第二章

并发进程

方　钰

主 要 内 容

2.1 进程基本概念

2.2 UNIX的进程

2.3 中断的基本概念及UNIX中断处理

2.4 处理机调度与死锁

2.5 进程通信机制

UNIX中进程的两种执行状态

用户态
User Mode

执行用户程序，
提供用户功能

核心态
Kernel Mode

执行内核程序，
提供系统功能

可以在一定时机相互转换

内核不是与用户进程平行运行的孤立的进程集合。

在核心态下执行内核代码的进程完成了内核功能！

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”

进程是程序的一次运行过程!!!

- 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！！！

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

文件头

Text段 1K字节

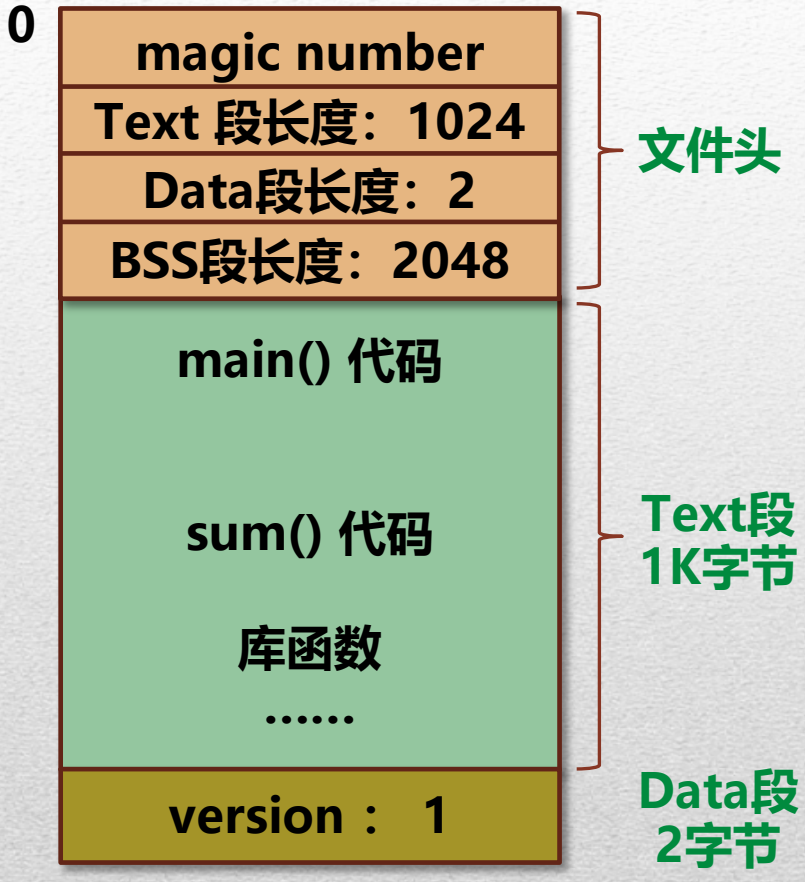
Data段 2字节

可执行程序被保存在磁盘上

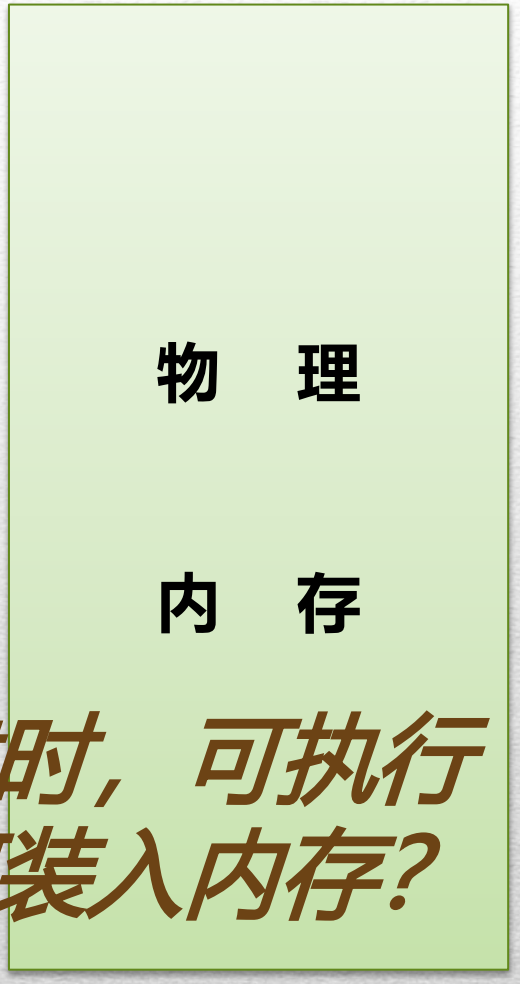
Data段和BSS段可能因为链接的其他文件中的全局变量而更长

经过编译、链接形成可执行程序

看下面这个简单的程序 ⇒ UNIX的可执行文件

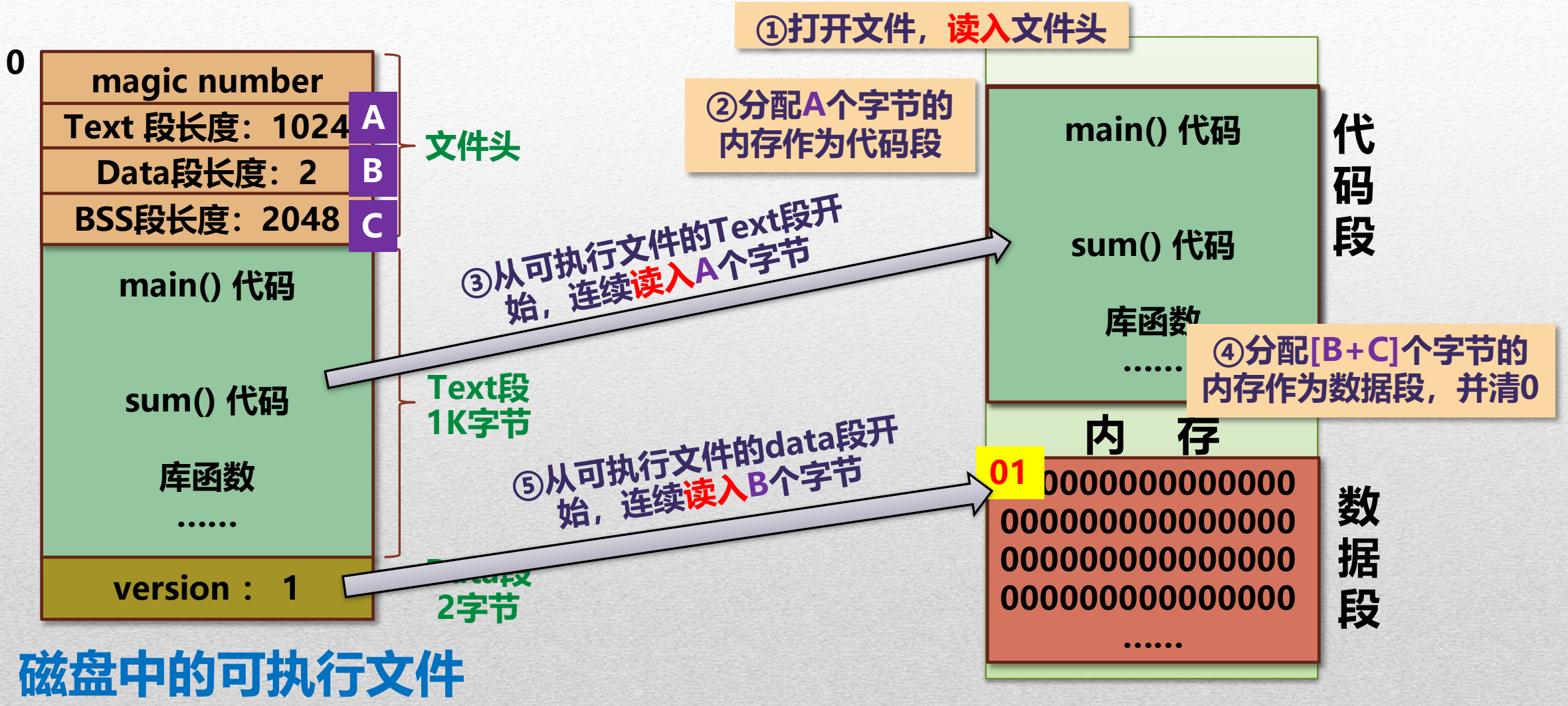


磁盘中的可执行文件

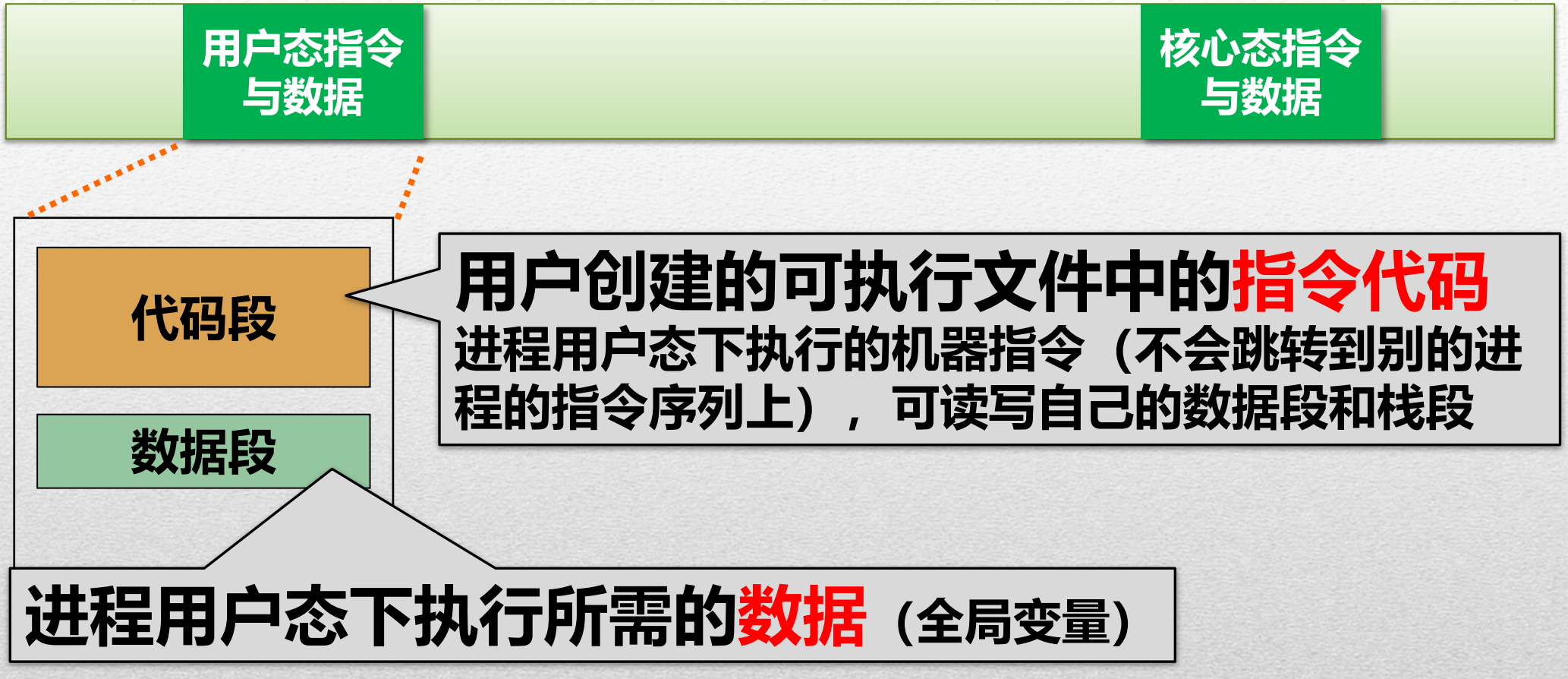


进程创建时，可执行文件如何装入内存?

看下面这个简单的程序 ⇒ UNIX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

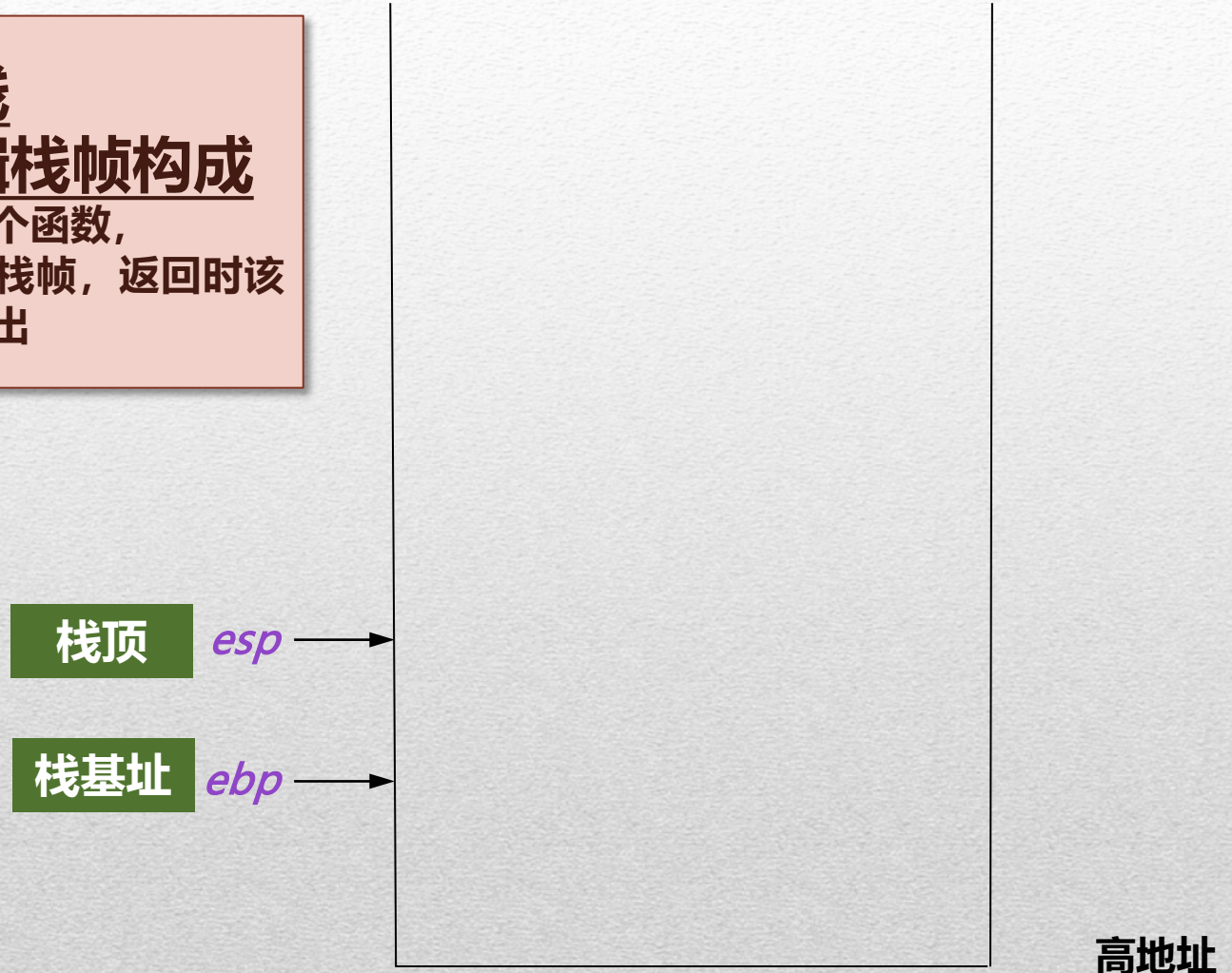
```
mov     [%ebp-4],  %eax
mov     %ebp,      %esp
pop     %ebp
ret
...
```

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

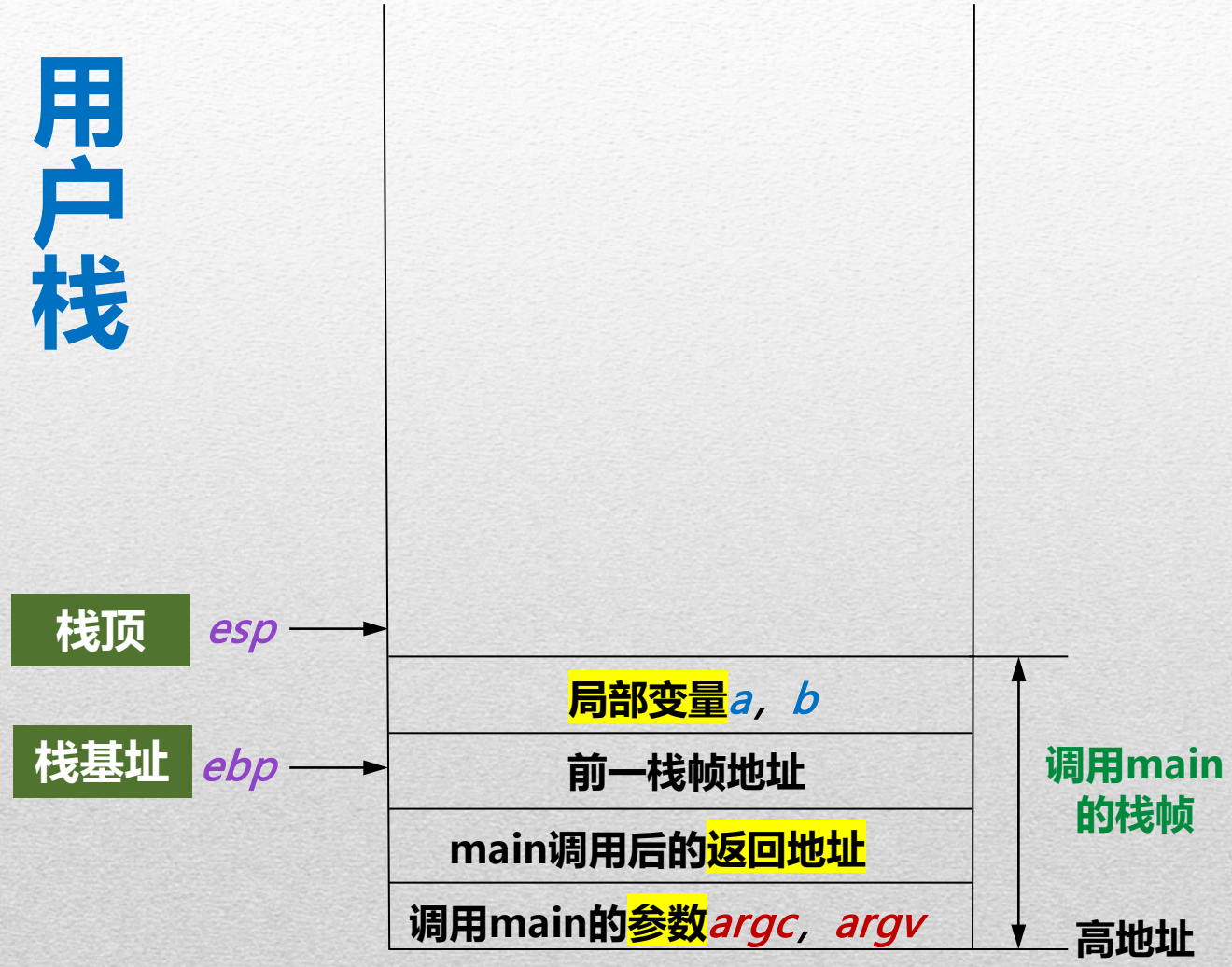
```
#include < fcntl.h>
char buffer[2048];
int version = 1;
main( argc, argv)
int   argc;
char *argv[];
{
    int a, b;
    ……;
    sum(a, b);
    exit(0);
}
int sum( var1, var2)
int var1, var2;
{
    int count;
    count = var1 +var2;
    return(count);
}
```

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



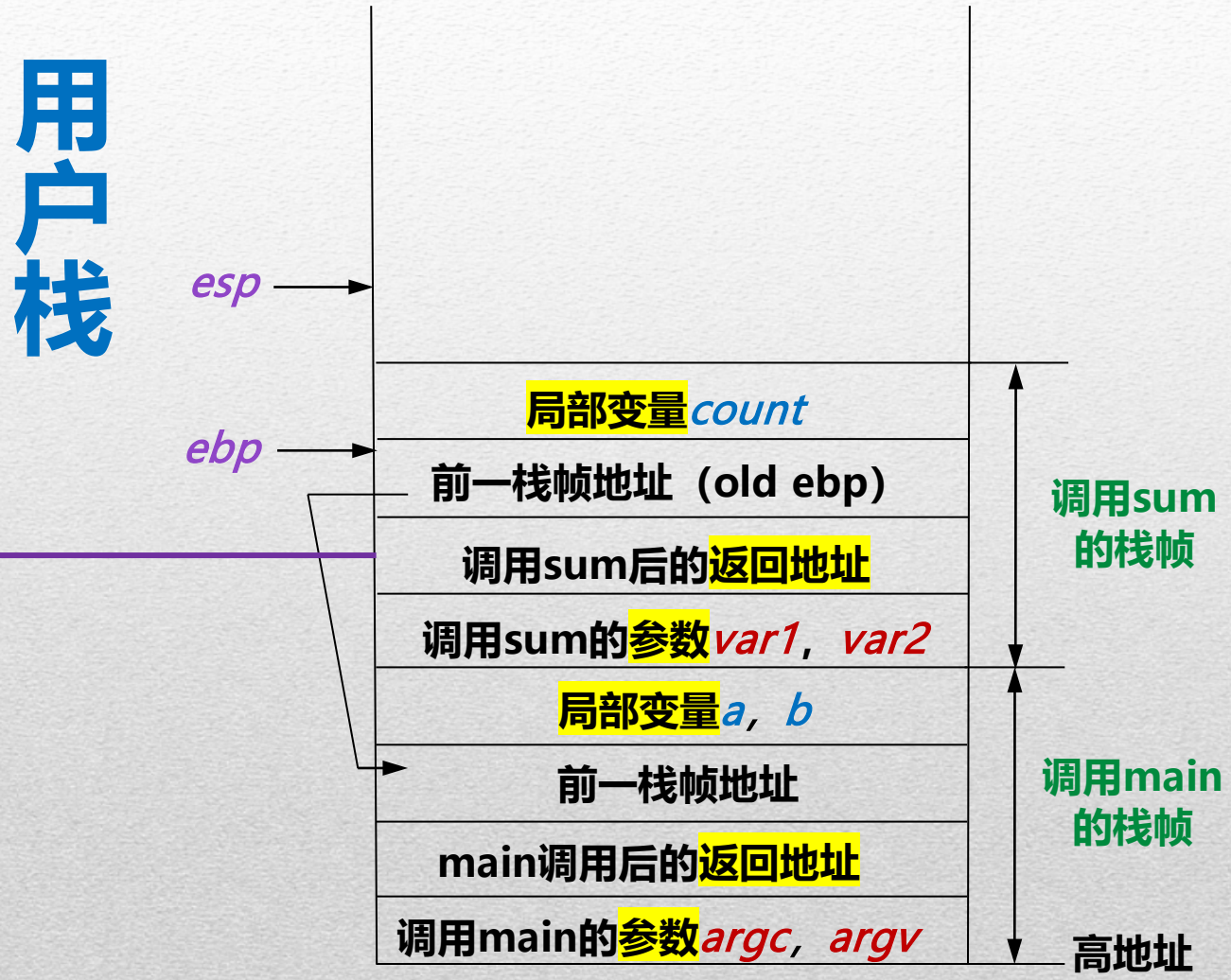
UNIX进程工作区

```
#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);
}
```



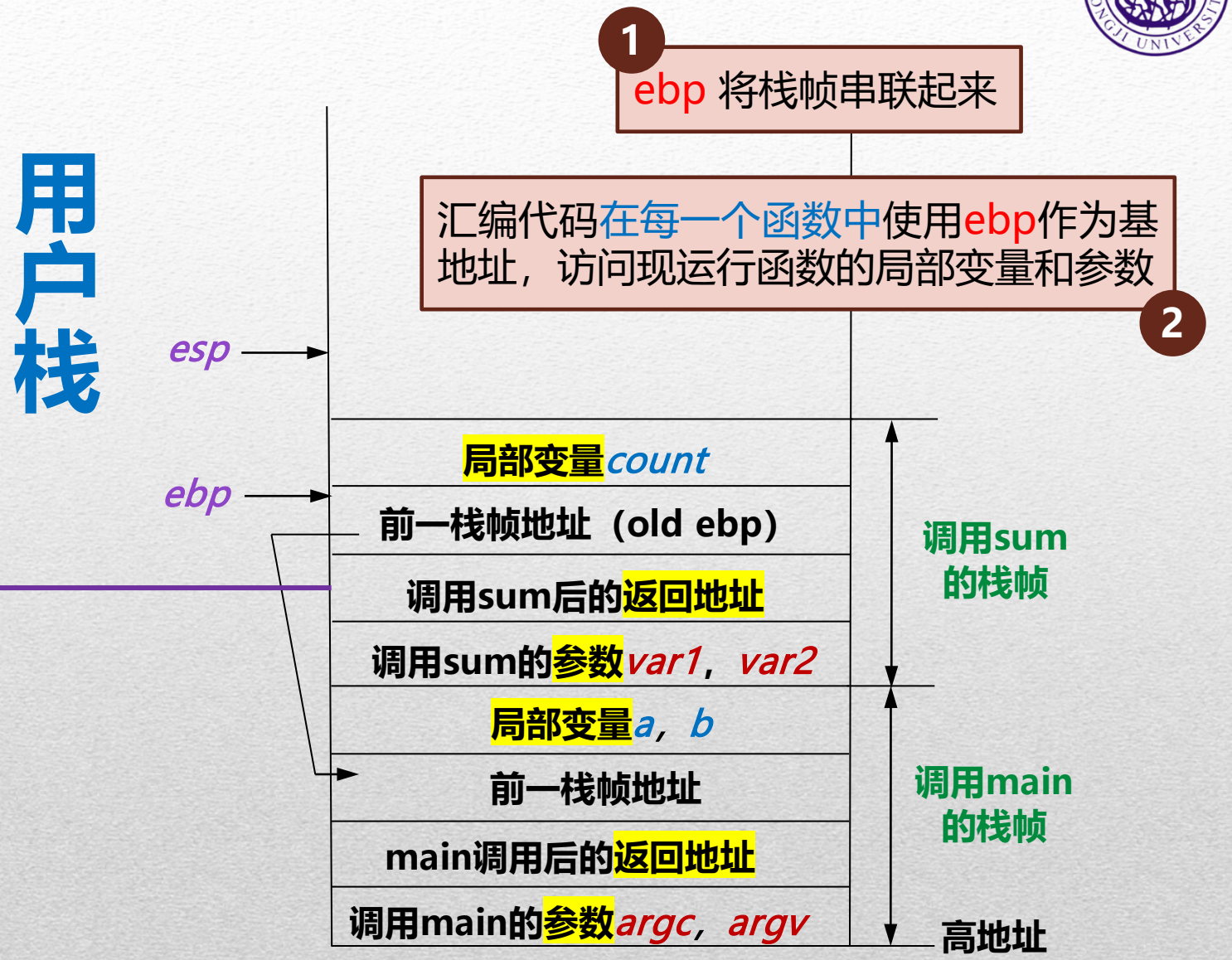
UNIX进程工作区

```
#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);
}
```



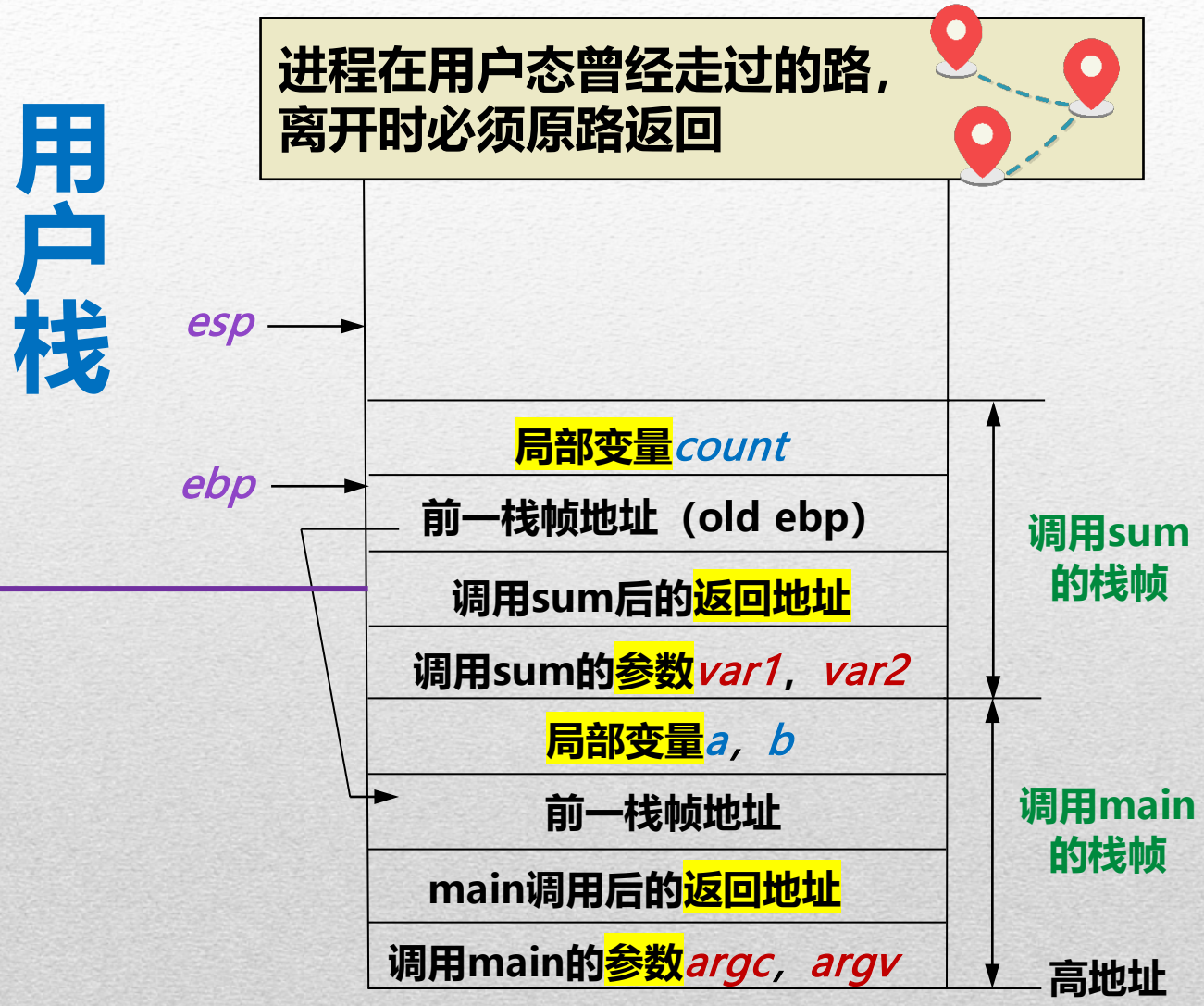
UNIX进程工作区

```
#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);
}
```



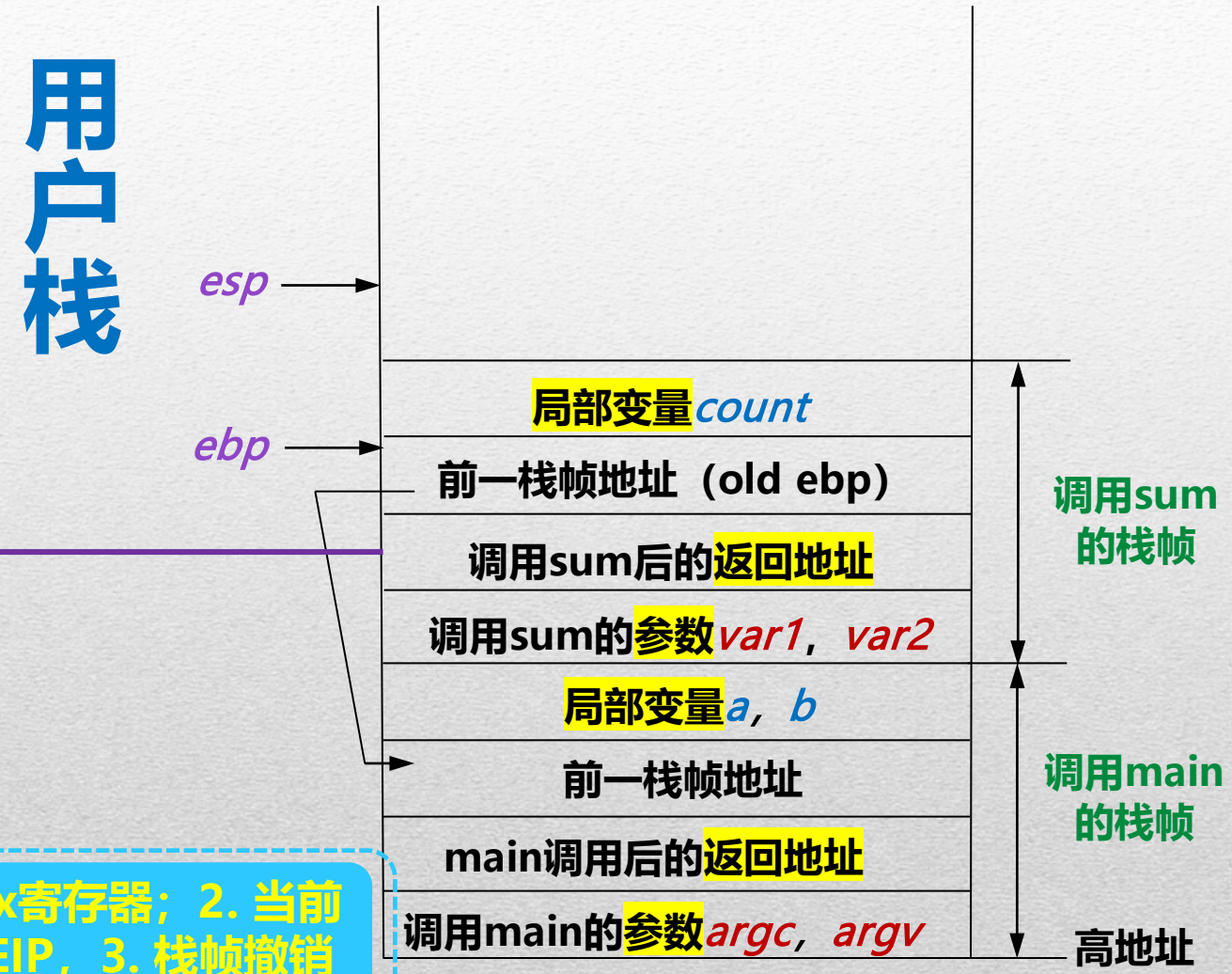
UNIX进程工作区

```
#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);
}
```



UNIX进程工作区

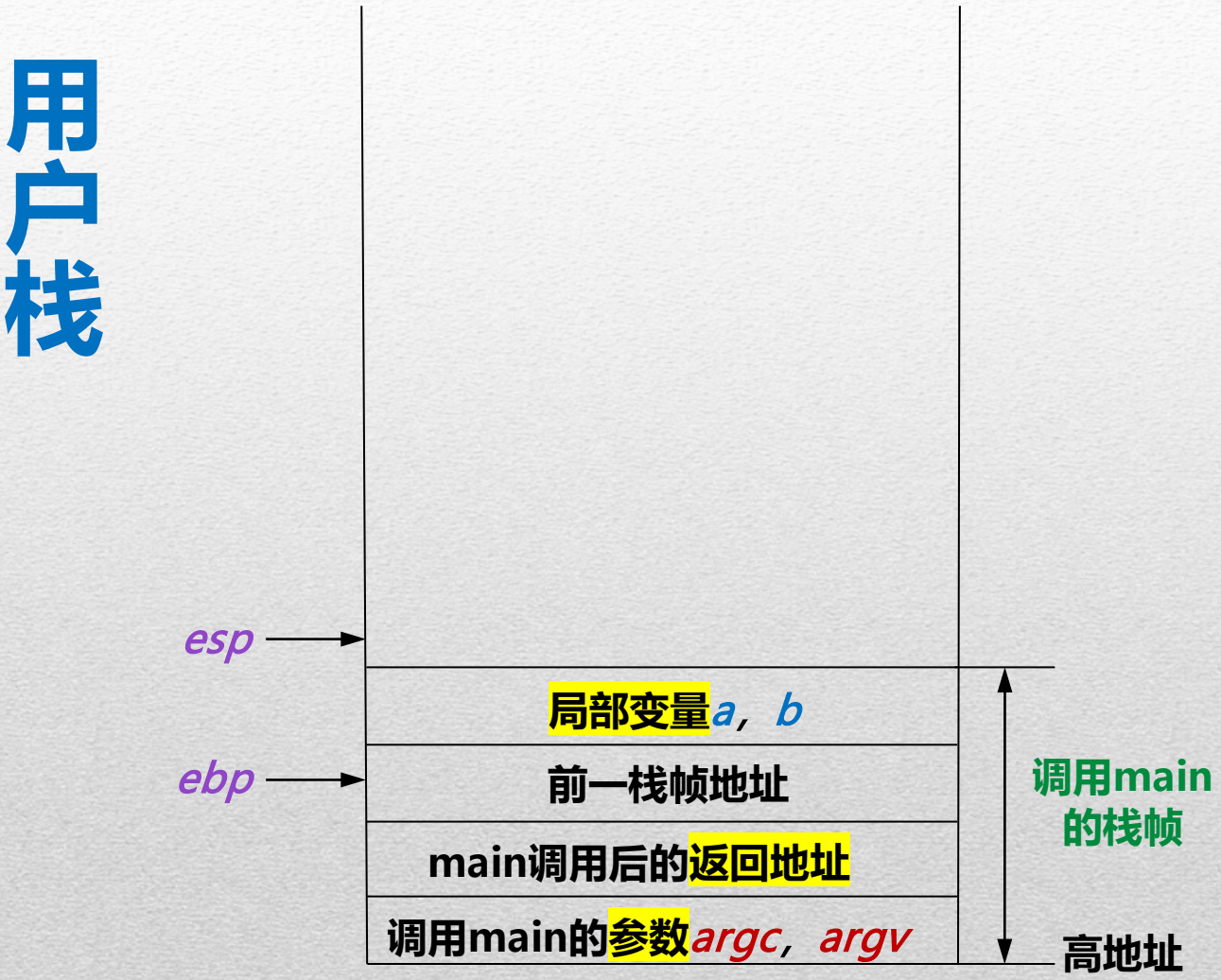
```
#include < fcntl.h>
char buffer[2048];
int version = 1;
main( argc, argv)
int   argc;
char *argv[];
{
    int a, b;
    ……;
    sum(a, b);
    exit(0);
}
int sum( var1, var2)
int var1, var2;
{
    int count;
    count = var1 +var2;
    return(count);
}
```



1. 将返回值放入eax寄存器；2. 当前栈帧返回地址装入EIP，3. 栈帧撤销

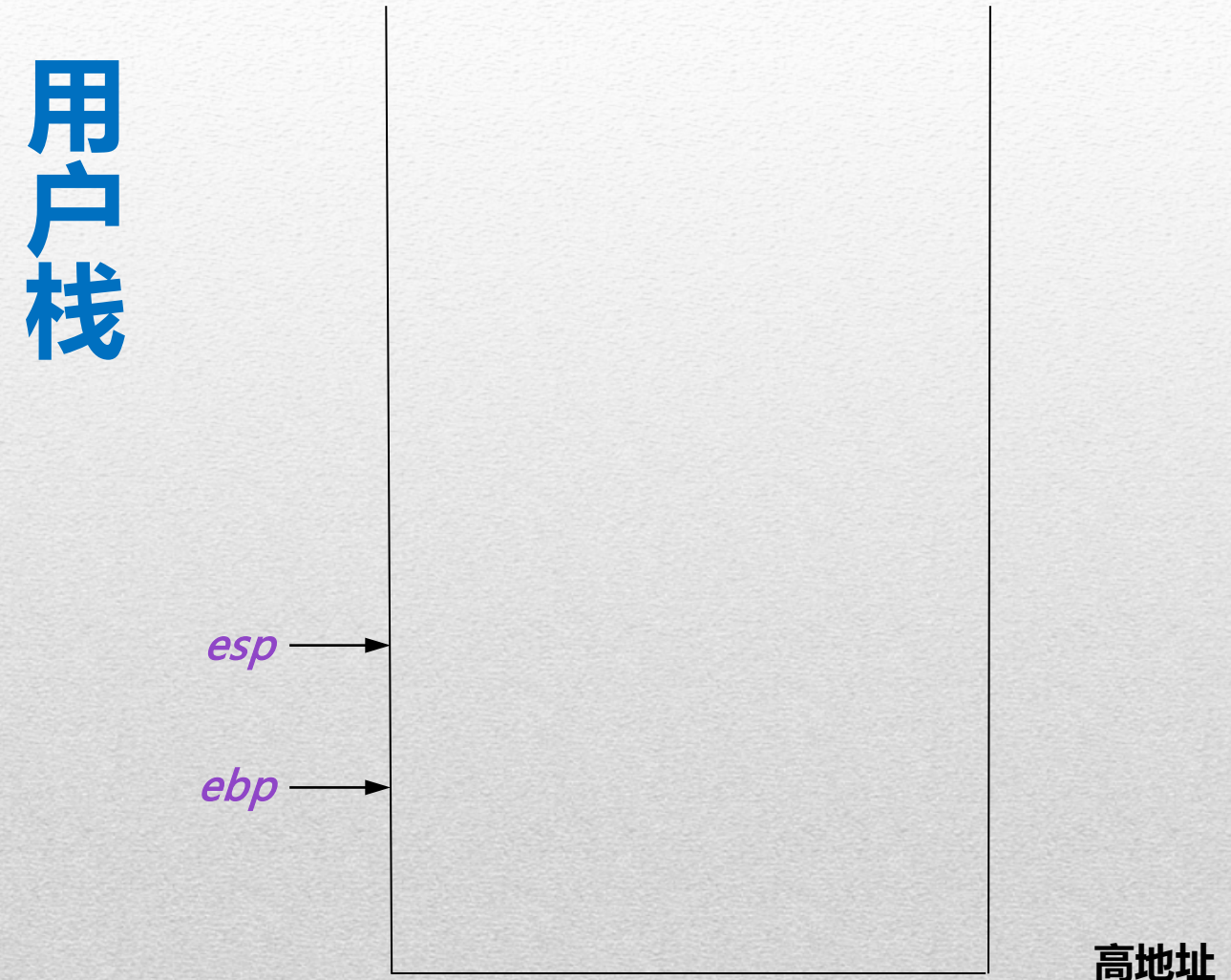
UNIX进程工作区

```
#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);
}
```

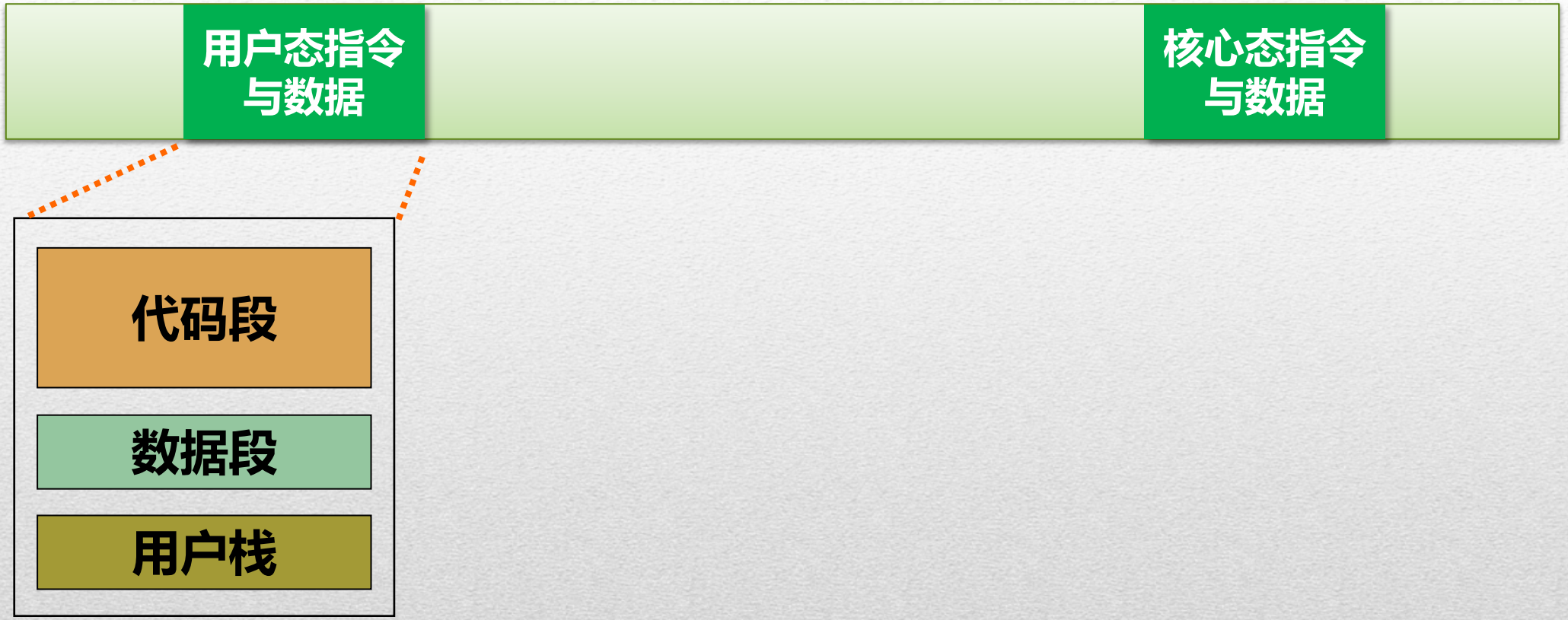


UNIX进程工作区

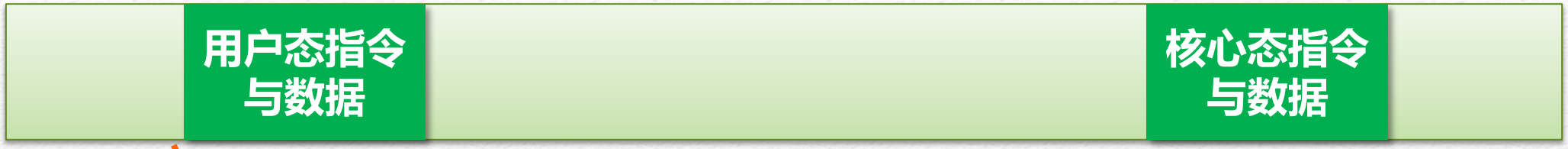
```
#include < fcntl.h>
char buffer[2048];
int version = 1;
main( argc, argv)
int   argc;
char *argv[];
{
    int a, b;
    ……;
    sum(a, b);
    exit(0);
}
int sum( var1, var2)
int var1, var2;
{
    int count;
    count = var1 +var2;
    return(count);
}
```



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



代码段

数据段

用户栈

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

程序

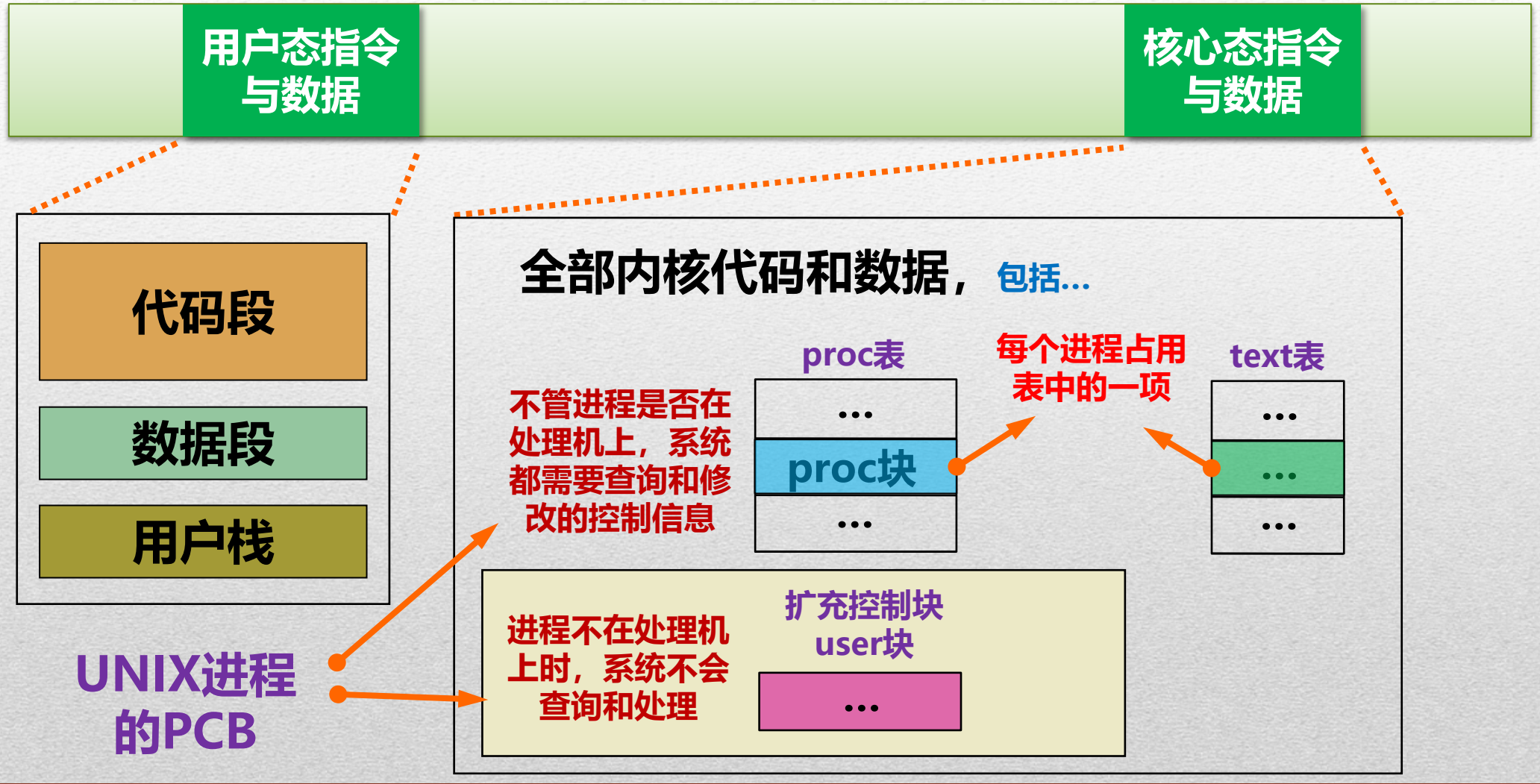
进程

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

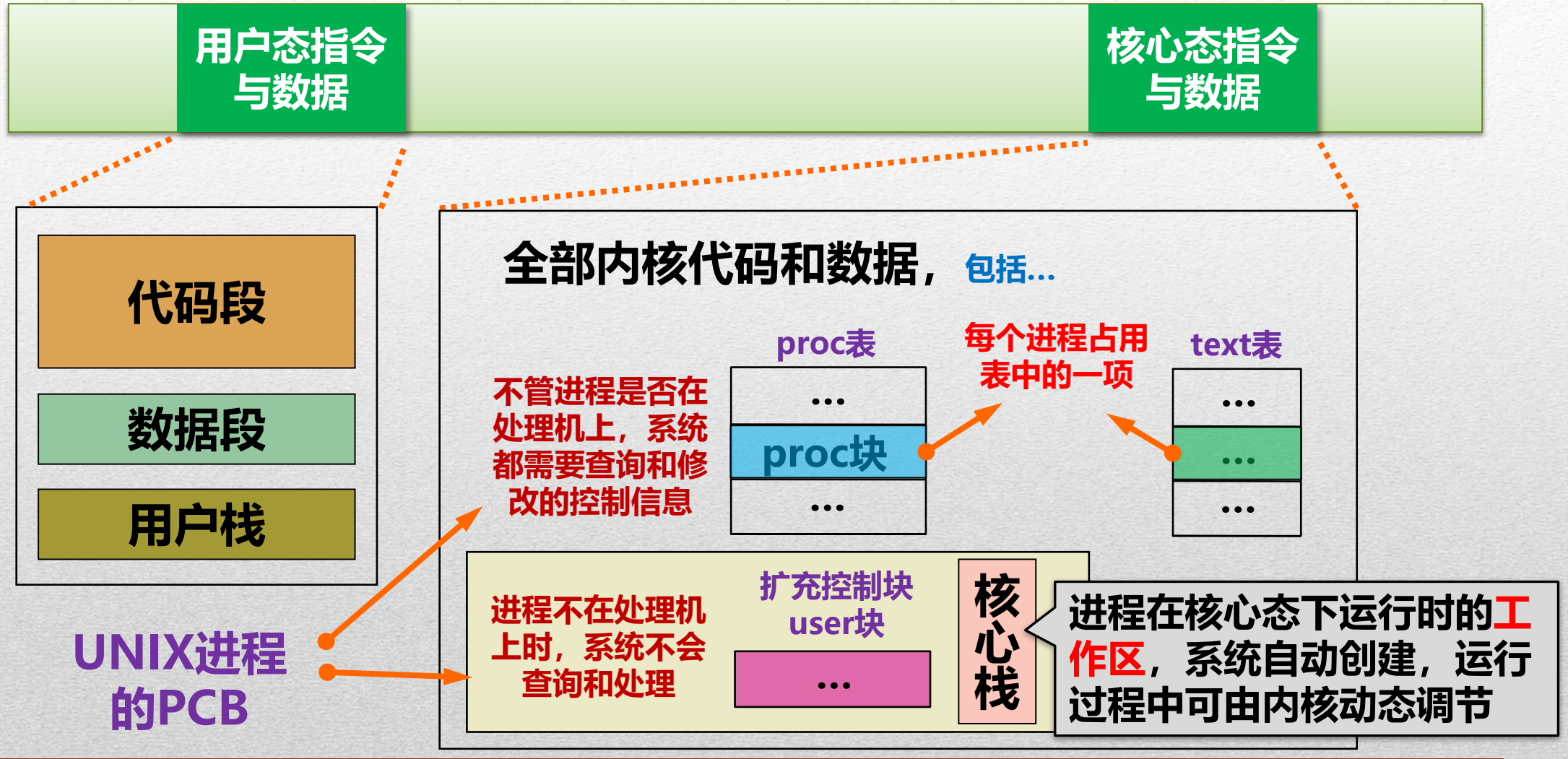
程序执行之前变身为进程

当进程运行在核心态呢？？？

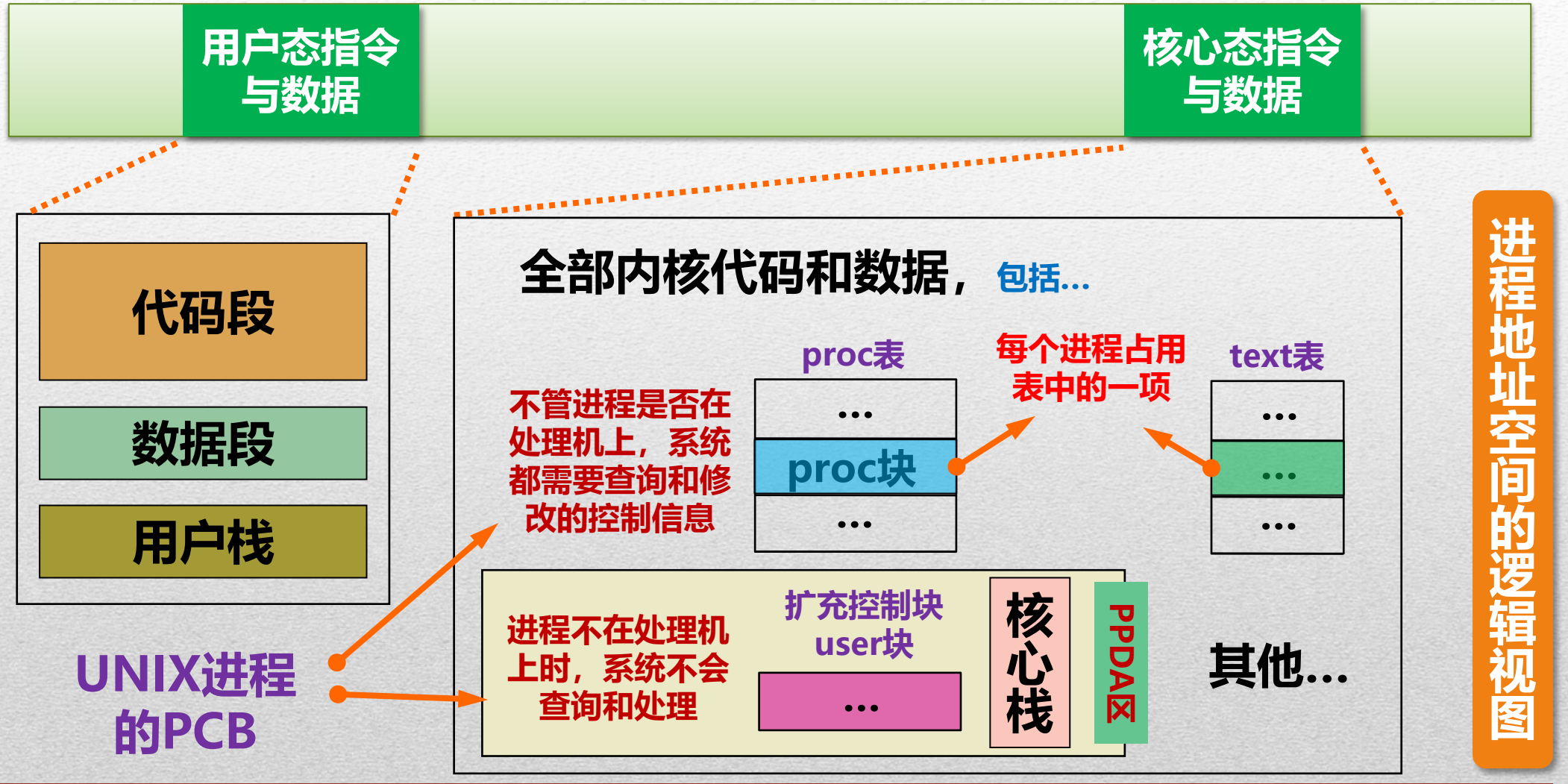
UNIX中进程核心态地址空间的构成



UNIX中进程核心态地址空间的构成

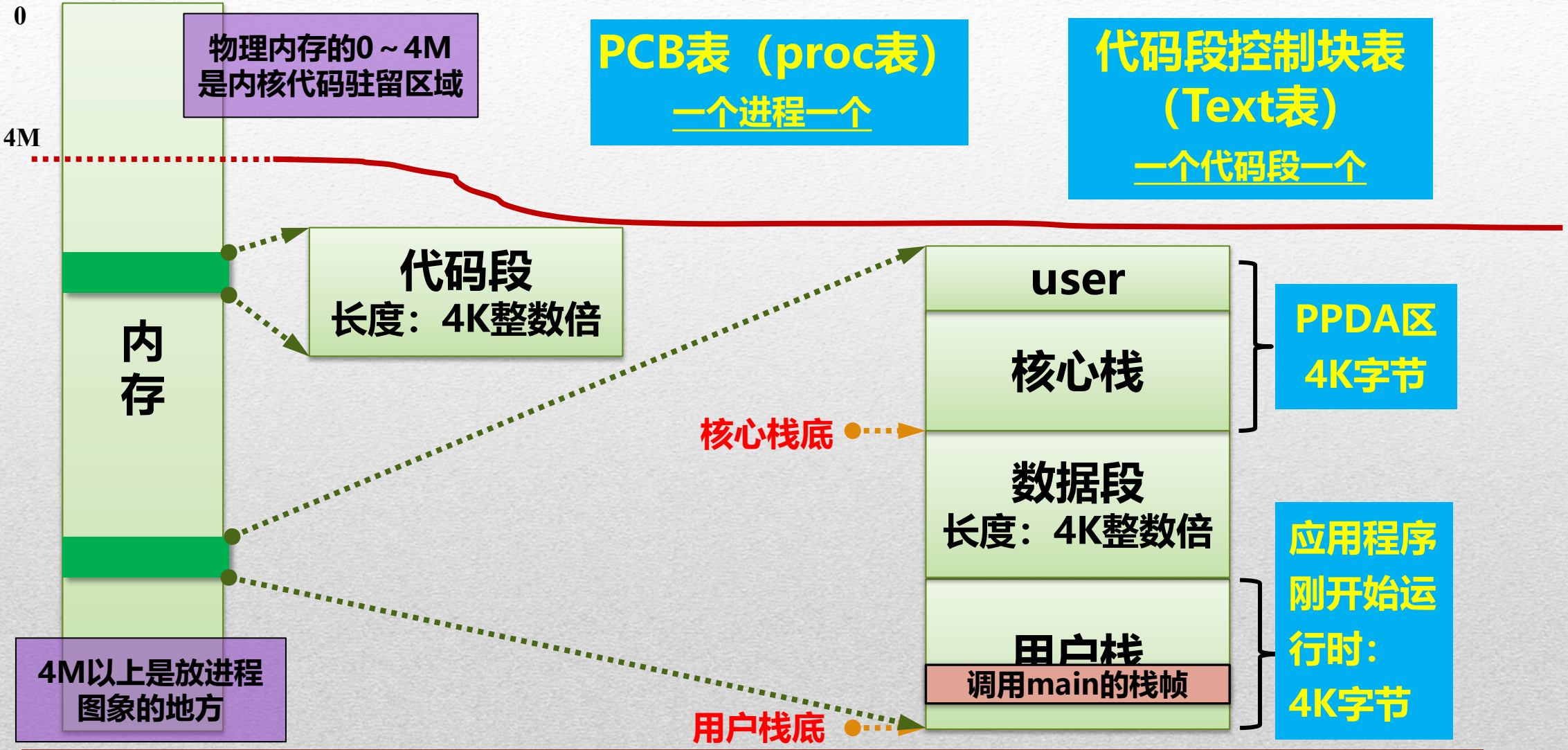


UNIX中进程核心态地址空间的构成

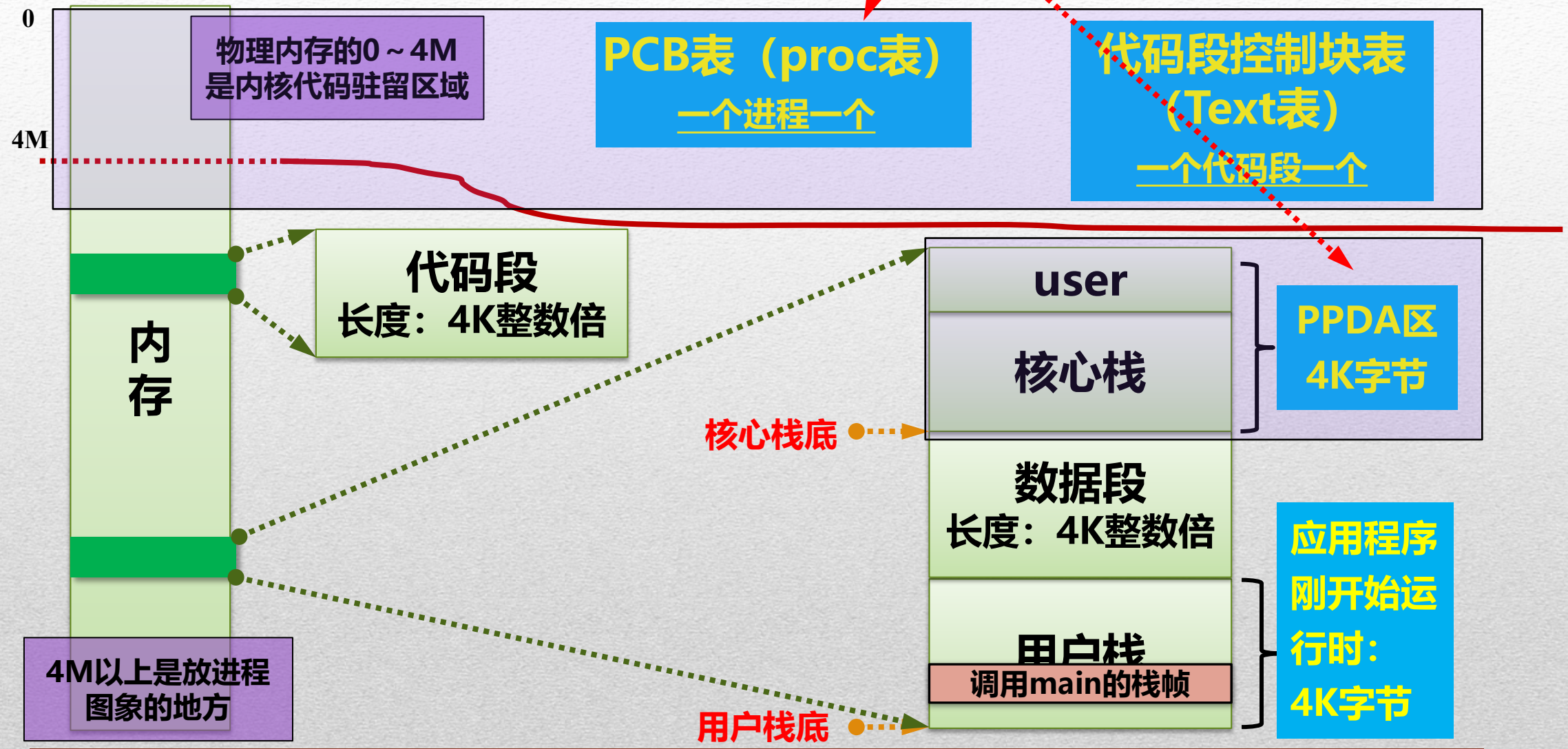


UNIX中进程的构成（进程图象）

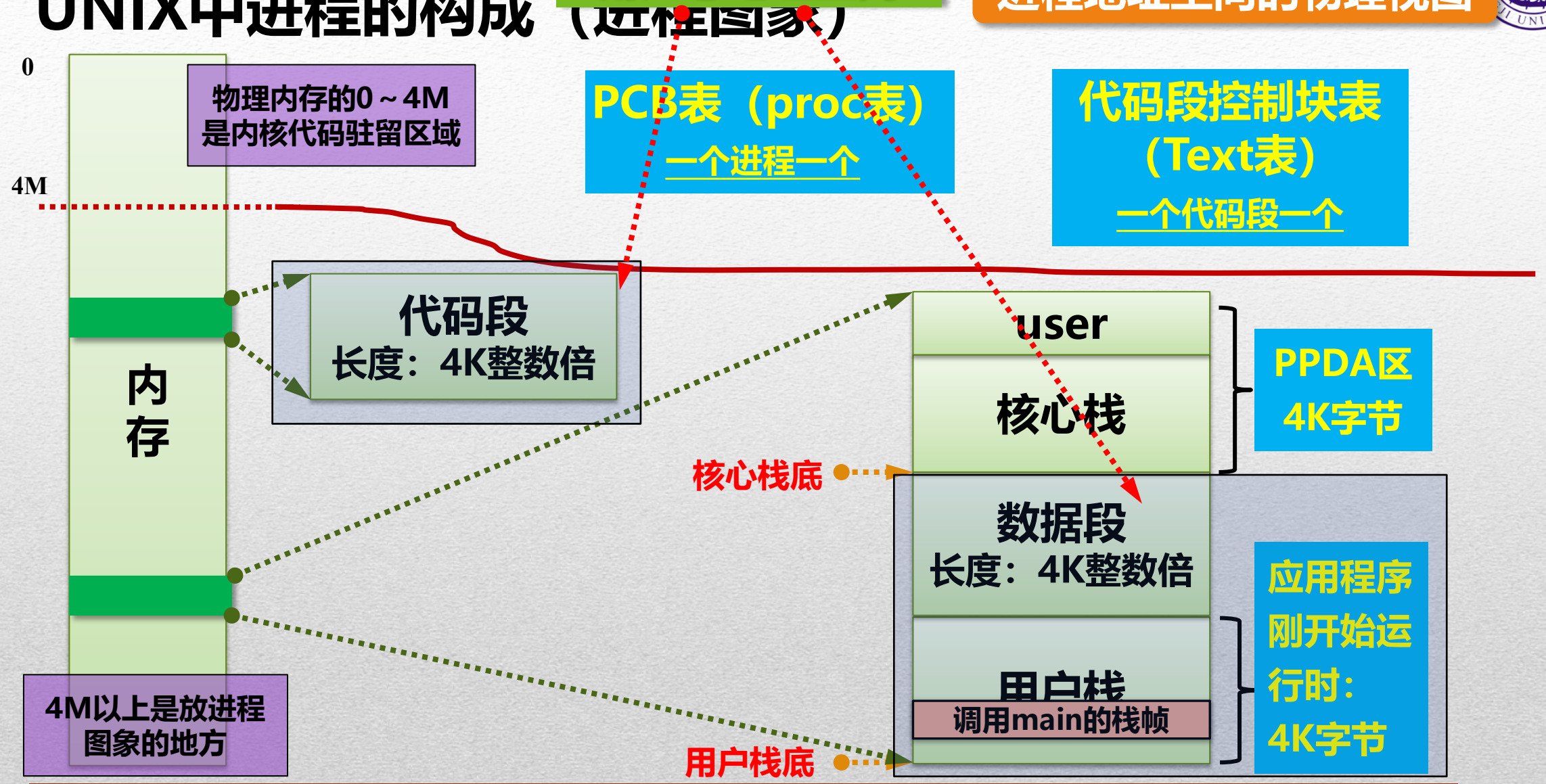
进程地址空间的物理视图



UNIX中进程的构成（进程图象）



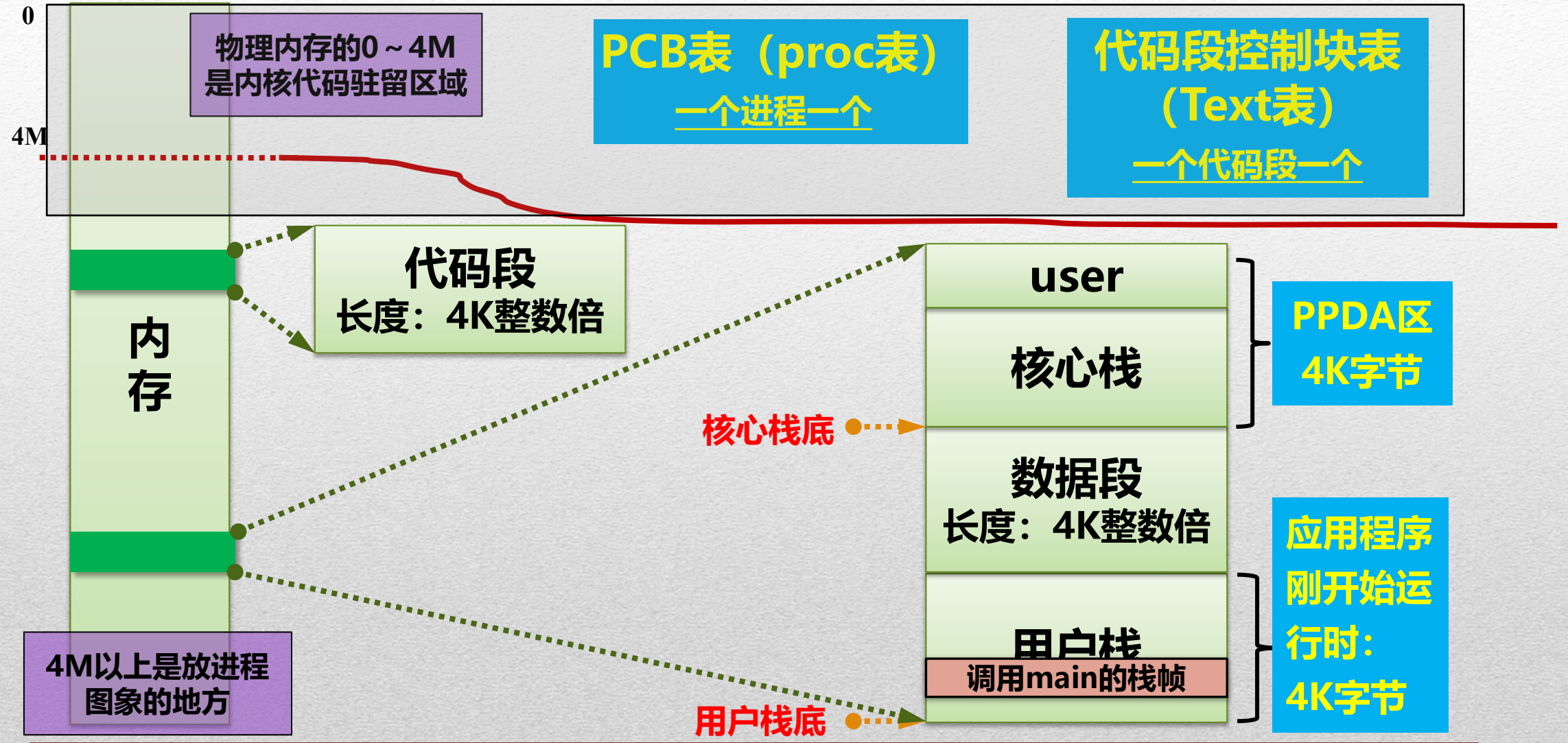
UNIX中进程的构成（进程图象）



UNIX中进程的构成（进程图象）

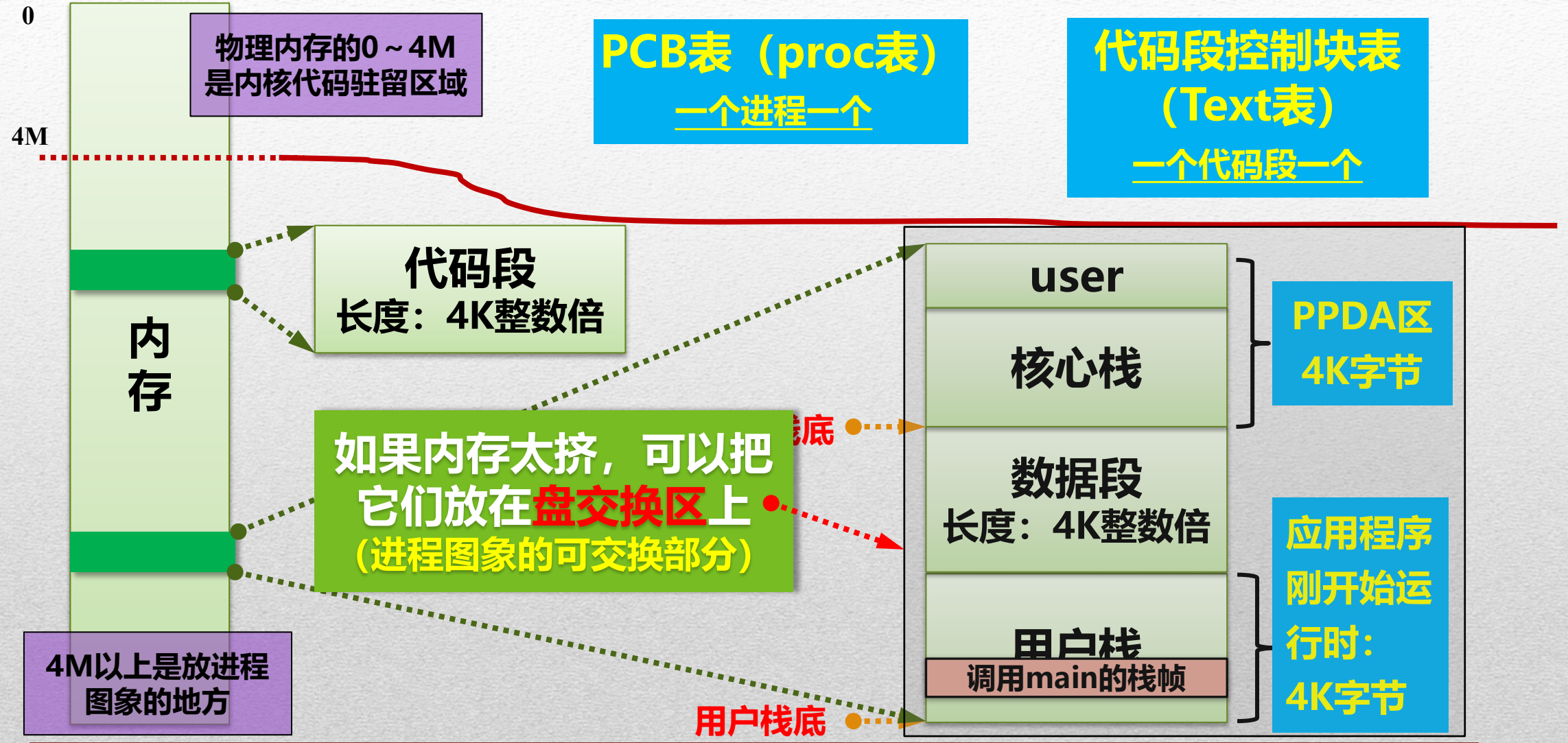
常驻内存部分

进程地址空间的物理视图



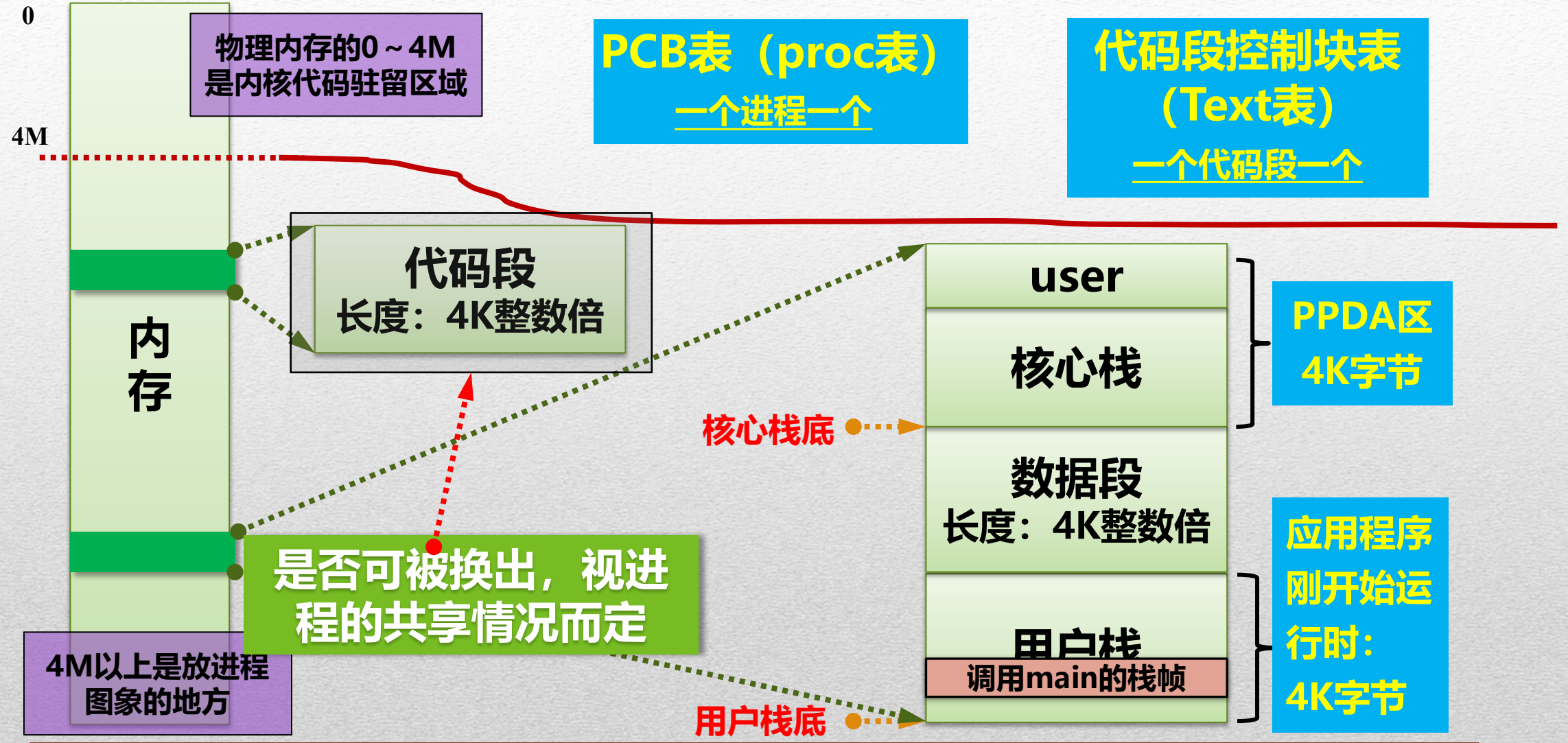
UNIX中进程的构成（进程图象）

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

请阅读讲义：71页 ~ 72页（2.7.1节），100页 ~ 103页（3.3.3节）

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

```
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
...
```

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

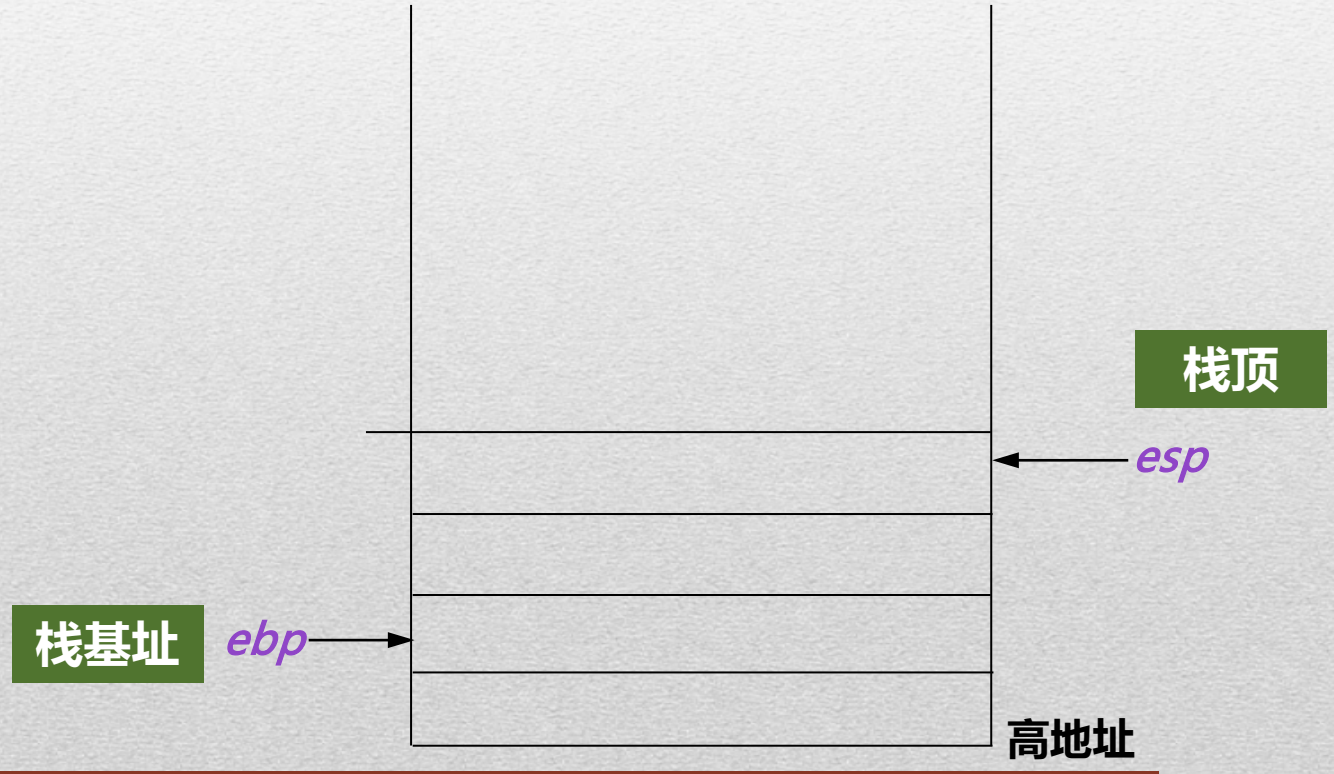
C语言编译器对函数调用的处理方式

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
        ...
```



C语言编译器对函数调用的处理方式

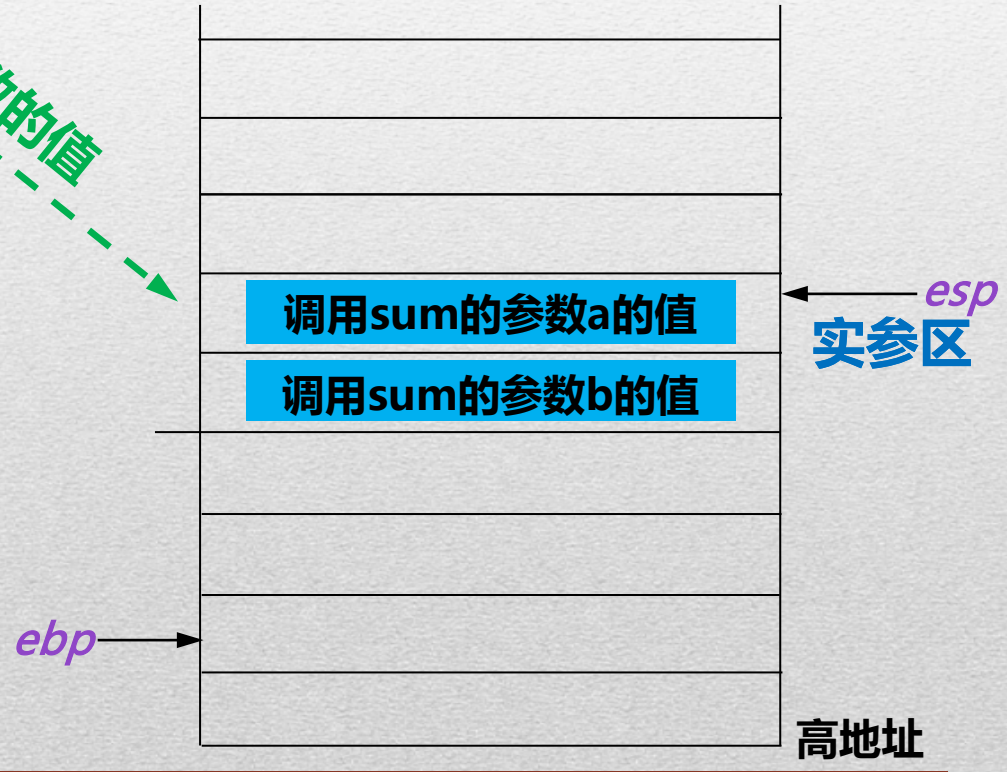
main

```
...
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



C语言编译器对函数调用的处理方式

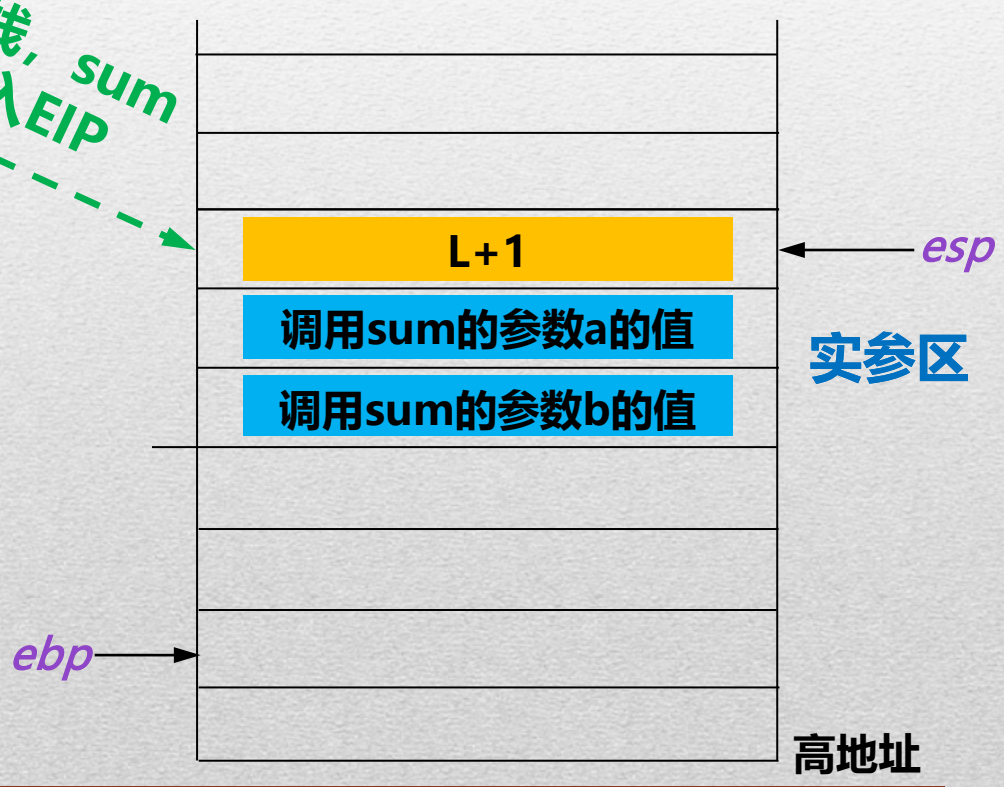
main

```
...
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

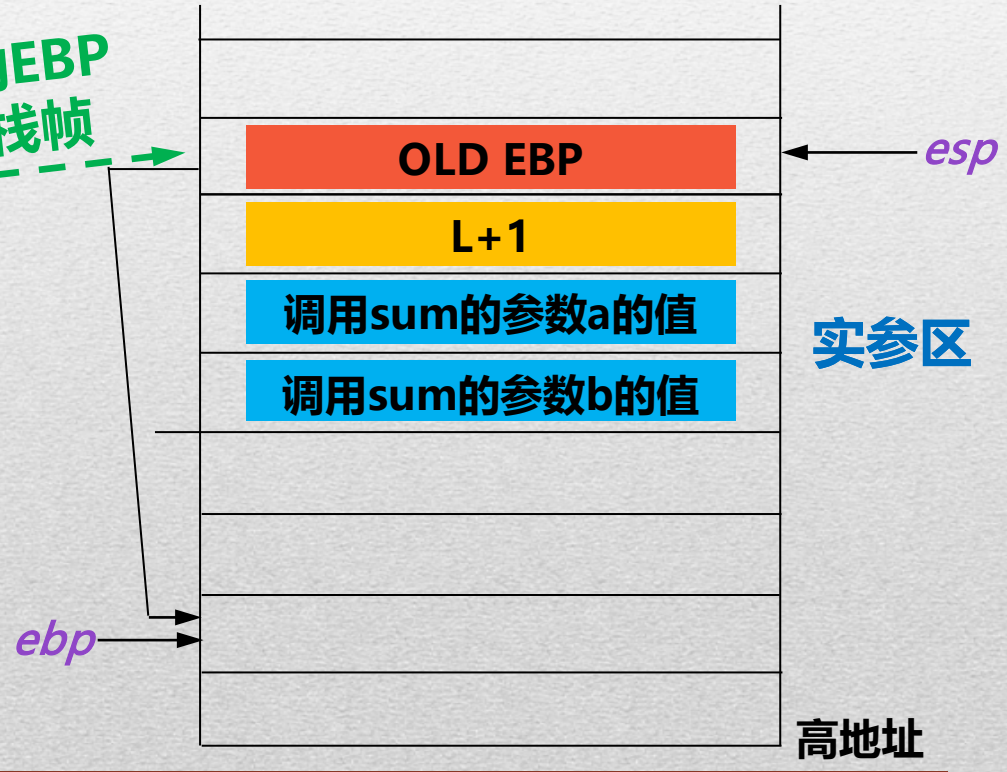
main

```
...
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

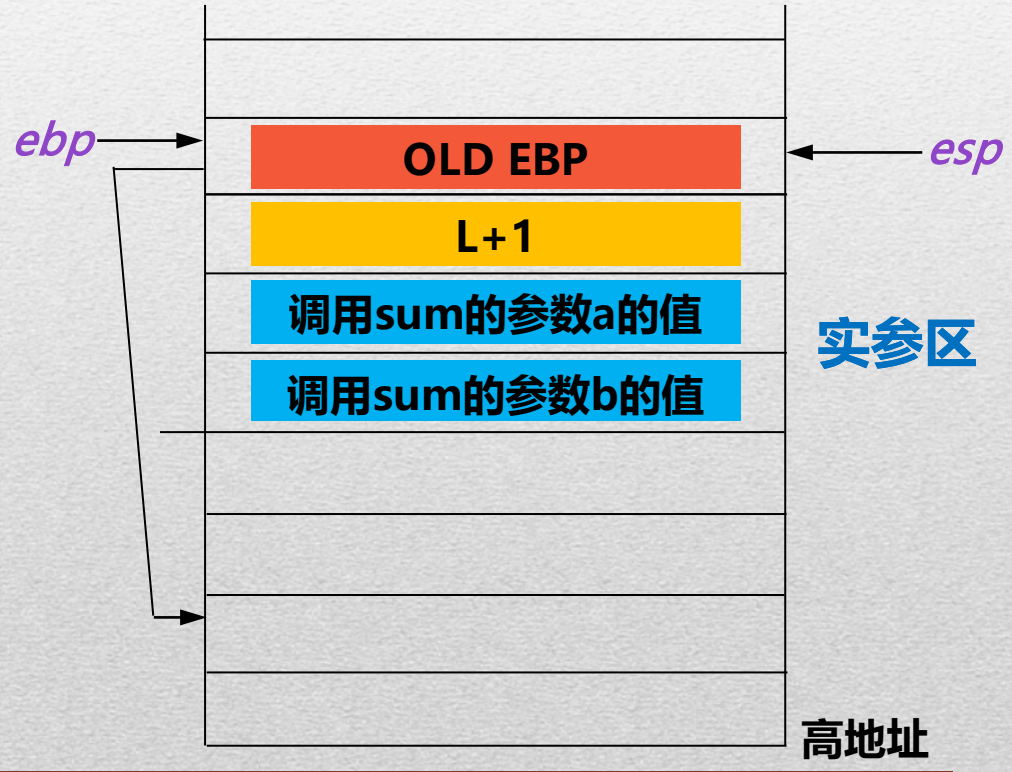
main

```
...
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

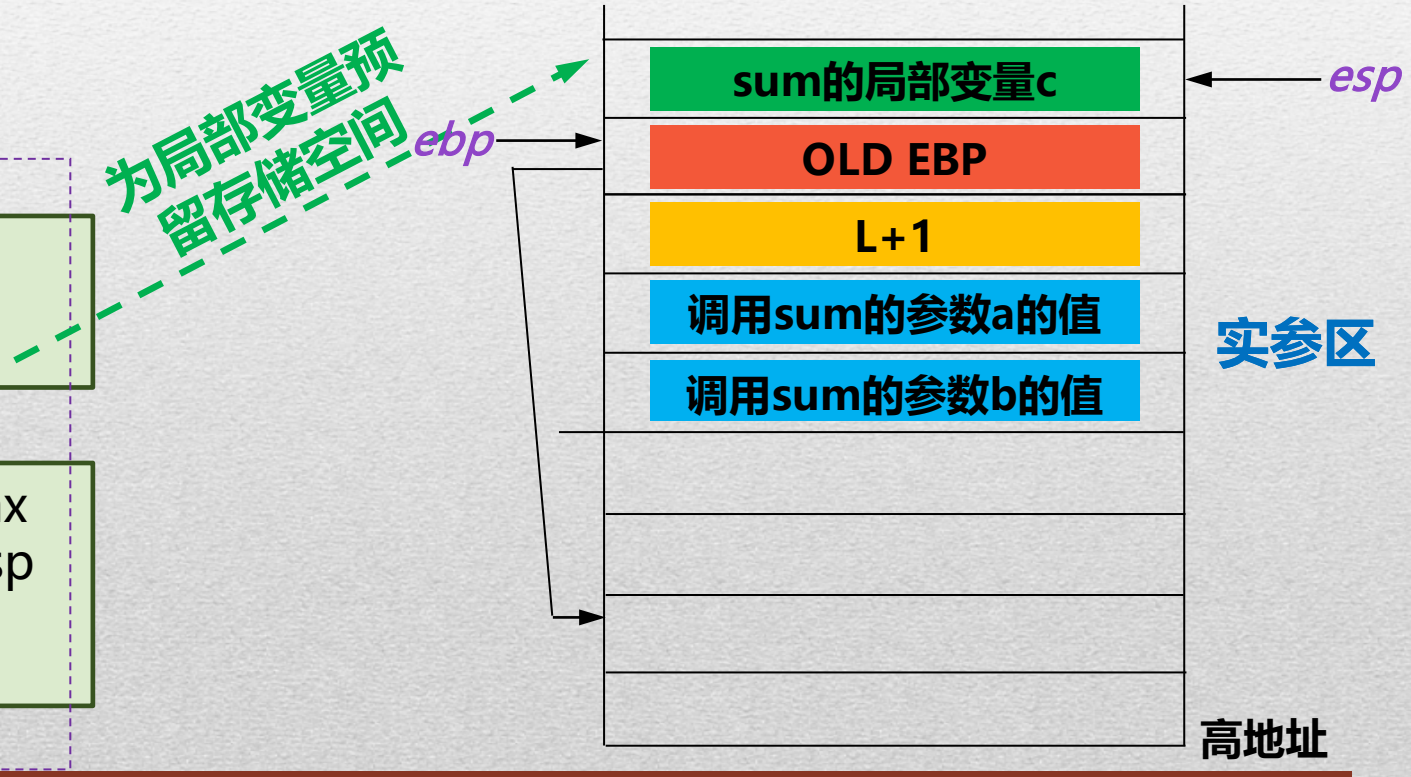
```
...
L-2:    push [%ebp - 8]
L-1:    push [%ebp - 4]
L:      call   sum
L+1:    add    %esp, 8
...
```

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



C语言编译器对函数调用的处理方式

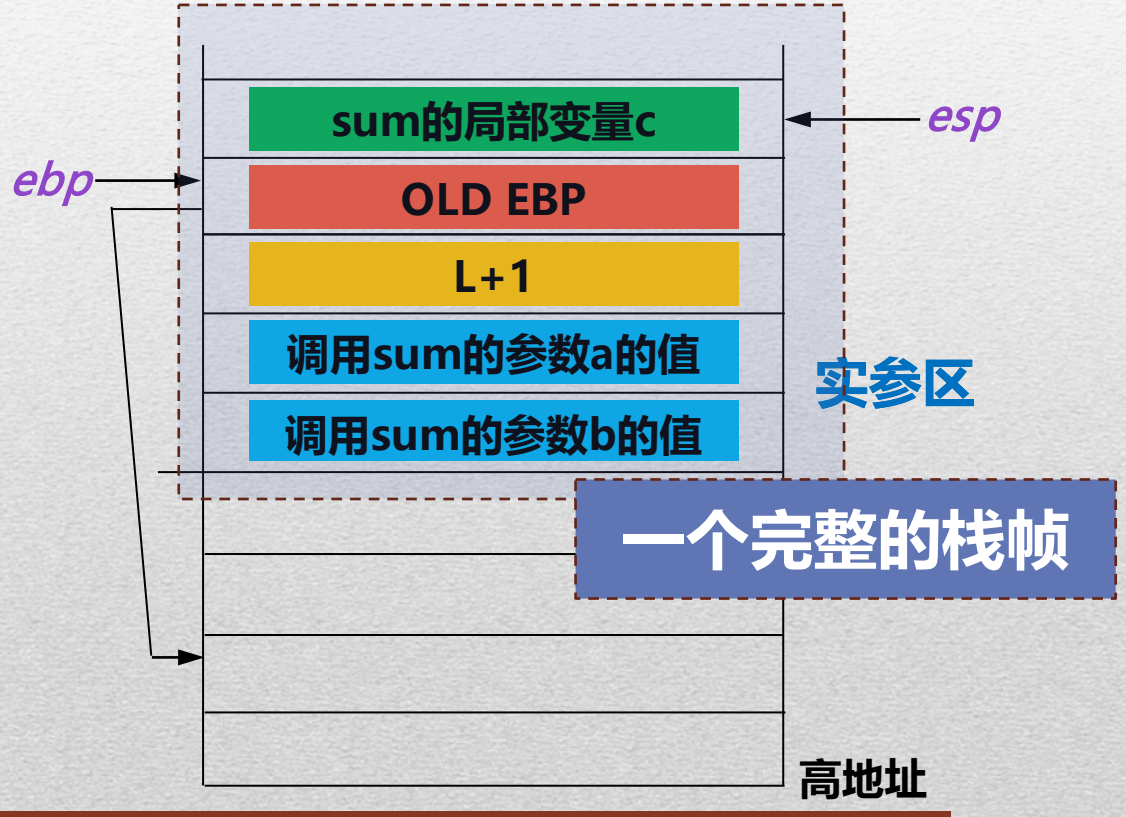
main

```
...
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

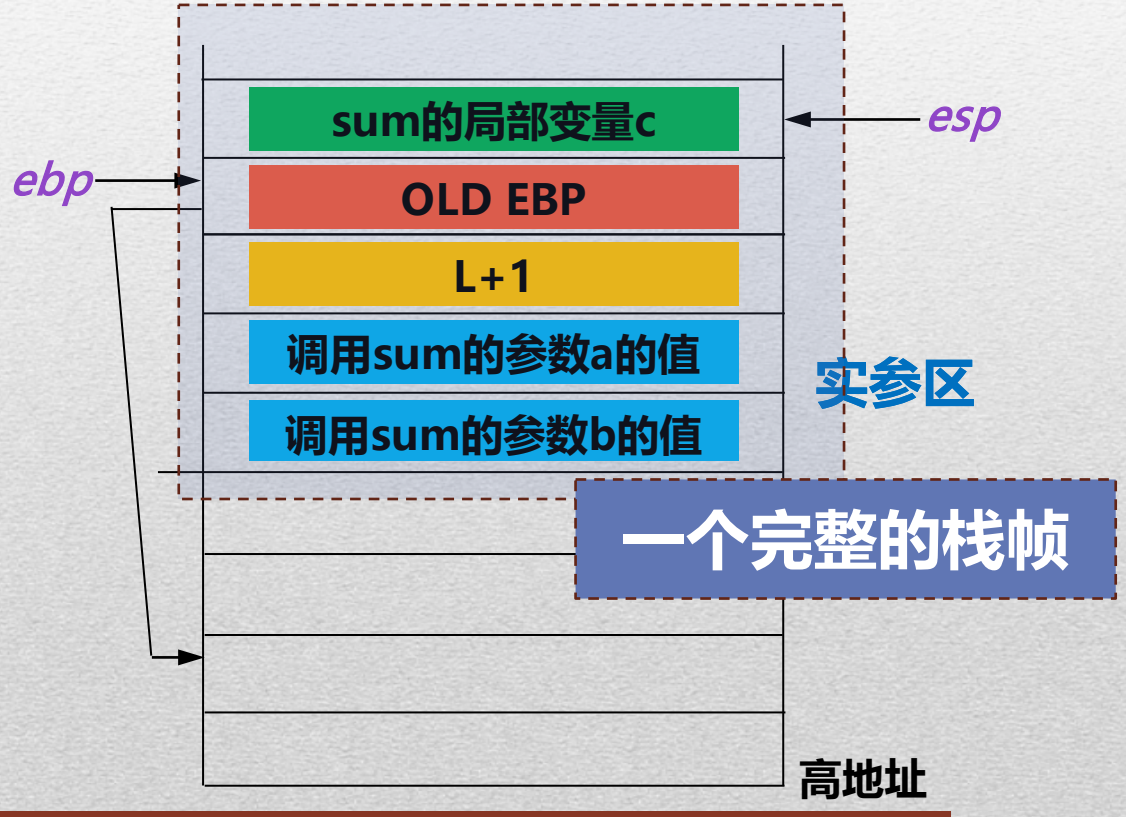
```
...
L-2:    push [%ebp - 8]
L-1:    push [%ebp - 4]
L:      call   sum
L+1:    add    %esp, 8
...
```

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]、……访问现运行函数栈帧中的第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

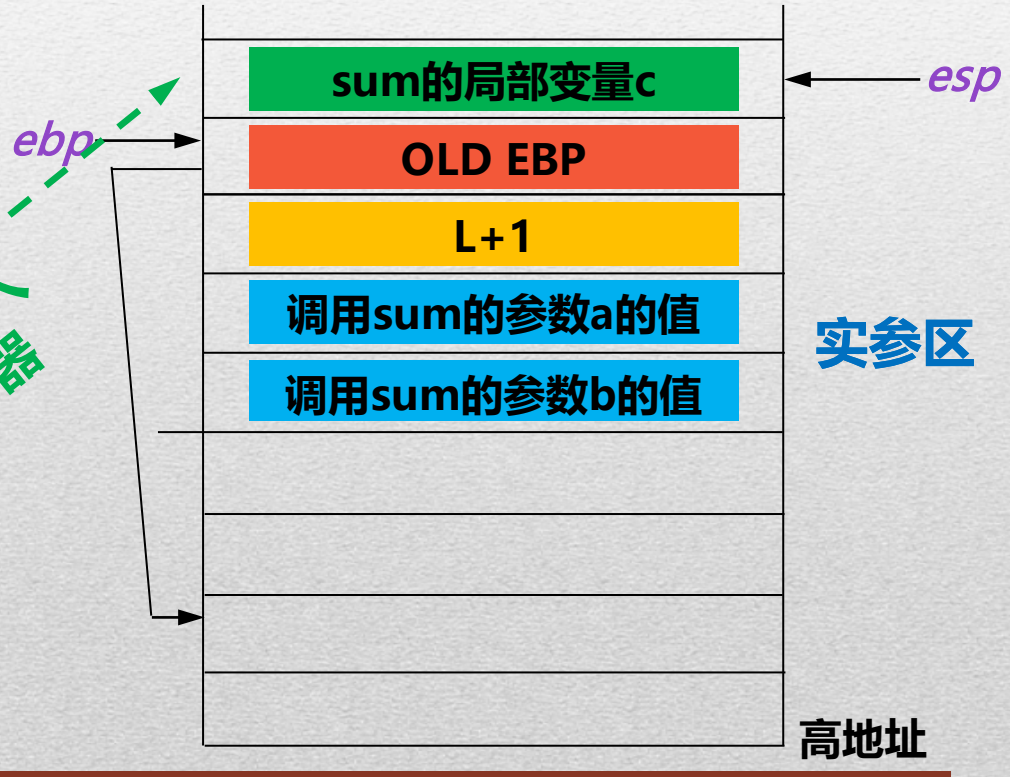
main

```
...
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

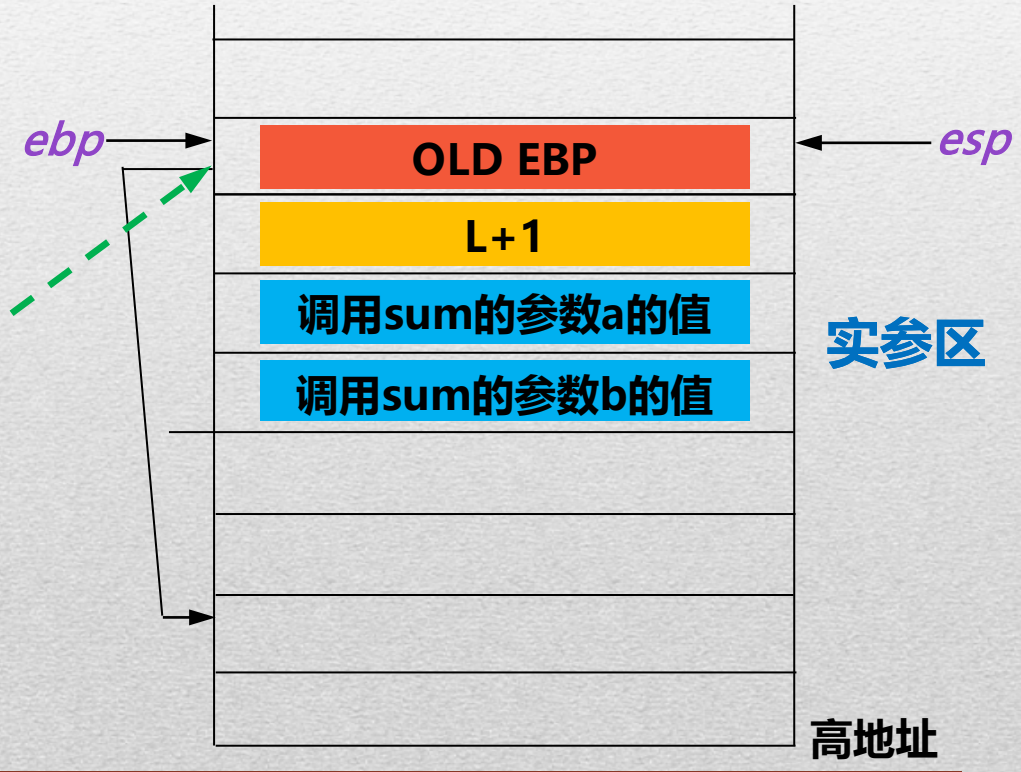
main

```
...
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

C语言编译器对函数调用的处理方式

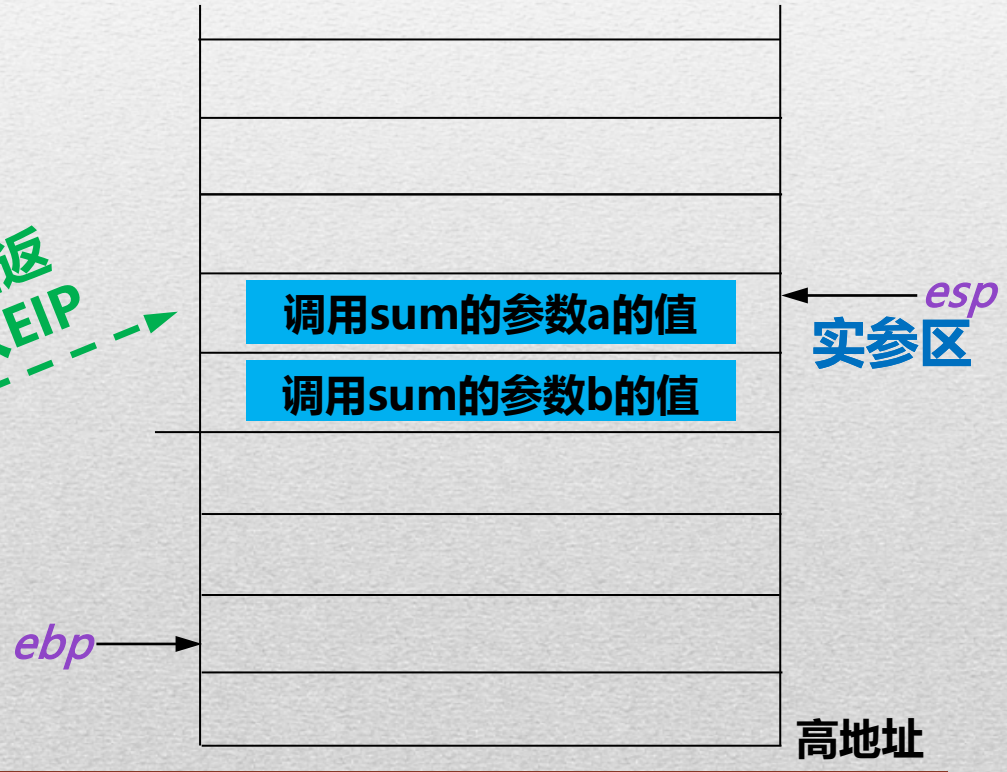
main

```
...
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



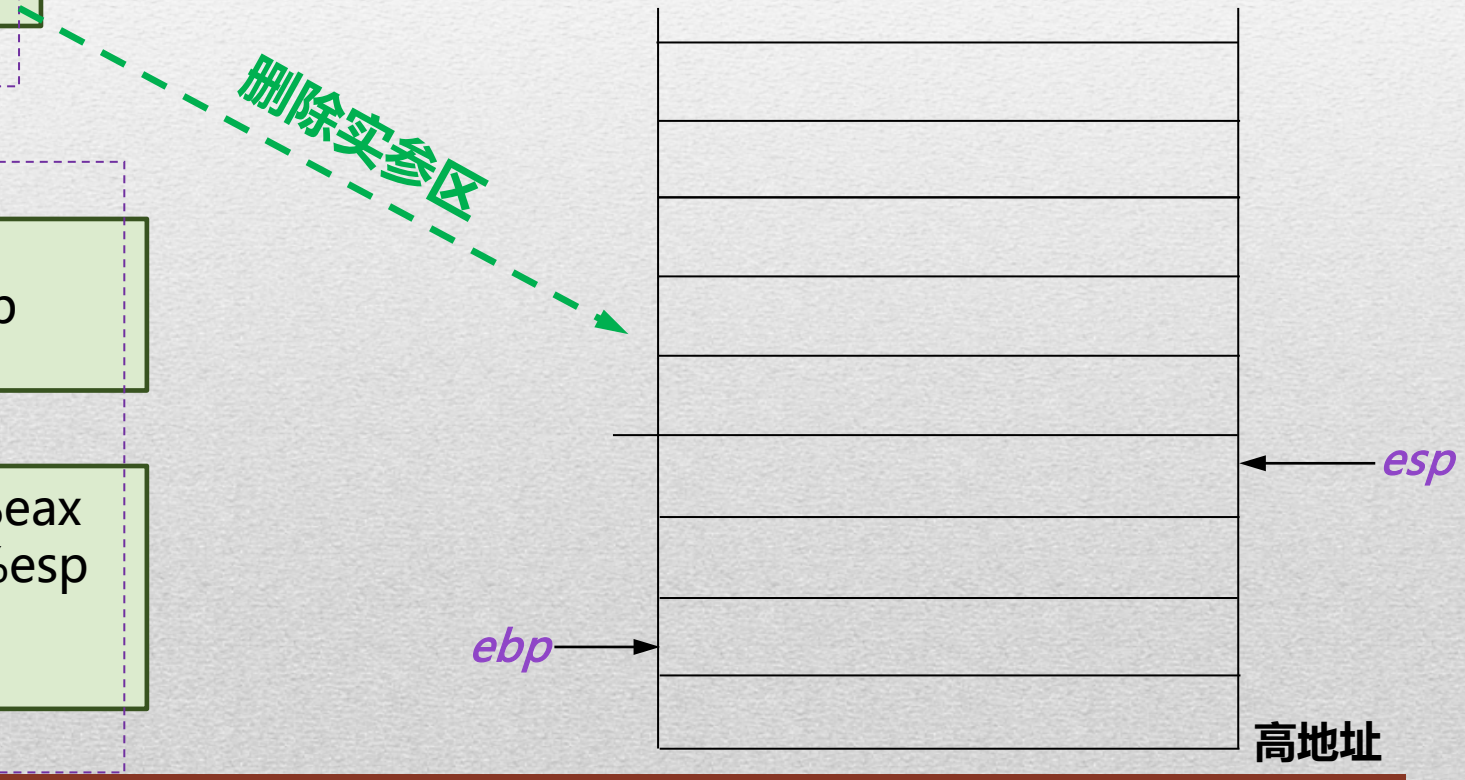
C语言编译器对函数调用的处理方式

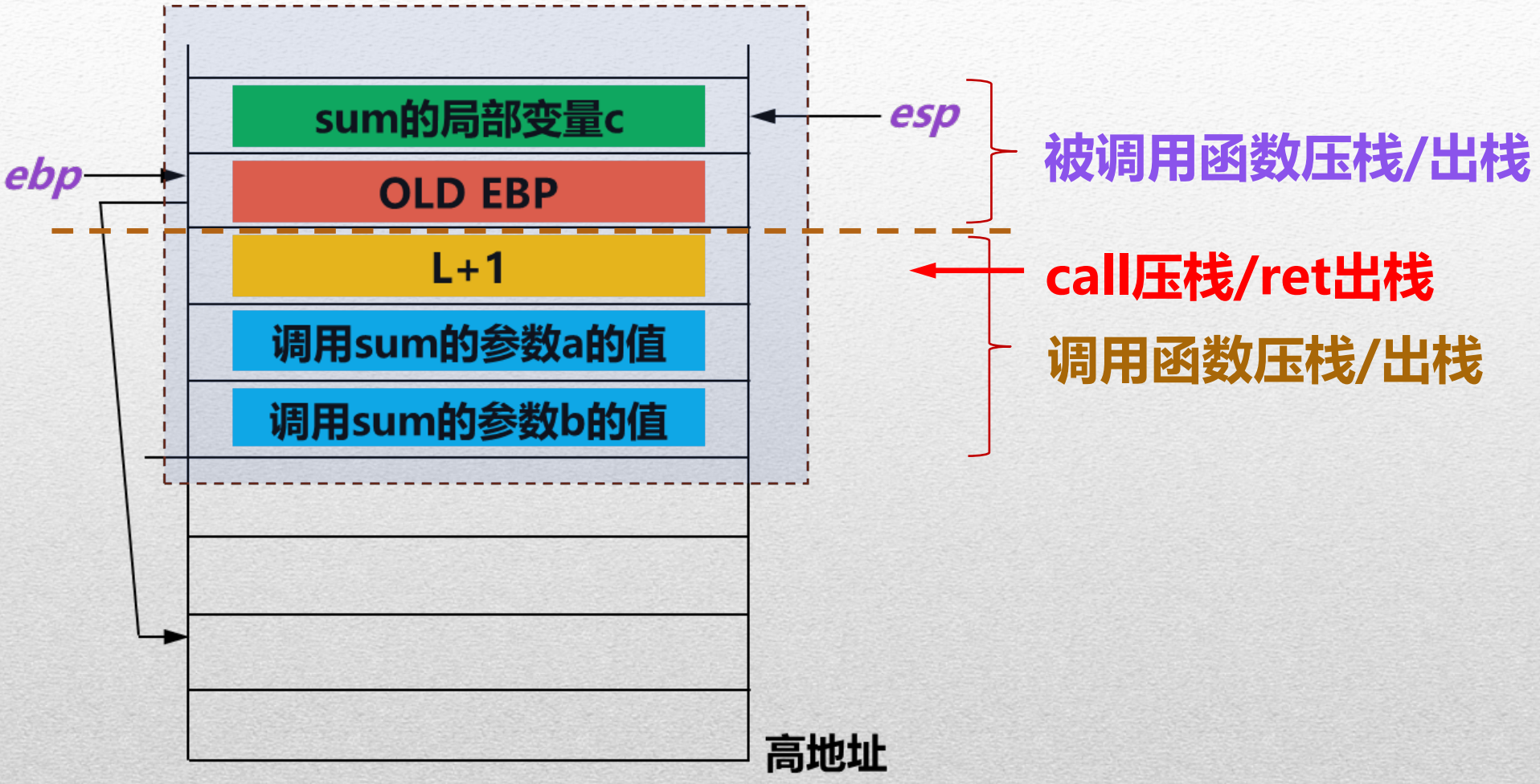
main

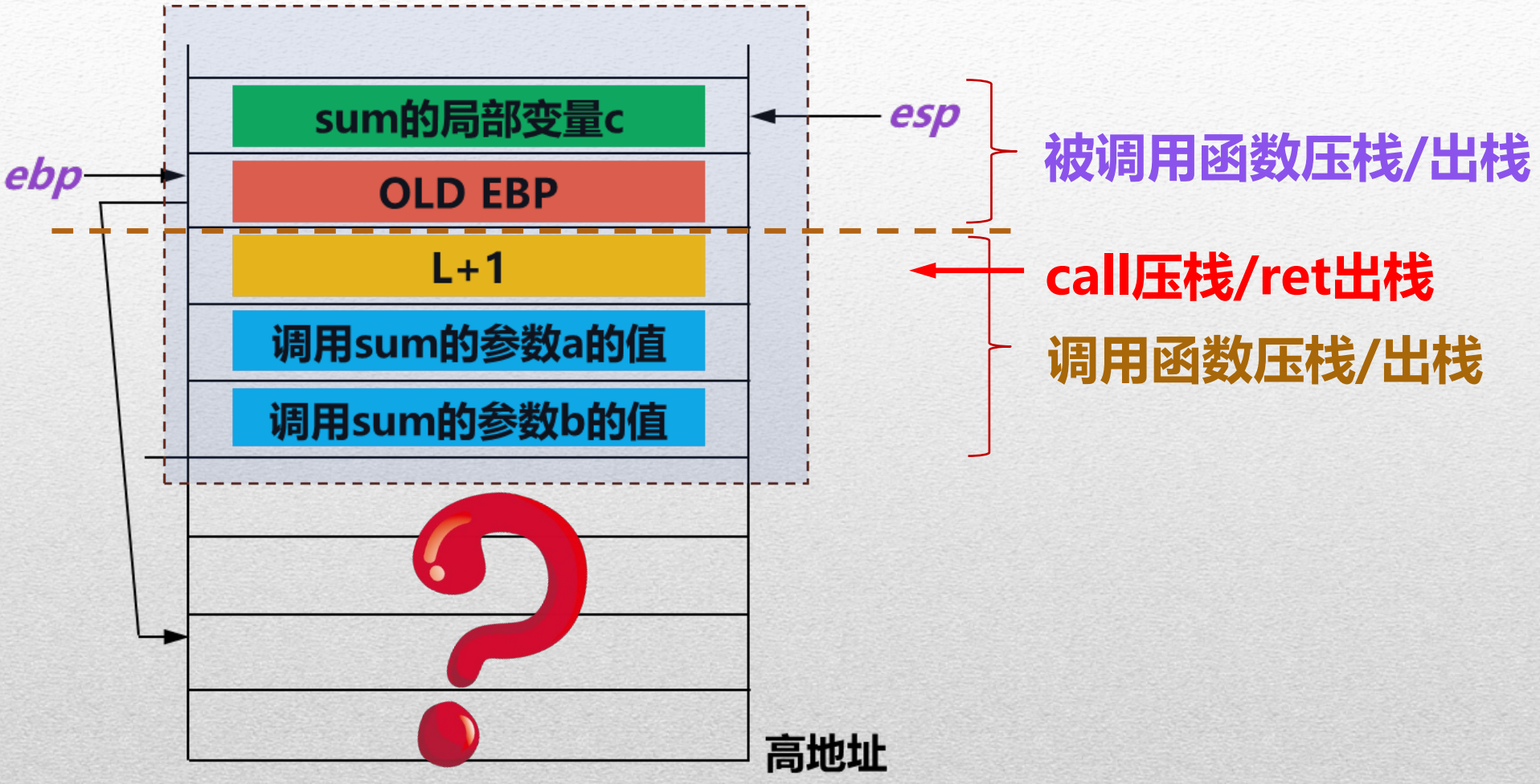
```
...
L-2:    push [%ebp - 8]
L-1:    push [%ebp - 4]
L:      call   sum
L+1:    add    %esp, 8
...
```

sum

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




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
ebp
高地址
被调用函数压栈/出栈
call压栈/ret出栈
调用函数压栈/出栈


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
ebp
高地址
被调用函数压栈/出栈
call压栈/ret出栈
调用函数压栈/出栈

C语言编译器对函数调用的处理方式

main

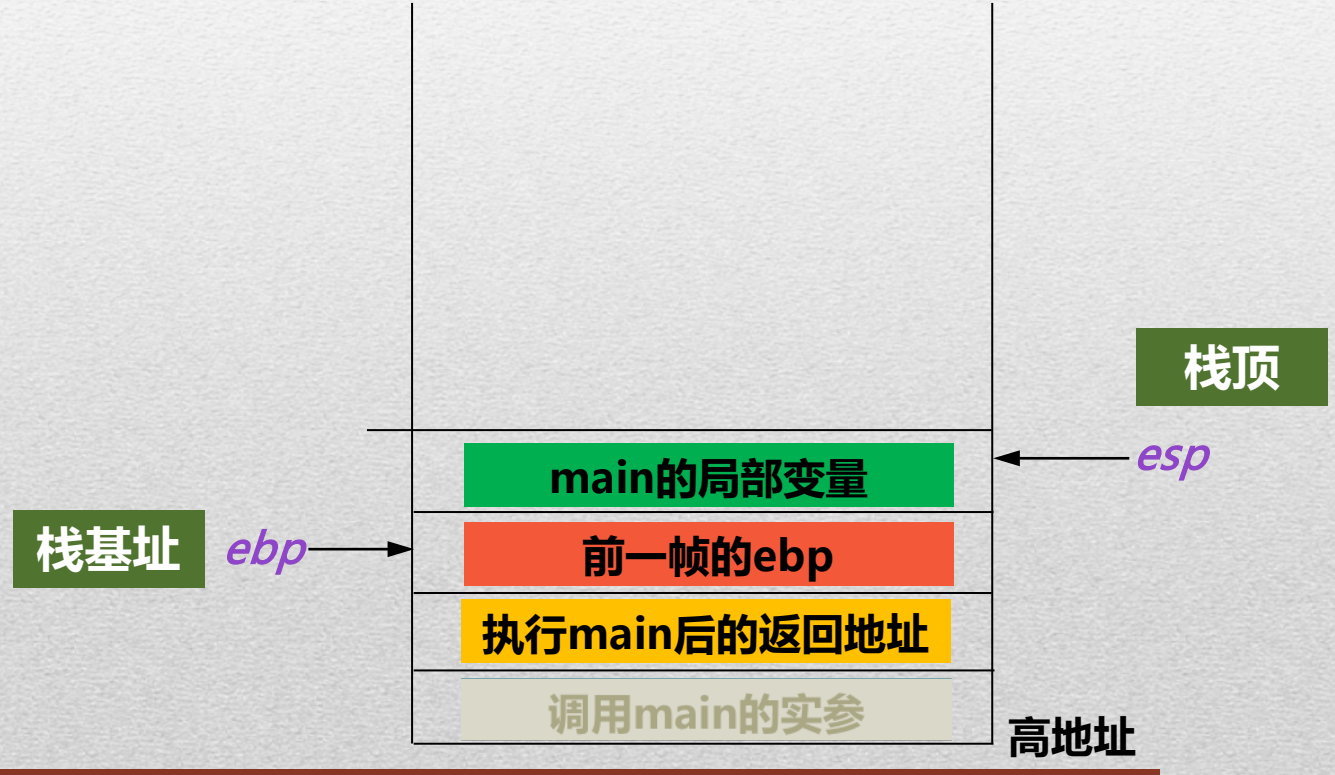
```
...
L-2:    push [%ebp - 8]
L-1:    push [%ebp - 4]
L:      call   sum
L+1:    add    %esp, 8
...
```

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);
}
```



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

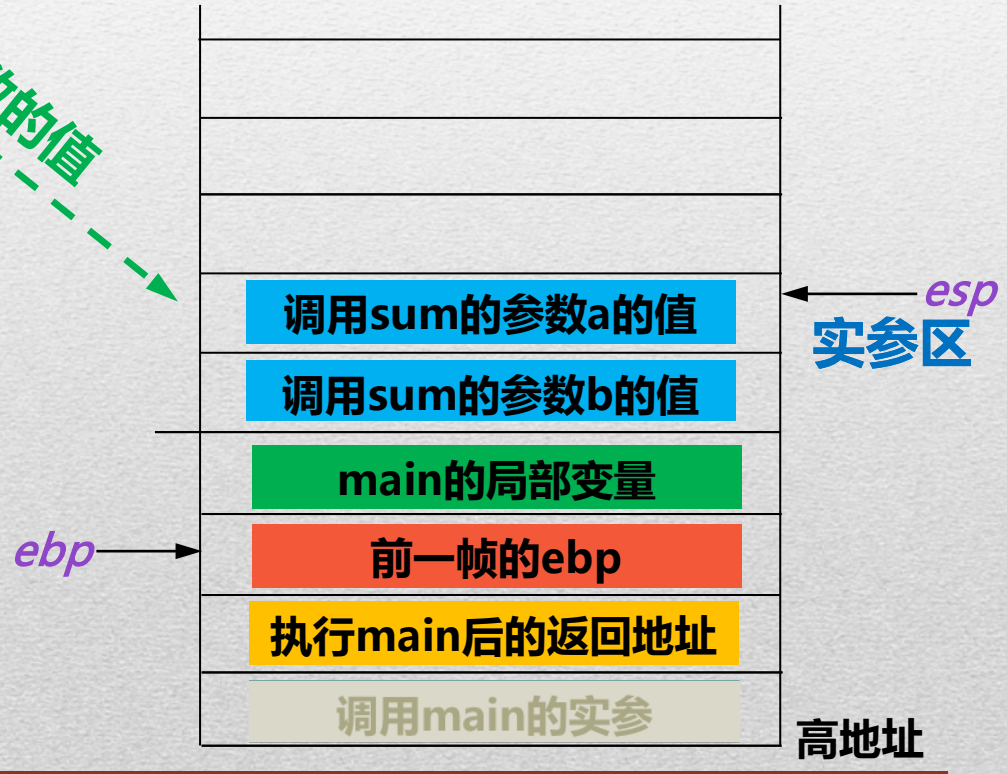
```
...
T:      push   %ebp
T+1:    mov    %esp,   %ebp
T+2:    sub    %esp,    4
...
        mov    [%ebp-4],  %eax
        mov    %ebp,      %esp
        pop    %ebp
        ret
...
```

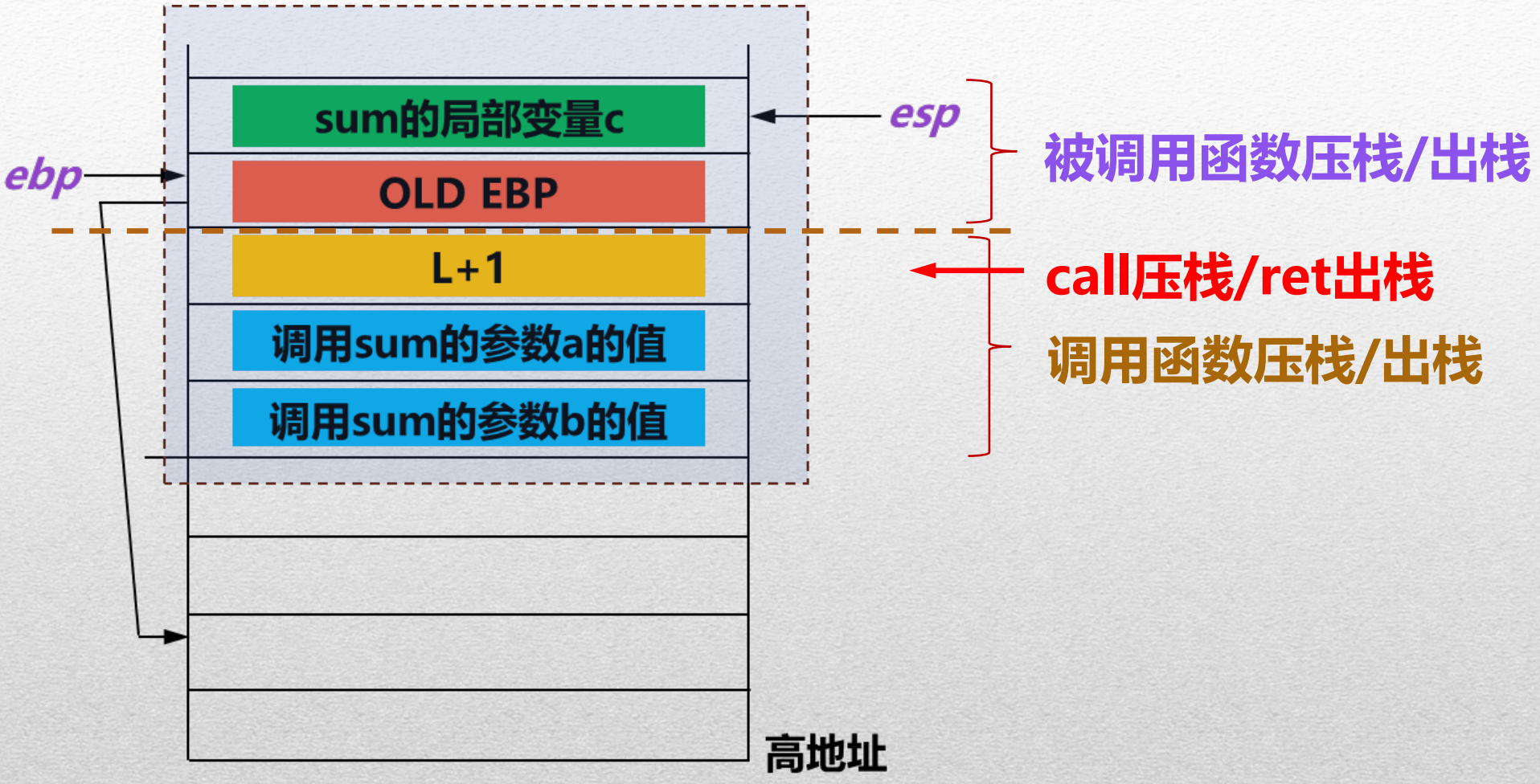
[%ebp - 8]，[%ebp - 4]分别是main的局部变量b和a的值

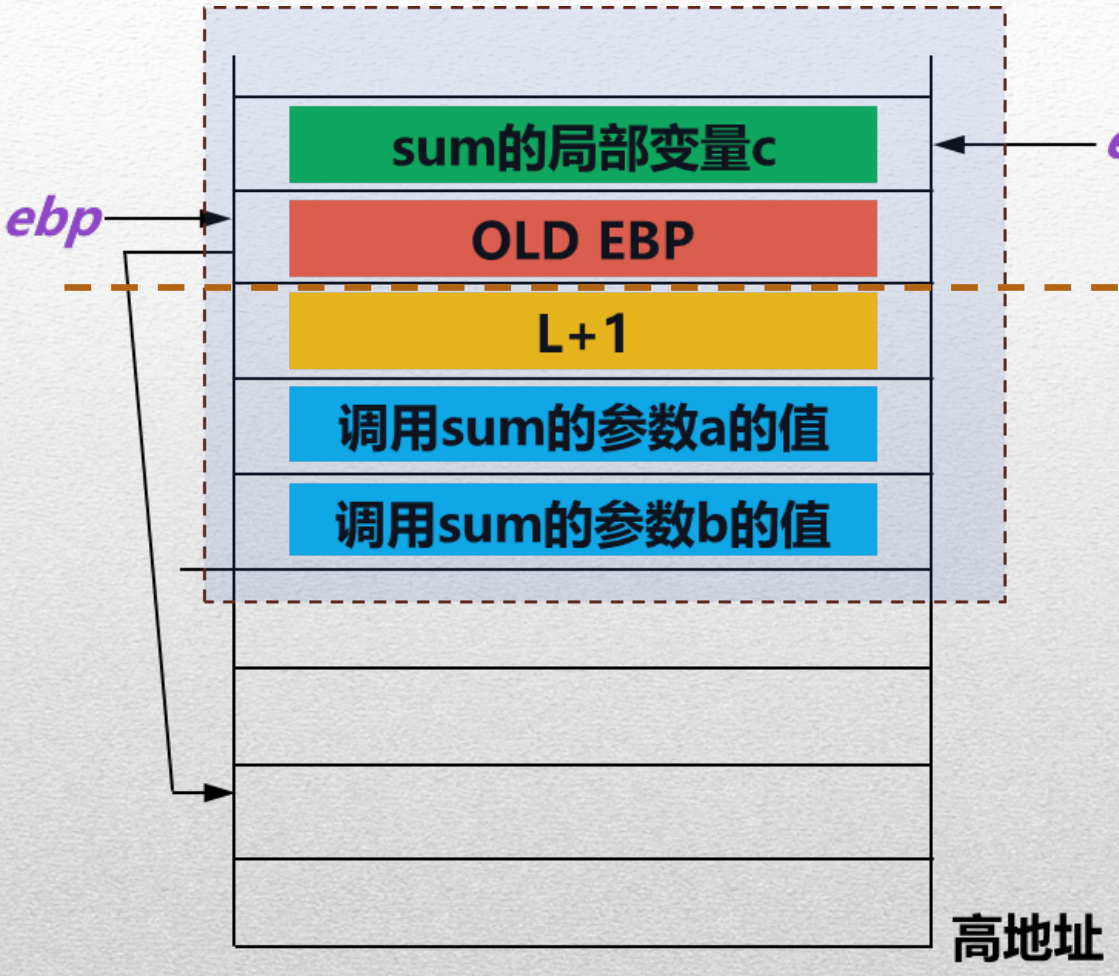
按逆序压入参数的值

```
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);
}
```




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
ebp
高地址
被调用函数压栈/出栈
call压栈/ret出栈
调用函数压栈/出栈



被调用函数压栈/出栈

call压栈/ret出栈

调用函数压栈/出栈

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?