第二章

并发进程

主 要 内 容

2.1 进程基本概念

2.2 处理机调度与死锁

2.3 UNIX的进程

2.4 中断的基本概念及UNIX中断处理

2.5 进程通信

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

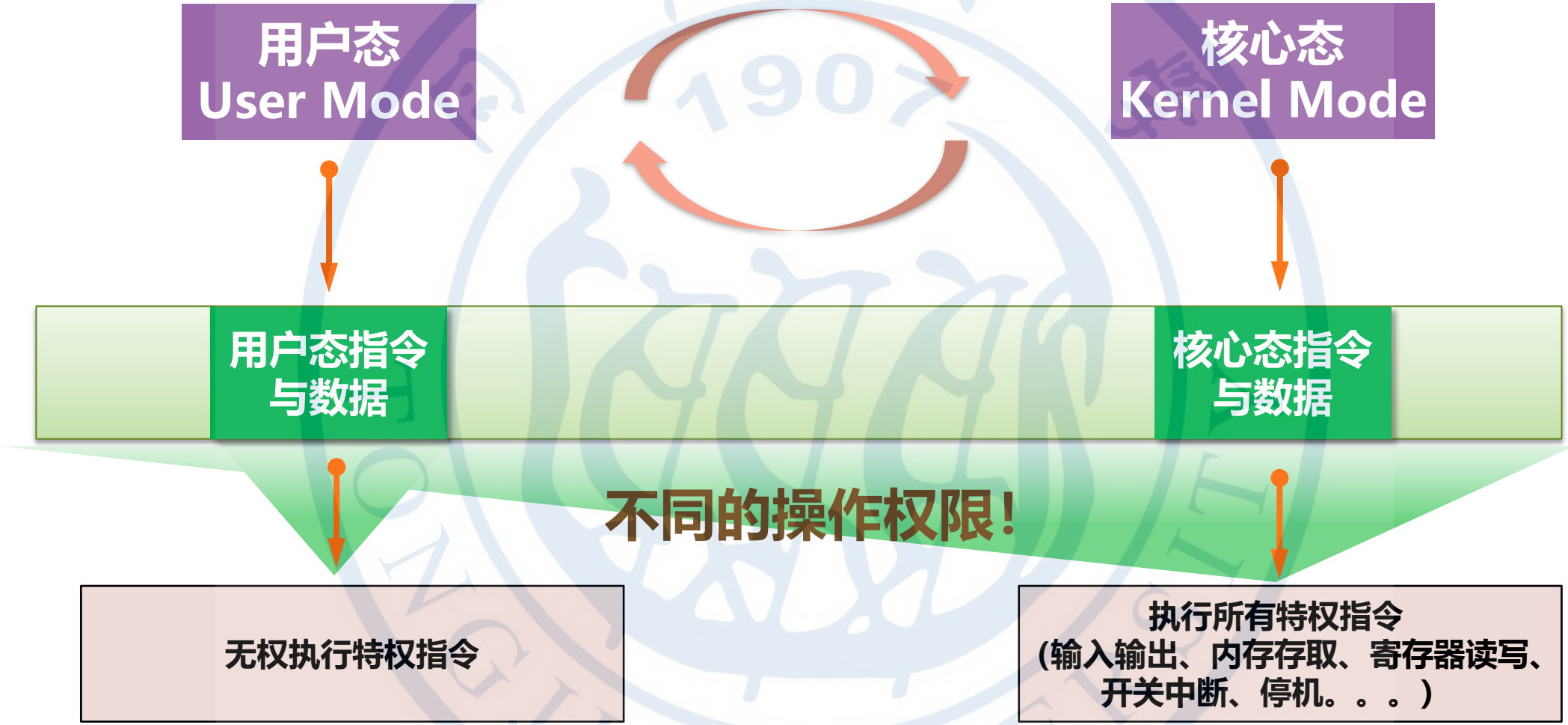
执行内核程序，
提供系统功能

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



程序的基本概念

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！！！

UNIX进程的结构特征

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);
}
```

UNIX进程的结构特征

可执行文件结构

```
#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

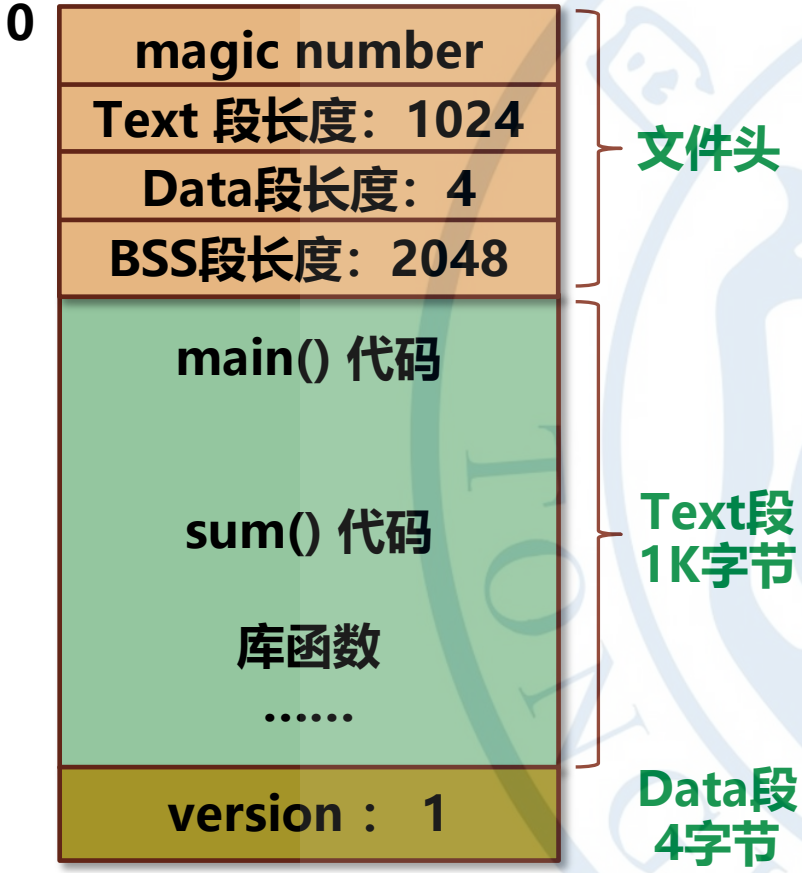
Data段
4字节

可执行程序被保存在磁盘上

经过编译、链接形成可执行文件

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

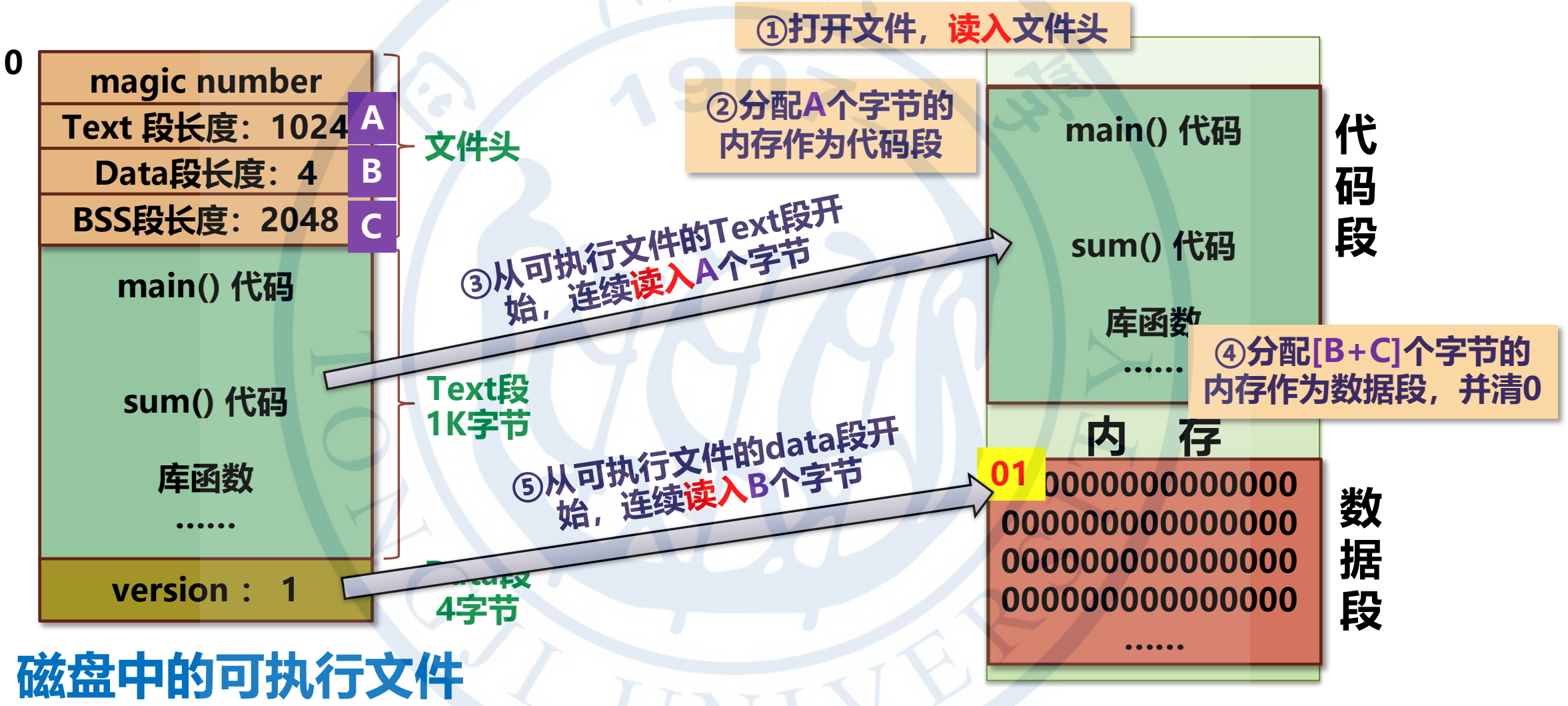
UNIX进程的结构特征



磁盘中的可执行文件

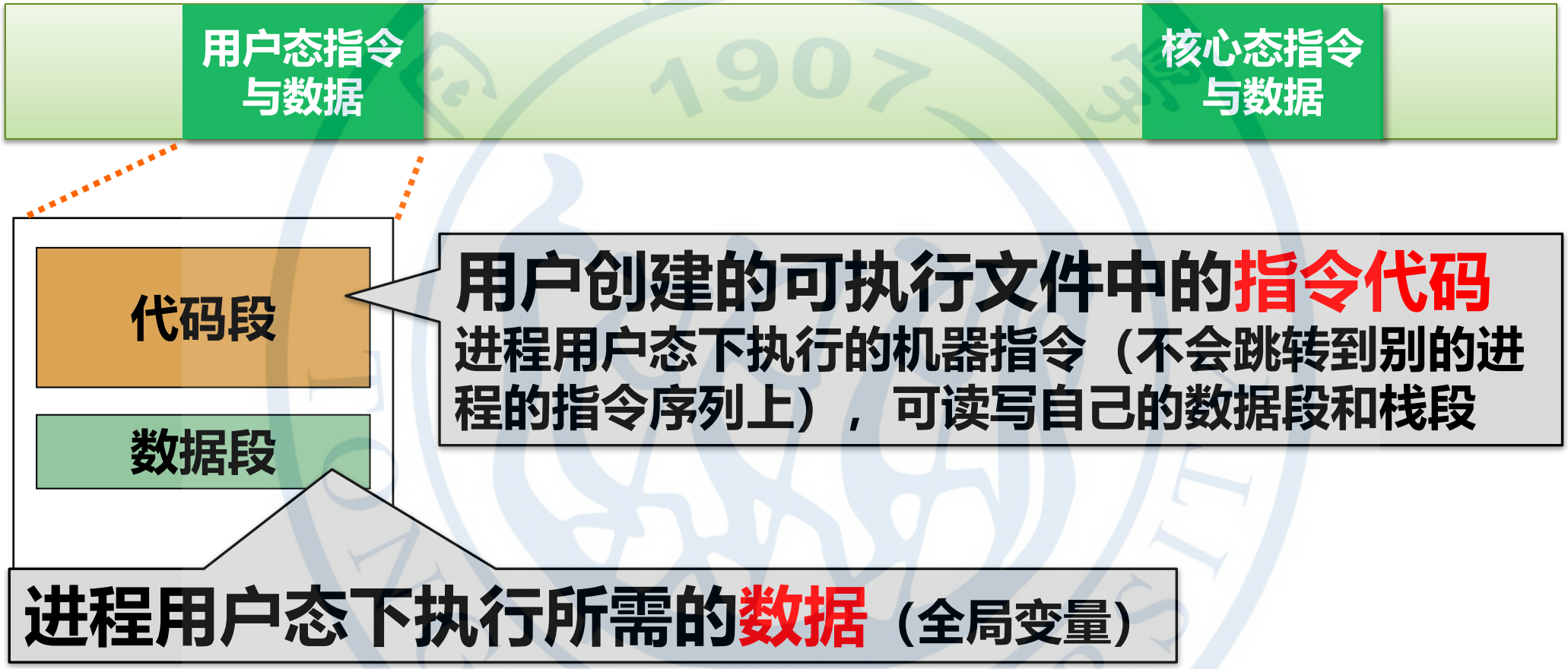
进程创建时，可执行文件如何装入内存?

UNIX进程的结构特征



磁盘中的可执行文件

UNIX进程的结构特征



程序中所有的内容都在内存了么？

UNIX进程的结构特征

```
#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进程的结构特征

用户态指令与数据

核心态指令与数据

代码段

数据段

用户栈

UNIX进程的结构特征


```
#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

```
        …
L-2:  push DWORD PTR [ebp-0x8]
L-1:  push DWORD PTR [ebp-0x4]
L:    call   sum
L+1:  add    esp, 0x8
        …
```

sum编译后

```
T:    push  ebp
T+1:  mov   ebp,   esp
T+2:  sub   esp,   0x10
```

完成加法计算

```
      mov eax, DWORD PTR [ebp-0x4]
      leave
      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

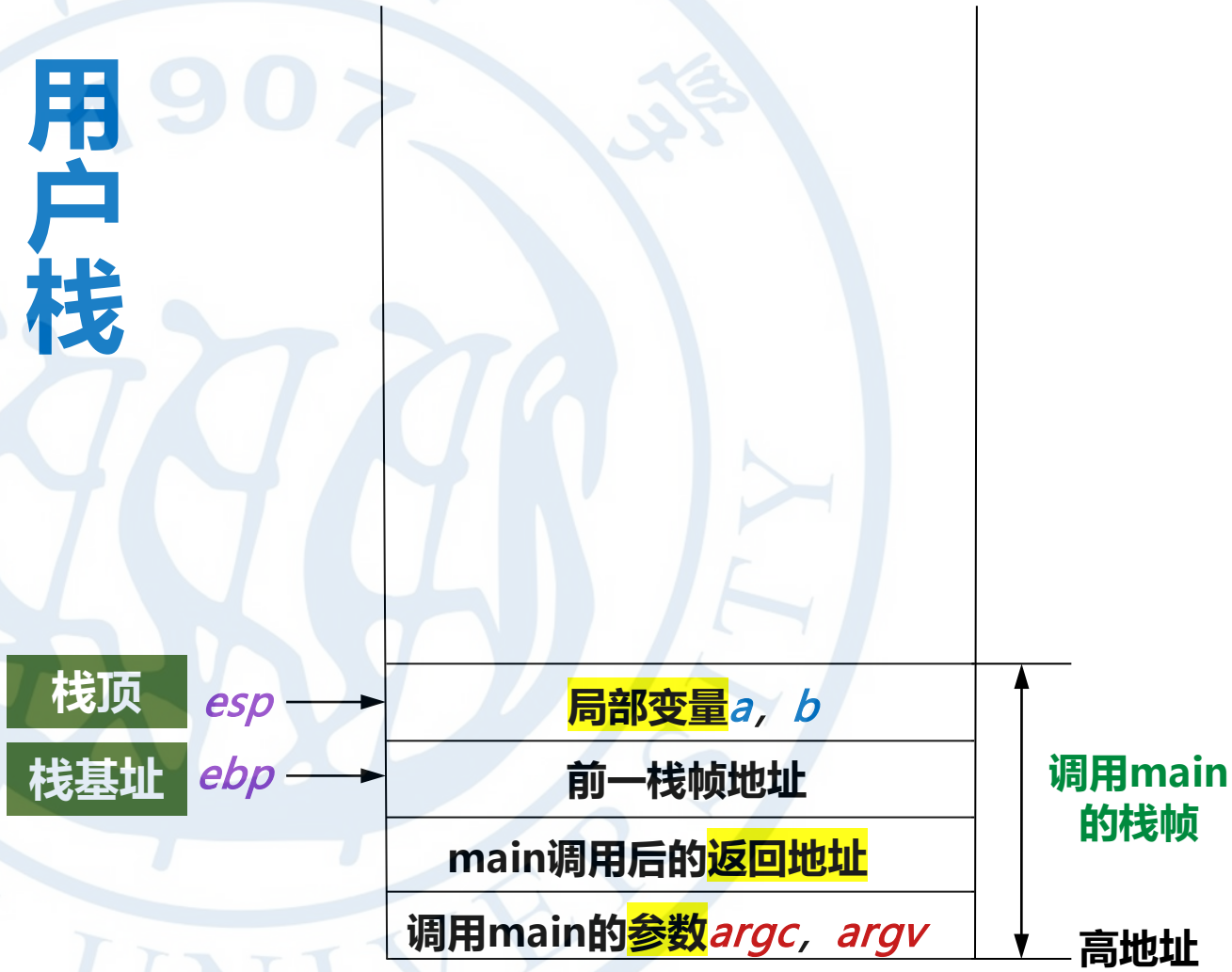
UNIX进程的结构特征

```
#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进程的结构特征

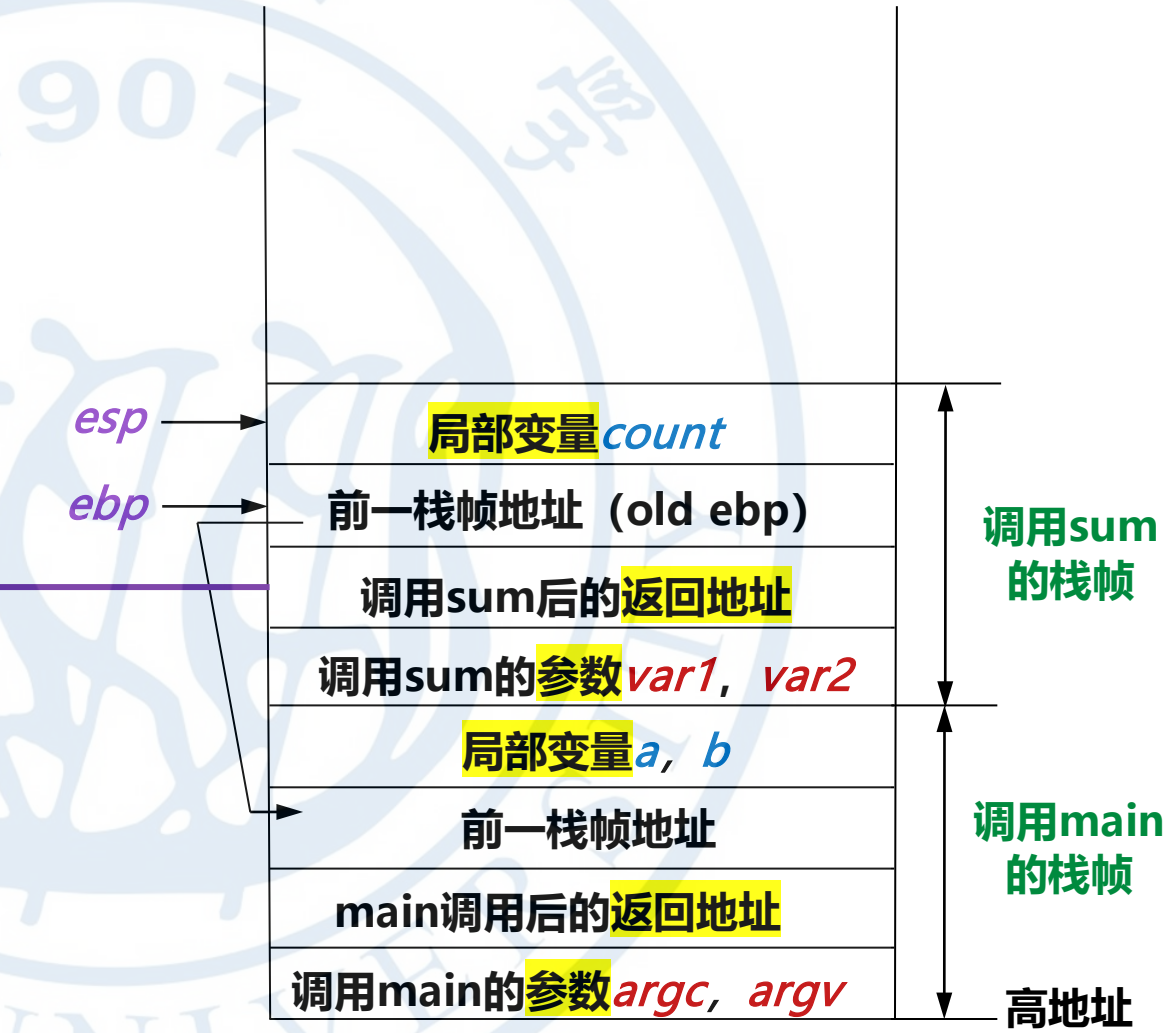

```
#include < fcntl.h>
char buffer[2048];
int version = 1;
main( argc, argv)
int   argc;
char *argv[];
{
    int a, b;
    ……;
    sum(a, b);
    exit(0);
}
int sum( var1, var2)
int var1, var2;
{
    int count;
    count = var1 +var2;
    return(count);
}
```



UNIX进程的结构特征

用户态地址空间的逻辑结构

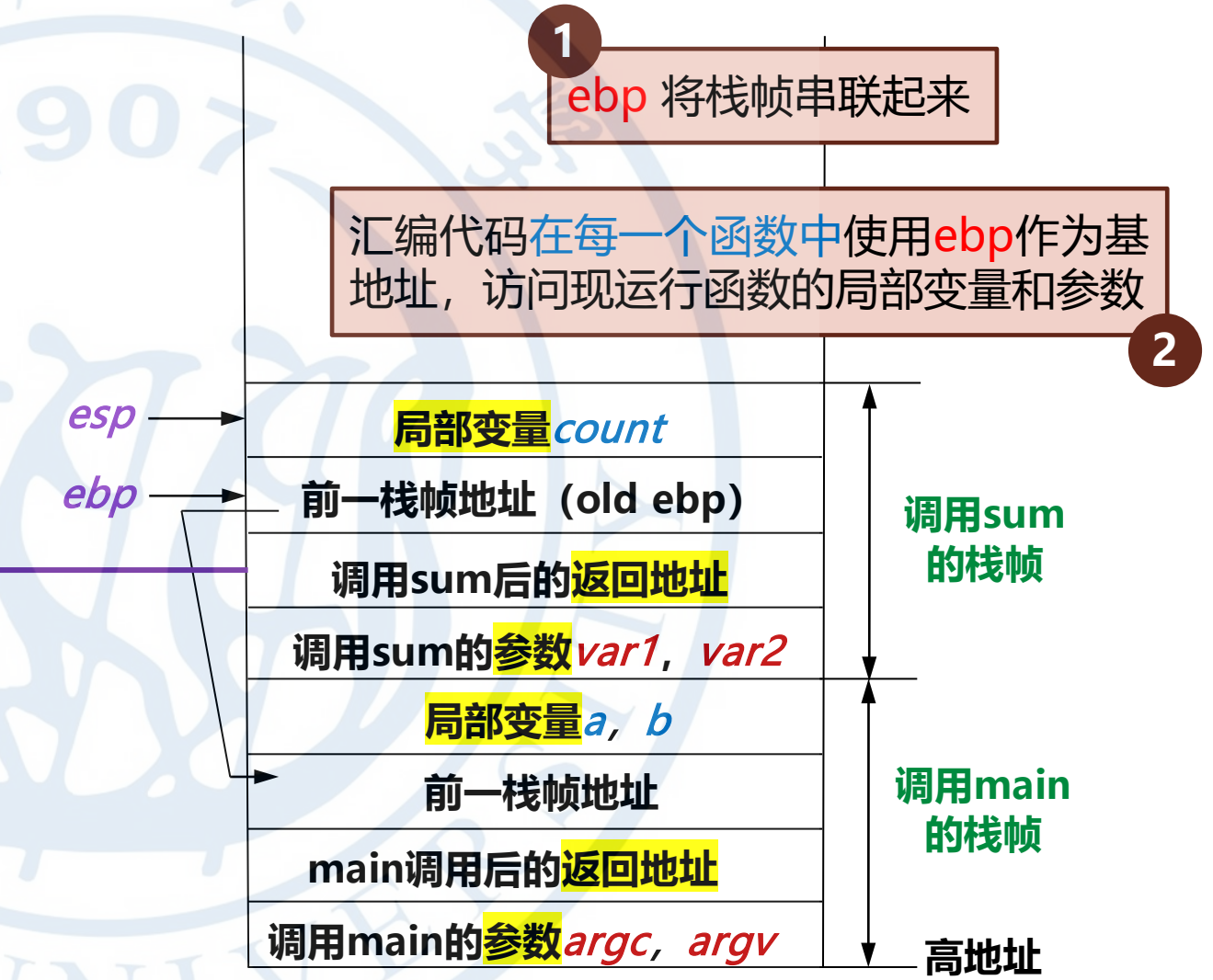
```
#include < fcntl.h>
char buffer[2048];
int version = 1;
main( argc, argv)
int   argc;
char *argv[];
{
    int a, b;
    ……;
    sum(a, b);
    exit(0);
}
int sum( var1, var2)
int var1, var2;
{
    int count;
    count = var1 +var2;
    return(count);
}
```



UNIX进程的结构特征

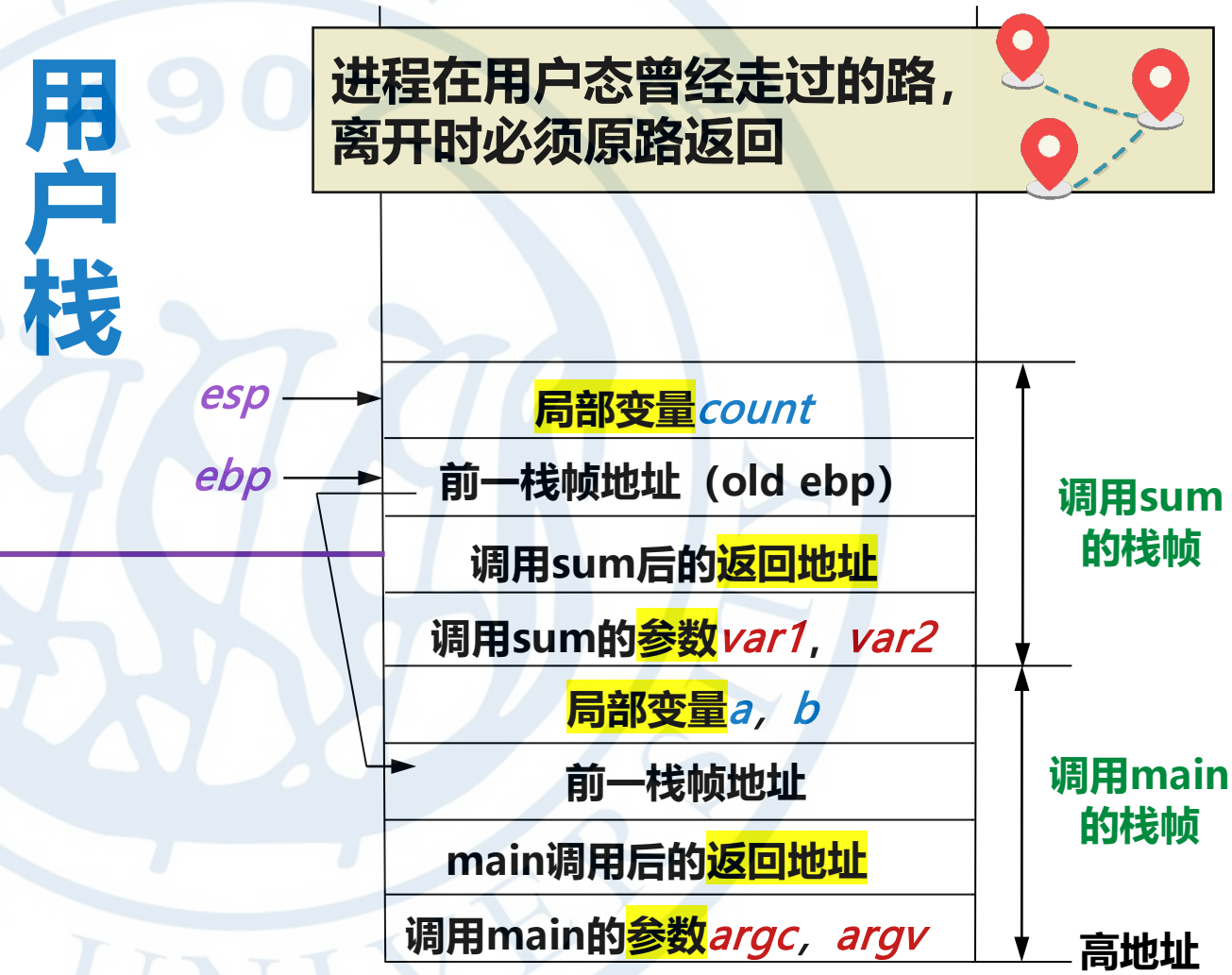
用户态地址空间的逻辑结构

```
#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);
}
```



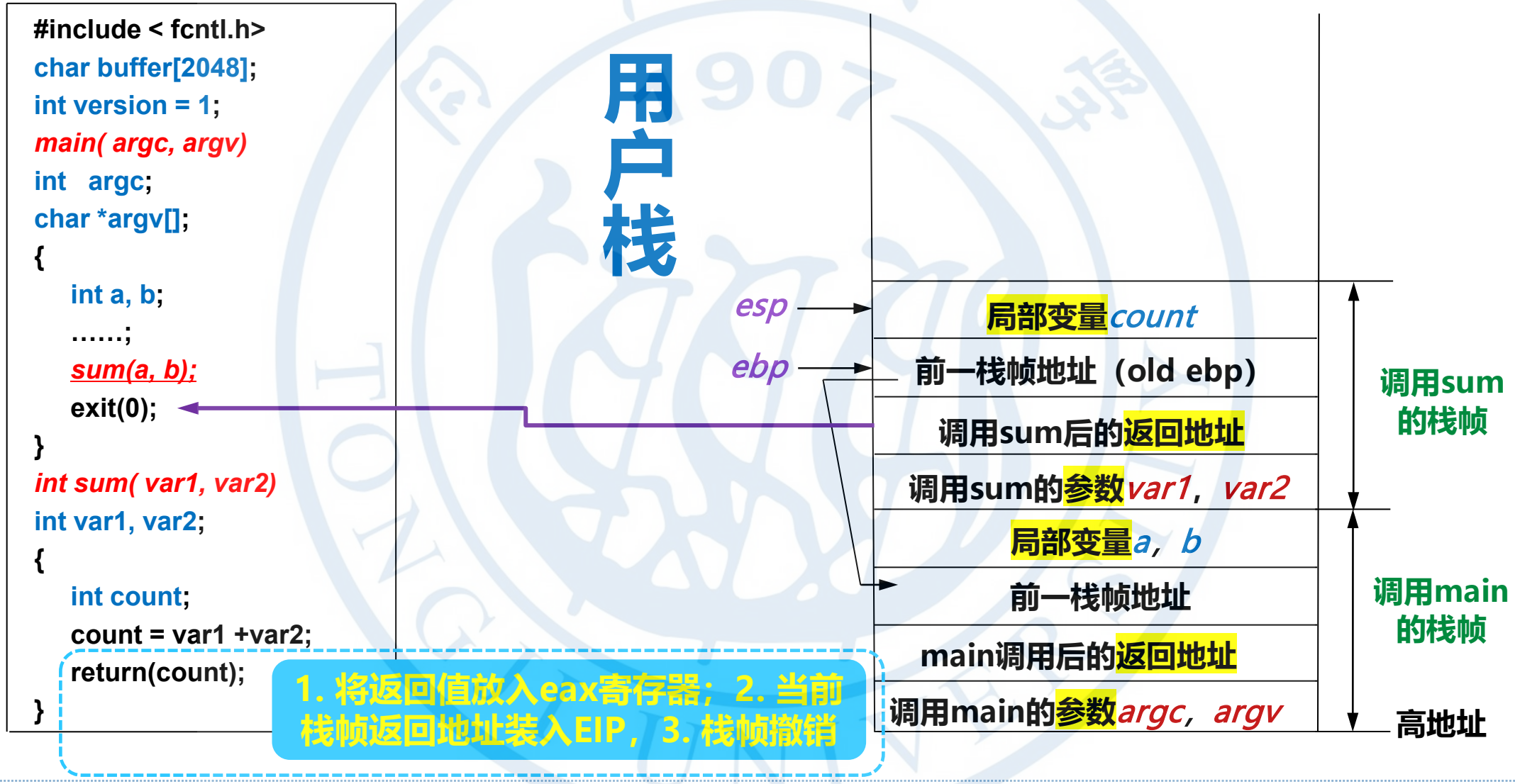
UNIX进程的结构特征


```
#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进程的结构特征


```
#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);
}
```

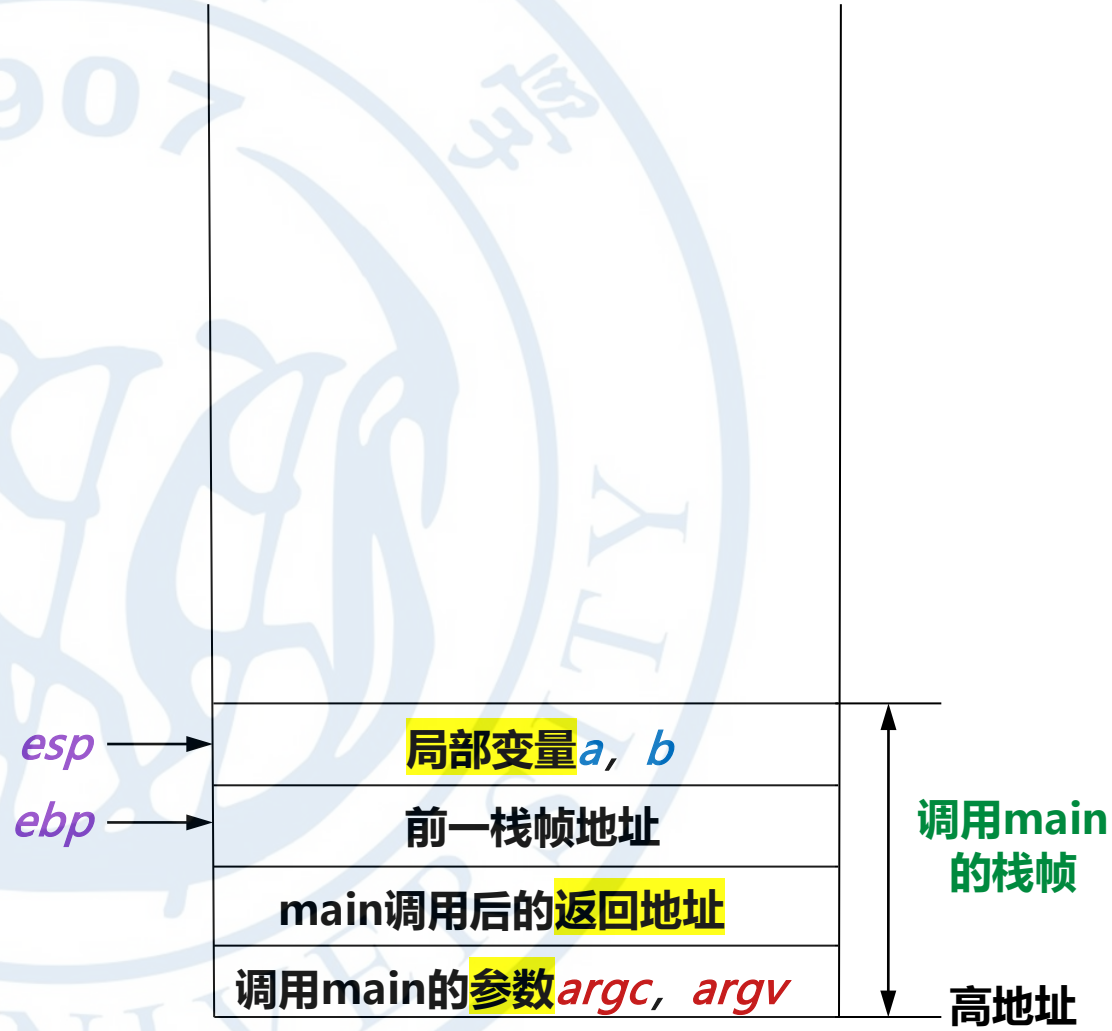


1. 将返回值放入eax寄存器；2. 当前栈帧返回地址装入EIP，3. 栈帧撤销

UNIX进程的结构特征


```
#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

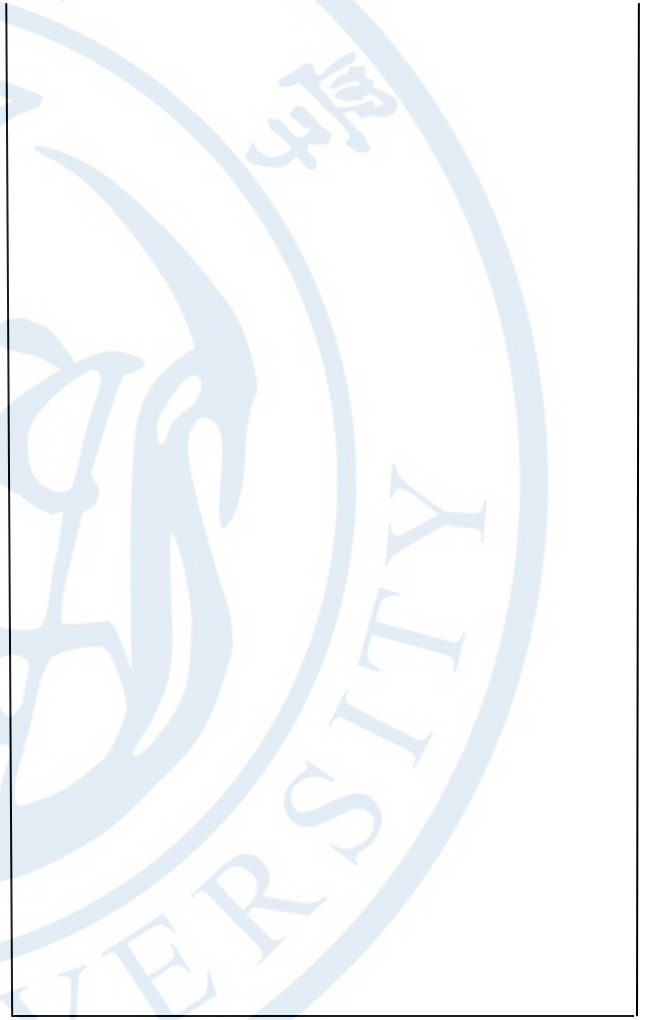


UNIX进程的结构特征

用户态地址空间的逻辑结构

```
#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

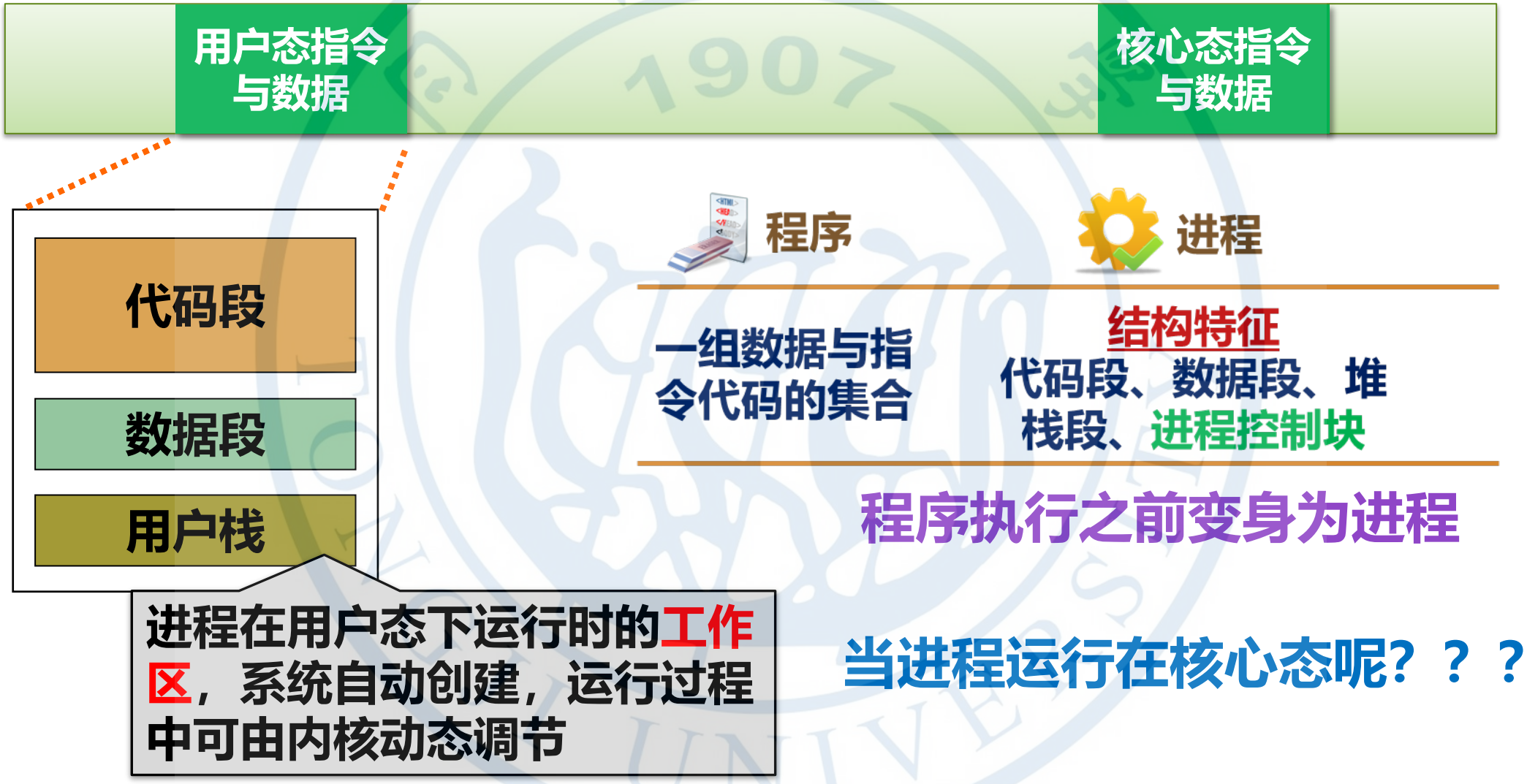


高地址

UNIX进程的结构特征

用户态地址空间的逻辑结构

UNIX进程的结构特征



程序

一组数据与指令代码的集合

进程

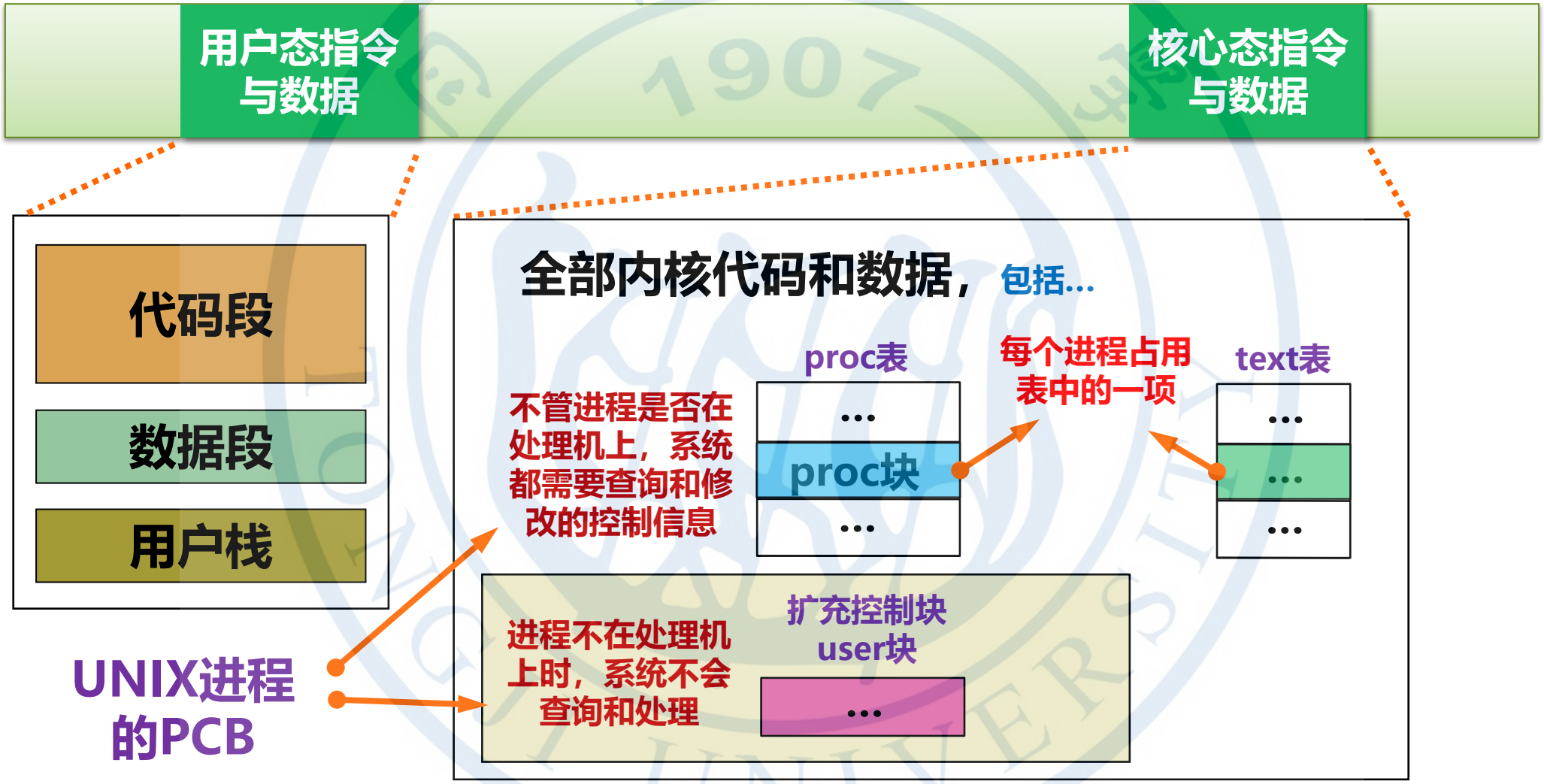
结构特征

代码段、数据段、堆栈段、进程控制块

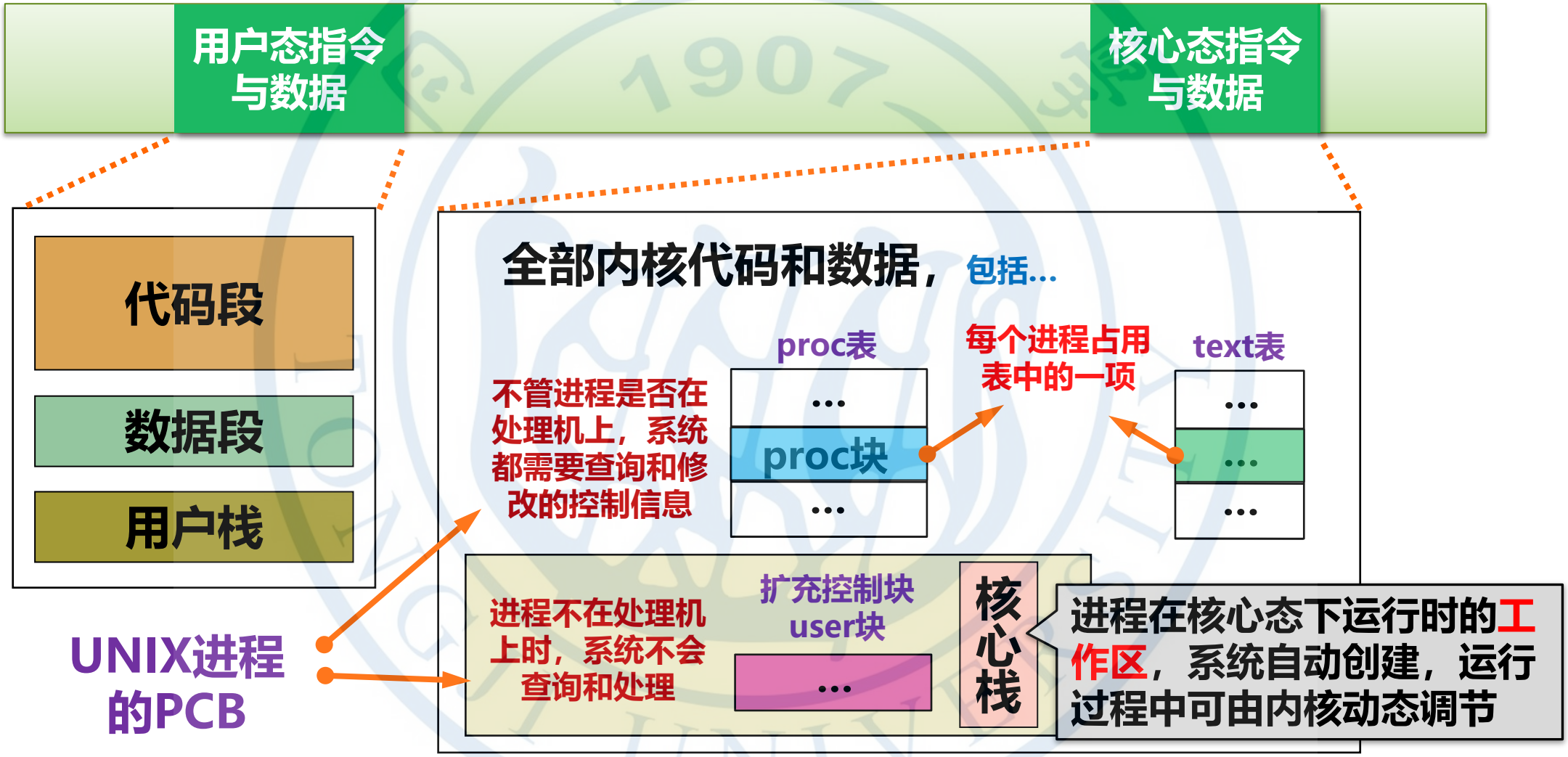
程序执行之前变身为进程

当进程运行在核心态呢？？？

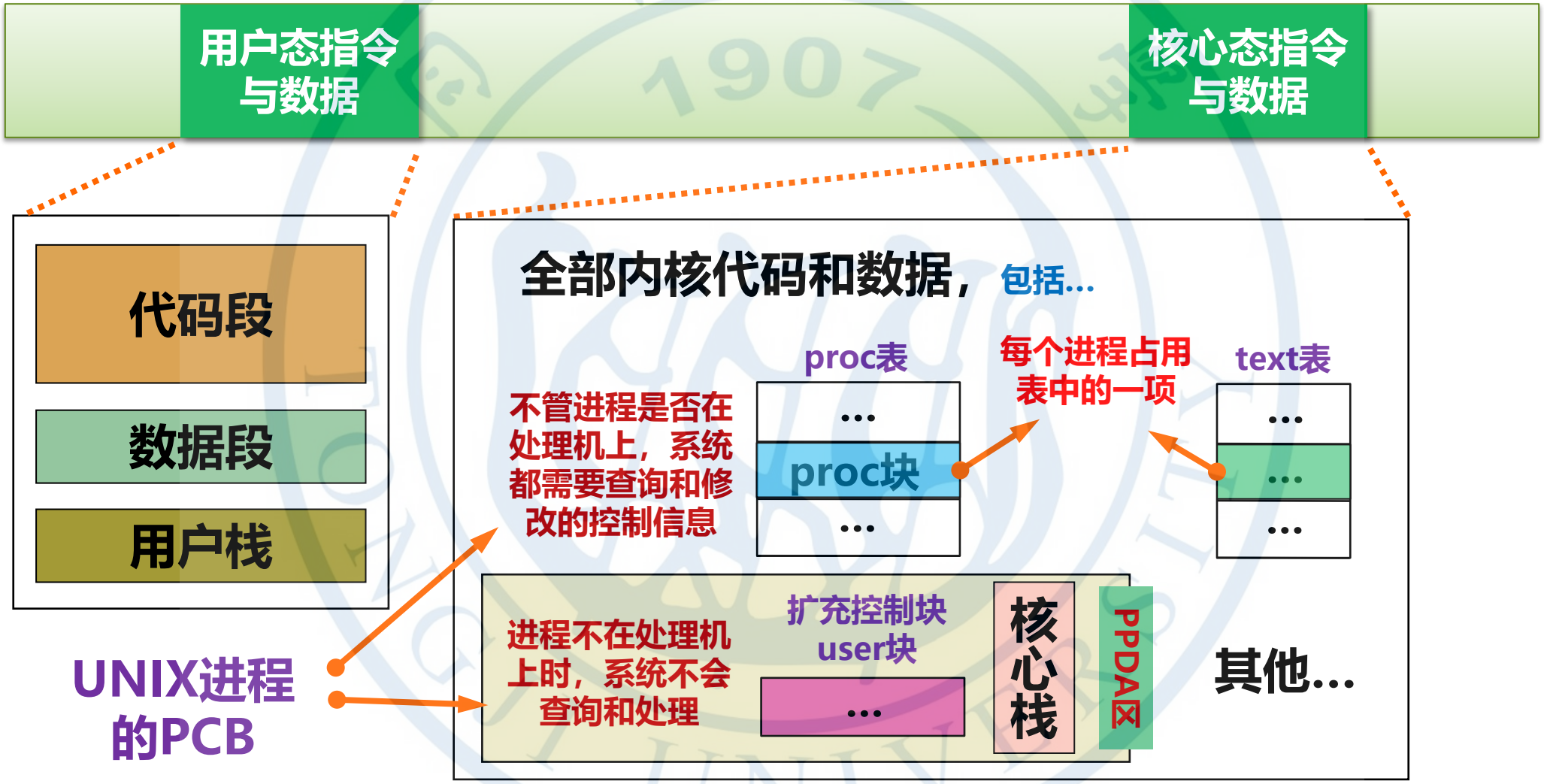
UNIX进程的结构特征



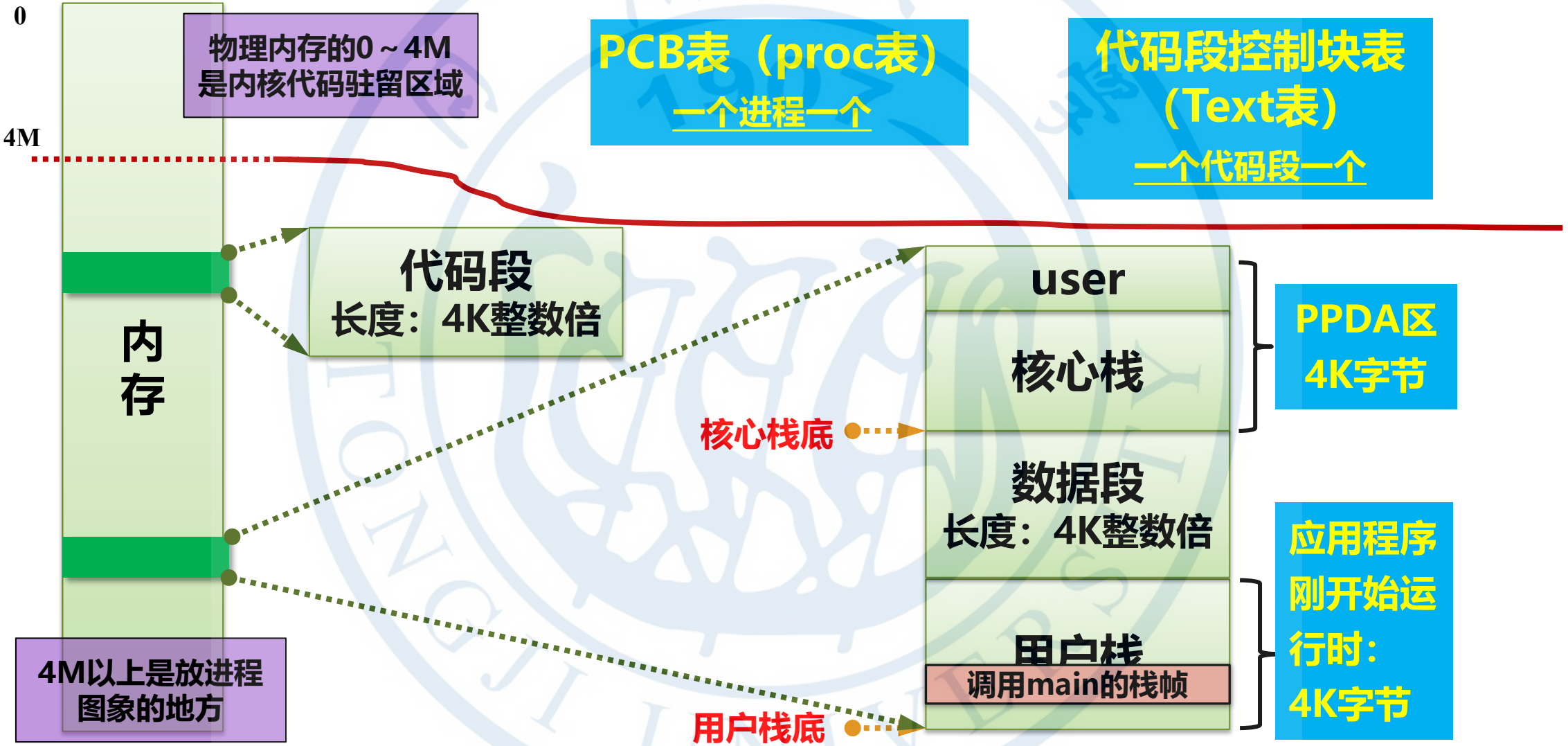
UNIX进程的结构特征



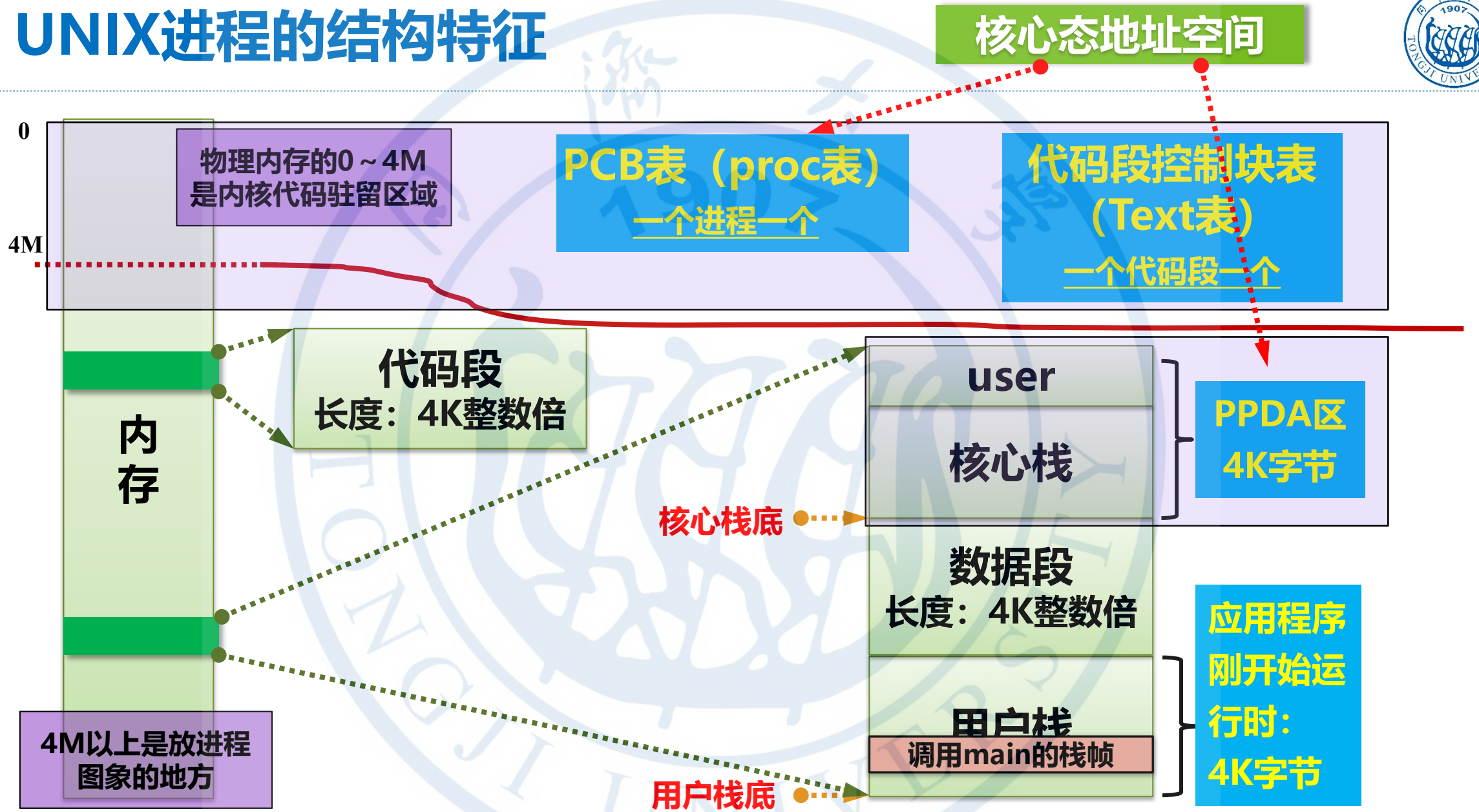
UNIX进程的结构特征



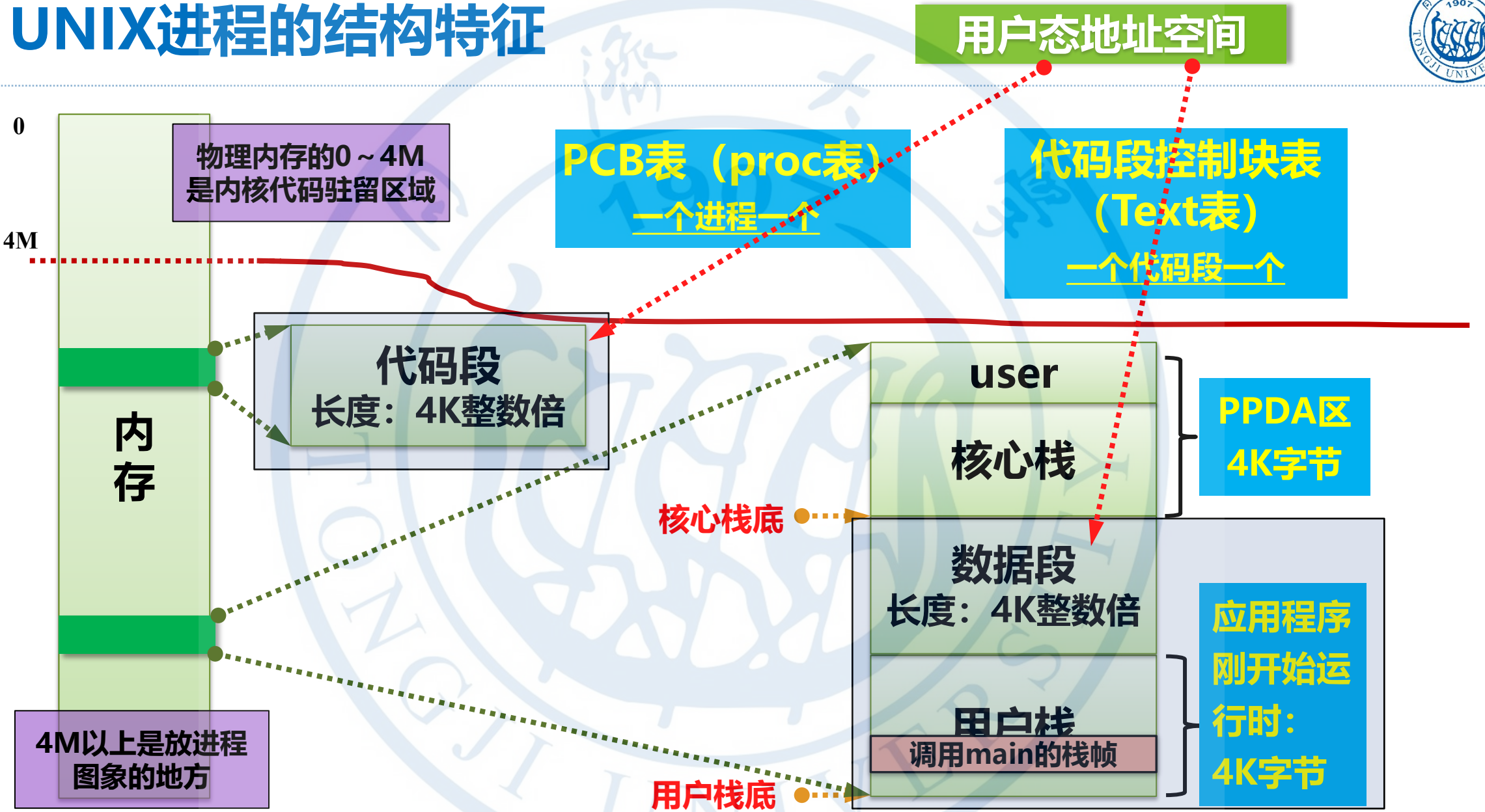
UNIX进程的结构特征



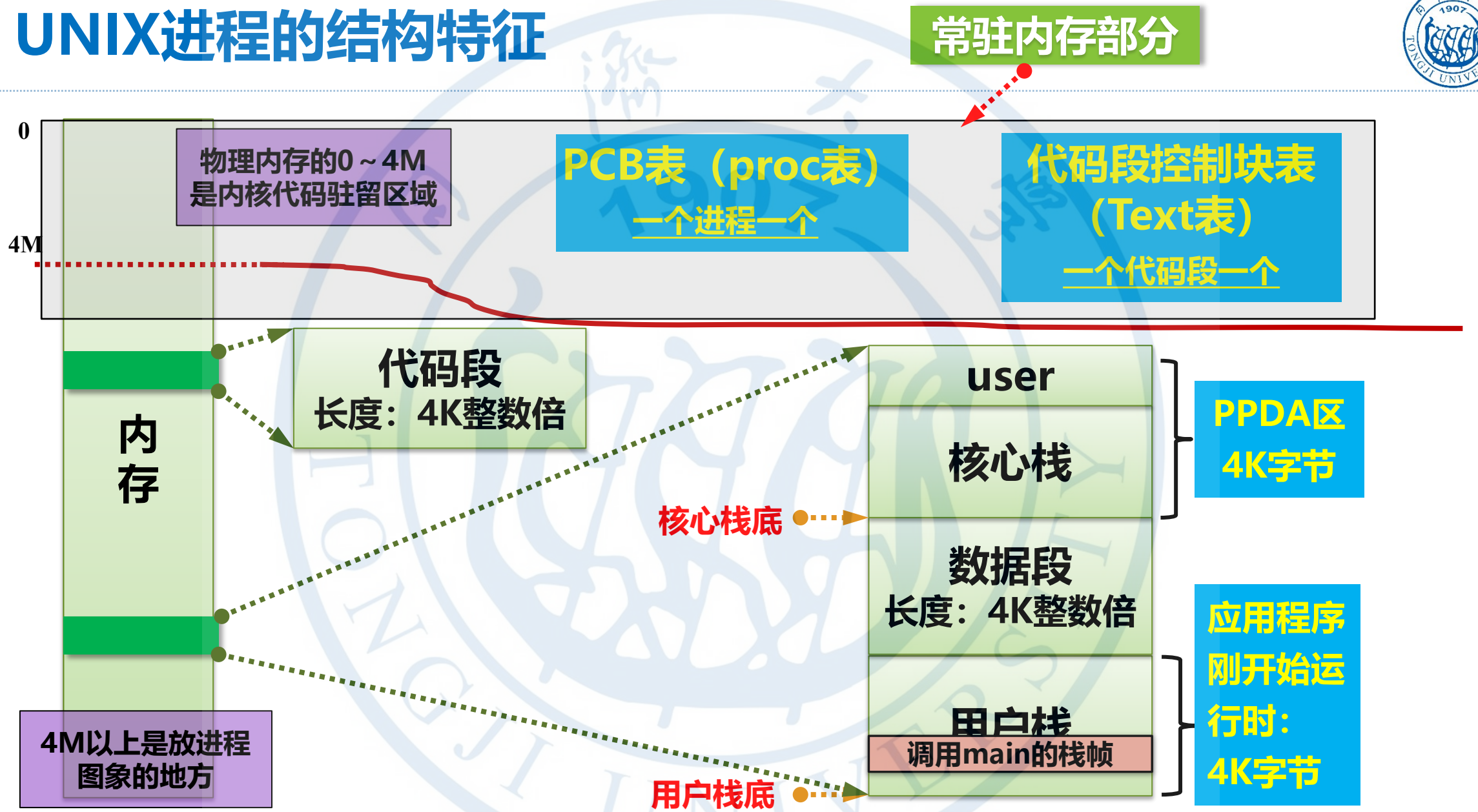
UNIX进程的结构特征



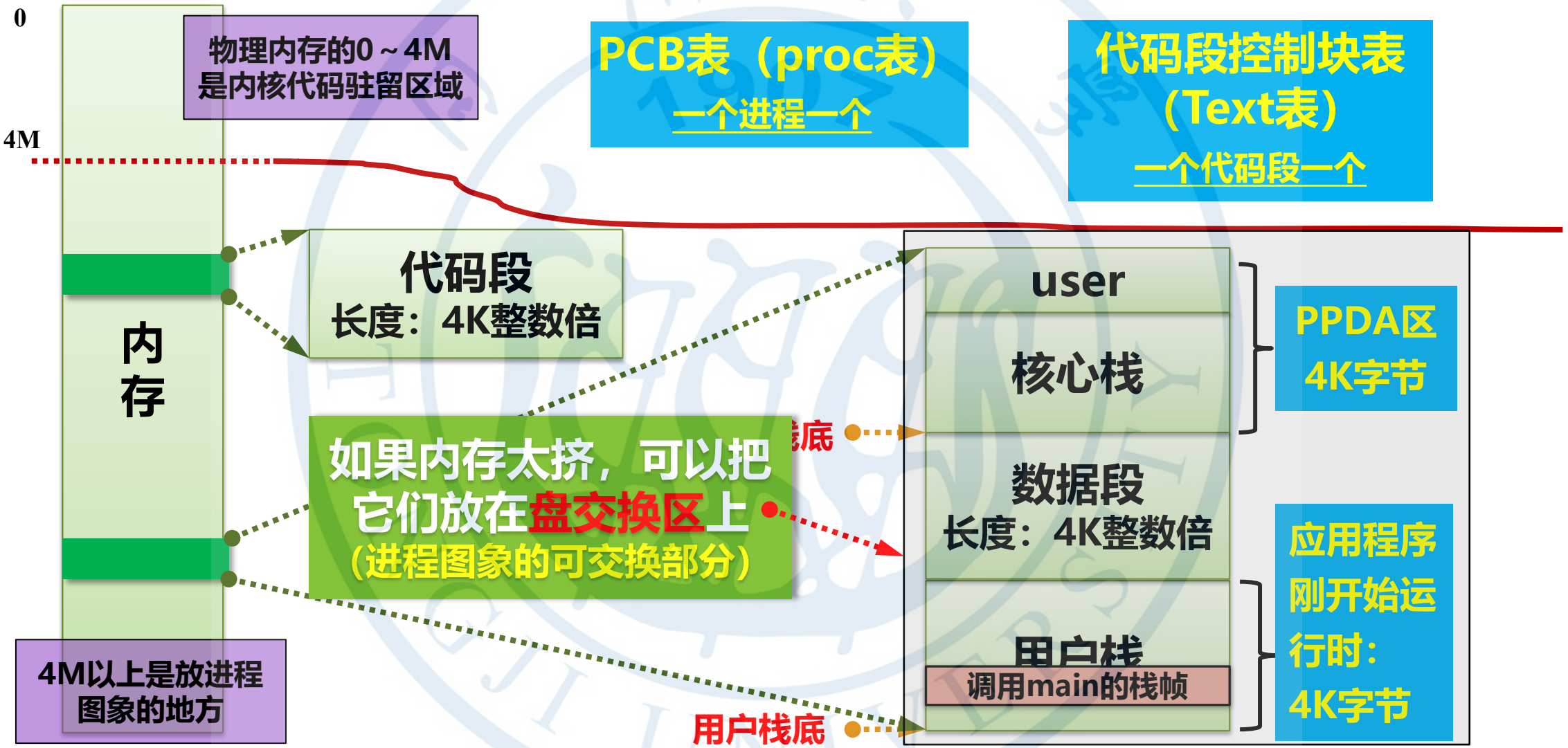
UNIX进程的结构特征



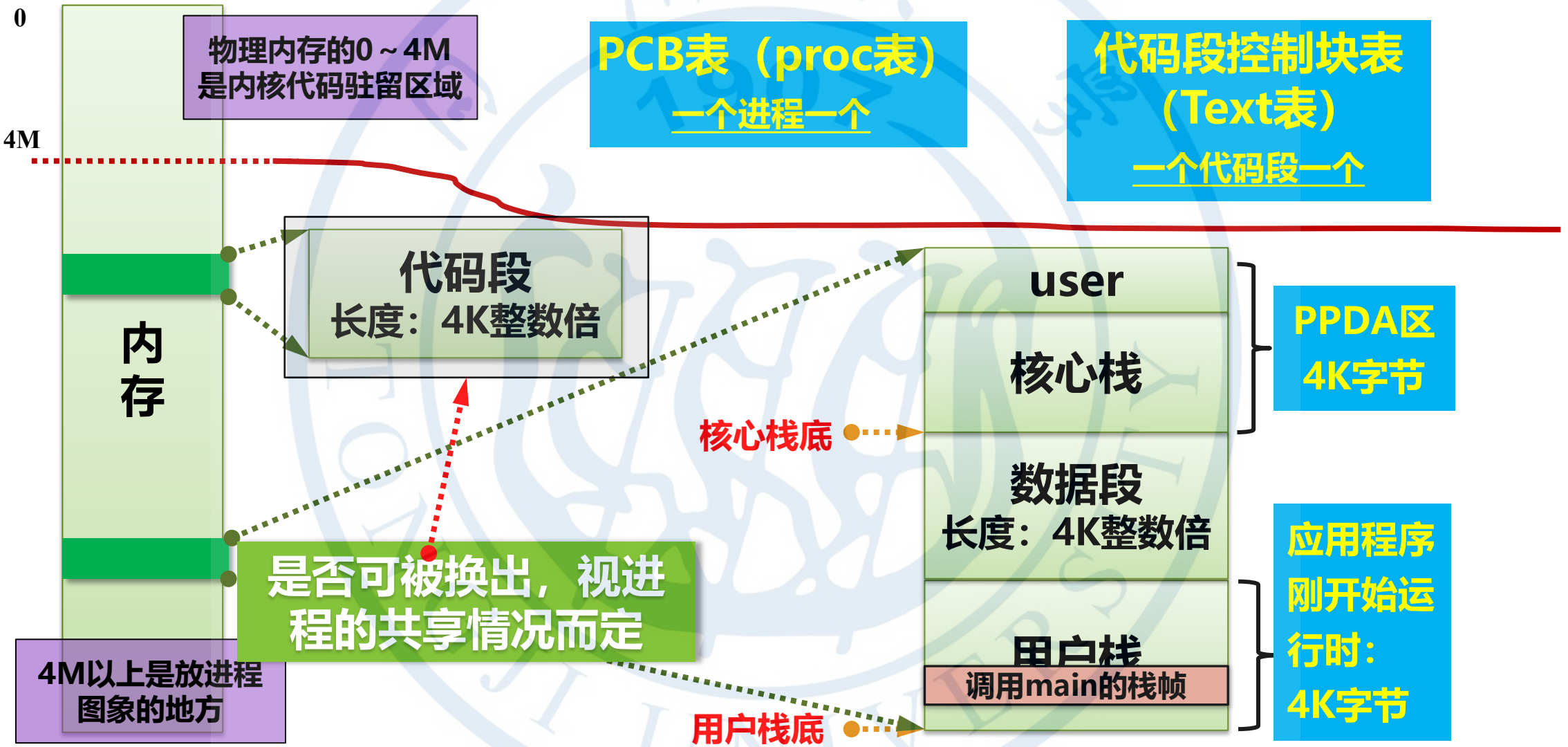
UNIX进程的结构特征



UNIX进程的结构特征



UNIX进程的结构特征



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

阅读讲义：71页 ~ 74页；100页 ~ 103页

栈的形成过程

C语言编译器对函数调用的处理方式

```
void main1()
{
    int a,b,result;
    a=1;
    b=2;
    result=sum(a,b);
}
```

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

```
        ...
L-2:  push DWORD PTR [ebp-0x8]
L-1:  push DWORD PTR [ebp-0x4]
L:    call   sum
L+1:  add    esp, 0x8
        ...
```

```
T:    push  ebp
T+1:  mov   ebp,  esp
T+2:  sub   esp,  0x10
```

完成加法计算

```
        mov eax, DWORD PTR [ebp-0x4]
        leave
        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

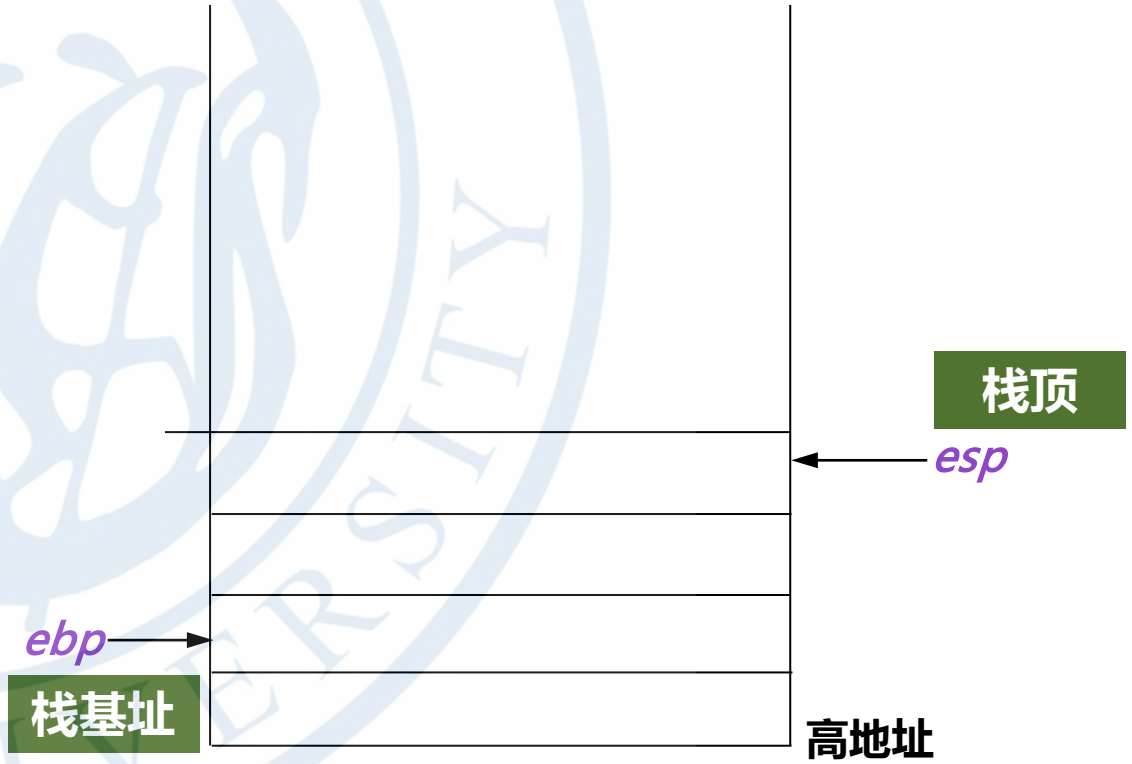
```
...
L-2:  push DWORD PTR [ebp-0x8]
L-1:  push DWORD PTR [ebp-0x4]
L:    call   sum
L+1:  add    esp, 0x8
...
```

sum

```
T:    push  ebp
T+1:  mov   ebp,  esp
T+2:  sub   esp,  0x10
```

完成加法计算

```
      mov eax, DWORD PTR [ebp-0x4]
      leave
      ret
```



C语言编译器对函数调用的处理方式

main

```
...
L-2:  push DWORD PTR [ebp-0x8]
L-1:  push DWORD PTR [ebp-0x4]
L:    call  sum
L+1:  add   esp, 0x8
...
```

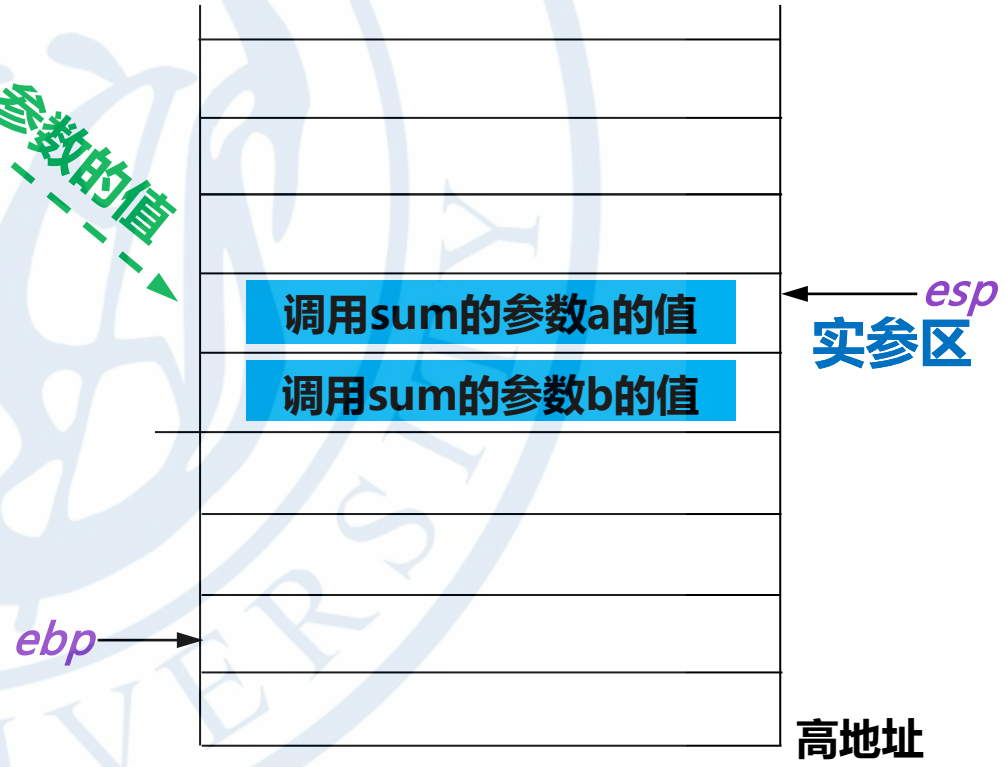
sum

```
T:    push  ebp
T+1:  mov   ebp,  esp
T+2:  sub   esp,  0x10
```

完成加法计算

```
      mov eax, DWORD PTR [ebp-0x4]
      leave
      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 DWORD PTR [ebp-0x8]
L-1:  push DWORD PTR [ebp-0x4]
L:    call  sum
L+1:  add   esp, 0x8
...
```

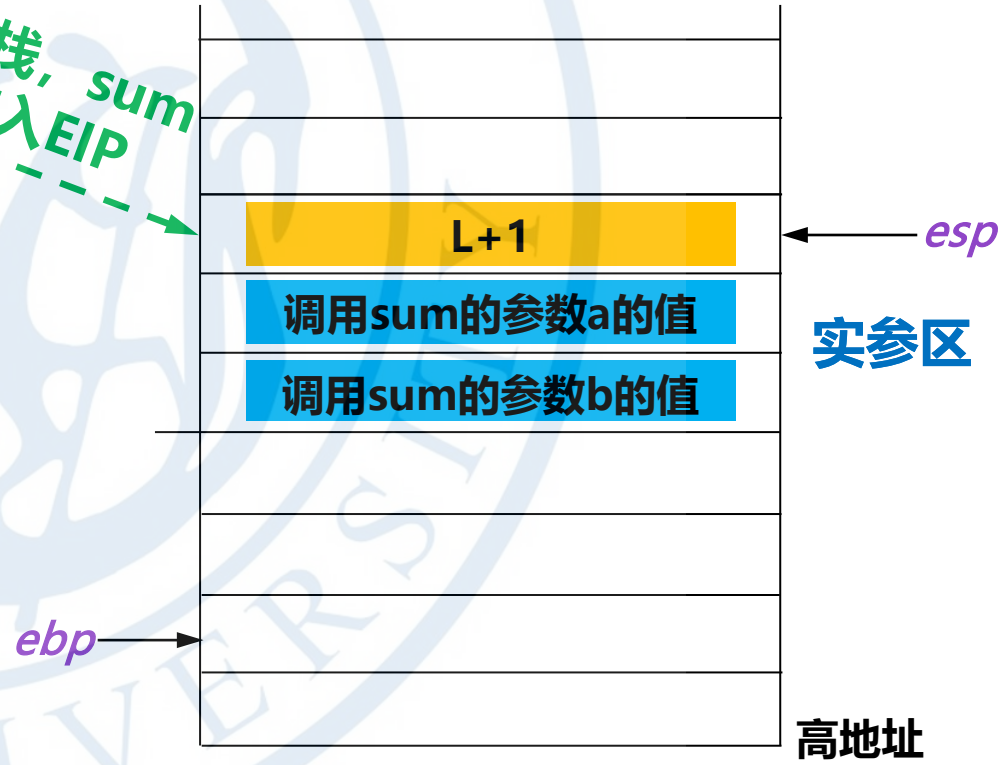
sum

```
T:    push  ebp
T+1:  mov   ebp,  esp
T+2:  sub   esp,  0x10
```

完成加法计算

```
      mov eax, DWORD PTR [ebp-0x4]
      leave
      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 DWORD PTR [ebp-0x8]
L-1:  push DWORD PTR [ebp-0x4]
L:    call  sum
L+1:  add   esp, 0x8
...
```

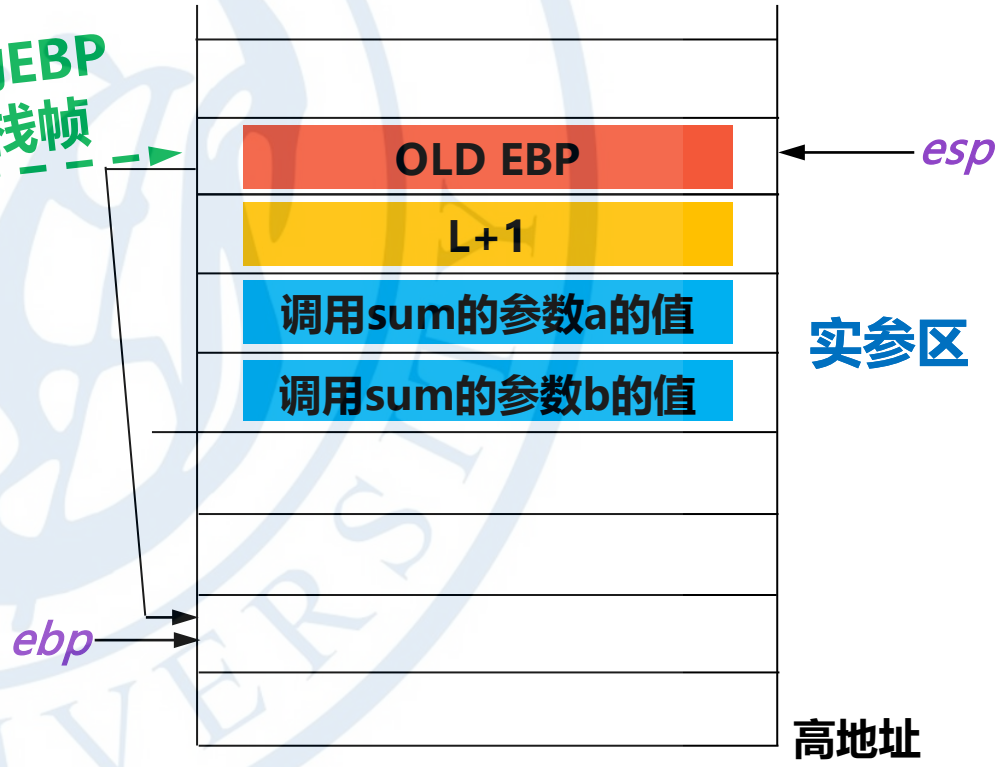
sum

```
T:    push  ebp
T+1:  mov   ebp,  esp
T+2:  sub   esp,  0x10
```

完成加法计算

```
      mov eax, DWORD PTR [ebp-0x4]
      leave
      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 DWORD PTR [ebp-0x8]
L-1:  push DWORD PTR [ebp-0x4]
L:    call   sum
L+1:  add    esp, 0x8
...
```

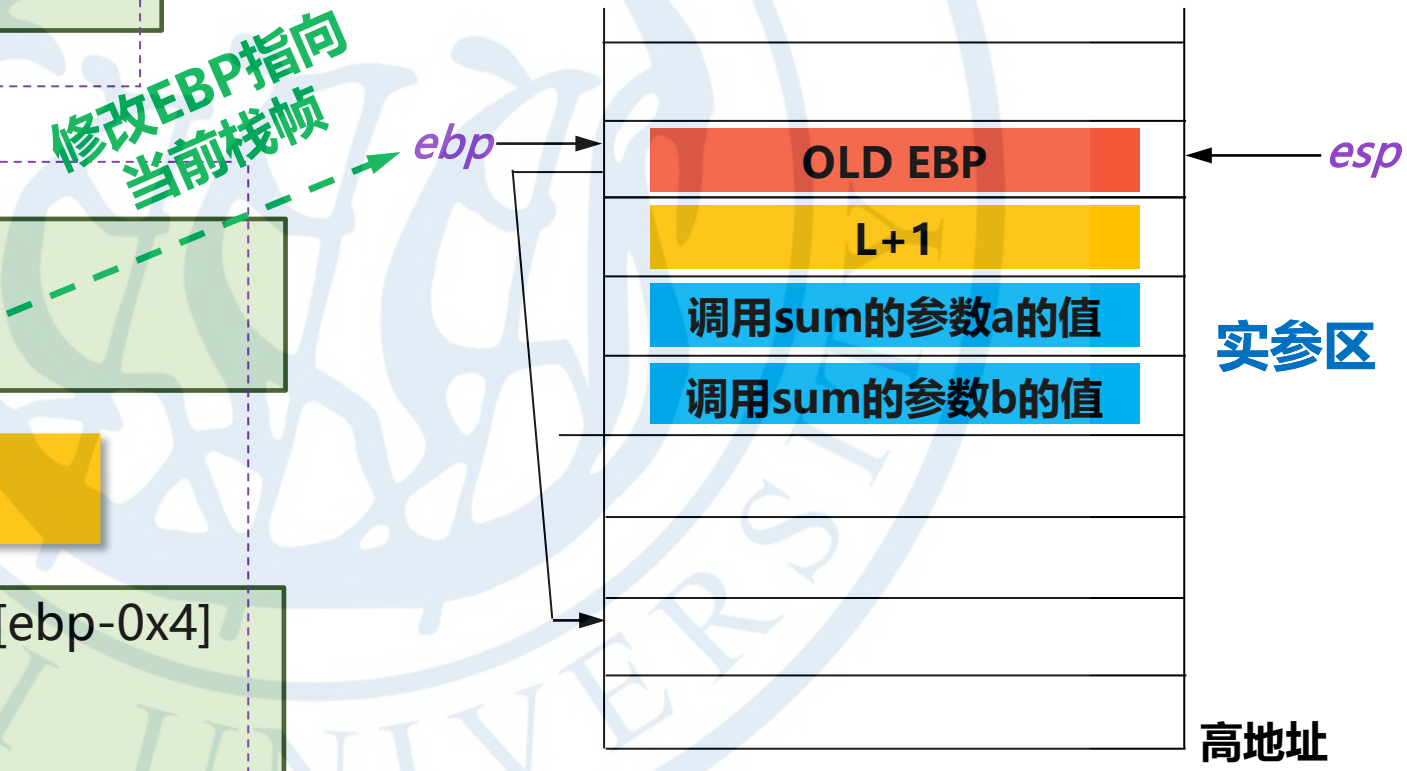
sum

```
T:    push  ebp
T+1:  mov   ebp,  esp
T+2:  sub   esp,  0x10
```

完成加法计算

```
      mov eax, DWORD PTR [ebp-0x4]
      leave
      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 DWORD PTR [ebp-0x8]
L-1:  push DWORD PTR [ebp-0x4]
L:    call   sum
L+1:  add    esp, 0x8
...
```

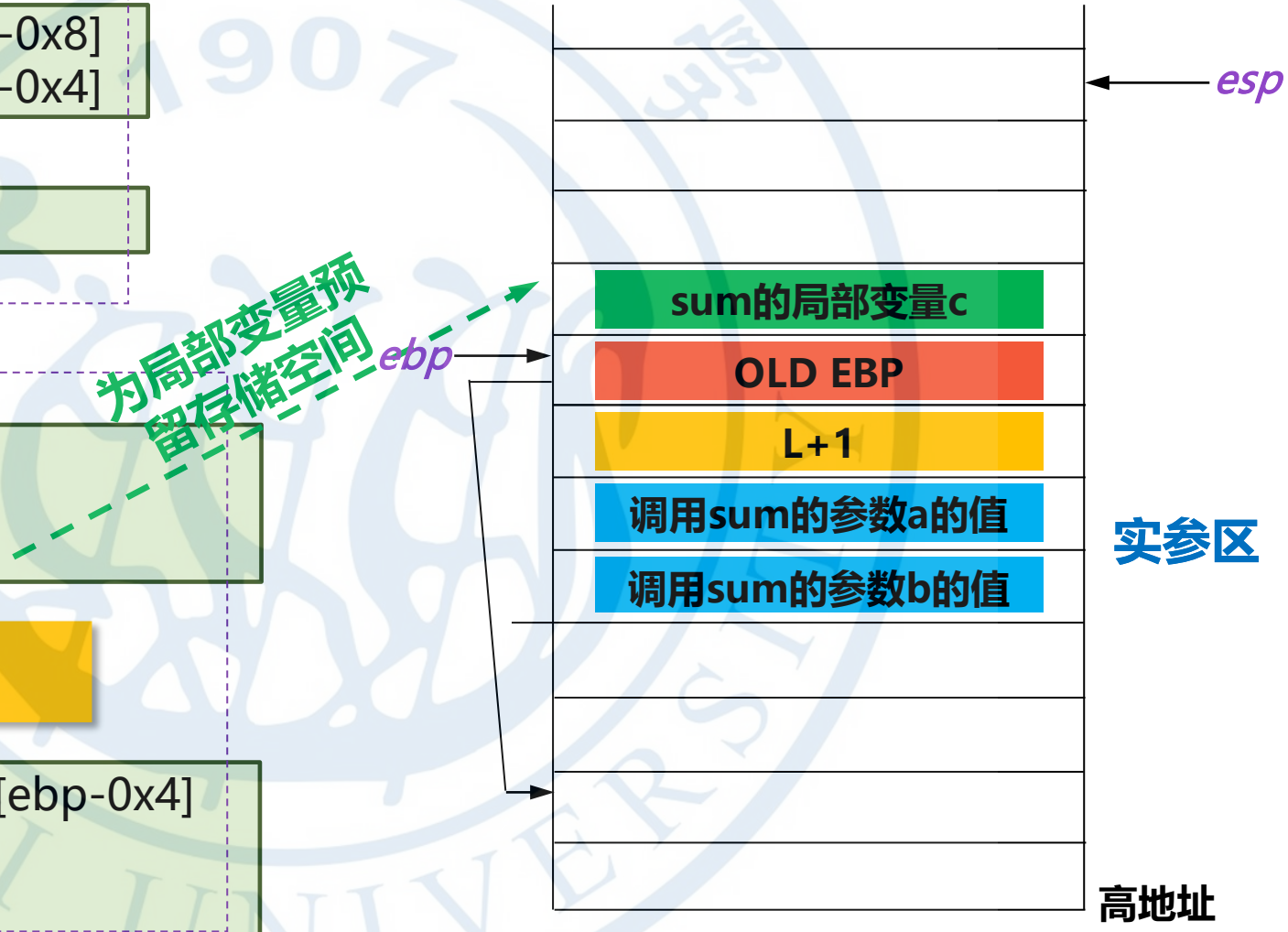
sum

```
T:    push  ebp
T+1:  mov   ebp,  esp
T+2:  sub   esp,  0x10
```

完成加法计算

```
      mov eax, DWORD PTR [ebp-0x4]
      leave
      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

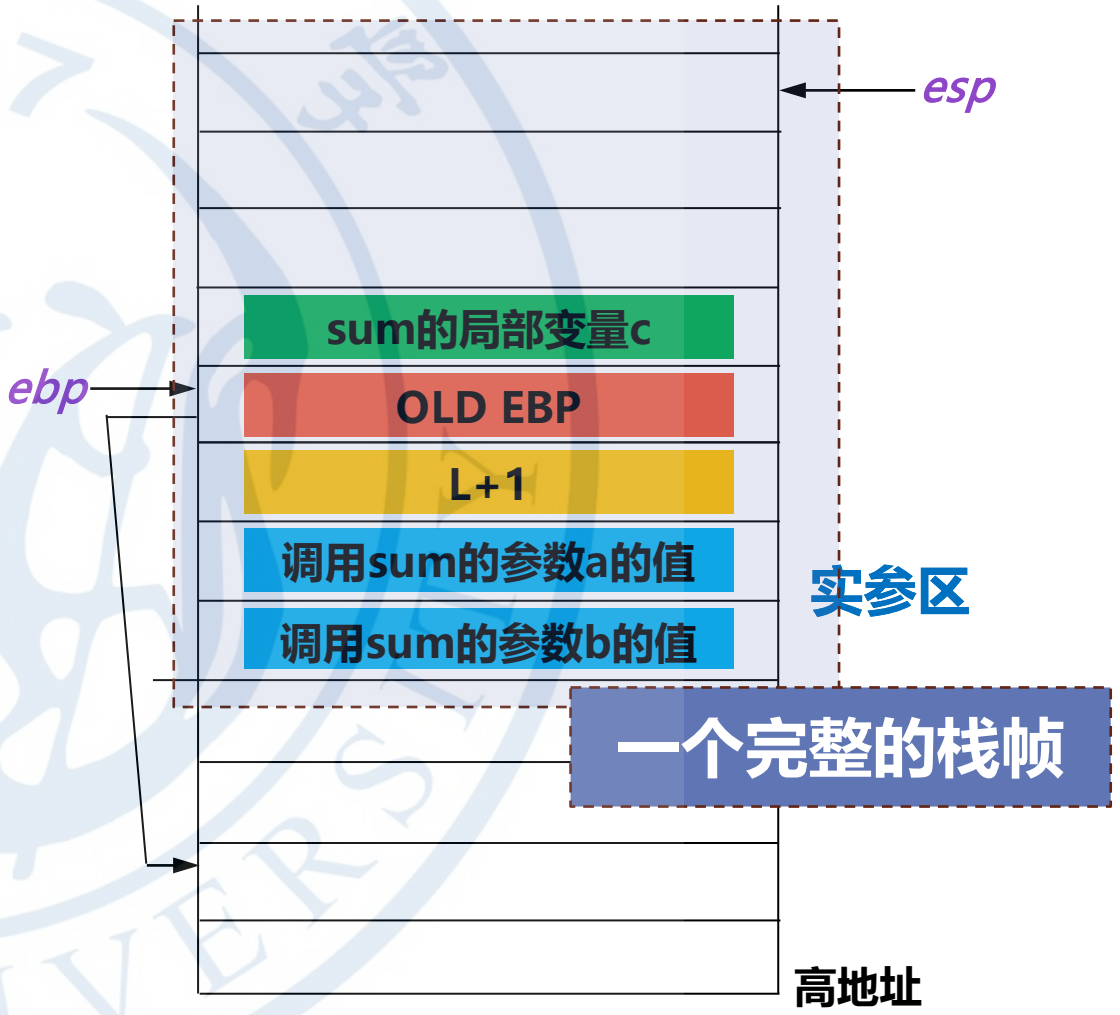
```
...
L-2:  push DWORD PTR [ebp-0x8]
L-1:  push DWORD PTR [ebp-0x4]
L:    call   sum
L+1:  add    esp, 0x8
...
```

sum

```
T:    push  ebp
T+1:  mov   ebp,  esp
T+2:  sub   esp,  0x10
```

完成加法计算

```
      mov eax, DWORD PTR [ebp-0x4]
      leave
      ret
```



C语言编译器对函数调用的处理方式

main

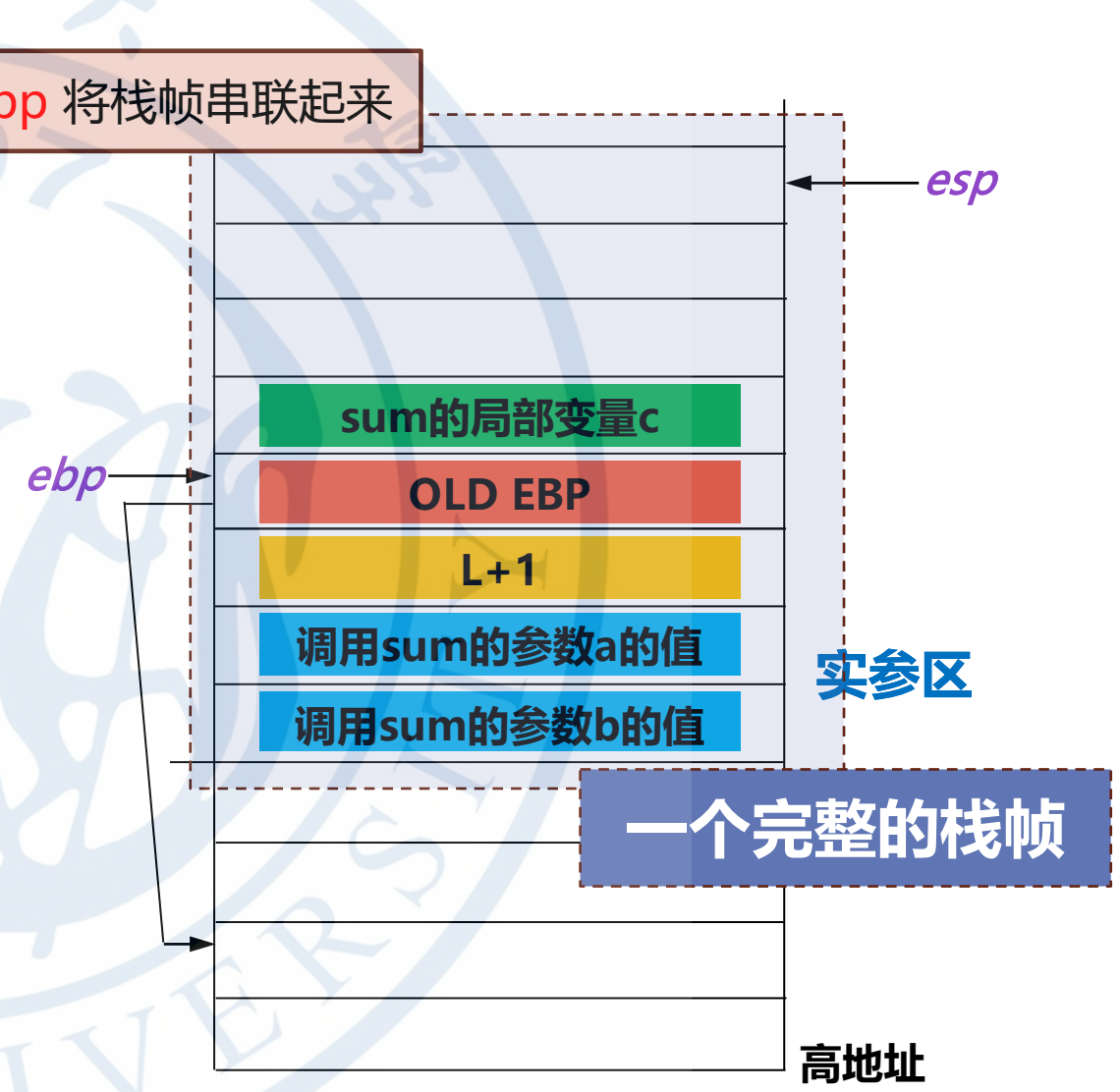
```
...
L-2:  push DWORD PTR [ebp-0x8]
L-1:  push DWORD PTR [ebp-0x4]
L:    call   sum
L+1:  add    esp, 0x8
...
```

sum

```
T:    push  ebp
T+1:  mov   ebp,  esp
T+2:  sub   esp,  0x10
```

完成加法计算

```
      mov eax, DWORD PTR [ebp-0x4]
      leave
      ret
```



C语言编译器对函数调用的处理方式

main

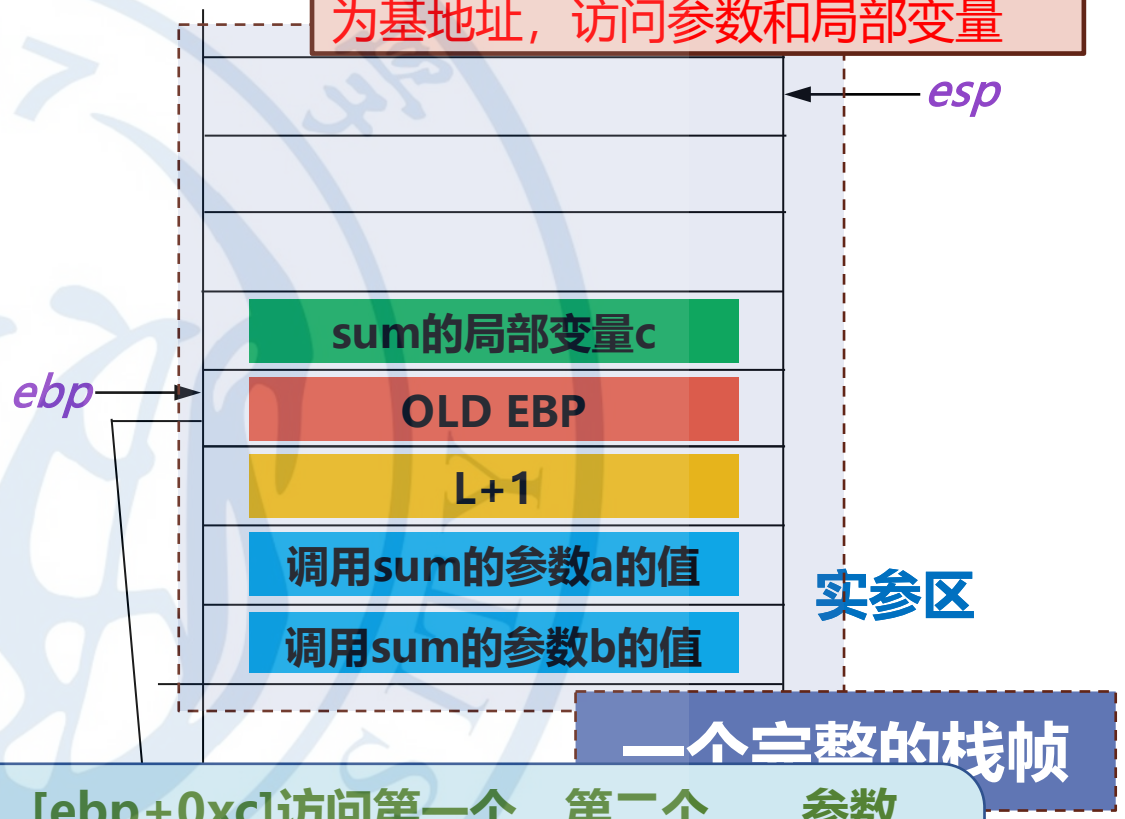
```
...
L-2:  push DWORD PTR [ebp-0x8]
L-1:  push DWORD PTR [ebp-0x4]
L:    call  sum
L+1:  add   esp, 0x8
...
```

sum

```
T:    push  ebp
T+1:  mov   ebp,  esp
T+2:  sub   esp,  0x10
```

完成加法计算

汇编代码在每一个函数内部以ebp为基地址，访问参数和局部变量 2




```
mov edx, DWORD PTR [ebp+0x8] // [ebp+0x8]，  [ebp+0xc]访问第一个，第二个……参数
mov eax, DWORD PTR [ebp+0xc]
add eax, edx
mov DWORD PTR [ebp-0x4], eax // [ebp-0x4]，  [ebp-0x8]访问第一个，第二个……局部变量
```

C语言编译器对函数调用的处理方式

main

```
...
L-2:  push DWORD PTR [ebp-0x8]
L-1:  push DWORD PTR [ebp-0x4]
L:    call   sum
L+1:  add    esp, 0x8
...
```

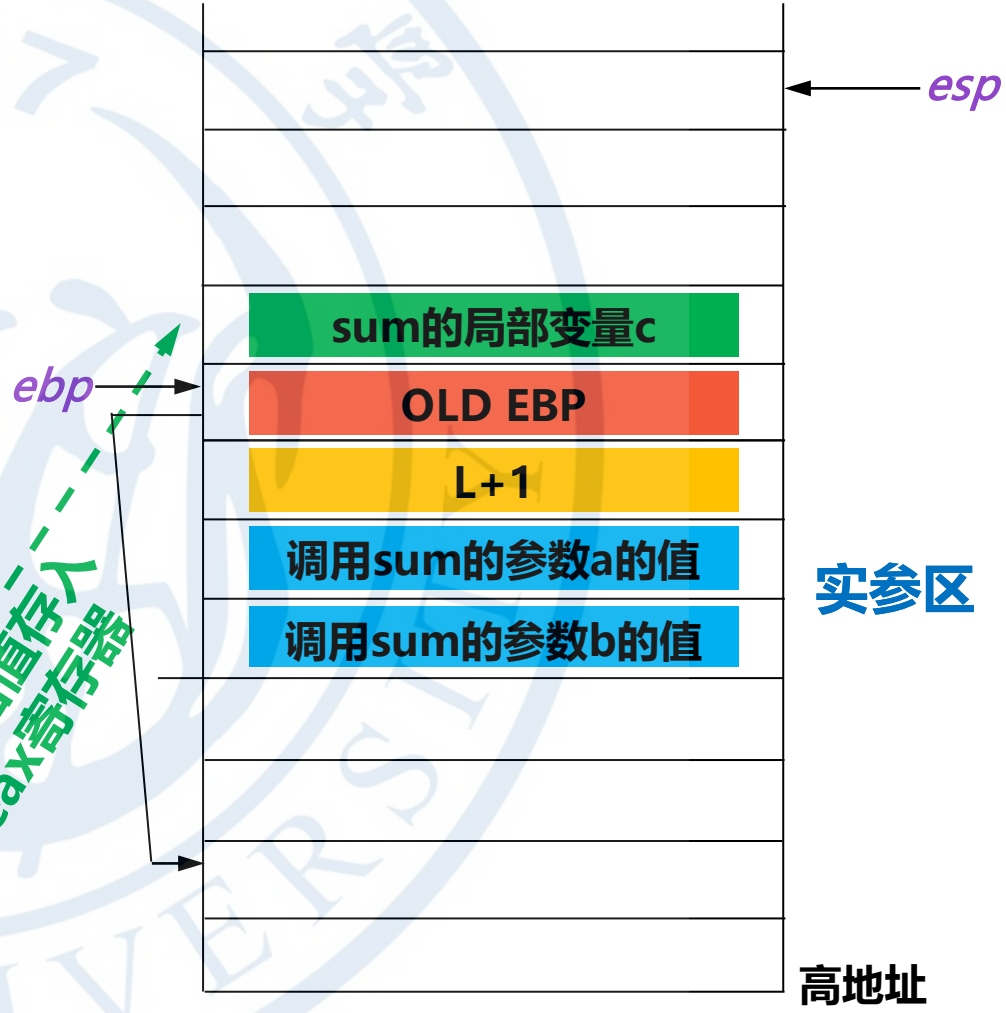
sum

```
T:    push  ebp
T+1:  mov   ebp,  esp
T+2:  sub   esp,  0x10
```

完成加法计算

```
      mov eax, DWORD PTR [ebp-0x4]
      leave
      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 DWORD PTR [ebp-0x8]
L-1:  push DWORD PTR [ebp-0x4]
L:    call   sum
L+1:  add    esp, 0x8
...
```

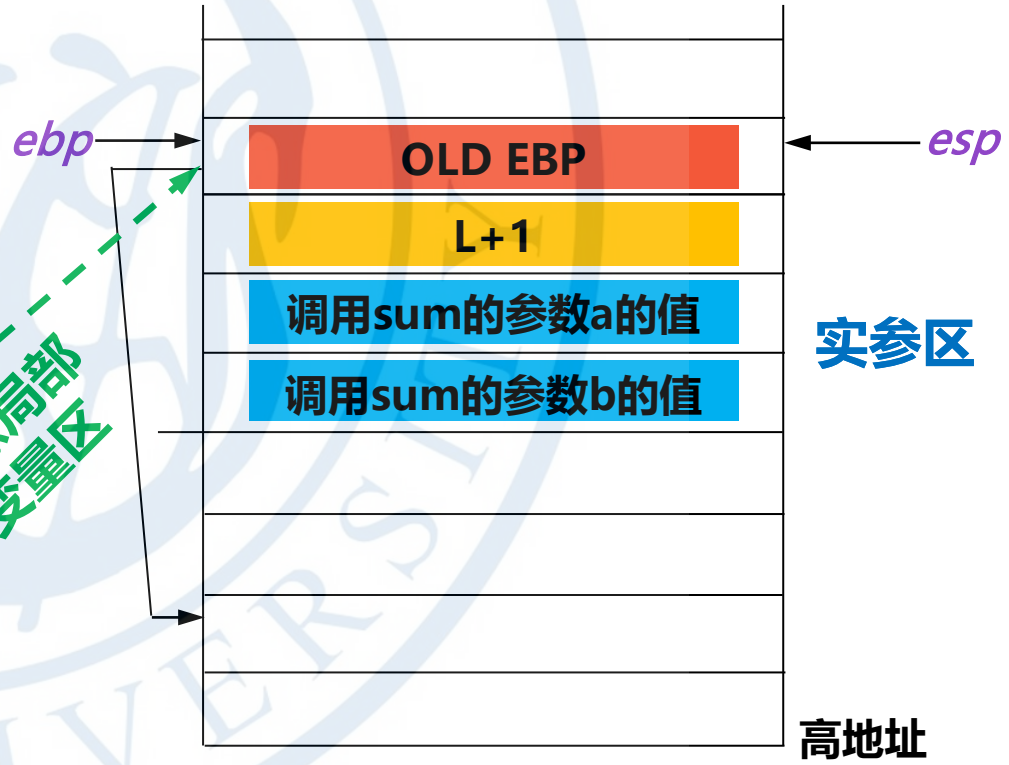
sum

```
T:    push  ebp
T+1:  mov   ebp,  esp
T+2:  sub   esp,  0x10
```

完成加法计算

```
mov eax, DWORD PTR [ebp-0x4]
leave
ret
```

mov esp,ebp;
pop ebp



C语言编译器对函数调用的处理方式

main

```
...
L-2:  push DWORD PTR [ebp-0x8]
L-1:  push DWORD PTR [ebp-0x4]
L:    call   sum
L+1:  add    esp, 0x8
...
```

sum

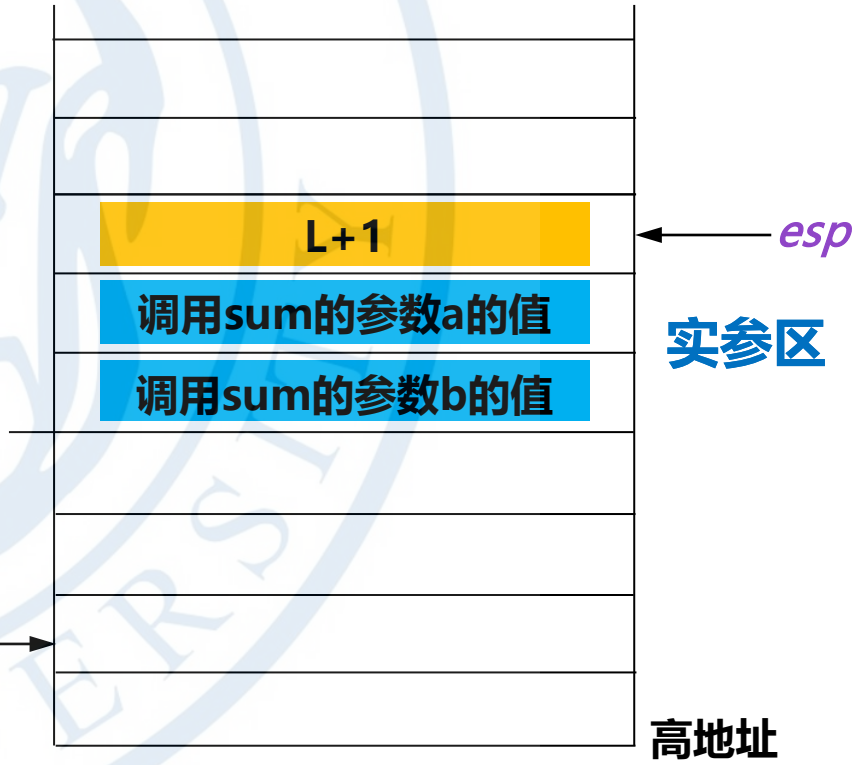
```
T:    push  ebp
T+1:  mov   ebp,  esp
T+2:  sub   esp,  0x10
```

完成加法计算

```
mov eax, DWORD PTR [ebp-0x4]
leave
ret
```

mov esp,ebp;
pop ebp

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 DWORD PTR [ebp-0x8]
L-1:  push DWORD PTR [ebp-0x4]
L:    call   sum
L+1:  add    esp, 0x8
...
```

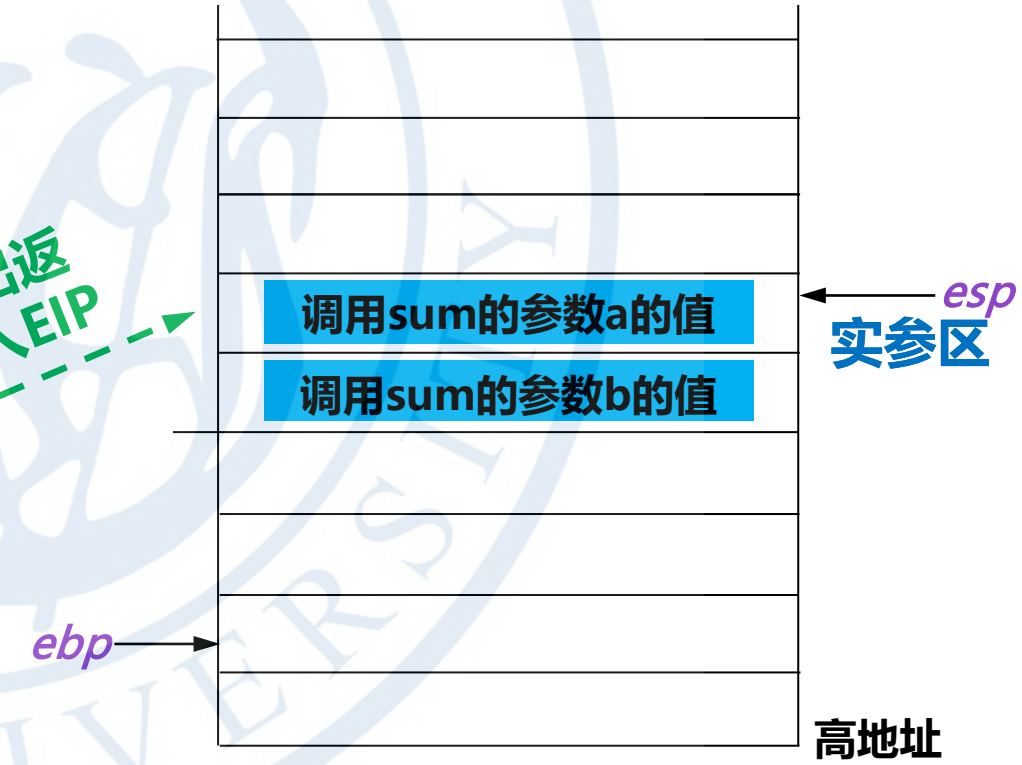
sum

```
T:    push  ebp
T+1:  mov   ebp,  esp
T+2:  sub   esp,  0x10
```

完成加法计算

```
mov eax, DWORD PTR [ebp-0x4]
leave
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 DWORD PTR [ebp-0x8]
L-1:  push DWORD PTR [ebp-0x4]
L:    call   sum
L+1:  add    esp, 0x8
...
```

sum

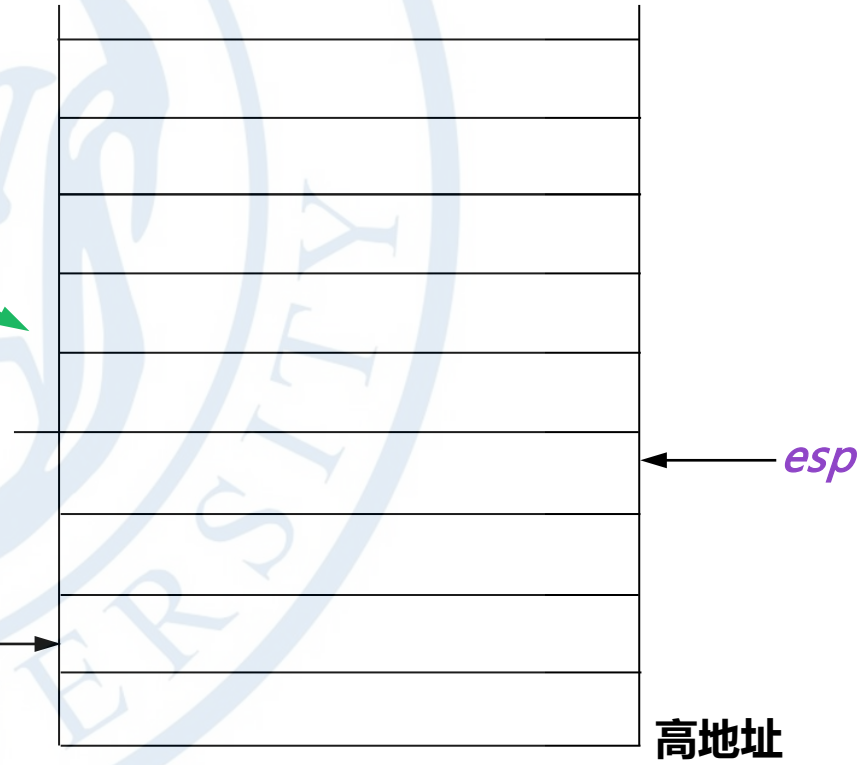
```
T:    push  ebp
T+1:  mov   ebp,  esp
T+2:  sub   esp,  0x10
```

完成加法计算

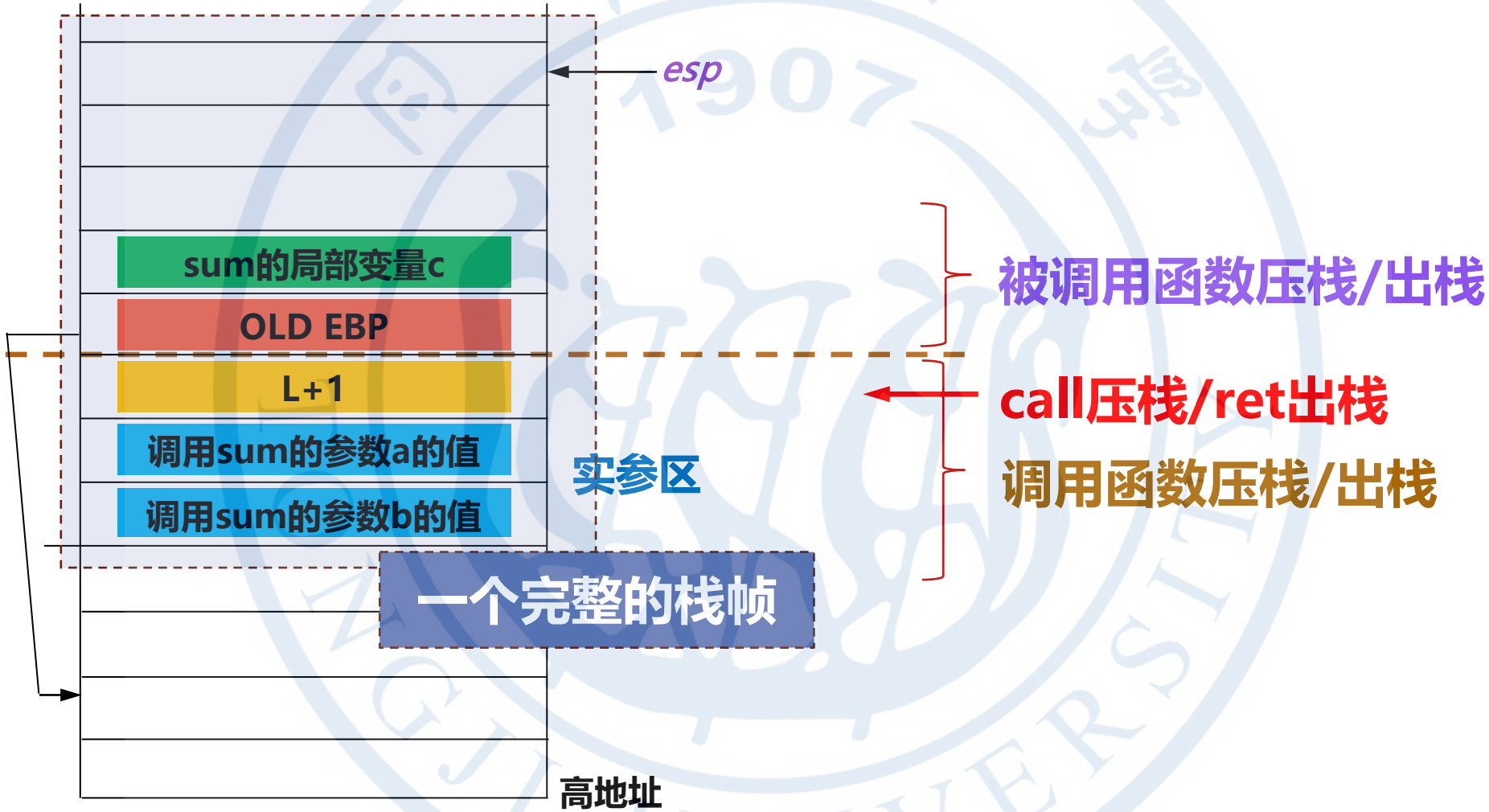
```
      mov eax, DWORD PTR [ebp-0x4]
      leave
      ret
```

返回值在EAX寄存器中

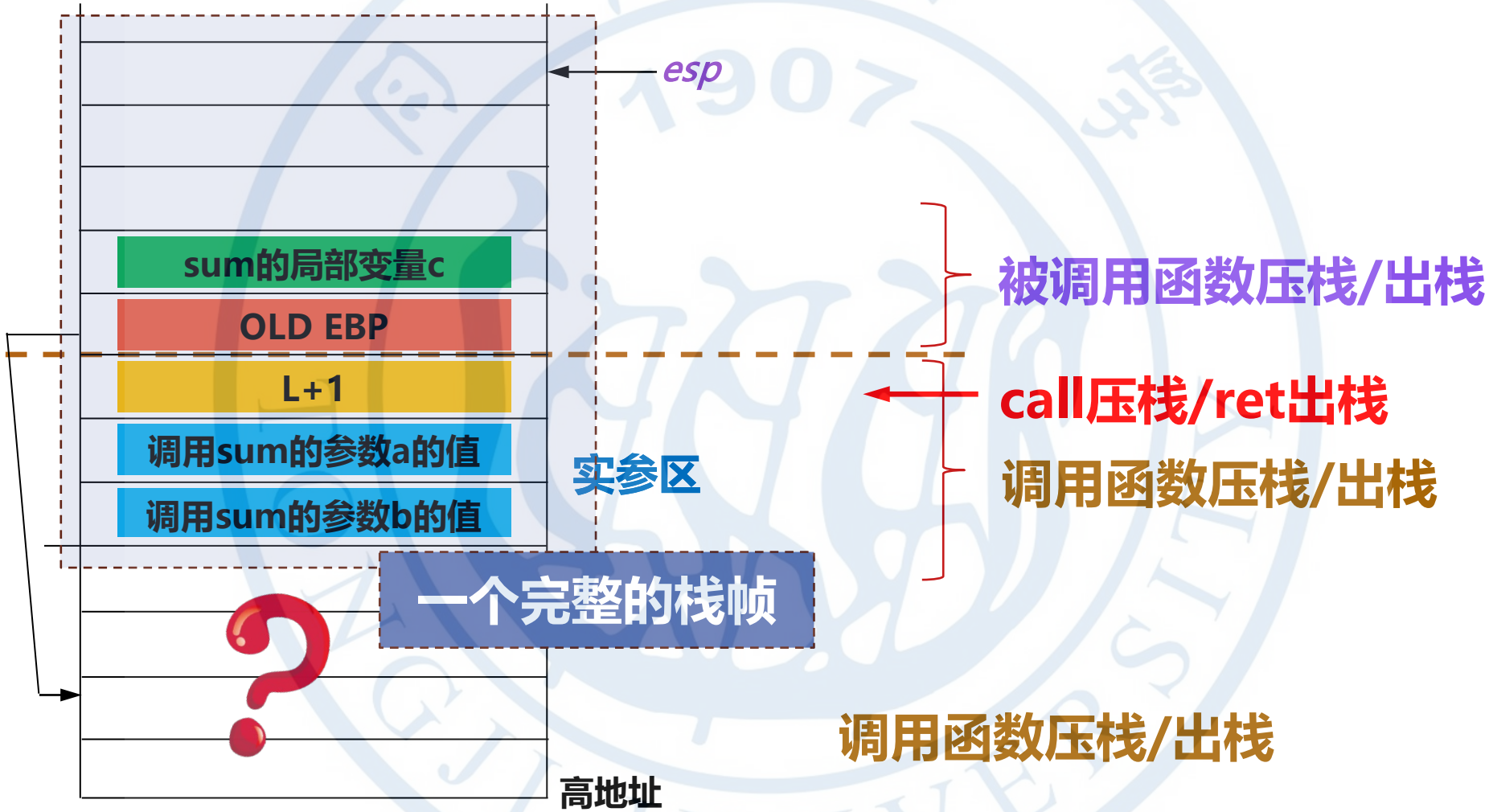
删除实参区



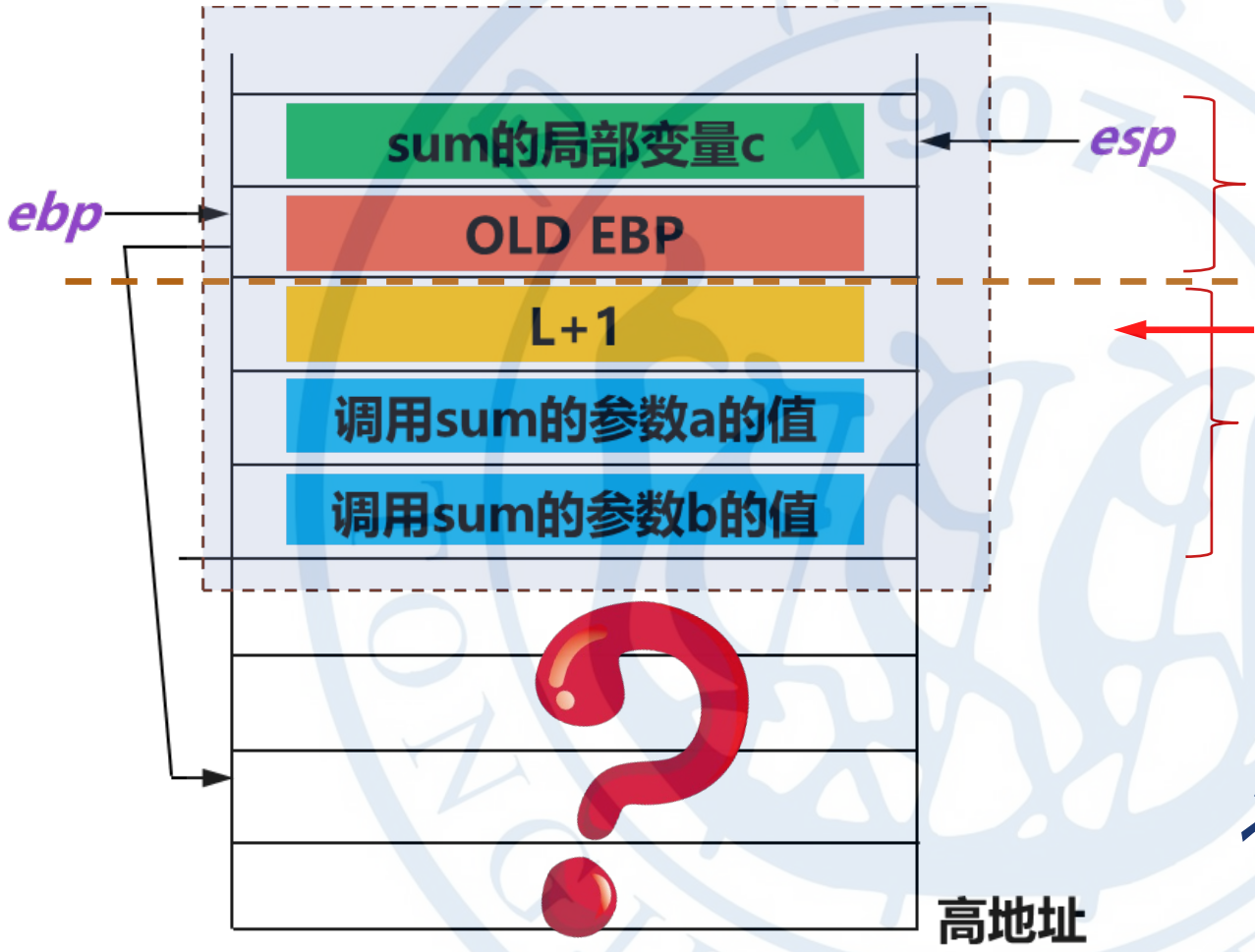
C语言编译器对函数调用的处理方式



C语言编译器对函数调用的处理方式



C语言编译器对函数调用的处理方式



被调用函数压栈/出栈

call压栈/ret出栈

调用函数压栈/出栈

这里是什么？

C语言编译器对函数调用的处理方式

main

```
...
L-2:  push DWORD PTR [ebp-0x8]
L-1:  push DWORD PTR [ebp-0x4]
L:    call   sum
L+1:  add    esp, 0x8
...
```

```
push  ebp
mov   ebp,   esp
sub   esp,   0x10
mov  DWORD PTR [ebp-0x4], 0x1
mov  DWORD PTR [ebp-0x8], 0x2
```

sum

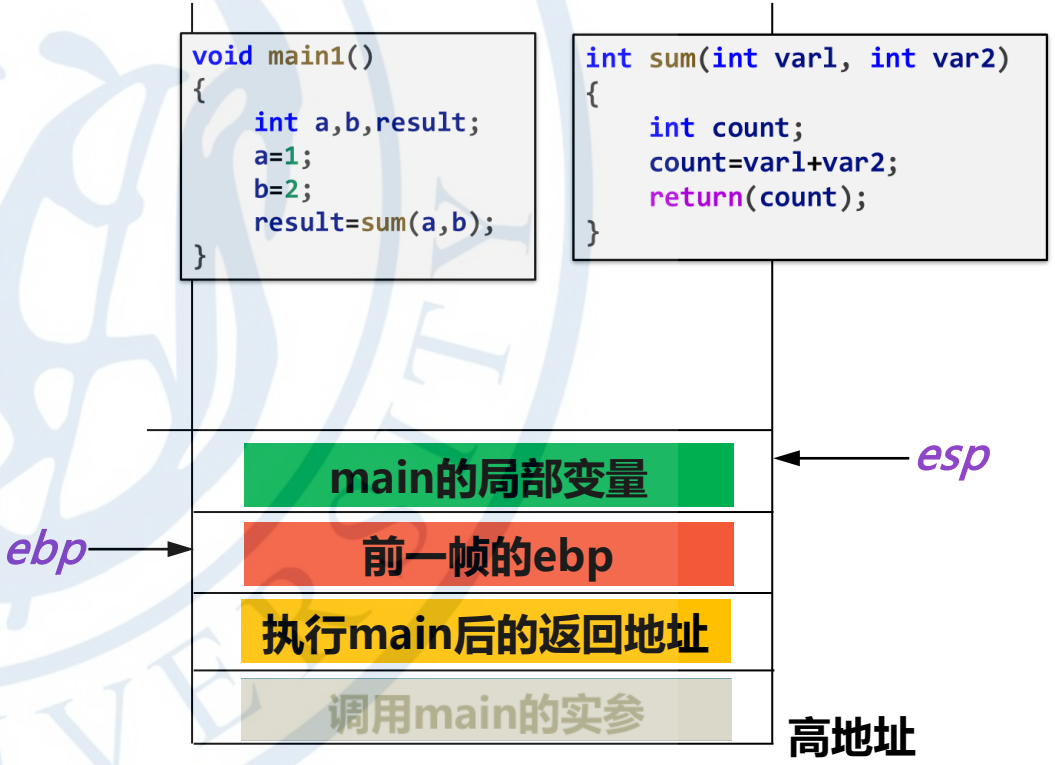
```
T:    push  ebp
T+1:  mov   ebp,   esp
T+2:  sub   esp,   0x10
```

完成加法计算

```
mov eax, DWORD PTR [ebp-0x4]
leave
ret
```

```c
void main1()
{
    int a,b,result;
    a=1;
    b=2;
    result=sum(a,b);
}
```

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



C语言编译器对函数调用的处理方式

main

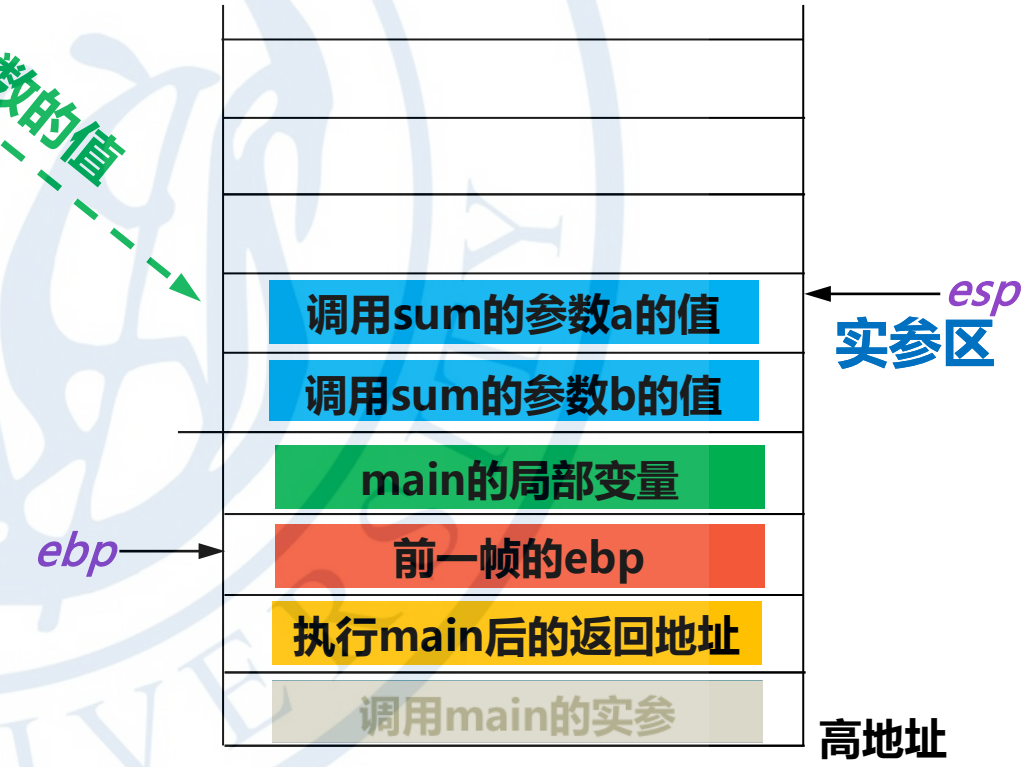
```
        ...
L-2:    push [%ebp - 8]
L-1:    push [%ebp - 4]
L:      call   sum
L+1:    add    %esp, 8
        ...
```

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

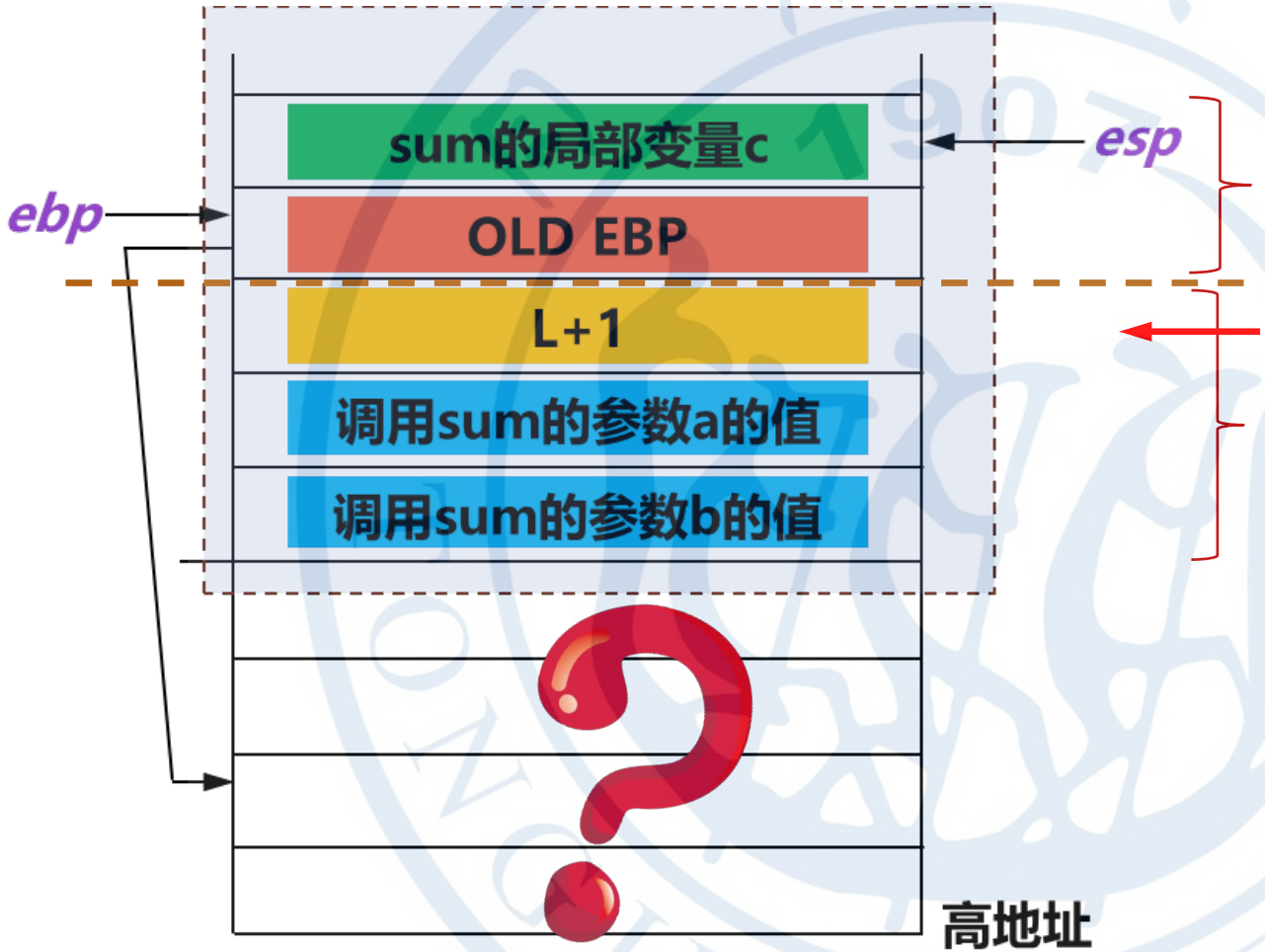
按逆序压入参数的值

sum

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



C语言编译器对函数调用的处理方式



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？