程有**自己的生命周期**，它由于任务的启动而创建，随着任务的完成（或终止）而消亡，它所占用的资源也随着进程的终止而释放。
- 一个可执行目标文件（即程序）可被加载执行多次，也即，一个程序可能对应多个不同的进程。
  - 例如，用word程序编辑一个文档时，相应的用户进程就是winword.exe，如果多次启动同一个word程序，就得到多个winword.exe进程，**处理不同的数据**。

# 进程的概念

---

- 操作系统（管理任务）以外的都属于“用户”的任务。
- 计算机处理的所有“用户”的任务由进程完成。
- 为强调进程完成的是用户的任务，通常将进程称为用户进程。
- 计算机系统中的任务通常就是指进程。例如，
  - Linux内核中通常把进程称为任务，每个进程主要通过一个称为进程描述符（process descriptor）的结构来描述，其结构类型定义为task\_struct，包含了一个进程的所有信息。
  - 所有进程通过一个双向循环链表实现的任务列表（task list）来描述，任务列表中每个元素是一个进程描述符。
  - IA-32中的任务状态段（TSS）、任务门（task gate）等概念中所称的任务，实际上也是指进程。

# 回顾：Linux将虚存空间组织成“区域”的集合



# 引入“进程”的好处

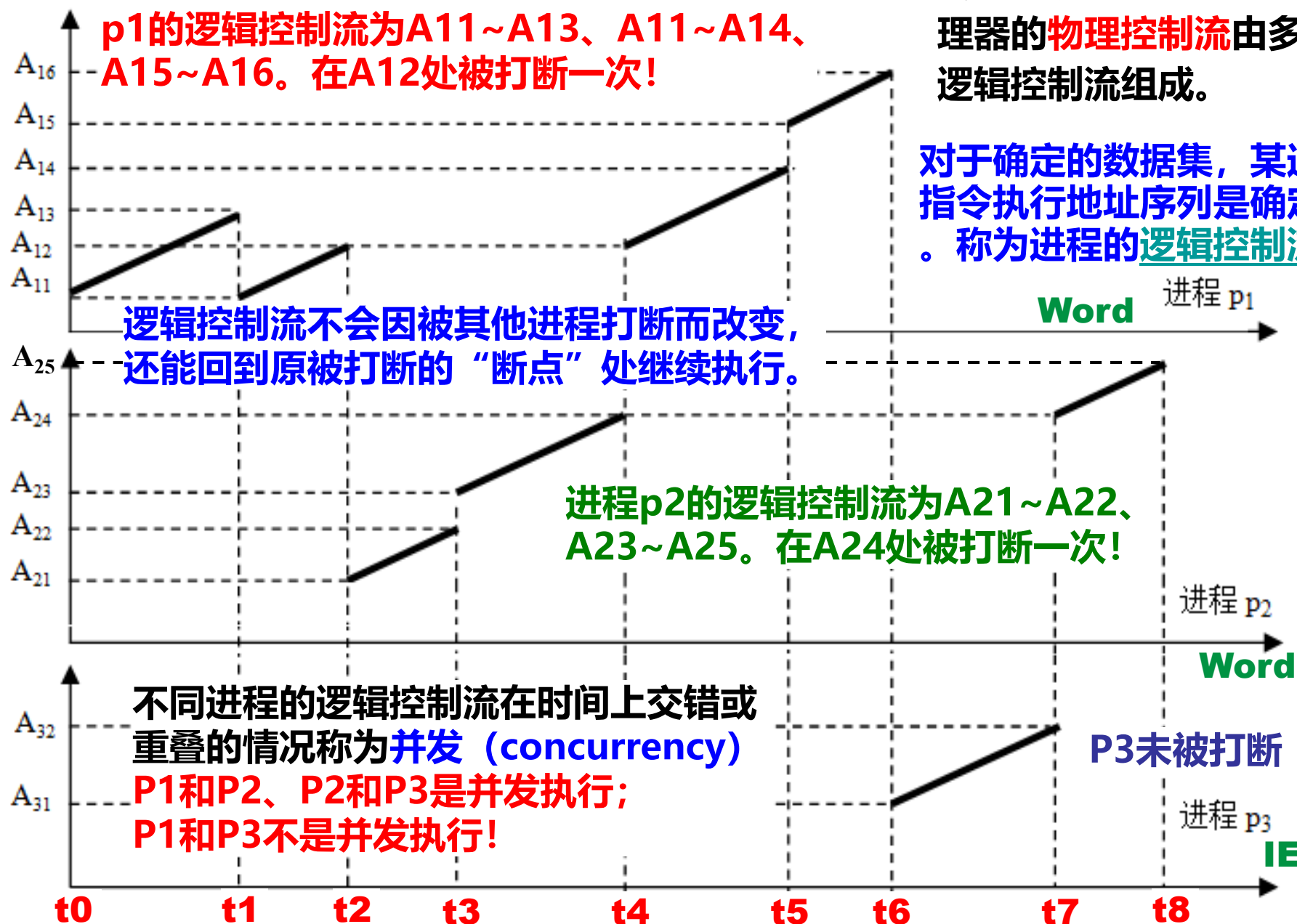
---

- “进程”的引入为应用程序提供了以下两方面的抽象：
  - 一个**独立的逻辑控制流**
    - 每个进程拥有一个独立的逻辑控制流，使得程序员以为自己的程序在执行过程中**独占使用处理器**
  - 一个**私有的虚拟地址空间**
    - 每个进程拥有一个私有的虚拟地址空间，使得程序员以为自己的程序在执行过程中**独占使用存储器**
- “进程”的引入简化了程序员的编程以及语言处理系统的处理，即**简化了编程、编译、链接、共享和加载等整个过程。**

# 逻辑控制流

对于单处理器系统，进程会轮流使用处理器，即处理器的物理控制流由多个逻辑控制流组成。

对于确定的数据集，某进程指令执行地址序列是确定的。称为进程的逻辑控制流。





# “进程” 与 “上下文切换”

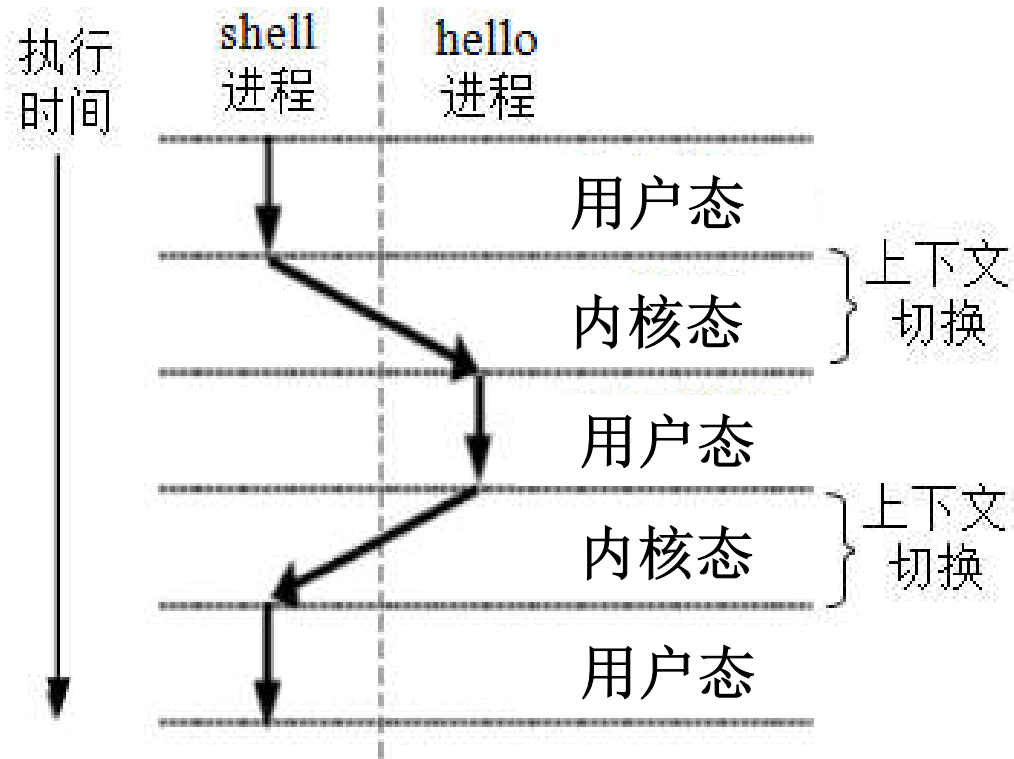
OS通过处理器调度让处理器轮流执行多个进程。实现不同进程中指令交替执行的机制称为**进程的上下文切换 (context switching)**

```
./hello  
hello, world  
$
```

“\$” 是shell命令行提示符，说明正在运行shell进程。

在一个进程的生命周期中，可能会有其他不同进程在处理器上交替运行！

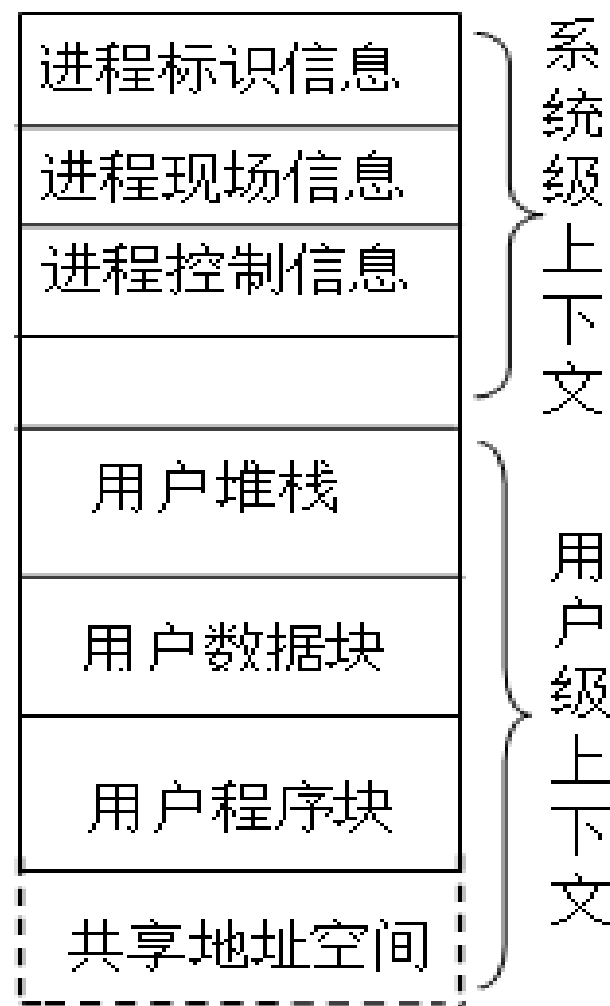
感觉到的运行时间比真实执行时间要长！



处理器调度等事件会引起用户进程正常执行被打断，因而形成异常控制流。进程的上下文切换机制很好地解决了这类异常控制流，实现了从一个进程安全切换到另一个进程执行的过程。

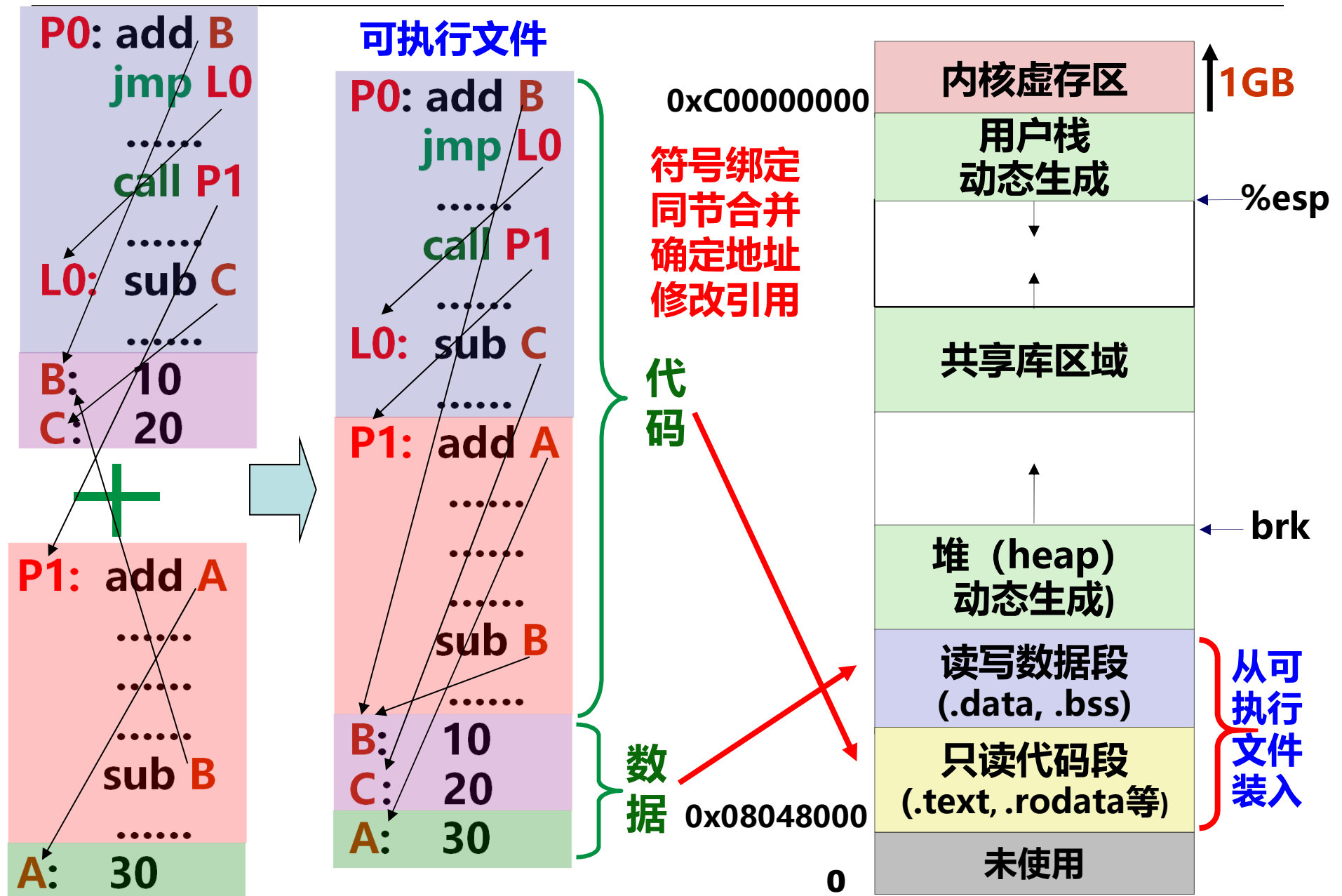
# “进程” 的 “上下文”

- 进程的物理实体（代码和数据等）和支持进程运行的环境合称为**进程的上下文**。
- 由进程的**程序块**、**数据块**、运行时的堆和用户栈（两者通称为**用户堆栈**）等组成的用户空间信息被称为**用户级上下文**；
- 由**进程标识信息**、**进程现场信息**、**进程控制信息**和系统内核栈等组成的内核空间信息被称为**系统级上下文**；
- 处理器中各寄存器的内容被称为**寄存器上下文**（也称**硬件上下文**），即进程的现场信息。
- 在进行进程上下文切换时，操作系统把换下进程的寄存器上下文保存到系统级上下文中的现场信息位置。
- 用户级上下文地址空间和系统级上下文地址空间一起构成了一个进程的整个存储器映像



进程的存储器映像

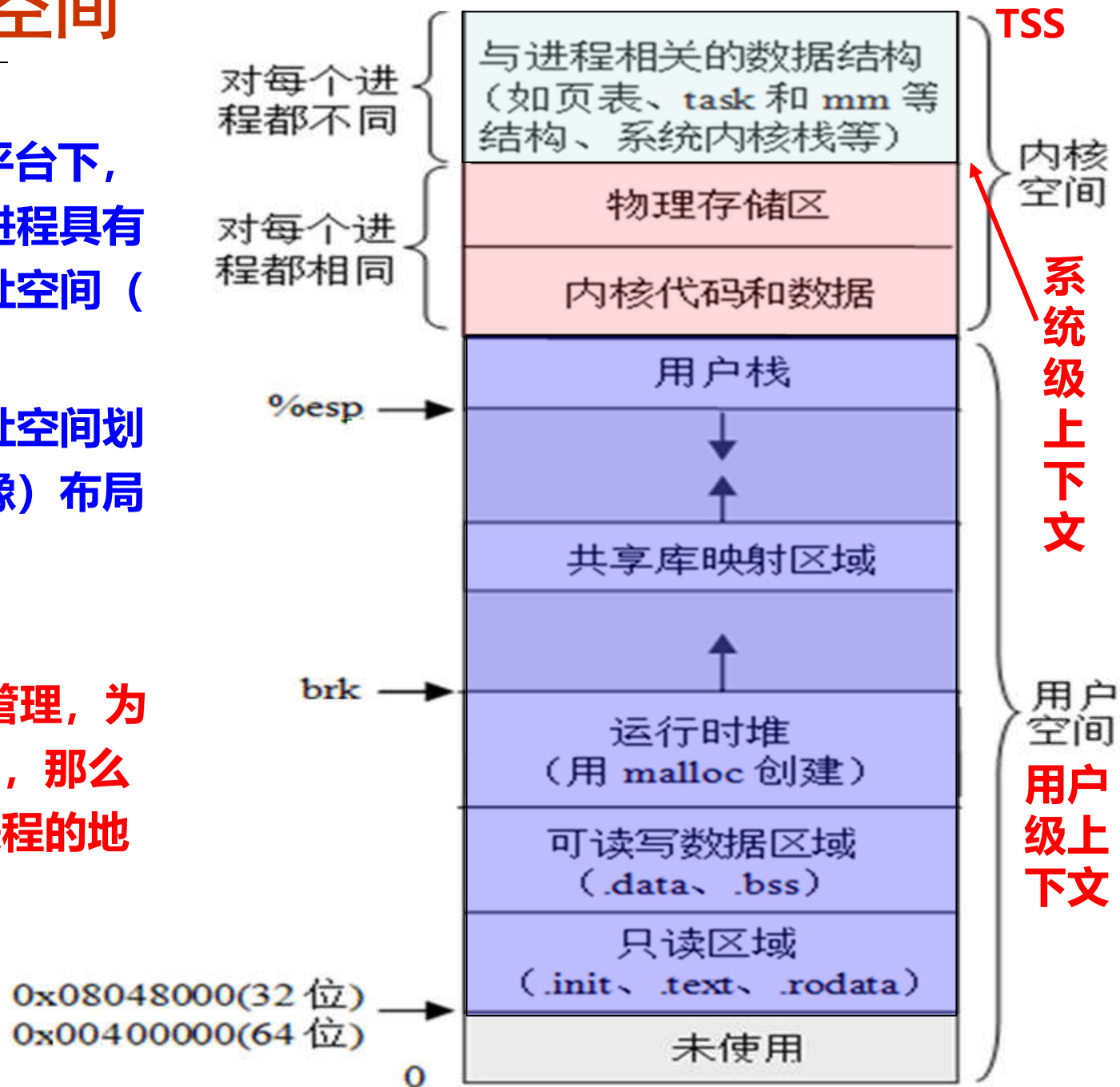
# 回顾：链接操作的步骤



# 进程的地址空间

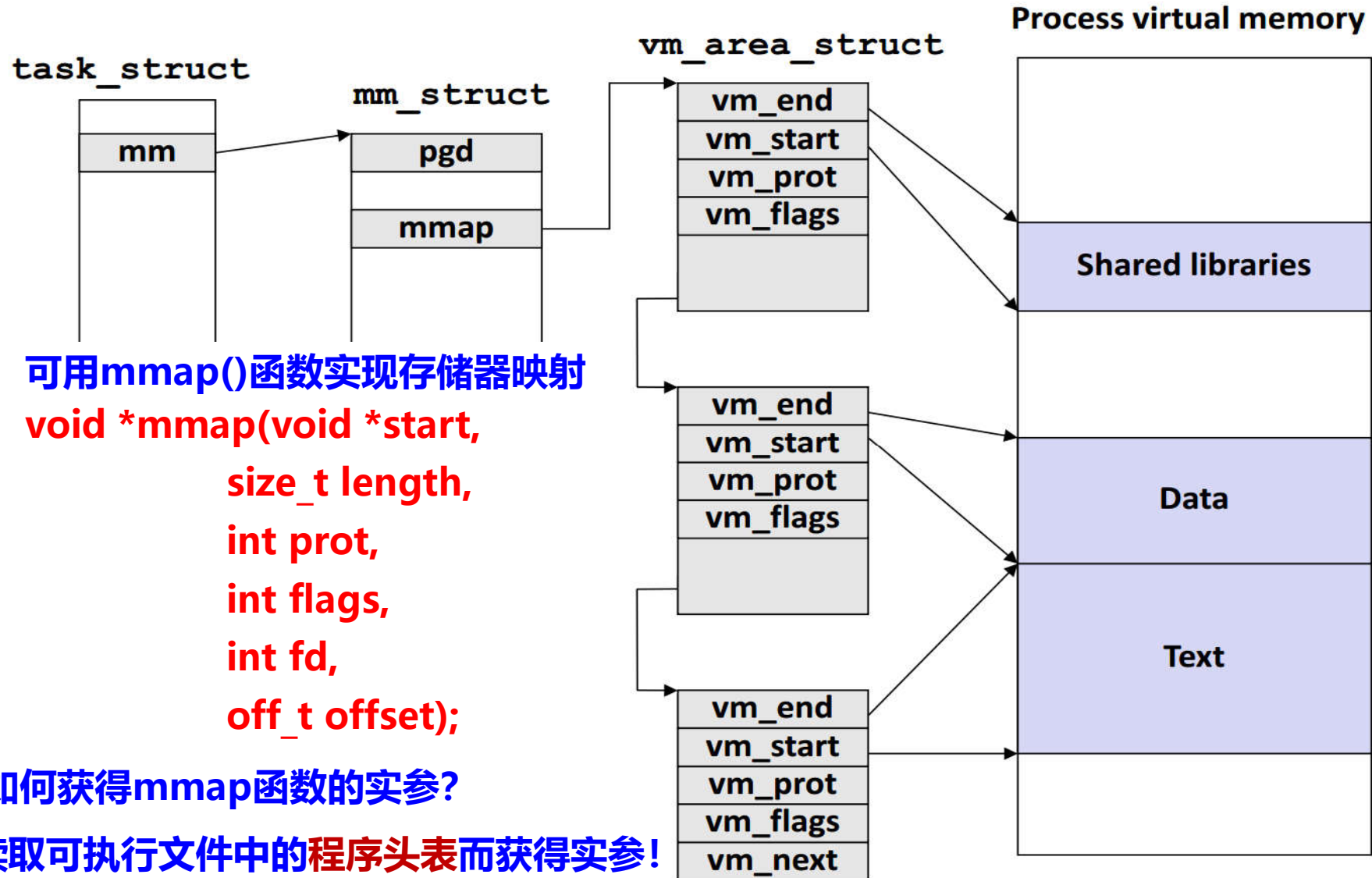
- IA-32/Linux平台下, 每个 (用户) 进程具有独立的私有地址空间 (虚拟地址空间)
- 每个进程的地址空间划分 (即存储映像) 布局相同 (如右图)

OS要对进程进行管理, 为进程分配主存空间, 那么它如何描述一个进程的地址空间呢?

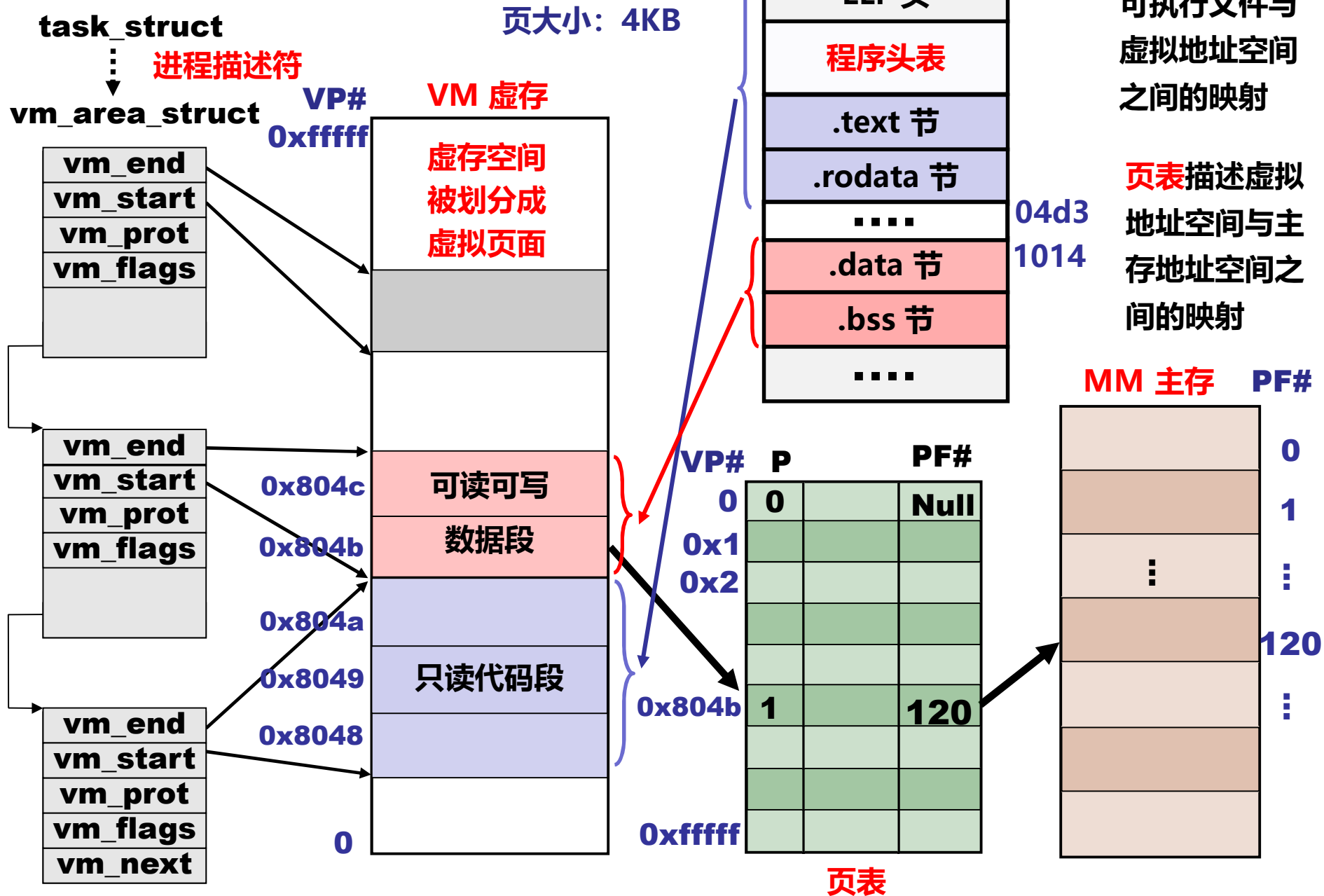


# 进程的存储器映射

**存储器映射 (memory mapping)** 是指将进程虚拟地址空间中的一个区域与硬盘上的一个对象建立关联 (生成页表项), 并初始化一个 `vm_area_struct` 结构信息



# 回顾：存储管理全局图



# 进程的存储器映射

可用mmap()函数实现存储器映射

**void \*mmap(void \*start, size\_t length, int prot, int flags, int fd, off\_t offset);**

**功能：**将指定文件fd中偏移量offset开始的长度为length个字节的一块信息映射到虚拟空间中起始地址为start、长度为length个字节的一块区域，得到vm\_area\_struct结构的信息，并初始化相应页表项，建立文件地址和区域之间的映射关系。

**prot**指定该区域内页面的访问权限位，对应vm\_area\_struct结构中的vm\_prot字段

**PROT\_EXE：**页面内容由指令组成

**PROT\_READ：**区域内页面可读

**PROT\_WRITE：**区域内页面可写

**PROT\_NONE：**区域内页面不能被访问

**flags**指定所映射的对象的类型，对应vm\_area\_struct结构中的vm\_flags字段

**MAP\_PRIVATE：**私有对象，采用写时拷贝技术，对应可执行文件中只读代码区域（.init、.text、.rodata）和已初始化数据区域（.data）

**MAP\_SHARED：**共享对象，对应共享库文件中的信息

**MAP\_ANON：**请求0的页，对应内核创建的匿名文件，相应页框用0初始化并驻留内存

**MAP\_PRIVATE | MAP\_ANON：**未初始化数据（.bss）、堆和用户栈等对应区域

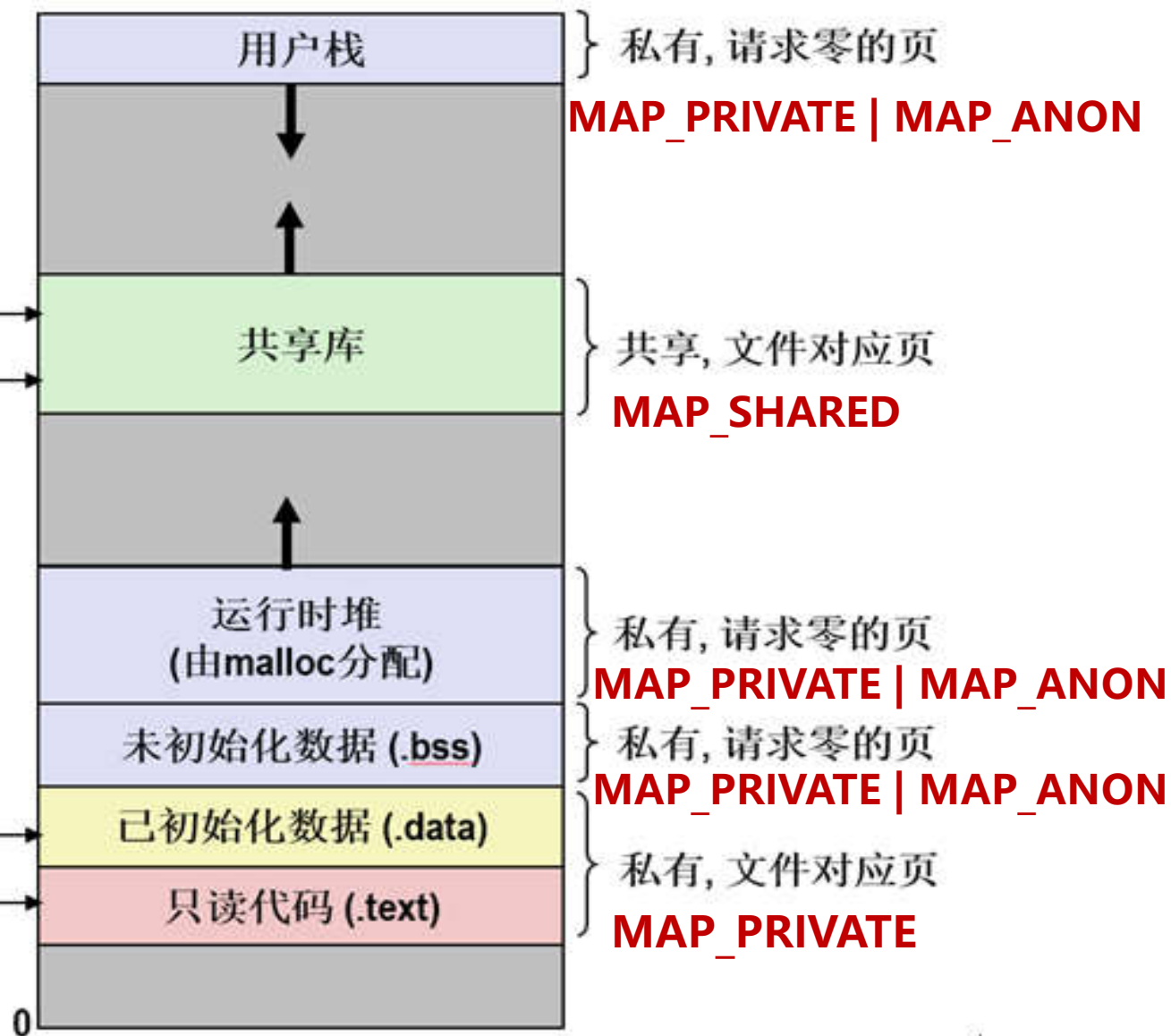
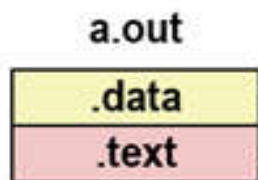
虚页第一次被装入内存后，不管是用普通文件还是匿名文件对其进行初始化，以后都是在主存页框和硬盘中的交换文件（swap file）之间进行调进调出。交换文件由内核管理和维护，称为交换分区（swap area）或交换空间（swap space）。

# Linux中虚拟地址空间中的区域

**匿名文件：**内核创建、  
无实际磁盘文件，无需  
从磁盘读入、对应请求0  
的页面



**普通文件：**可执行文件  
和共享库文件





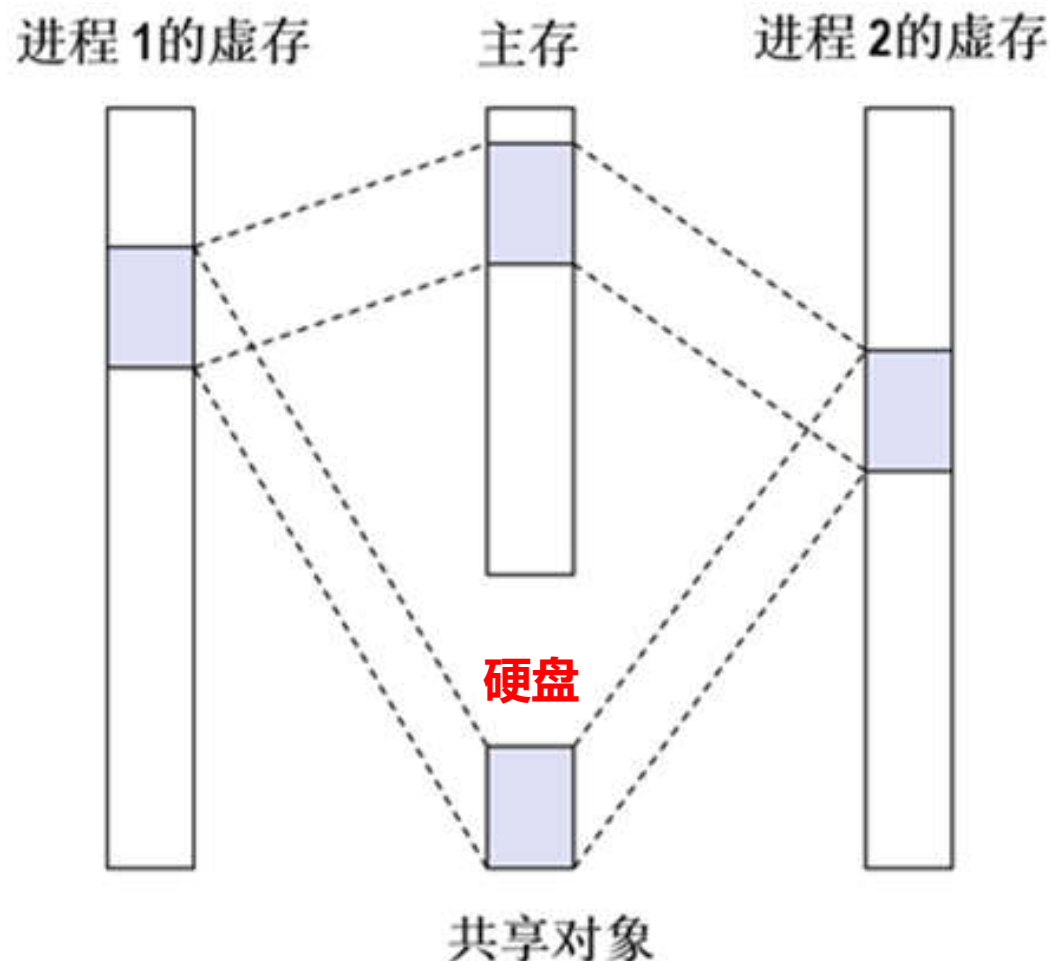
# 共享库文件中的共享对象

**多个进程调用共享库文件中的代码，但共享库代码在内存和硬盘都只需要一个副本**

**进程1运行过程中，内核为共享对象分配若干页框**

**进程2运行过程中，内核只要将进程2对应区域内页表项中的页框号直接填上即可**

**一个进程对共享区域进行的写操作结果，对于所有共享同一个共享对象的进程都是可见的，而且结果也会反映在硬盘上对应的共享对象中**



**所分配的页框在主存不一定连续，为简化示意图，这里图中所示页框是连续的**

# 私有对象的写时拷贝技术

同一个可执行文件对应不同进程时，只读代码区一样，可读可写数据区开始也一样，但属于私有对象为节省主存，多采用写时拷贝技术

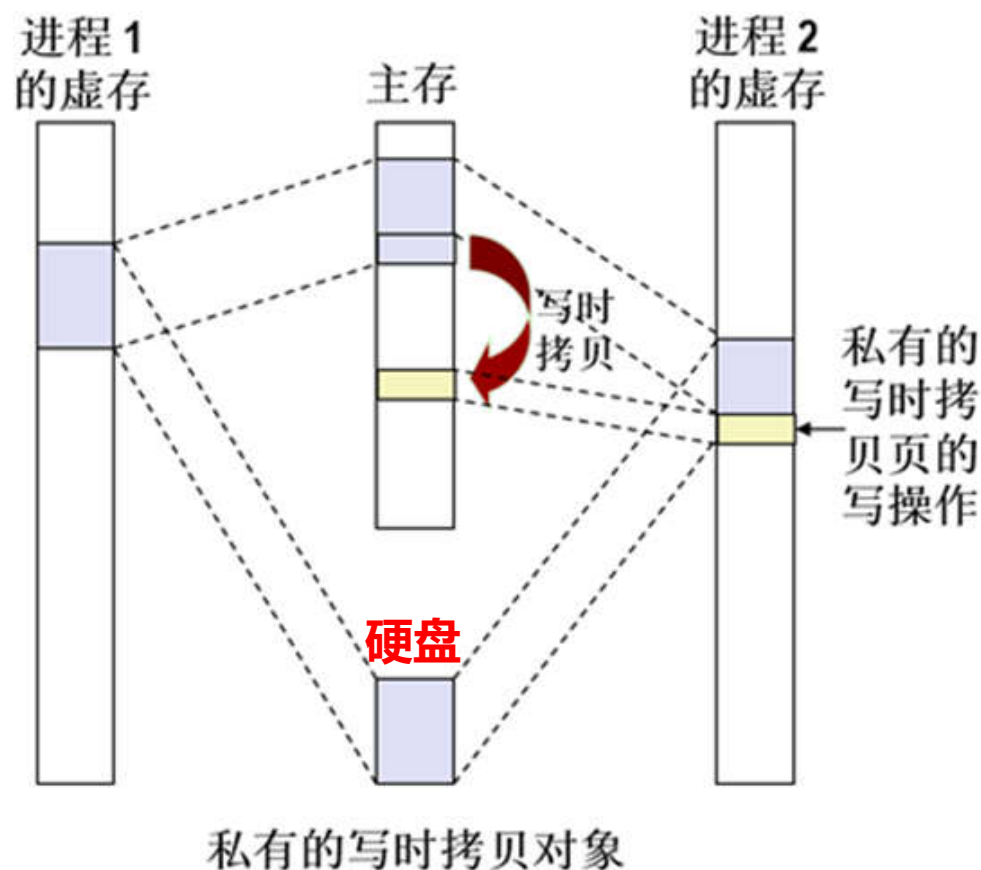
进程1运行过程中，内核为对象分配若干页框，并标记为只读

进程2运行过程中，内核只要将进程2对应区域内页表项中的页框号直接填上，并标记为只读

若两个进程都只是读或执行，则在内存只有一个副本，节省主存；

若进程2进行写操作，则发生访问违例，此时，内核判断异常原因是进程试图写私有的写时拷贝页，就会分配一个新页框，把内容拷贝到新页框，并修改进程2的页表项

所分配的页框在主存不一定连续



采用写时拷贝技术，能保证：

- (1) 只读代码区在内存只有一个副本
- (2) 可读可写数据区占用尽量少的页框

# 回顾：用户模式和内核模式

---

- 为了使OS能够起到管理程序执行的目的，在一些时候处理器中必须运行**内核代码**
- 为了区分处理器运行的是用户代码还是内核代码，必须有一个状态位来标识，这个状态位称为**模式位**
- **处理器模式**分**用户模式（用户态）**和**内核模式（核心态）**
- 用户模式（也称**目态、用户态**）下，处理器运行用户进程，此时不允许使用特权指令
- 内核模式（有时称**系统模式、管理模式、超级用户模式、管态、内核态、核心态**）下处理器运行内核代码，允许使用**特权指令**，例如：停机指令、开/关中断指令、Cache冲刷指令等。

# 程序的加载和运行

---

- UNIX/Linux系统中，可通过调用**execve()**函数来启动加载器。
- **execve()**函数的功能是在当前进程上下文中加载并运行一个新程序。  
**execve()**函数的用法如下：

```
int execve(char *filename, char *argv[], *envp[]);
```

**filename**是加载并运行的可执行文件名(如./hello)，可带参数列表**argv**和环境变量列表**envp**。若错误（如找不到指定文件**filename**），则返回-1，并将控制权交给调用程序；若函数执行成功，则不返回，最终将控制权传递到可执行目标中的主函数**main**。

- 主函数**main()**的原型形式如下：

```
int main(int argc, char **argv, char **envp); 或者：
```

```
int main(int argc, char *argv[], char *envp[]);
```

**argc**指定参数个数，参数列表中第一个总是命令名（可执行文件名）

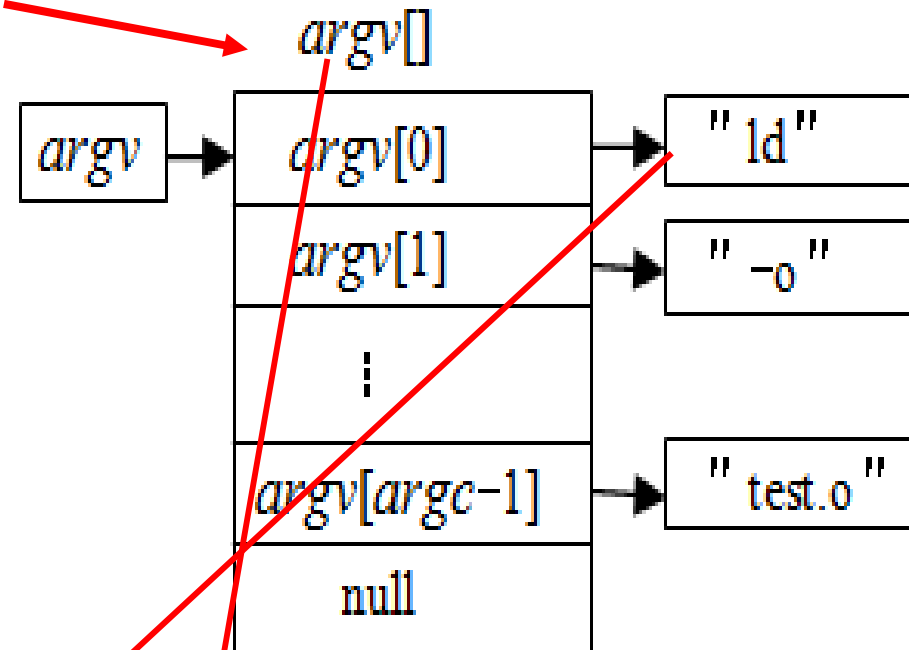
例如：命令行为“**ld -o test main.o test.o**” 时，**argc=5**

# 回顾：程序的加载和运行

若在shell命令行提示符下输入以下命令行

Unix>**ld -o test main.o test.o**

ld是可执行文件名（即命令名），随后是命令的若干参数，argv是一个以null结尾的指针数组，argc=5



在shell命令行提示符后键入命令并按“enter”键后，便构造argv和envp，然后调用**execve()**函数来启动加载器，最终转**main()**函数执行

```
int execve(char *filename, char *argv[], *envp[]);
```

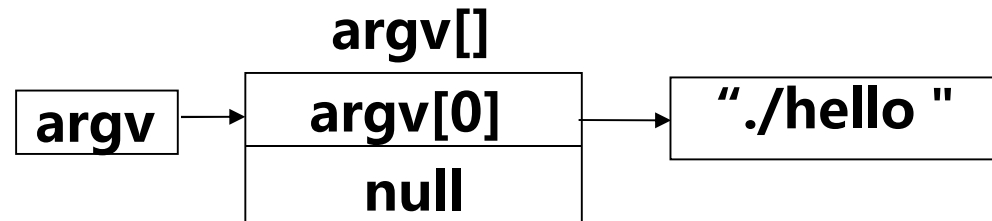
```
int main(int argc, char *argv[], char *envp[]);
```

# 回顾：程序的加载和运行

问题：hello程序的加载和运行过程是怎样的？

Step1: 在shell命令行提示符后输入命令：`$/./hello[enter]`

Step2: shell命令行解释器构造argv和envp



Step3: 调用**fork()**函数，创建一个子进程，与父进程shell完全相同（只读/共享），包括只读代码段、可读写数据段、堆以及用户栈等。

Step4: 调用**execve()**函数,在当前进程（新创建的子进程）的上下文中加载并运行hello程序。将hello中的.text节、.data节、.bss节等内容加载到当前进程的虚拟地址空间（仅修改当前进程上下文中关于存储映像的一些数据结构，不从磁盘拷贝代码和数据等内容）

Step5: 调用hello程序的**main()**函数，hello程序开始在一个进程的上下文中运行。  
`int main(int argc, char *argv[], char *envp[]);`

# 回顾：可执行文件的加载

- 通过调用execve系统调用函数来调用加载器
- 加载器 (loader) 根据可执行文件的程序 (段) 头表中的信息, 将可执行文件的代码和数据从磁盘“拷贝”到存储器中 (实际上不会真正拷贝, 仅建立一种映像, 这涉及到许多复杂的过程和一些重要概念, 将在后续课上学习)
- 加载后, 将PC (EIP) 设定指向 Entry point (即符号\_start处), 最终执行main函数, 以启动程序执行。

程序被启动

如 \$ ./P

调用fork()

以构造的argv和envp  
为参数调用execve()

execve()调用加载器  
进行可执行文件加载,  
并最终转去执行main

\_start: \_\_libc\_init\_first → \_init → atexit → main → \_exit

# ELF文件信息举例

**\$ readelf -h main**

可执行目标文件的ELF头

ELF Header:

Magic: 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00

Class: ELF32

Data: 2's complement, little endian

Version: 1 (current)

OS/ABI: UNIX - System V

ABI Version: 0

Type: EXEC (Executable file)

Machine: Intel 80386

Version: 0x1

**Entry point address: x8048580**

Start of program headers: 52 (bytes into file)

Start of section headers: 3232 (bytes into file)

Flags: 0x0

Size of this header: 52 (bytes)

Size of program headers: 32 (bytes)

Number of program headers: 8

Size of section headers: 40 (bytes)

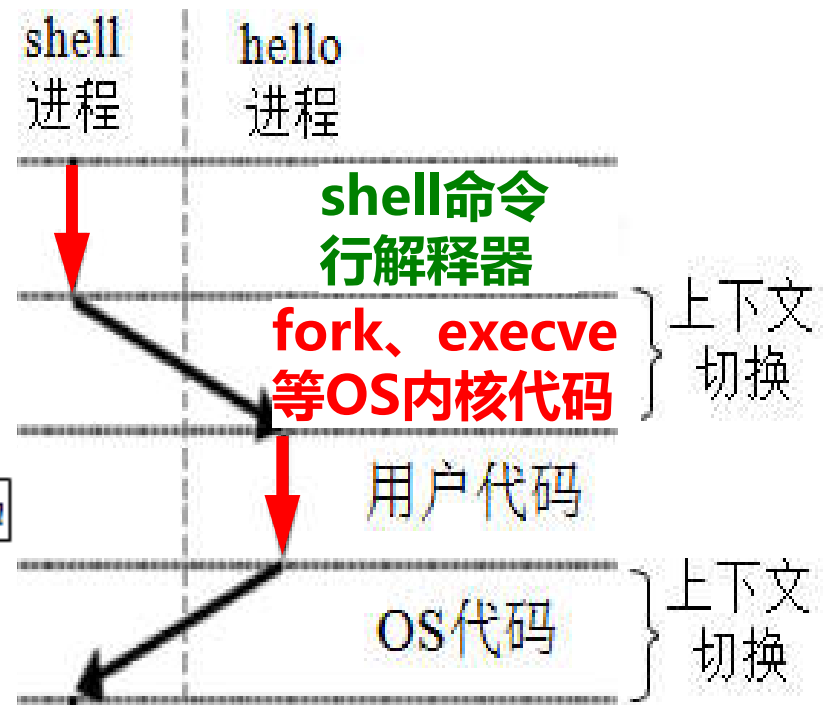
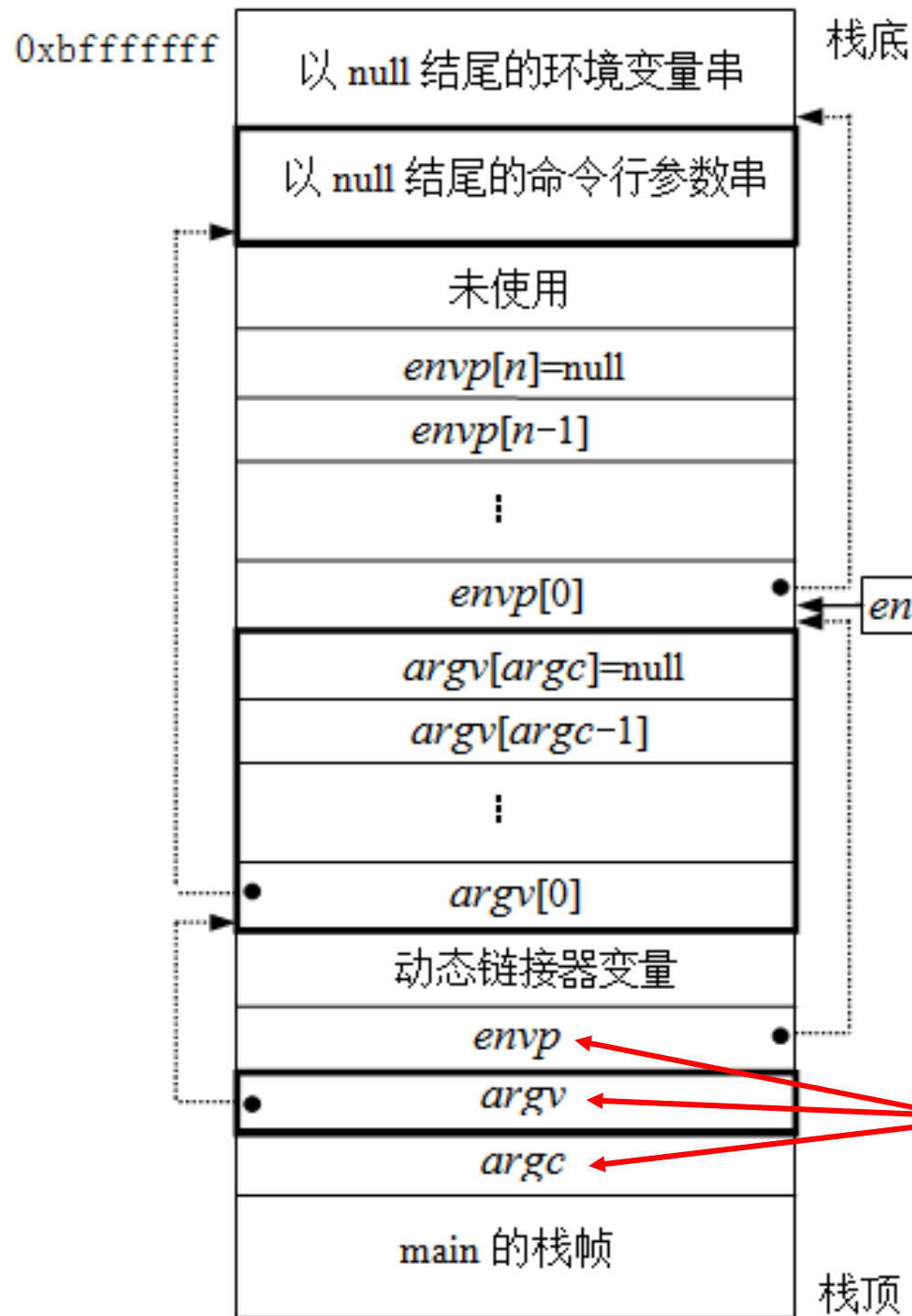
Number of section headers: 29

Section header string table index: 26

ELF 头
程序头表
.init 节
.text 节
.rodata 节
.data 节
.bss 节
.symtab 节
.debug 节
.line 节
.strtab 节
节头表



# 程序加载和运行



当 IA-32/Linux 系统开始执行 main() 函数时，在虚拟地址空间的 **用户栈** 中的结构如左图所示

```
int main(int argc,  
char *argv[],  
char *envp[]);
```