

## 第二章

# 并发进程

方 钰

---



# 主要内容

2.1 进程的基本概念

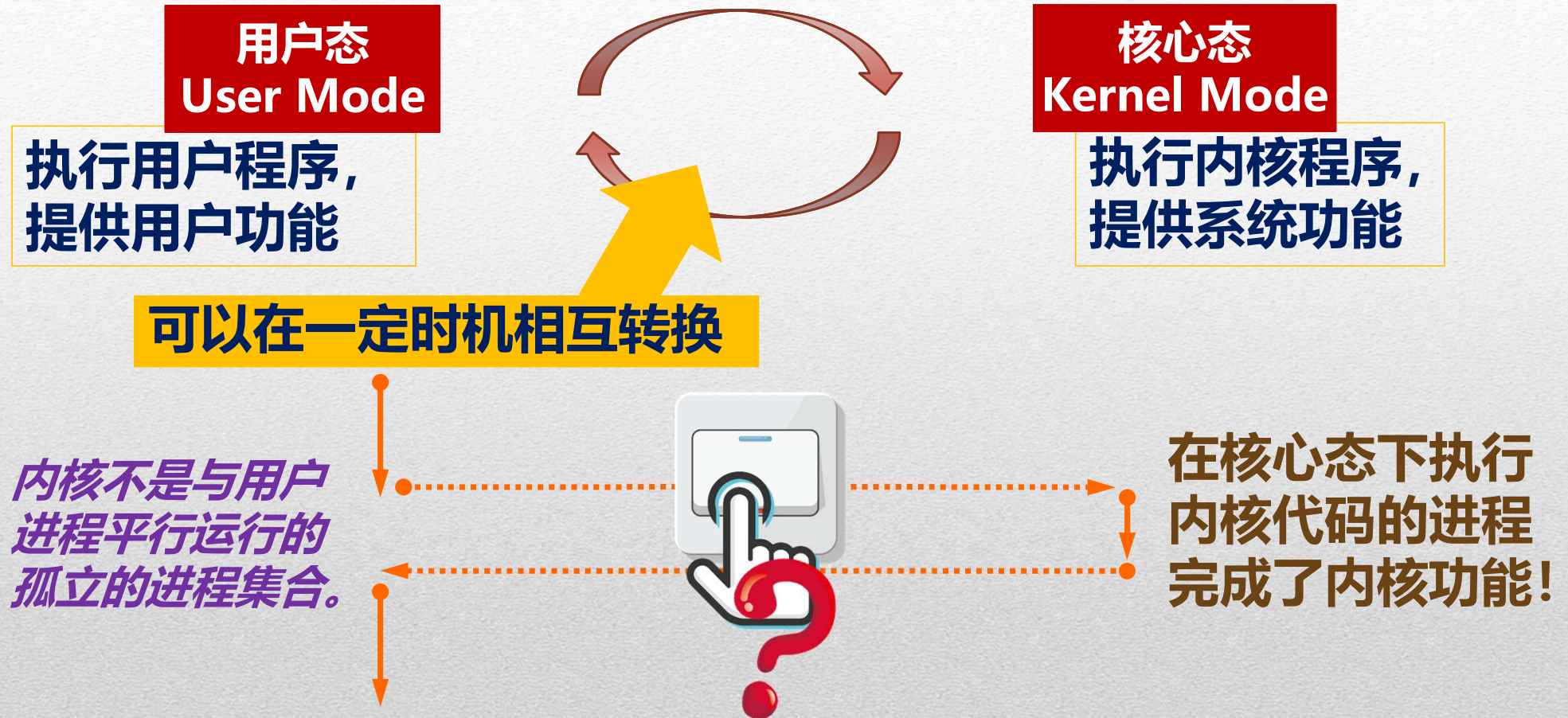
2.2 **UNIX的进程**

2.3 中断基本概念及UNIX中断处理

2.4 进程通信



# UNIX中进程的两种执行状态





# UNIX中进程的两个地址空间





# 程序并发执行带来的问题.....



资源共享



各种程序活动的相互依赖与制约

为了解决程序并发执行带来的问题:



程序



进程

一组数据与指令代码的集合

**结构特征**

代码段、数据段、堆栈段、**进程控制块**

静态的  
存放在某种介质上

**动态性**, 具有生命周期  
“由创建而产生, 由调度而执行, 由撤销而消亡”

- 多个进程实体可同时存在于内存中**并发执行**
- 独立运行、独立分配资源和独立接受调度的**基本单位**
- 按**不可预知 (异步)**的速度向前推进

今天继续解决进程的  
结构特征问题!!!

进程是程序的一次运行过程!!!





## 看下面这个简单的程序

### 用户态 User Mode

```
#include <fcntl.h>
char buffer[2048];
int version = 1;
main( argc, argv)
int  argc;
char *argv[];
{
    int a, b;
    .....;
    sum(a, b);
    exit(0);
}
int sum( var1, var2)
int var1, var2;
{
    int count;
    count = var1 +var2;
    return(count);
}
```

## 看下面这个简单的程序



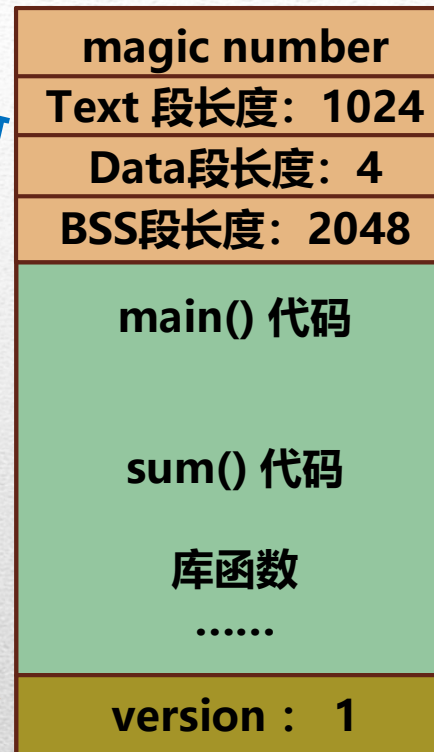
## UNIX的可执行文件

```
#include <fcntl.h>
char buffer[2048];
int version = 1;
main( argc, argv)
int  argc;
char *argv[];
{
    int a, b;
    .....;
    sum(a, b);
    exit(0);
}
int sum( var1, var2)
int var1, var2;
{
    int count;
    count = var1 + var2;
    return(count);
}
```

未赋初值的全局变量在可  
执行文件中只记录大小

被赋初值的全局变量在可  
执行文件中分配存储单元

0



文件头

Text段  
1K字节Data段  
2字节

可执行程序  
被保存在磁  
盘上

经过编译、  
链接形成可  
执行程序

Data段和BSS段可能  
因为链接的其他文件  
中的全局变量而更长

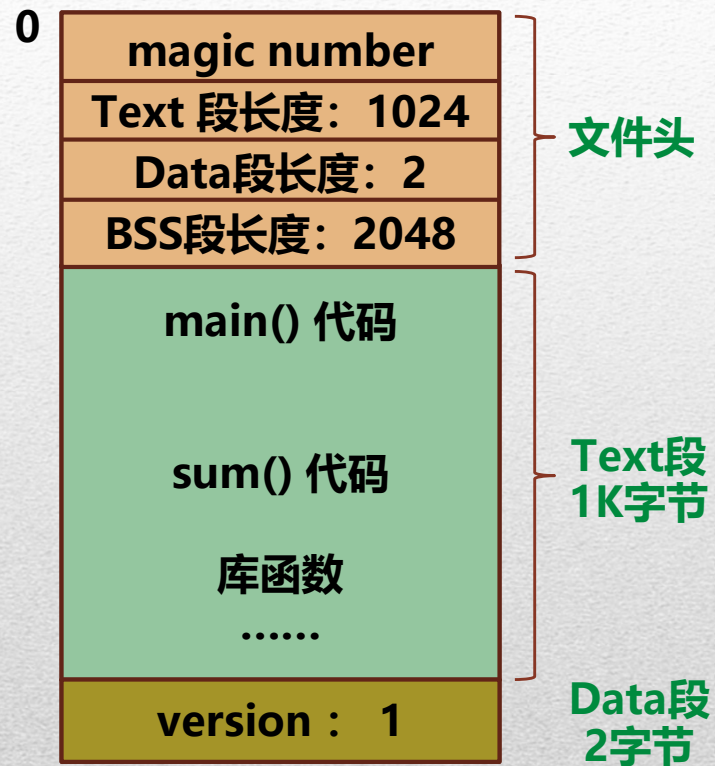




看下面这个简单的程序



UNIX的可执行文件



物 理

内 存

进程创建时, 可执行  
文件如何装入内存?

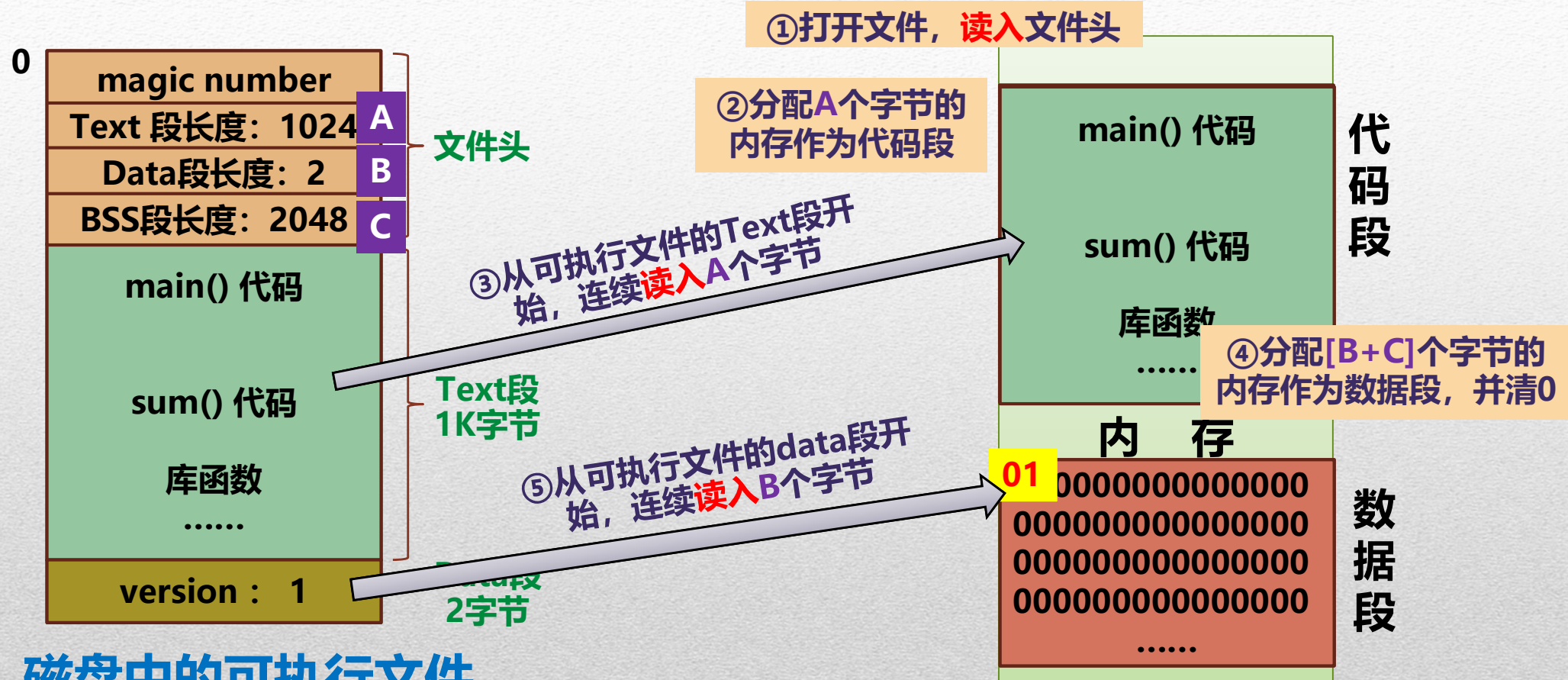
磁盘中的可执行文件



# 看下面这个简单的程序



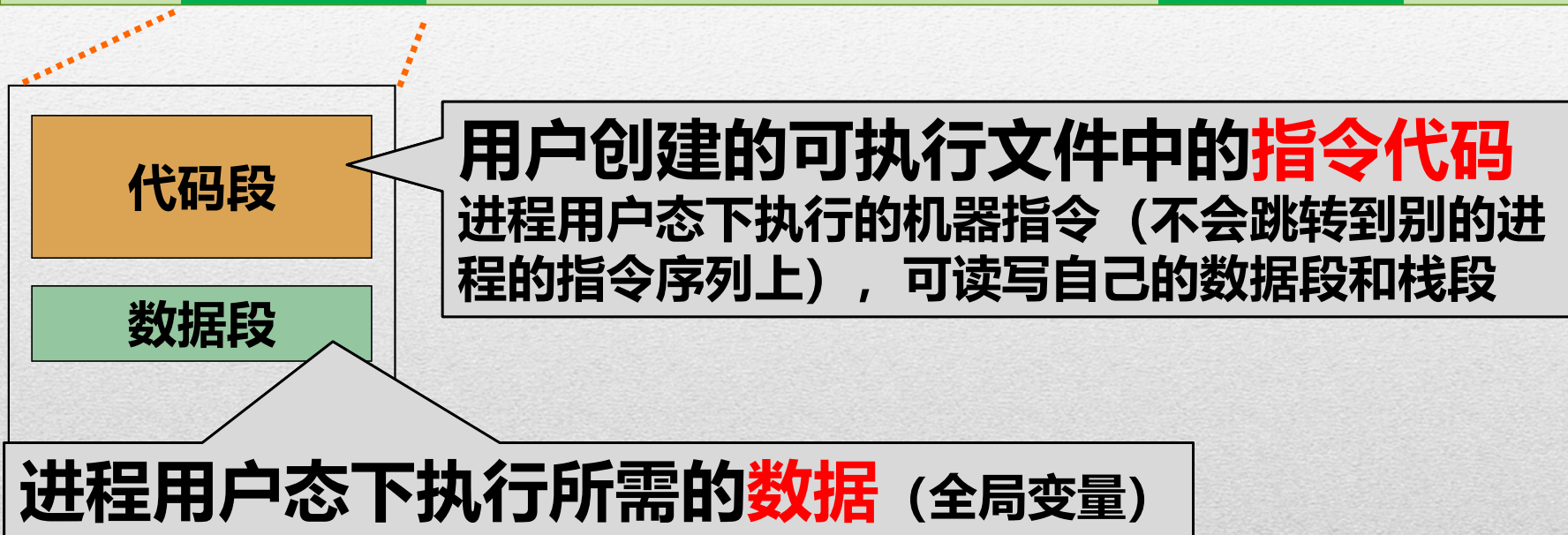
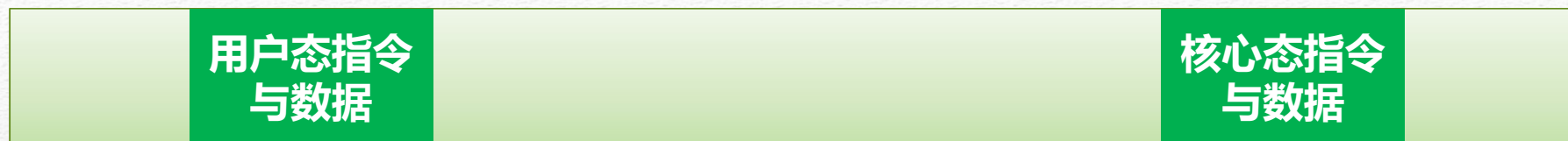
# UNIX的可执行文件



## 磁盘中的可执行文件



# UNIX进程用户态地址空间的构成



程序中所有的内容都在内存了么？





# UNIX进程用户态地址空间的构成

```
#include <fcntl.h>
char buffer[2048];
int version = 1;
main( argc, argv)
{
    int argc;
    char *argv[];
    {
        int a, b;
        .....;
        sum(a, b);
        exit(0);
    }
    int sum( var1, var2)
    {
        int var1, var2;
        {
            int count;
            count = var1 + var2;
            return(count);
        }
    }
}
```

函数调用需要传递的参数在哪里？

函数内部的局部变量在哪里？



# UNIX进程用户态地址空间的构成







# UNIX进程工作区

```
#include <fcntl.h>
char buffer[2048];
int version = 1;
main( argc, argv)
int  argc;
char *argv[];
{
    int a, b;
    .....;
    sum(a, b);
    exit(0);
}
int sum( var1, var2)
int var1, var2;
{
    int count;
    count = var1 +var2;
    return(count);
}
```

## main编译后

```
...
push    [%ebp - 8]
push    [%ebp - 4]
call    sum
add     %esp, 8
...
```

## sum编译后

```
...
push    %ebp
mov     %esp, %ebp
sub     %esp, 4
```

## 加法计算

```
mov     [%ebp-4], %eax
mov     %ebp, %esp
pop     %ebp
ret
```

编译器在调用函数和被调用函数前后自动生成一组汇编指令



# UNIX进程工作区

```
#include <fcntl.h>
char buffer[2048];
int version = 1;
main( argc, argv)
int  argc;
char *argv[];
{
    int a, b;
    .....;
    sum(a, b);
    exit(0);
}
int sum( var1, var2)
int var1, var2;
{
    int count;
    count = var1 + var2;
    return(count);
}
```

**用户栈**  
**由逻辑栈帧构成**  
每调用一个函数，  
压入一个栈帧，返回时该  
栈帧被弹出

栈顶 *esp* →

栈基址 *ebp* →

高地址





# UNIX进程工作区

```
#include <fcntl.h>
char buffer[2048];
int version = 1;
main( argc, argv)
int  argc;
char *argv[];
{
    int a, b;
    .....;
    sum(a, b);
    exit(0);
}
int sum( var1, var2)
int var1, var2;
{
    int count;
    count = var1 + var2;
    return(count);
}
```

用户栈

栈顶

esp

栈基址

ebp

局部变量 *a, b*

前一栈帧地址

main调用后的返回地址

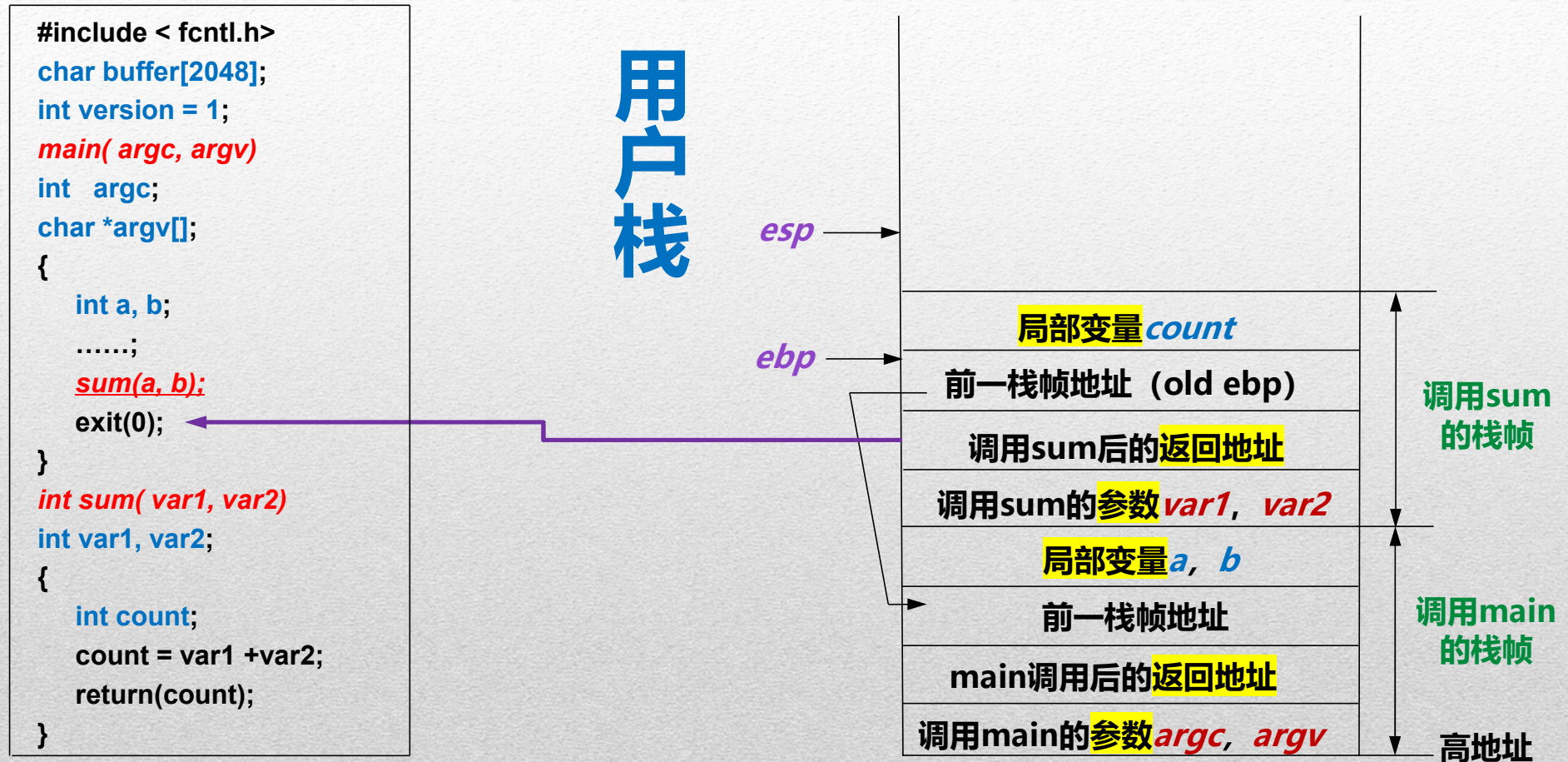
调用main的参数 *argc, argv*

调用main的栈帧

高地址

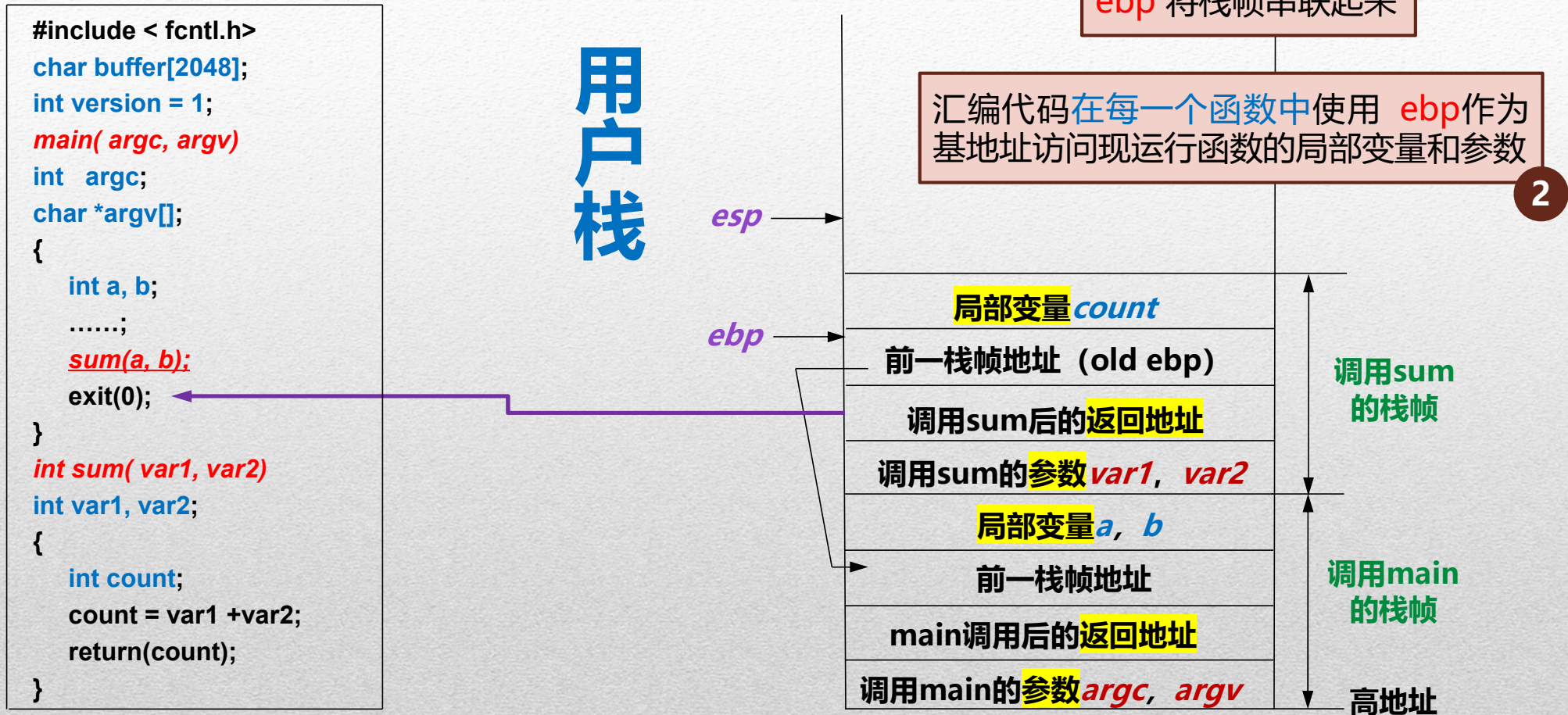


# UNIX进程工作区





# UNIX进程工作区



# UNIX进程工作区

```
#include <fcntl.h>
char buffer[2048];
int version = 1;
main( argc, argv)
int  argc;
char *argv[];
{
    int a, b;
    .....;
    sum(a, b);
    exit(0);
}
int sum( var1, var2)
int var1, var2;
{
    int count;
    count = var1 + var2;
    return(count);
}
```

## 用户栈

进程在用户态曾经走过的路，  
离开时必须原路返回

esp

ebp

局部变量 *count*

前一栈帧地址 (old ebp)

调用sum后的返回地址

调用sum的参数 *var1*, *var2*

局部变量 *a*, *b*

前一栈帧地址

main调用后的返回地址

调用main的参数 *argc*, *argv*

调用sum  
的栈帧

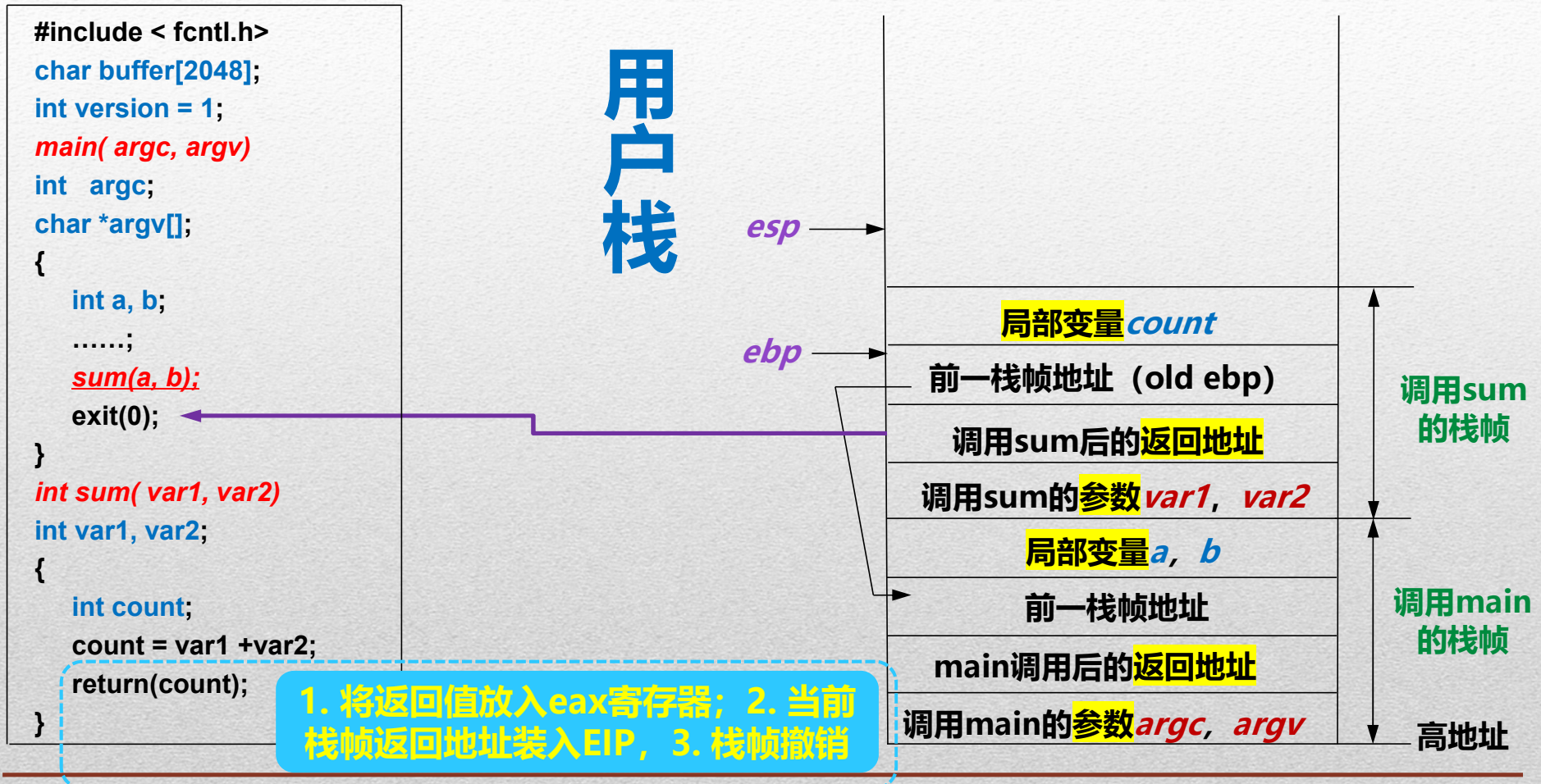
调用main  
的栈帧

高地址





# UNIX进程工作区

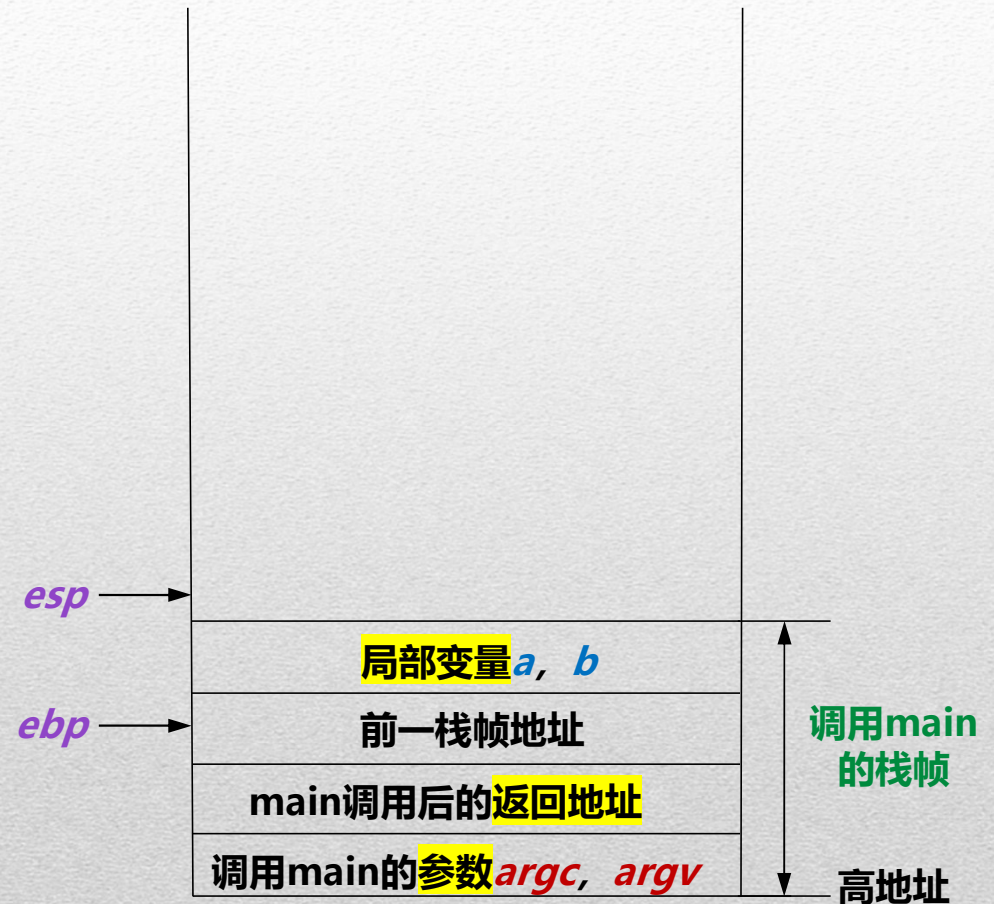




# UNIX进程工作区

```
#include <fcntl.h>
char buffer[2048];
int version = 1;
main( argc, argv)
int  argc;
char *argv[];
{
    int a, b;
    .....;
    sum(a, b);
    exit(0);
}
int sum( var1, var2)
int var1, var2;
{
    int count;
    count = var1 + var2;
    return(count);
}
```

用户栈







# UNIX进程工作区

```
#include <fcntl.h>
char buffer[2048];
int version = 1;
main( argc, argv)
int  argc;
char *argv[];
{
    int a, b;
    .....;
    sum(a, b);
    exit(0);
}
int sum( var1, var2)
int var1, var2;
{
    int count;
    count = var1 + var2;
    return(count);
}
```

用户栈

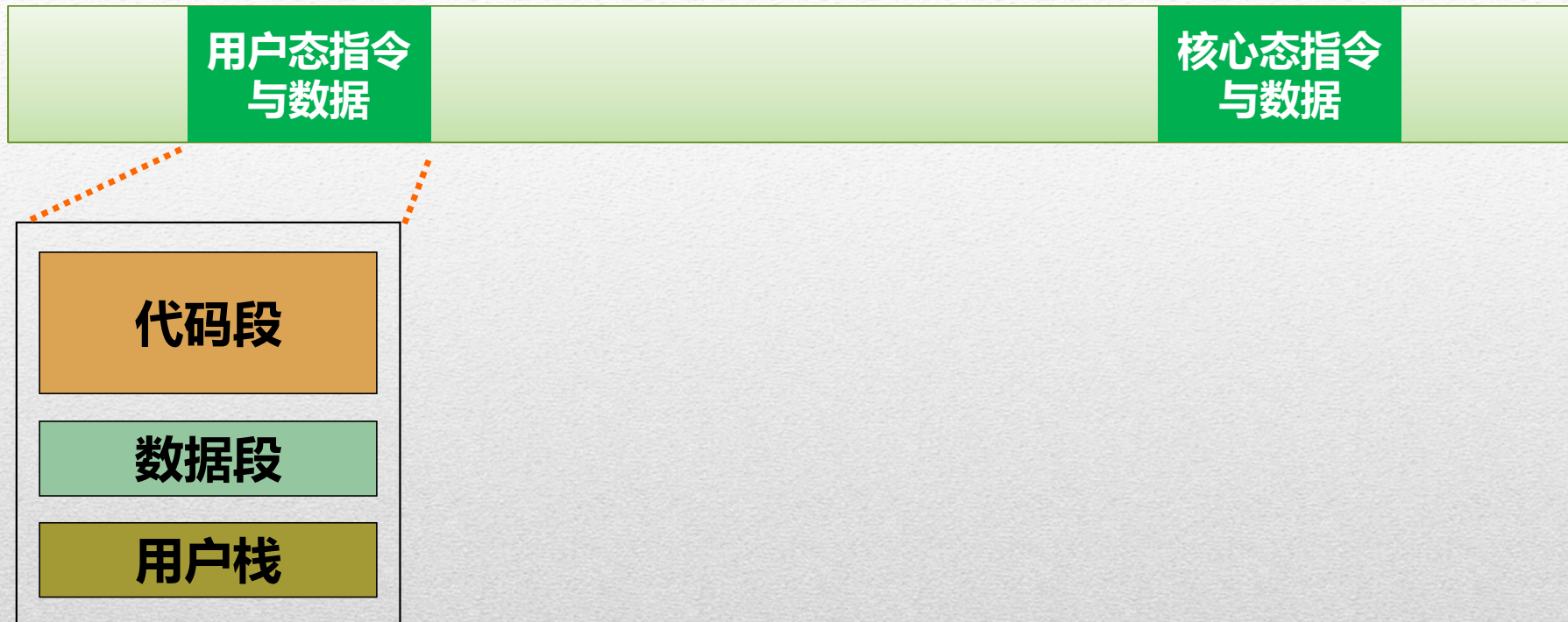
esp →

ebp →

高地址

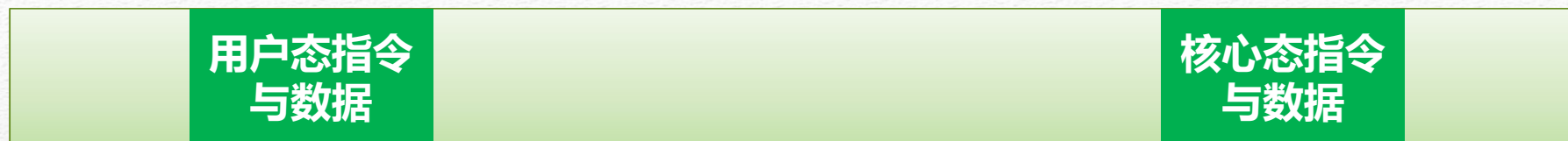


# UNIX中进程用户态地址空间的构成





# UNIX中进程用户态地址空间的构成



程序



进程

一组数据与指令代码的集合

结构特征  
代码段、数据段、堆栈段、**进程控制块**

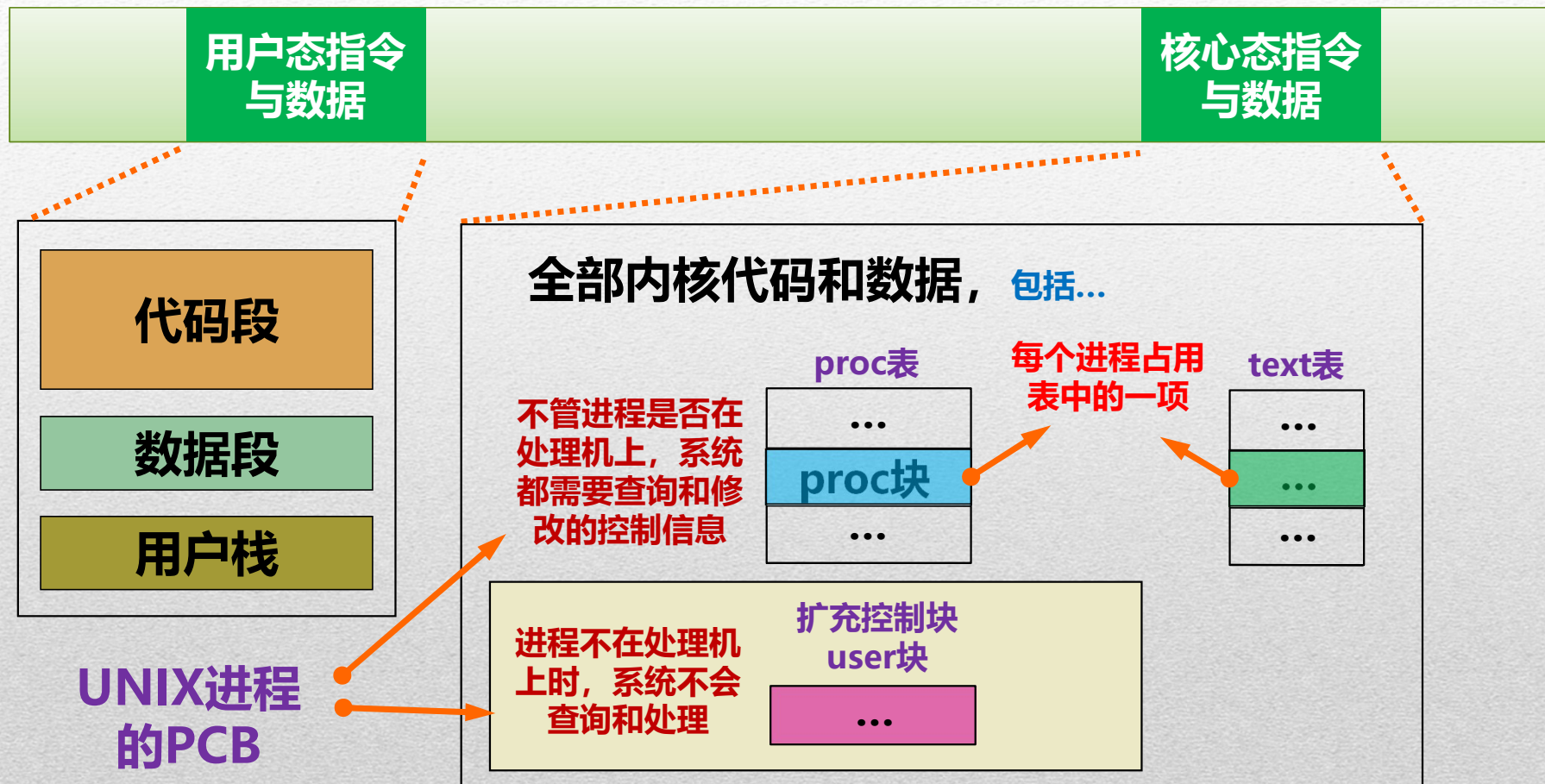
程序执行之前变身为进程

进程在用户态下运行时的**工作区**，系统自动创建，运行过程中可由内核动态调节

当进程运行在核心态呢？？？



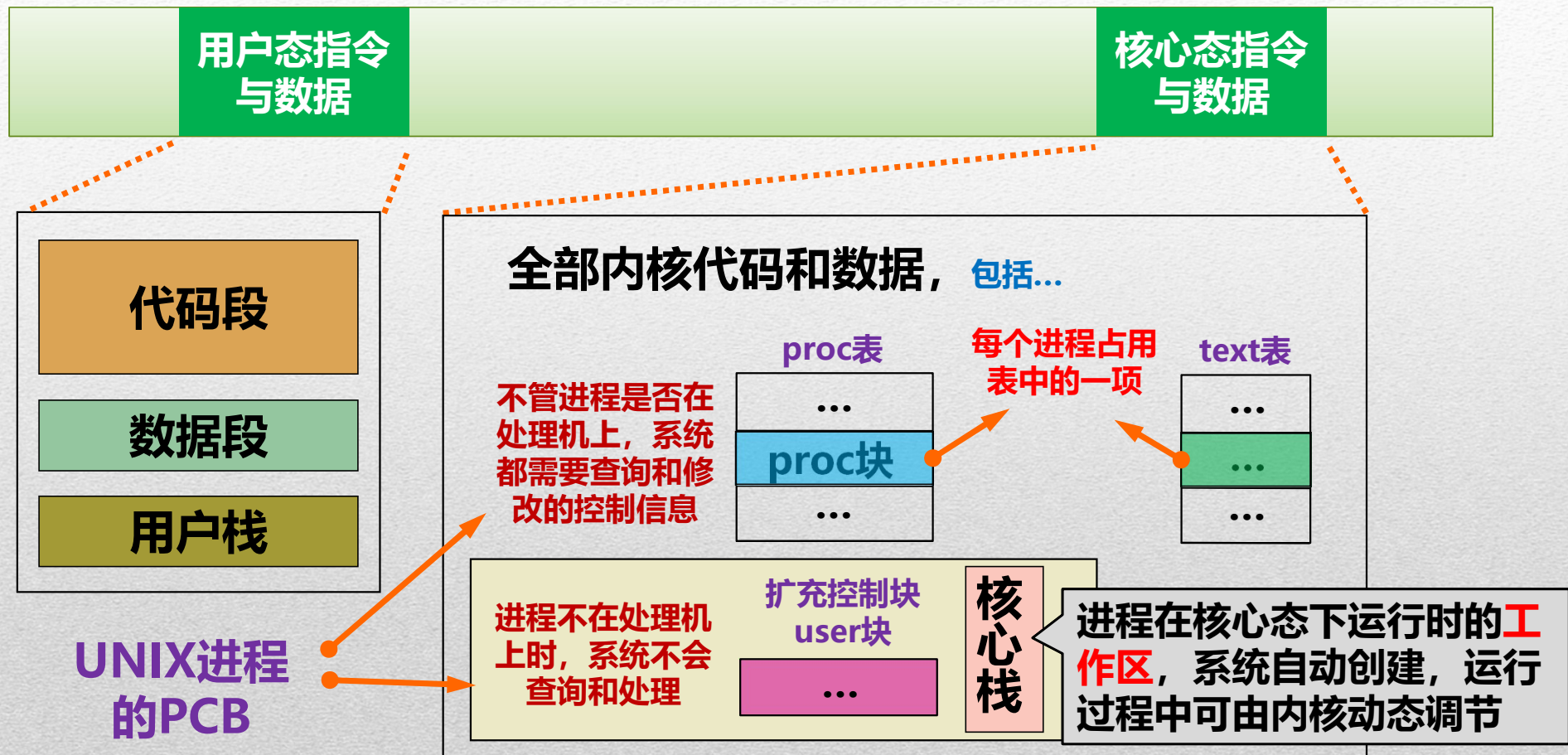
# UNIX中进程核心态地址空间的构成





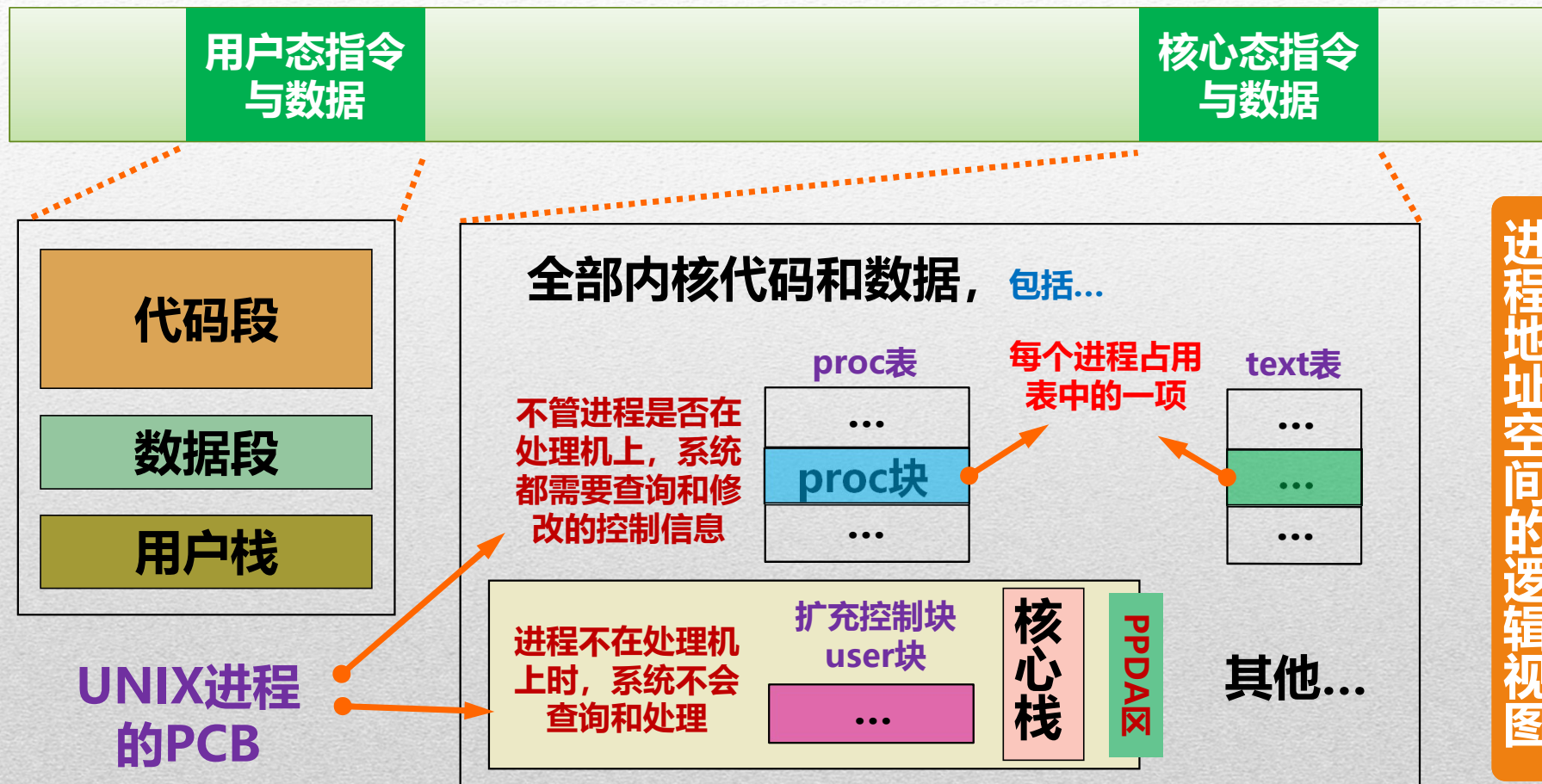


# UNIX中进程核心态地址空间的构成



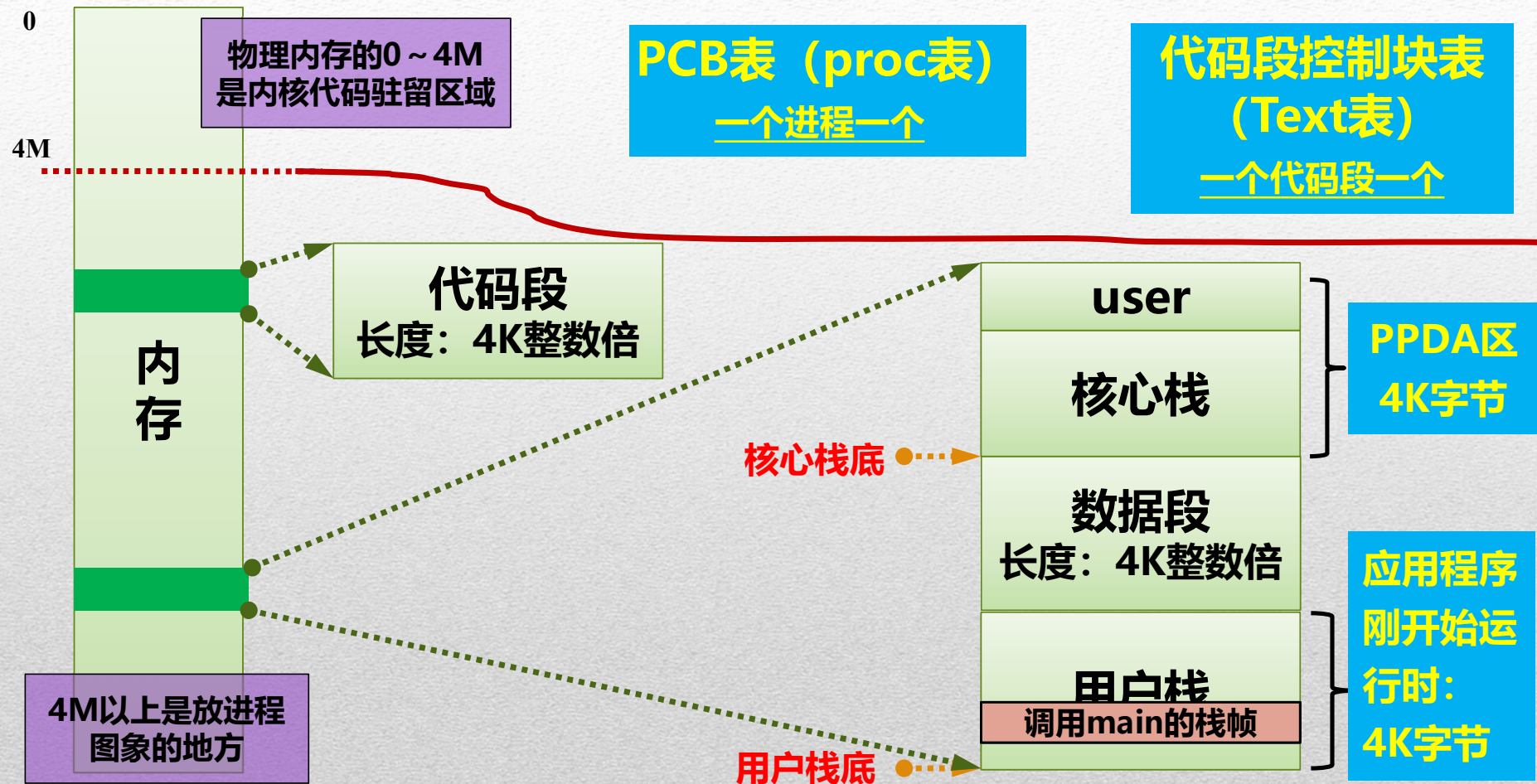


# UNIX中进程核心态地址空间的构成





## UNIX中进程的构成 (进程图象)

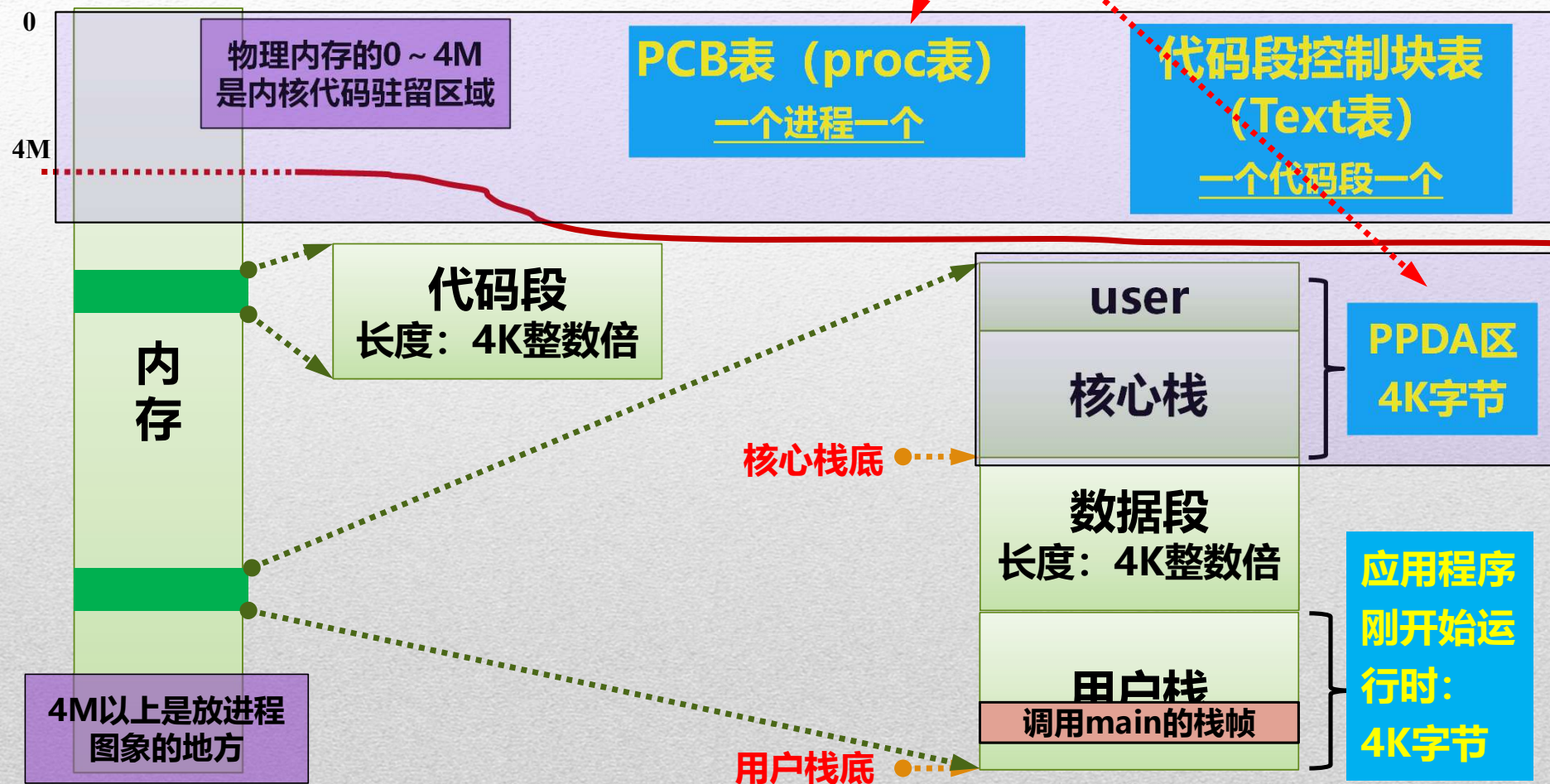




# UNIX中进程的构成 (进程图家)

核心态地址空间

进程地址空间的物理视图

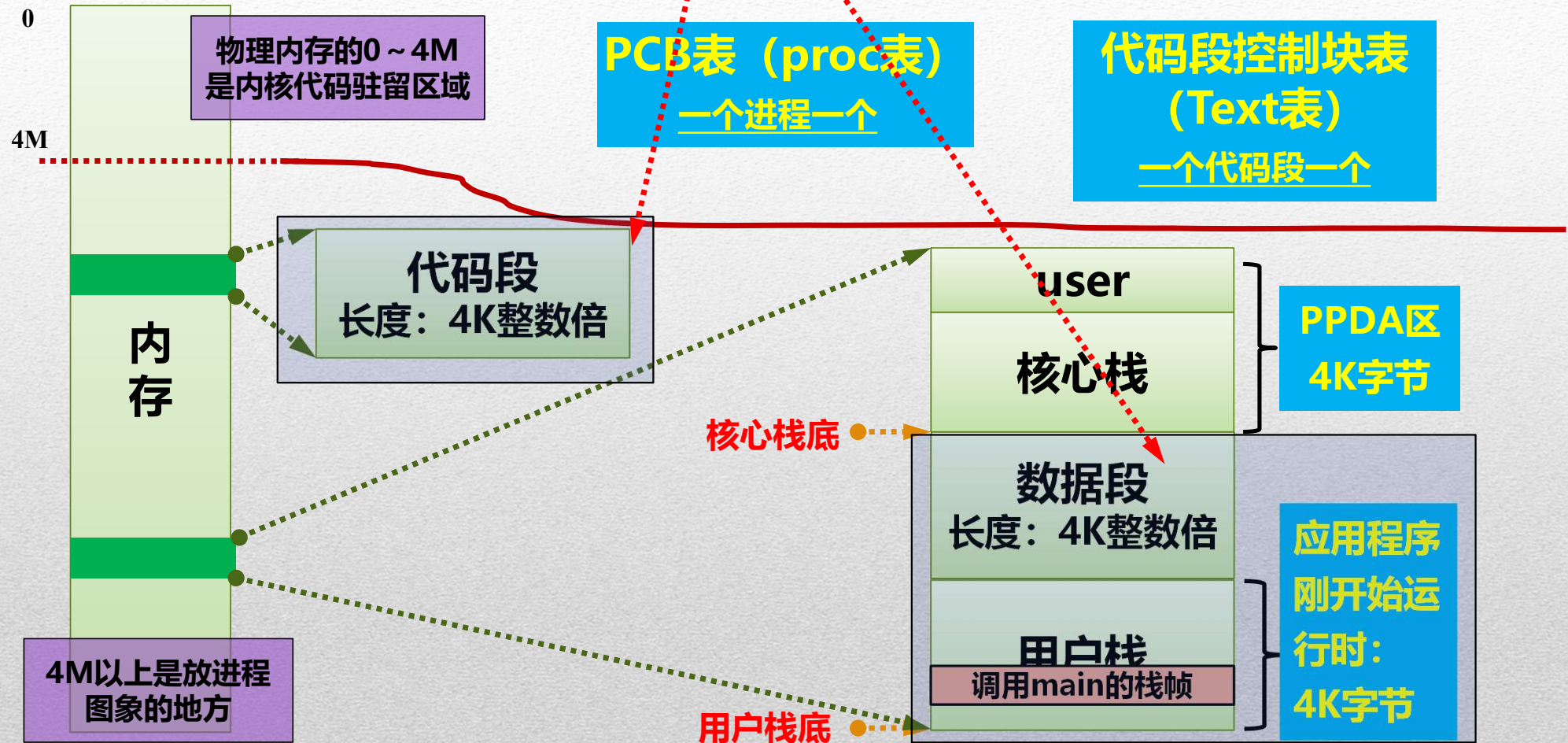




# UNIX中进程的构成

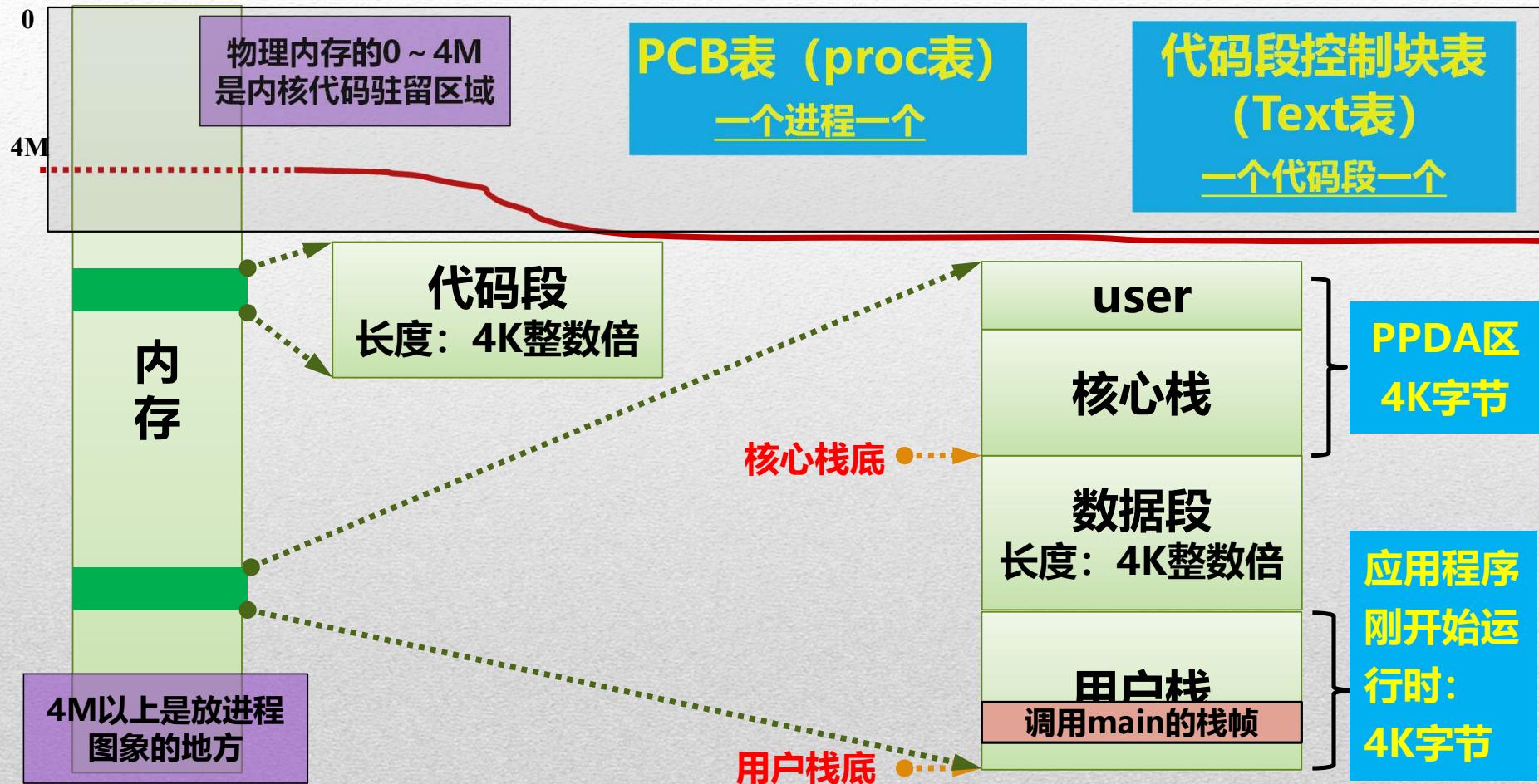
用户态地址空间  
(进程图家)

进程地址空间的物理视图



# UNIX中进程的构成 (进程图景)

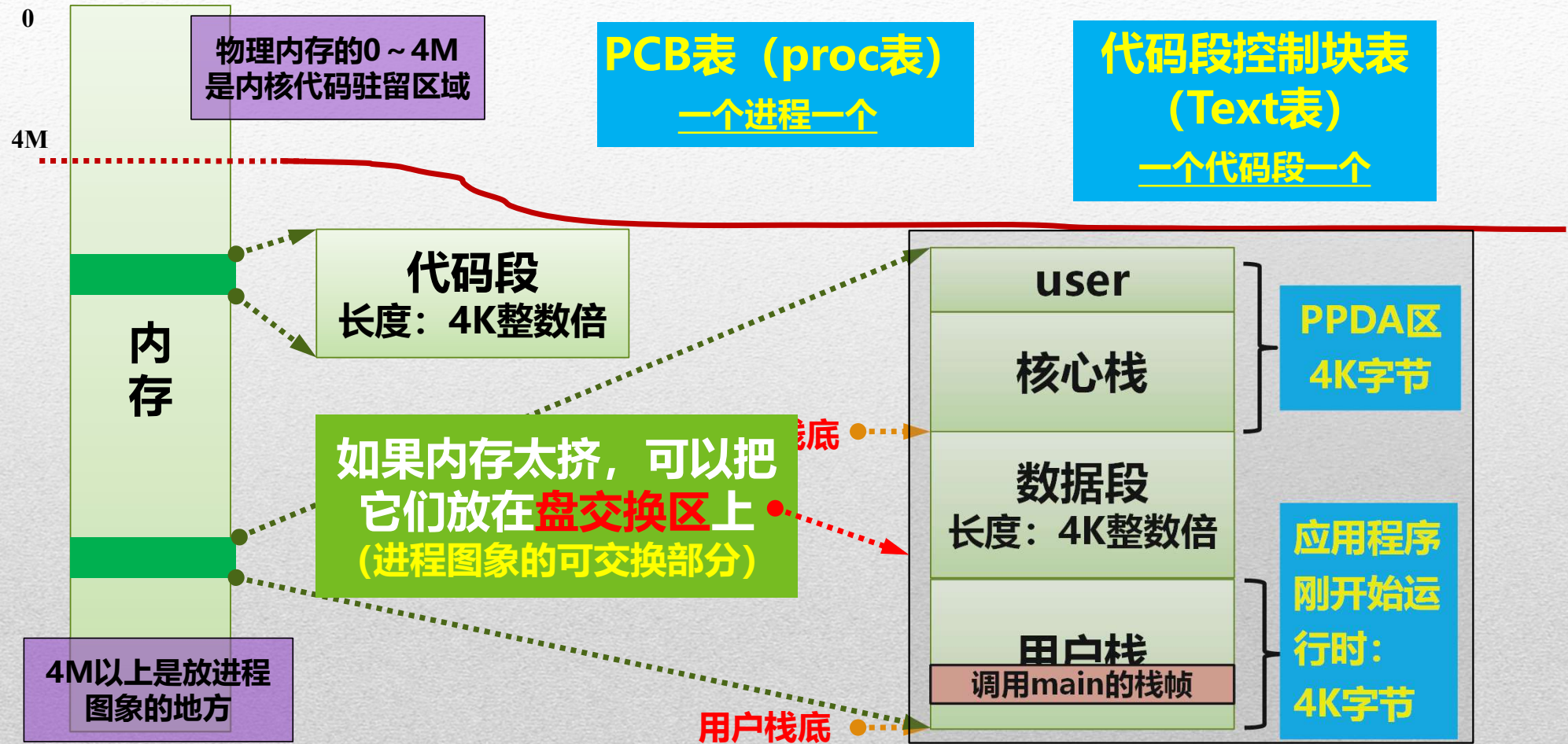
## 进程地址空间的物理视图





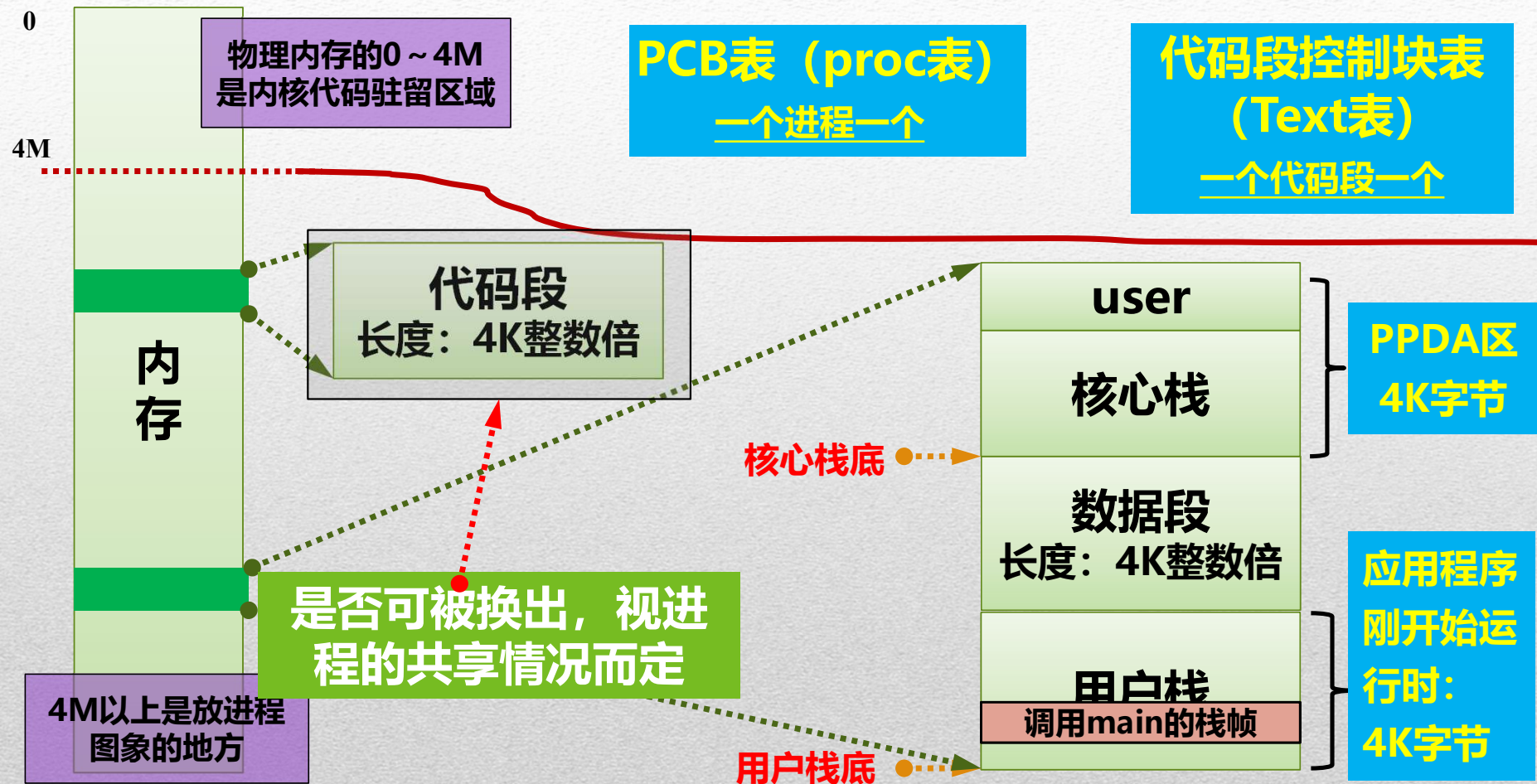
# UNIX中进程的构成 (进程图象)

## 进程地址空间的物理视图



# UNIX中进程的构成 (进程图象)

## 进程地址空间的物理视图







## 本节小结:

- 1 UNIX进程的两个执行状态及两个地址空间
- 2 UNIX进程的进程图象
- 3 C语言函数调用与返回过程



# 栈的形成过程





## C语言编译器对函数调用的处理方式

```
void main()
{
    int a,b;
    a = 16;
    b = 32;
    a = sum(a, b);
}
```

...

```
push    [%ebp - 8]
push    [%ebp - 4]
call     sum
add     %esp, 8
...
```

编译后

编译后

编译器在调用函数和被调用函数前后  
自动生成一组汇编指令

```
int sum(int var1, int var2)
{
    int c;
    c = var1 + var2;
    return(c);
}
```

...

```
push    %ebp
mov     %esp, %ebp
sub     %esp, 4
加法计算
mov     [%ebp-4], %eax
mov     %ebp, %esp
pop     %ebp
ret
...
```



## C语言编译器对函数调用的处理方式

**main**

```
...  
L-2:  push [%ebp - 8]  
L-1:  push [%ebp - 4]  
L:    call  sum  
L+1:  add  %esp, 8  
...
```

**sum**

```
...  
T:    push %ebp  
T+1:  mov  %esp, %ebp  
T+2:  sub  %esp, 4  
...
```

```
...  
      mov  [%ebp-4], %eax  
      mov  %ebp,    %esp  
      pop  %ebp  
      ret  
...
```

**栈基址***ebp***栈顶***esp***高地址**





## C语言编译器对函数调用的处理方式

**main**

```
...  
L-2: push [%ebp - 8]  
L-1: push [%ebp - 4]  
L:   call sum  
L+1: add  %esp, 8  
...
```

**sum**

```
...  
T:   push %ebp  
T+1: mov  %esp, %ebp  
T+2: sub  %esp, 4  
...
```

```
...  
    mov [%ebp-4], %eax  
    mov %ebp,    %esp  
    pop %ebp  
    ret  
...
```

按逆序压入参数的值

ebp

调用sum的参数a的值

调用sum的参数b的值

esp  
实参区

高地址



## C语言编译器对函数调用的处理方式

**main**

```
...  
L-2:  push [%ebp - 8]  
L-1:  push [%ebp - 4]  
L:    call  sum  
L+1:  add  %esp, 8  
...
```

**sum**

```
...  
T:    push %ebp  
T+1:  mov  %esp, %ebp  
T+2:  sub  %esp, 4  
...
```

```
...  
mov  [%ebp-4], %eax  
mov  %ebp, %esp  
pop  %ebp  
ret  
...
```

将返回地址压栈, sum  
入口地址T装入EIP







## C语言编译器对函数调用的处理方式

**main**

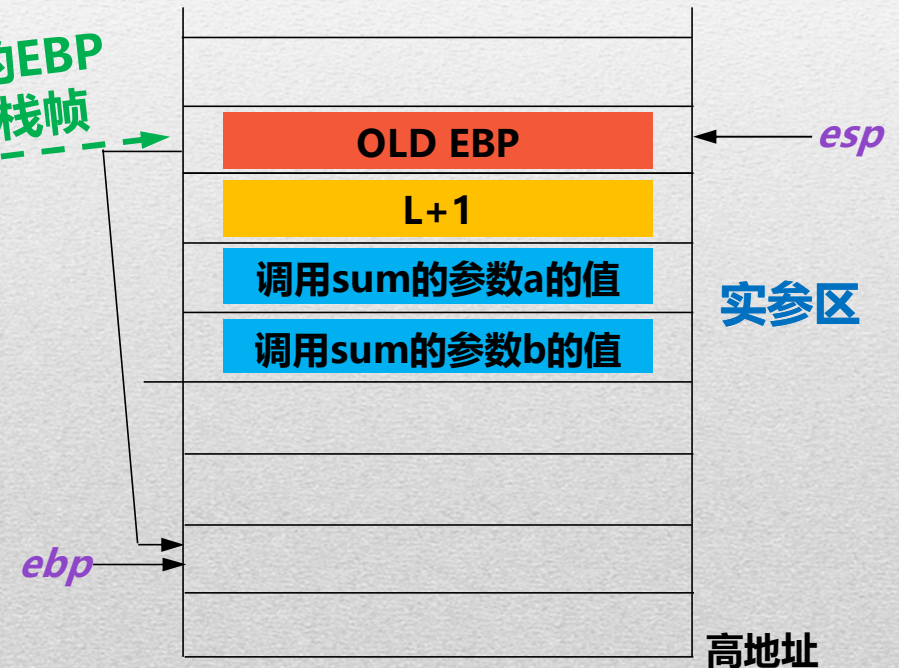
```
...  
L-2:  push [%ebp - 8]  
L-1:  push [%ebp - 4]  
L:    call  sum  
L+1:  add  %esp, 8  
...
```

**sum**

```
...  
T:    push %ebp  
T+1:  mov  %esp, %ebp  
T+2:  sub  %esp, 4  
...
```

```
...  
mov  [%ebp-4], %eax  
mov  %ebp, %esp  
pop  %ebp  
ret  
...
```

前一栈帧的EBP  
存入当前栈帧





## C语言编译器对函数调用的处理方式

**main**

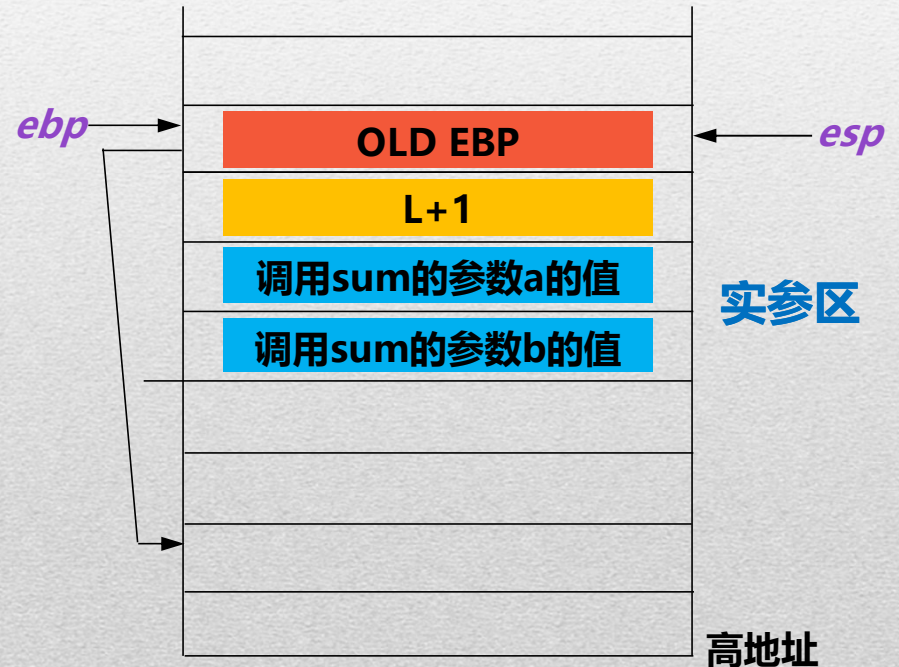
```
...  
L-2:  push [%ebp - 8]  
L-1:  push [%ebp - 4]  
L:    call  sum  
L+1:  add  %esp, 8  
...
```

**sum**

```
...  
T:    push %ebp  
T+1:  mov  %esp, %ebp  
T+2:  sub  %esp, 4  
...
```

```
...  
      mov  [%ebp-4], %eax  
      mov  %ebp,    %esp  
      pop  %ebp  
      ret  
...
```

修改EBP指向  
当前栈帧







## C语言编译器对函数调用的处理方式

**main**

```
...  
L-2:  push [%ebp - 8]  
L-1:  push [%ebp - 4]  
L:    call  sum  
L+1:  add  %esp, 8  
...
```

**sum**

```
...  
T:    push %ebp  
T+1:  mov  %esp, %ebp  
T+2:  sub  %esp, 4  
...
```

```
...  
mov  [%ebp-4], %eax  
mov  %ebp, %esp  
pop  %ebp  
ret  
...
```

为局部变量预留存储空间

ebp

sum的局部变量c

OLD EBP

L+1

调用sum的参数a的值

调用sum的参数b的值

esp

实参区

高地址



# C语言编译器对函数调用的处理方式

**main**

```
...  
L-2: push [%ebp - 8]  
L-1: push [%ebp - 4]  
L:   call sum  
L+1: add  %esp, 8  
...
```

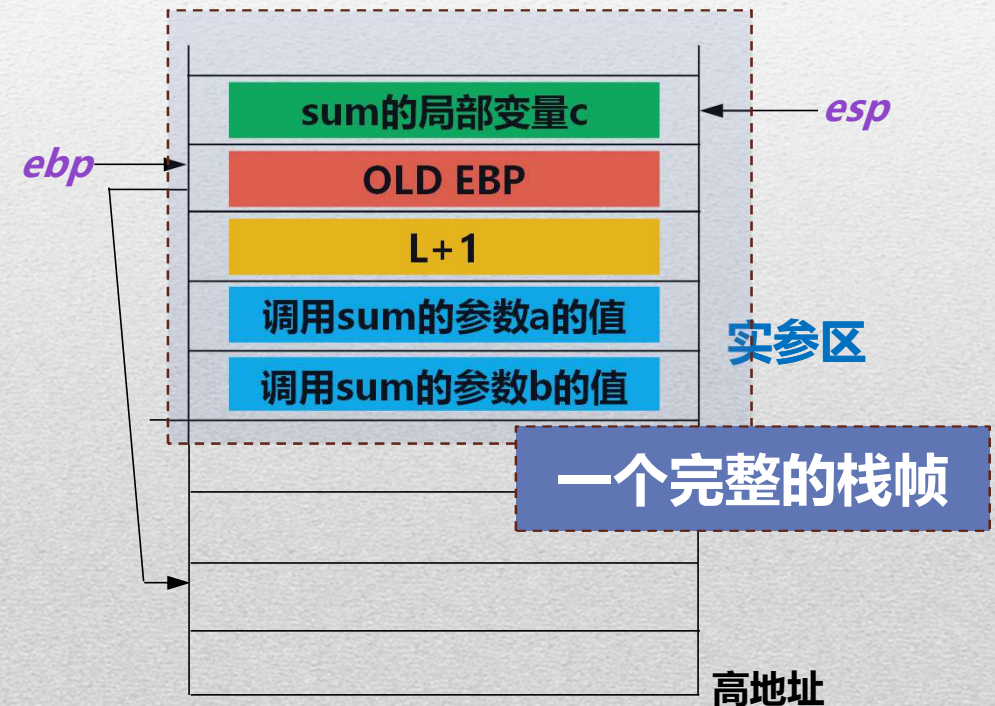
**sum**

```
...  
T:   push %ebp  
T+1: mov  %esp, %ebp  
T+2: sub  %esp, 4  
...
```

```
...  
mov  [%ebp-4], %eax  
mov  %ebp,    %esp  
pop  %ebp  
ret  
...
```

**1**

ebp 将栈帧串联起来







## C语言编译器对函数调用的处理方式

main

```

...
L-2:  push [%ebp - 8]
L-1:  push [%ebp - 4]
L:    call  sum
L+1:  add  %esp, 8
...

```

sum

```

...
T:    push %ebp
T+1:  mov  %esp, %ebp
T+2:  sub  %esp, 4
...

```

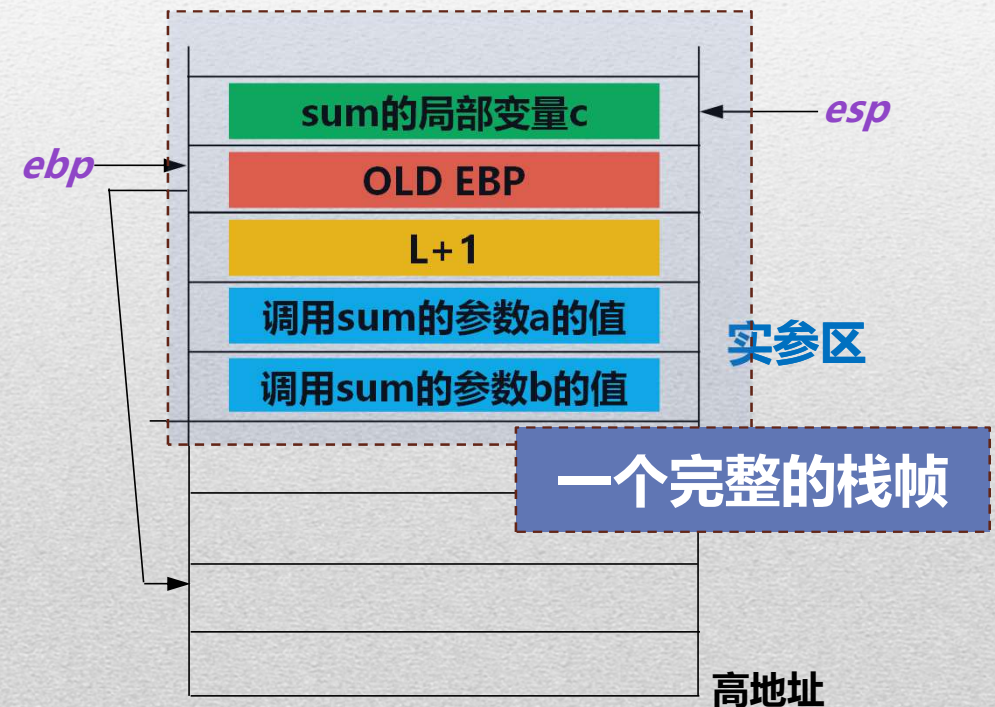
```

...
mov  [%ebp-4], %eax
mov  %ebp, %esp
pop  %ebp
ret
...

```

汇编代码在每一个函数中使用  $[\%ebp - 4]$ 、 $[\%ebp - 8]$ 、.....访问现运行函数的第1个、第2个、.....整形局部变量；使用  $[\%ebp + 8]$ 、 $[\%ebp + 12]$ 、.....访问现运行函数的第1个、第2个、.....整形参数。

2





## C语言编译器对函数调用的处理方式

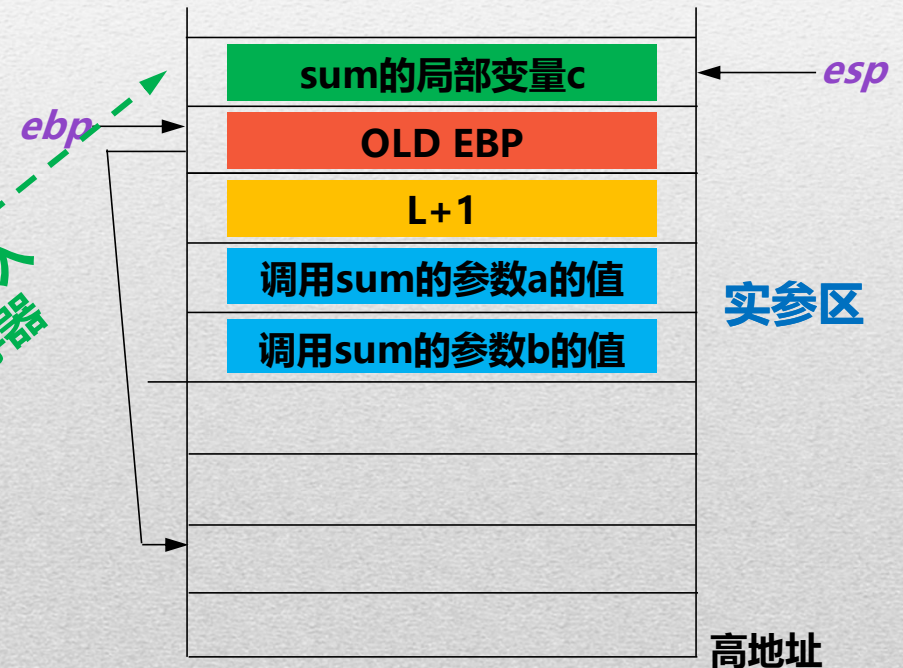
**main**

```
...  
L-2: push [%ebp - 8]  
L-1: push [%ebp - 4]  
L:   call sum  
L+1: add %esp, 8  
...
```

**sum**

```
...  
T:   push %ebp  
T+1: mov %esp, %ebp  
T+2: sub %esp, 4  
...  
mov [%ebp-4], %eax  
mov %ebp, %esp  
pop %ebp  
ret  
...
```

把返回值存入  
eax寄存器







# C语言编译器对函数调用的处理方式

main

```

...
L-2:  push [%ebp - 8]
L-1:  push [%ebp - 4]
L:    call  sum
L+1:  add  %esp, 8
...

```

sum

```

...
T:    push %ebp
T+1:  mov  %esp, %ebp
T+2:  sub  %esp, 4
...

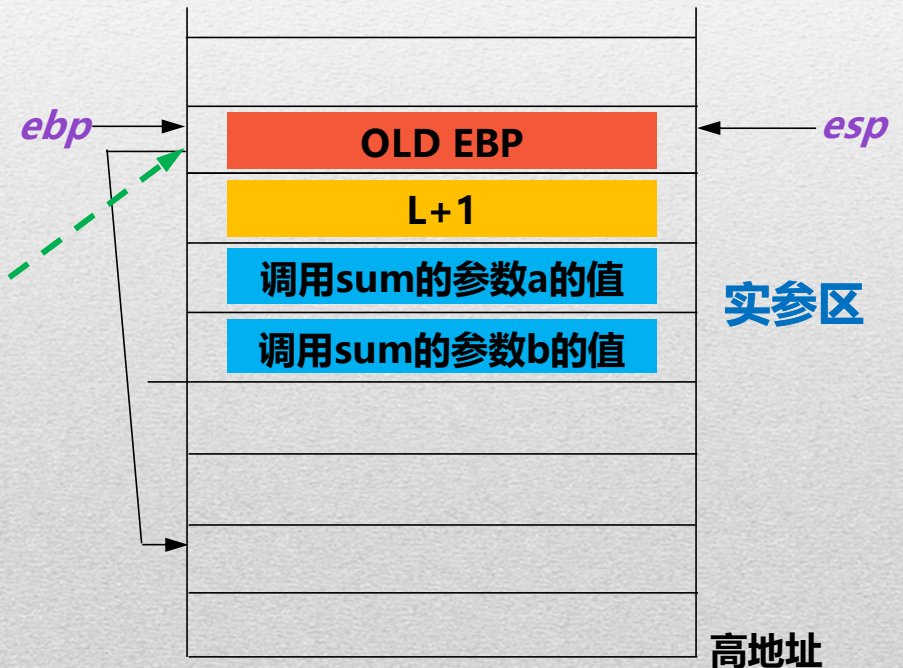
```

```

...
mov  [%ebp-4], %eax
mov  %ebp, %esp
pop  %ebp
ret
...

```

删除局部  
变量区





# C语言编译器对函数调用的处理方式

main

```

...
L-2:  push [%ebp - 8]
L-1:  push [%ebp - 4]
L:    call  sum
L+1:  add  %esp, 8
...

```

sum

```

...
T:    push %ebp
T+1:  mov  %esp, %ebp
T+2:  sub  %esp, 4
...

```

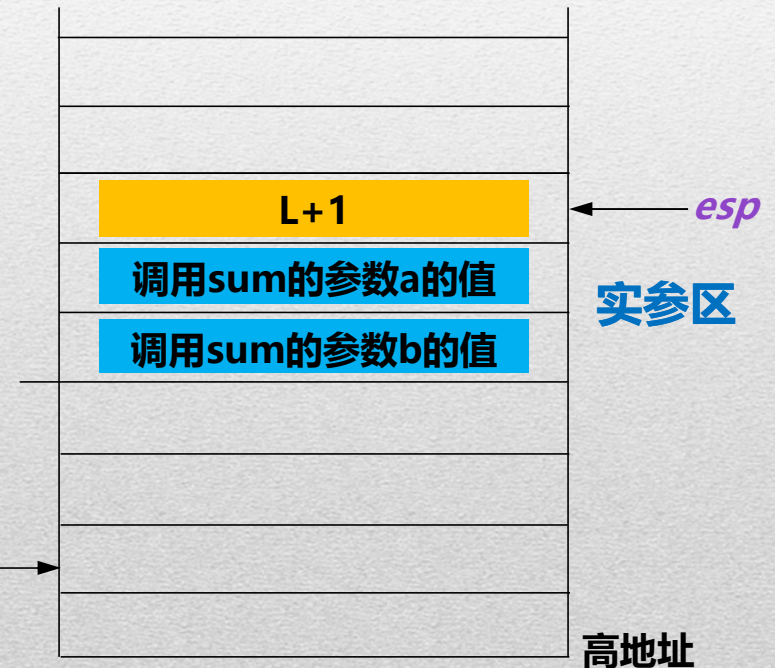
```

...
mov  [%ebp-4], %eax
mov  %ebp,    %esp
pop  %ebp
ret
...

```

恢复前一  
帧ebp

ebp







## C语言编译器对函数调用的处理方式

**main**

```
...  
L-2:  push [%ebp - 8]  
L-1:  push [%ebp - 4]  
L:    call  sum  
L+1:  add  %esp, 8  
...
```

**sum**

```
...  
T:    push %ebp  
T+1:  mov  %esp, %ebp  
T+2:  sub  %esp, 4  
...
```

```
...  
mov  [%ebp-4], %eax  
mov  %ebp, %esp  
pop  %ebp  
ret  
...
```

从栈顶弹出返回地址装入EIP

ebp

调用sum的参数a的值

调用sum的参数b的值

← esp  
实参区

高地址



## C语言编译器对函数调用的处理方式

main

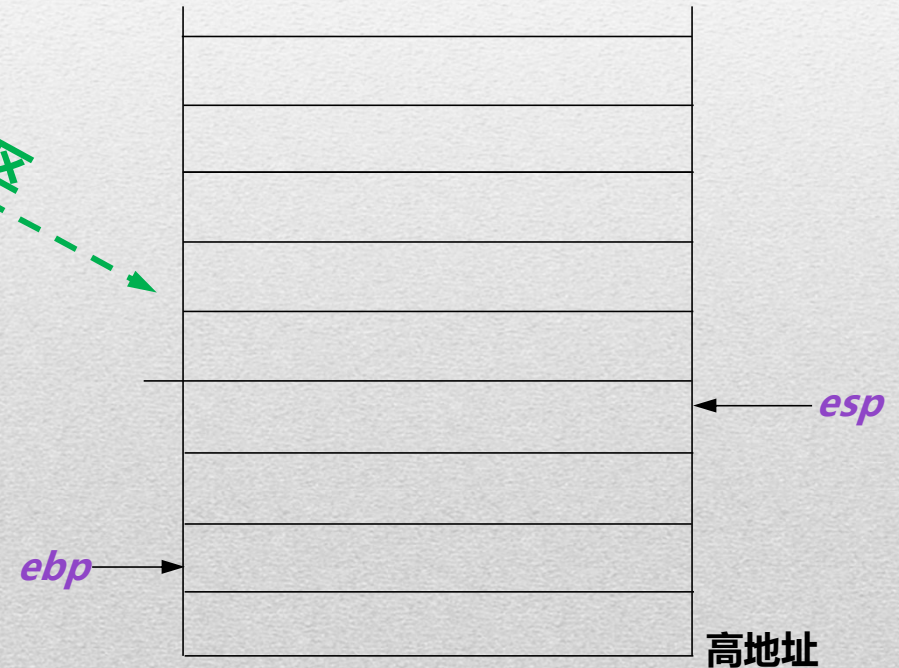
```
...  
L-2:  push [%ebp - 8]  
L-1:  push [%ebp - 4]  
L:    call  sum  
L+1:  add  %esp, 8  
...
```

sum

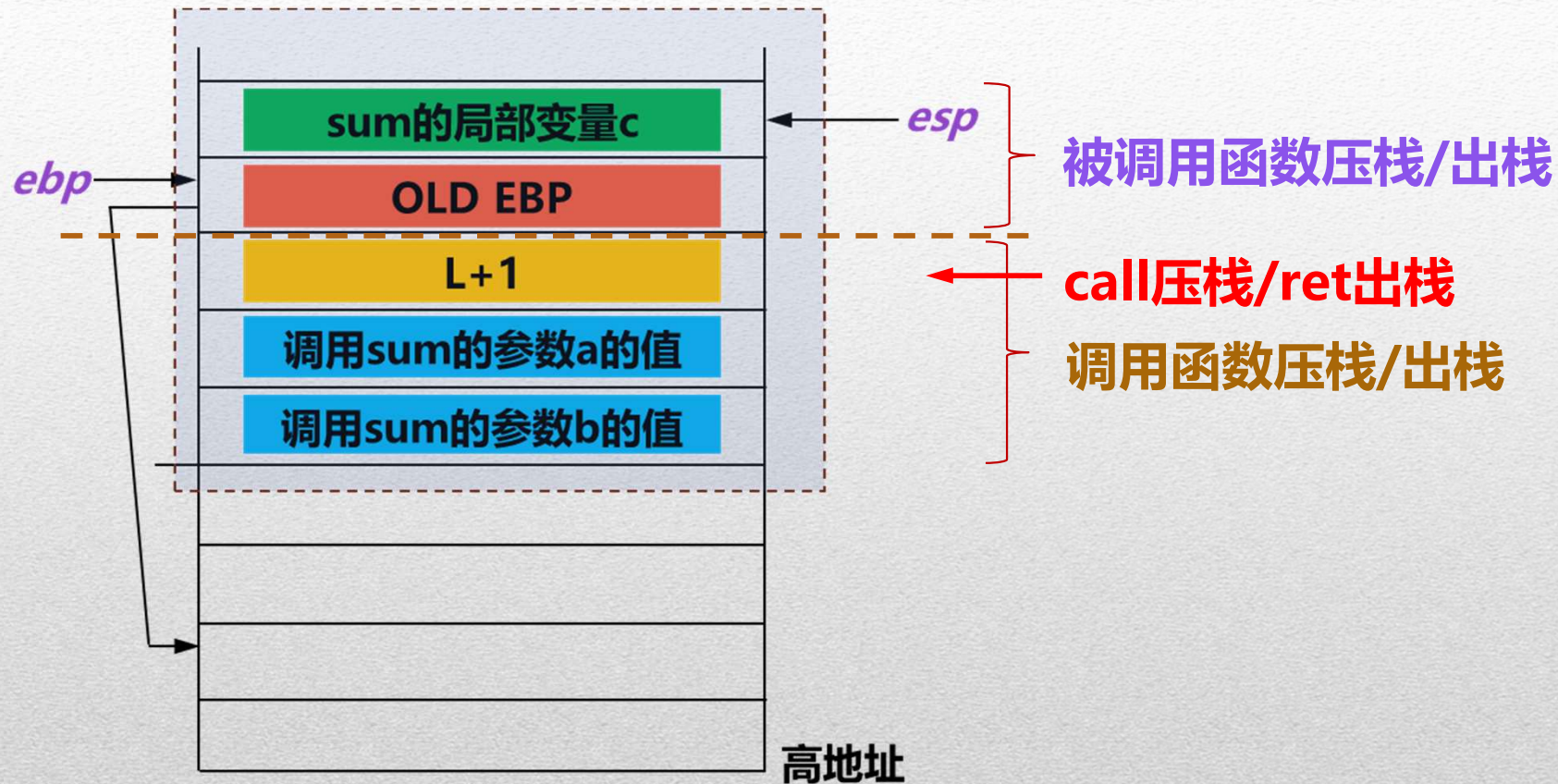
```
...  
T:    push %ebp  
T+1:  mov  %esp, %ebp  
T+2:  sub  %esp, 4  
...
```

```
...  
mov  [%ebp-4], %eax  
mov  %ebp, %esp  
pop  %ebp  
ret  
...
```

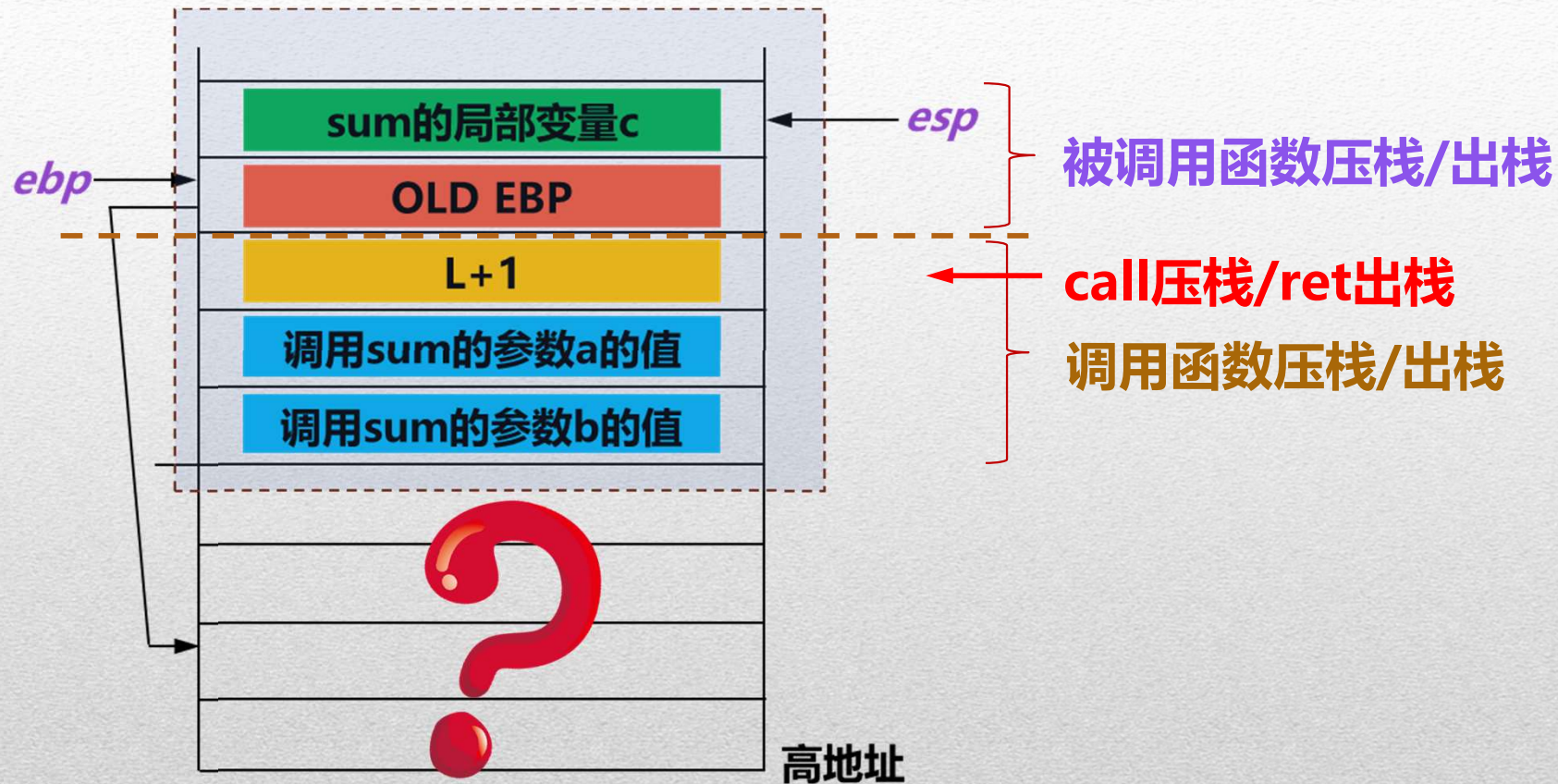
删除实参区















# C语言编译器对函数调用的处理方式

**main**

```

...
L-2:  push [%ebp - 8]
L-1:  push [%ebp - 4]
L:    call  sum
L+1:  add  %esp, 8
...
  
```

**sum**

```

...
T:    push %ebp
T+1:  mov  %esp, %ebp
T+2:  sub  %esp, 4
...
      mov  [%ebp-4], %eax
      mov  %ebp,   %esp
      pop  %ebp
      ret
...
  
```

```

void main()
{
    int a,b;
    a = 16;
    b = 32;
    a = sum(a, b);
}
  
```

```

int sum(int var1, int var2)
{
    int c;
    c = var1 + var2;
    return(c);
}
  
```

栈基址

*ebp*

栈顶

*esp*

main的局部变量

前一帧的ebp

执行main后的返回地址

调用main的实参

高地址



# C语言编译器对函数调用的处理方式

**main**

```

...
L-2: push [%ebp - 8]
L-1: push [%ebp - 4]
L:   call sum
L+1: add  %esp, 8
...

```

$[\%ebp - 8]$ ,  $[\%ebp - 4]$ 分别是main的局部变量b和a的值

**sum**

```

...
T:   push %ebp
T+1: mov  %esp, %ebp
T+2: sub  %esp, 4
...

```

```

...
mov  [%ebp-4], %eax
mov  %ebp,    %esp
pop  %ebp
ret
...

```

```

void main()
{
    int a,b;
    a = 16;
    b = 32;
    a = sum(a, b);
}

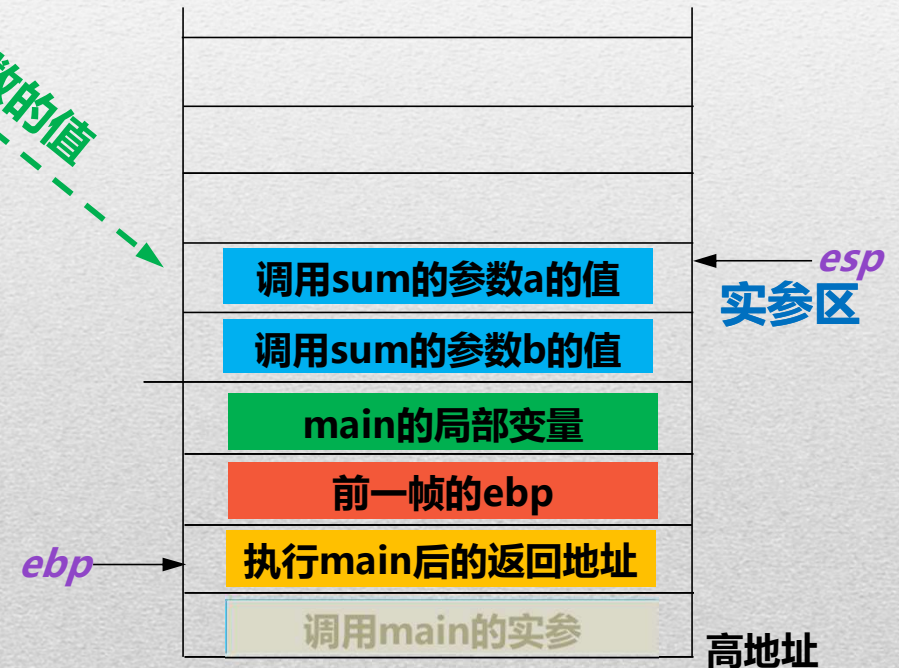
```

```

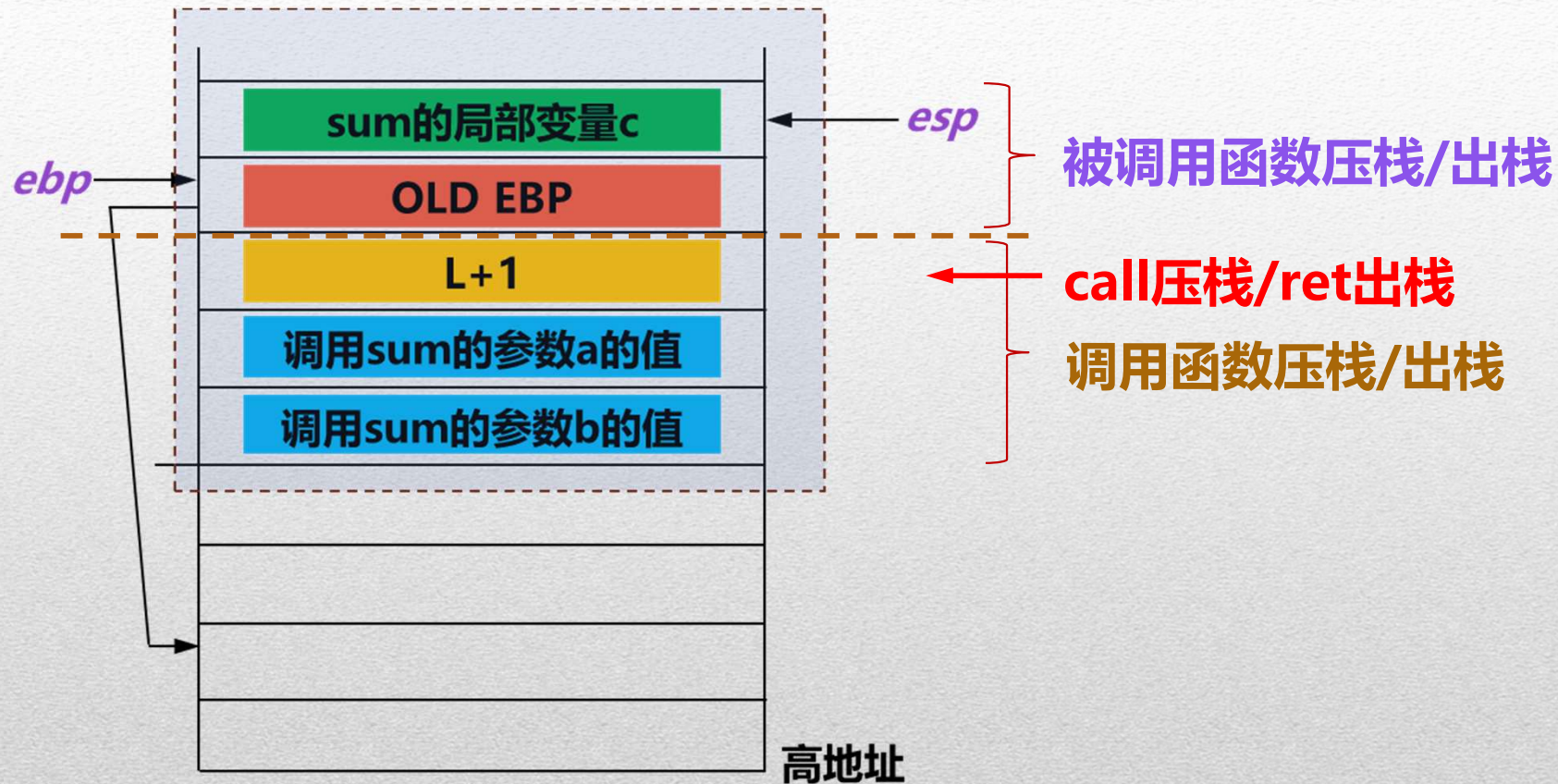
int sum(int var1, int var2)
{
    int c;
    c = var1 + var2;
    return(c);
}

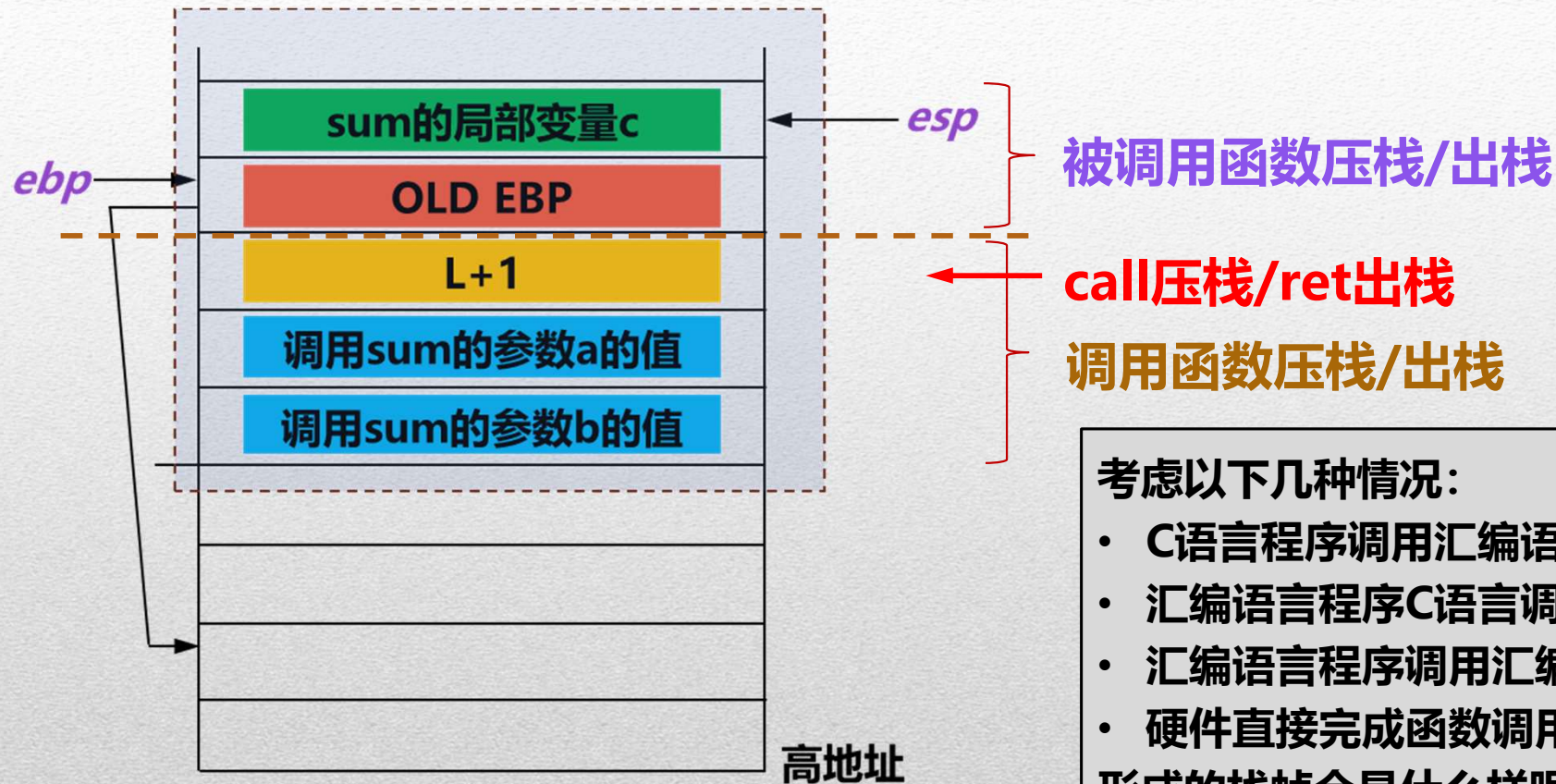
```

按逆序压入参数的值









考虑以下几种情况:

- C语言程序调用汇编语言程序
  - 汇编语言程序C语言调用程序
  - 汇编语言程序调用汇编语言程序
  - 硬件直接完成函数调用
- 形成的栈帧会是什么样呢?