

# 软件安全

## 第1章 软件安全基础

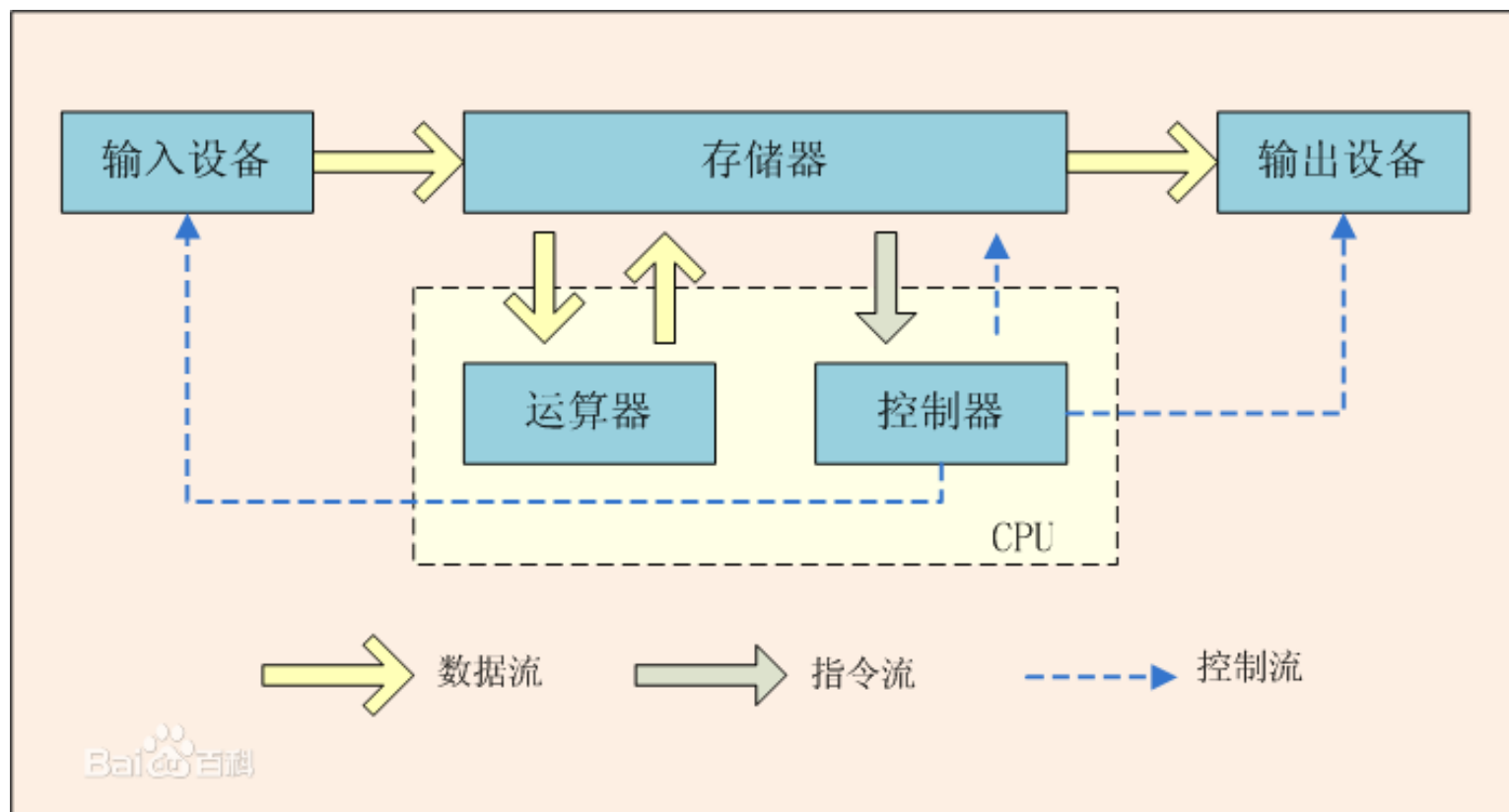
徐国胜

北京邮电大学

# 目录

- 1 计算机引导
- 2 Win32内存体系
- 3 PE文件结构
- 4 进程空间分区
- 5 系统栈工作

# 1.1 冯·诺依曼体系



## 1.2 BIOS启动程序

- (1)打开计算机的电源开关时,处理器进入复位(Reset) 状态
  - 将所有内存清零, 并执行内存同位测试
  - 将段寄存器CS的内容设为FFFFH
  - 其他寄存器都清零, IP=0000H
    - 因此第一个要执行的指令是位于CS:IP中的指令, 它的物理地址为0FFFF0H, 所以将存储器的高地址分配给ROM BIOS, 作为BIOS的入口地址

- (2) 随后BIOS启动一个程序，进行主机自检
  - 确保系统的每一个部分都得到了电源支持，内存存储器、主板上的其它芯片、键盘、鼠标、磁盘控制器及一些I/O端口正常可用

### ■ (3) 自检程序将控制权还给BIOS

■ BIOS读取BIOS设置，得到引导驱动器的顺序，依次检查

■ BIOS将所检查磁盘的第一个扇区（512B）载入内存，放在0x0000:0x7c00处，如果这个扇区的最后两个字节是“55AAH”，那么这就是一个引导扇区，磁盘也就是一块可引导盘，调用该驱动器上磁盘的引导扇区进行引导

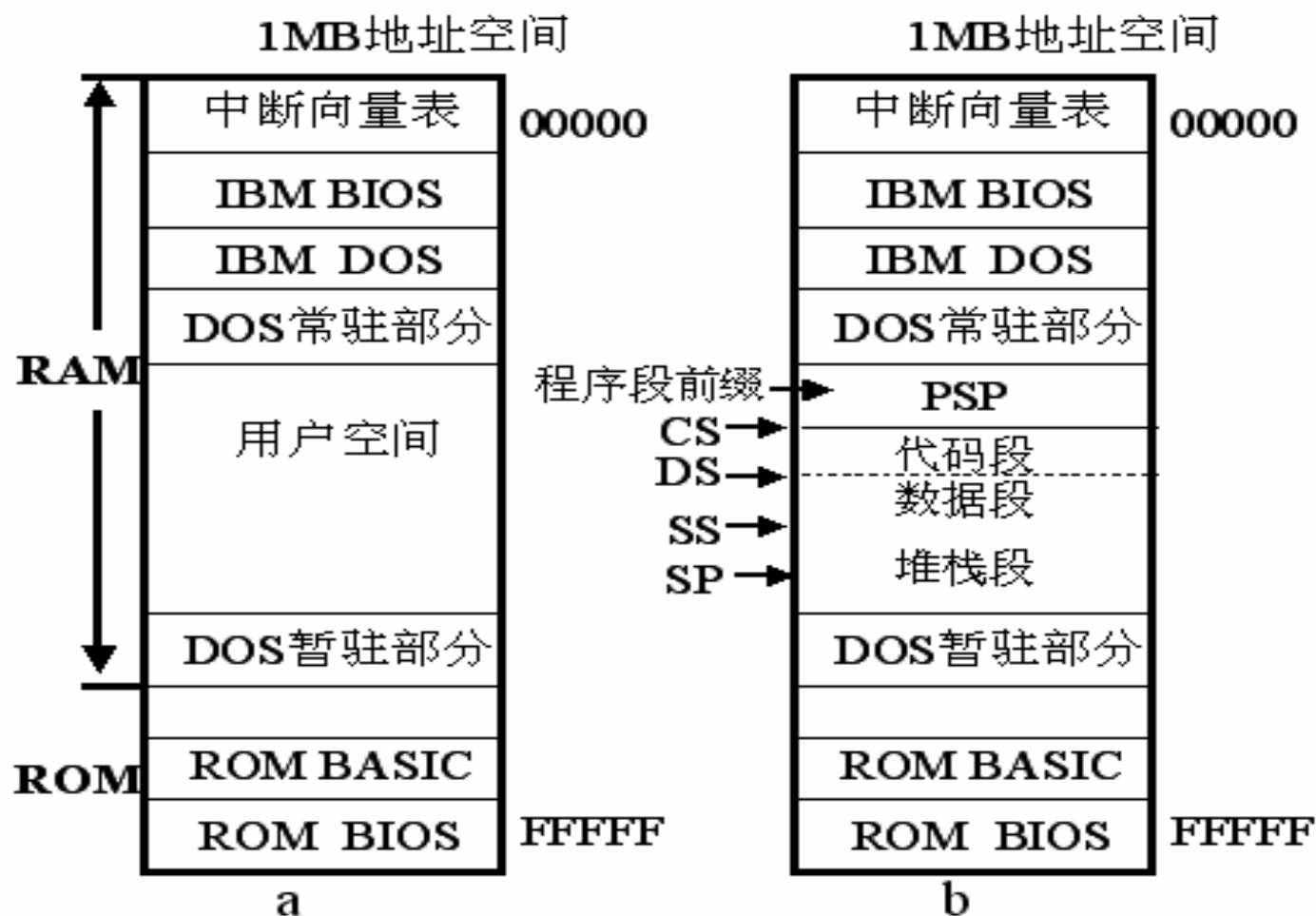
## 1.3 系统加载程序

- 一旦BIOS将控制权交给操作系统之后，就可以向操作系统申请运行程序了。可执行的程序有两种：  
\*.com程序及\*.exe程序。程序在内存的配置情况如图4-1所示。
- 一个\*.com的程序只包含一个段，段内包括指令、数据及堆栈，这种程序小而精，适合用于小的应用程序或常驻程序。汇编语言绝大部分的程序均使用\*.exe的格式。
- 这里\*.com程序被加载运行时有下列的特点：

- ①. 一个\*.com程序占用一个段（64KB）的运行空间，PSP（Program Segment Prefix）、指令、数据及堆栈都安排在这个段内。段寄存器CS、DS、ES及SS均指向这一段的起始地址。
- ②. 段内安排以程序段前缀PSP在前，指令及数据居中，堆栈则安排在段的高地址部分。PSP占用256个字节的空间，因此第一个指令的偏移地址必须从0100H开始，IP寄存器的初值即是0100H。



- ③ . PSP及堆栈都由系统配置，系统会在堆栈顶端存入两个字节，其内容为零值，SP就指到这个零值的数据项，这时SP的内容为OFFFEH。



PC-DOS的内存配置（a为DOS启动后，b为用户程序装入后）

# 1.4 Windows系统启动过程

- 预启动：通过BIOS进行自检
- 启动
- 装载内核
- 内核初始化
- 用户登录

## ①预启动

按电源开机

CPU初始化

POST加电自检

检测显卡

检测CPU

检测所有内存

检测标准设备

检测即插即用设备

显示系统配置表

系统BIOS

更新ESCD

系统BIOS最后按

指定驱动器启动

BIOS

软驱

光驱

硬盘  
引导区  
NTLDR

①系统BIOS读MBR（主引导记录）检查HD分区表

②系统BIOS调NTLDR（操作系统加载器）进内存

## ②启动

CPU初始化  
（实模式转为  
32位保护模式）读取  
BOOT.INI

内存

选择希望启动  
的操作系统内存中的  
NTLDRWindows  
2000 / XP非Windows  
2000 / XPNTLDR会  
继续引导  
进行以下过程NTLDR则会读取  
系统引导扇区副本  
BOOTSECT.DOS转入启动  
相应系统

## ③装载内核

引导过程装载

Windows2000/XP内核  
(NTOSKRNL.EXE)

硬件抽象层(HAL)

## ④初始化内核

Windows2000/XP  
内核完成初始化Windows2000/XP  
内核开始装载并初始化  
设备驱动程序

启动WIN32子系统

启动  
WINDOWS2000/XP  
服务

## ⑤用户登录

开始登录进程

WIN32子系统  
启动

WINLOGON.EXE

LOCAL  
SECURITY  
AUTHORITY  
(LSASS.EXE)  
显示登录对话框继续配置  
网络设备和用户环境  
个性化桌面

# 内容安排

- 1 计算机引导
- 2 Win32内存体系
- 3 PE文件结构
- 4 进程空间分区
- 5 系统栈工作

## 2.1 存储体系



- 高速缓存：
  - Data Cache
  - TLB(Translation Lookaside Buffer)
- 内存：DRAM, SDRAM等；
- 外存：软盘、硬盘、光盘、磁带等；

# 寄存器

Intel x86的寄存器可以分为下述的几类：

- 通用寄存器
  - 32位的通用寄存器有*EAX*、*EBX*、*ECX*、*EDX*、*ESP*、*EBP*、*ESI*和*EDI*，它们的使用方法不总是相同的。一些指令赋予它们特殊的功能。
- 段寄存器
  - 段寄存器被用于指向进程地址空间不同的段。
  - *CS*指向一个代码段的开始；
  - *SS*是一个堆栈段；
  - *DS*、*ES*、*FS*、*GS*和各种其他数据段，例如存储静态数据的段。
- 程序流控制寄存器
- 其他寄存器

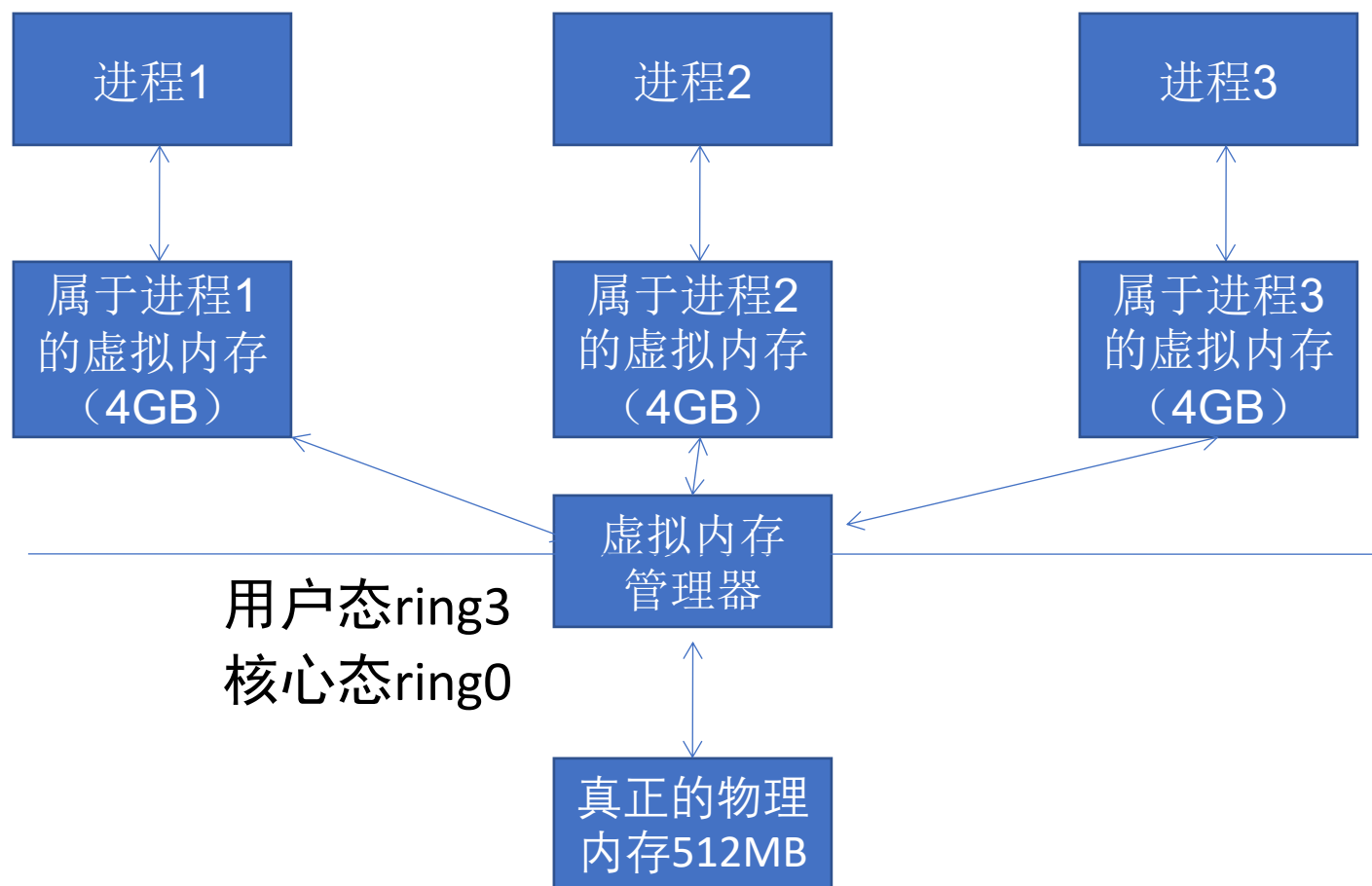
# 寄存器

- 在通用寄存器里面有很多寄存器虽然他们的功能和使用没有任何的区别，但是在长期的编程和使用中，在程序员习惯中已经默认的给每个寄存器赋上了特殊的含义，比如：
  - EAX一般用来做返回值
  - ECX用于记数
  - EIP：扩展指令指针。在调用一个函数时，这个指针被存储在堆栈中，用于后面的使用。在函数返回时，这个被存储的地址被用于决定下一个将被执行的指令的地址。
  - ESP：扩展堆栈指针。这个寄存器指向堆栈的当前位置，并允许通过使用push和pop操作或者直接的指针操作来对堆栈中的内容进行添加和移除。
  - EBP：扩展基指针。主要用与存放在进入call以后的ESP的值，便于退出的时候回复ESP的值，达到堆栈平衡的目的。



## 2.2 虚拟内存

- Windows的内存可以被分为两个层面：物理内存和虚拟内存。其中，物理内存比较复杂，需要进入Windows内核级别ring0才能看到。通常，在用户模式下，我们用调试器看到的地址都是虚拟内存。
- Windows让所有的进程都“相信”自己拥有独立的4GB内存空间。但是我们计算机中那跟实际的内存条可能只有512MB,怎么能为所有进程都分配4GB的内存呢？这一切都是通过虚拟内存管理器的映射做到的。



- 虽然每个进程都“相信”自己拥有4GB的空间，但实际上它们运行时真正能用到的空间根本没有那么多。
- 内存管理器只是分给进程一片“假地址”，或者说是“虚拟地址”，它们对进程来说只是一笔“无形的数字财富”；
- 当需要实际的内存操作时，内存管理器才会把“虚拟地址”和“物理地址”联系起来。

## 2.3 内存管理与银行的类比

我们将银行与内存管理机制进行一下类比来帮助大家理解

内存管理	银行类比
进程	储户
内存管理器	银行
物理内存	钞票
虚拟内存	存款
进程可能拥有大片内存，但使用往往很少	储户拥有大笔存款，但实际生活中的开销并没多少

## 我们将银行与内存管理机制进行一下类比来帮助大家理解

内存管理	银行类比
进程不使用虚拟内存时，这些内存只是些地址，是虚拟存在的，是一笔无形的数字财富。	用户不使用储蓄时，储蓄也只是是一些数字，是无形的数字财富。
进程使用内存时，内存管理器会为此虚拟地址映射实际的物理地址，虚拟内存地址和最终被映射到的物理内存地址之间没有什么必然联系	储户需要钱时，银行才会兑换一定的现金给储户，但物理钞票的号码与储户心目中的数字存款之间可能并没有任何联系。

## 我们将银行与内存管理机制进行一下类比来帮助大家理解

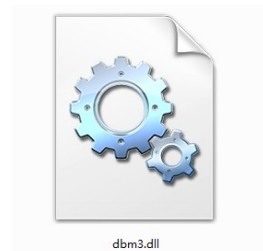
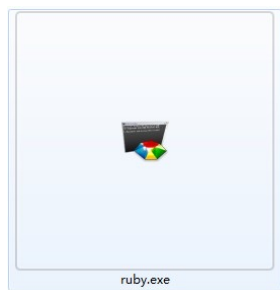
内存管理	银行类比
操作系统的实际物理内存空间可以远远小于进程的虚拟内存空间之和，仍能正常调度	银行的现金准备可以远远小于所有储户的储蓄额总和，仍能正常运转
很少出现所有程序要申请出全部物理内存	很少会出现所有储户同时要取出全部存款的现象
物理内存可以远小于虚拟内存之和	社会上实际流通的钞票可以远远小于社会的财富总额

# 内容安排

- 1 计算机引导
- 2 Win32内存体系
- 3 PE文件结构
- 4 进程空间分区
- 5 系统栈工作

## 3.1 PE文件格式简介

- PE(Portable Executable)是Win32平台下可执行文件遵守的数据格式。常见的可执行文件(如“\*.exe”文件和“\*.dll”文件)都是典型的PE文件。



- 一个可执行文件不光包含了二进制的机器代码，还会自带其他信息，如字符串、菜单、图标、位图、字体等。



- PE文件格式规定了所有的这些信息在可执行文件中如何组织。在程序被执行时，操作系统会按照PE文件格式的约定去相应的地方准确地定位各种类型的资源，并分别装入内存的不同区域。
- PE文件格式把可执行文件分成若干个数据节(section)，不同的资源被存放在不同的节中。一个典型的PE文件包含的节如下：

- .text 由编译器产生，存放着二进制的机器代码，也是我们反汇编和调试的对象。
- .data 初始化的数据块，如宏定义、全局变量、静态变量等。
- .idata 可执行文件所使用的动态链接库等外来函数与文件的信息。
- .rsrc 存放程序的资源，如图标、菜单等。
- 除此之外，还可能出现的节包括“.reloc”、“.edata”、“.tls”、“.rdata”等。



## PE文件简单构成

## 3.2 PE文件与虚拟内存之间的映射

- 静态反汇编工具看到的PE文件中某条指令的位置是相对于磁盘文件而言的，即所谓的文件偏移，我们可能还需要知道这条指令在内存中所处的位置，即虚拟内存的位置
- 反之，在调试时看到的某条指令的地址是虚拟内存地址，我们也经常需要回到PE文件中找到这条指令对应的机器码

我们首先要弄清楚几个概念

- 文件偏移地址(File Offset)

- 数据在PE文件中的地址叫做文件偏移地址。这是文件在磁盘上存放时相对于文件开头的偏移。

- 装载基址(Image Base)

- PE装入内存时的基地址。默认情况下, EXE文件在内存中的基地址是0x00400000, DLL文件是0x10000000。这些位置可以通过修改编译选项更改。

我们首先要弄清楚几个概念

- 虚拟内存地址(Virtual Address,VA)

- PE文件中的指令被装入内存后的地址。

- 相对虚拟地址(Relative Virtual Address,RVA)

- 相对虚拟地址是内存地址相对于映射基址的偏移量。
- 虚拟内存地址、映射基址、相对虚拟内存地址三者有如下关系

$$VA = \text{Image Base} + \text{RVA}$$

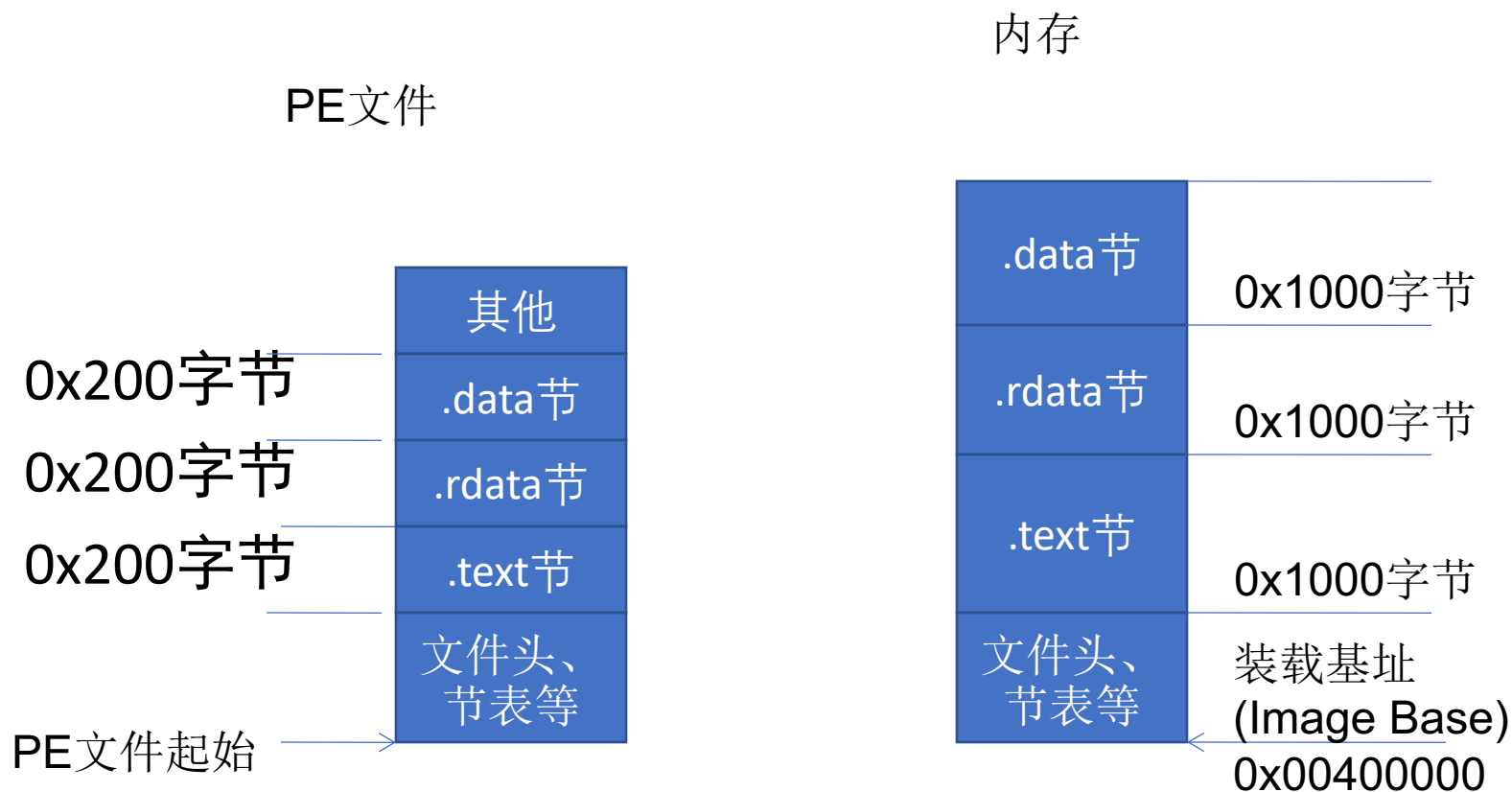
文件偏移地址在与他们计算时还需要考虑存放方式的不同。

- PE文件中的数据按照磁盘数据标准存放，以0x200字节为基本单位进行组织。当一个数据节不足0x200字节时，不足的地方将被0x00填充；当一个数据节超过0x200字节时，下一个0x200块将分配给这个节使用。因此PE数据节的大小永远是0x200的整数倍。

文件偏移地址在与他们计算时还需要考虑存放方式的不同。

- 当代码装入内存后，将按照内存数据标准存放，并以0X1000字节为基本单位进行组织。类似的，不足将被补全，若超出将分配下一个0x1000为其所用。因此，内存中的节总是0x1000的整数倍。





PE文件与虚拟内存的映射关系

节(section)	相对虚拟偏移量(RVA)	文件偏移量
.text	0x00001000	0x0400
.rdata	0x00007000	0x6200
.data	0x00009000	0x7400
.rsrc	0x0002D000	0x7800

我们把这种由存储单位差异引起的节基址差称做**节偏移**，在上图例中：

.text节偏移 =  $0x1000 - 0x400 = 0xc00$

.rdata节偏移 =  $0x7000 - 0x6200 = 0xE00$

.data节偏移 =  $0x9000 - 0x7400 = 0x1c00$

.rsrc节偏移 =  $0x2D000 - 0x7800 = 0x25800$

文件偏移地址 = 虚拟内存地址(VA) – 装载基址(Image Base)-节偏移  
= RVA – 节偏移

以上表为例，如果在调试时遇到虚拟内存中0x00404141处的一条指令，那么要换算出这条指令在文件中的偏移量，有：

文件偏移量 =  $0x00404141 - 0x00400000 - (0x1000 - 0x400) = 0x3541$

# 内容安排

- 1 计算机引导
- 2 Win32内存体系
- 3 PE文件结构
- 4 进程空间分区
- 5 系统栈工作

## 4.1 进程空间的功能分区

根据不同的操作系统，一个进程可能被分配到不同的内存区域去执行。但是不管什么样的操作系统、什么样的计算机架构，进程使用的内存都可以按照功能大致分成以下4个部分。

# 内存的不同用途

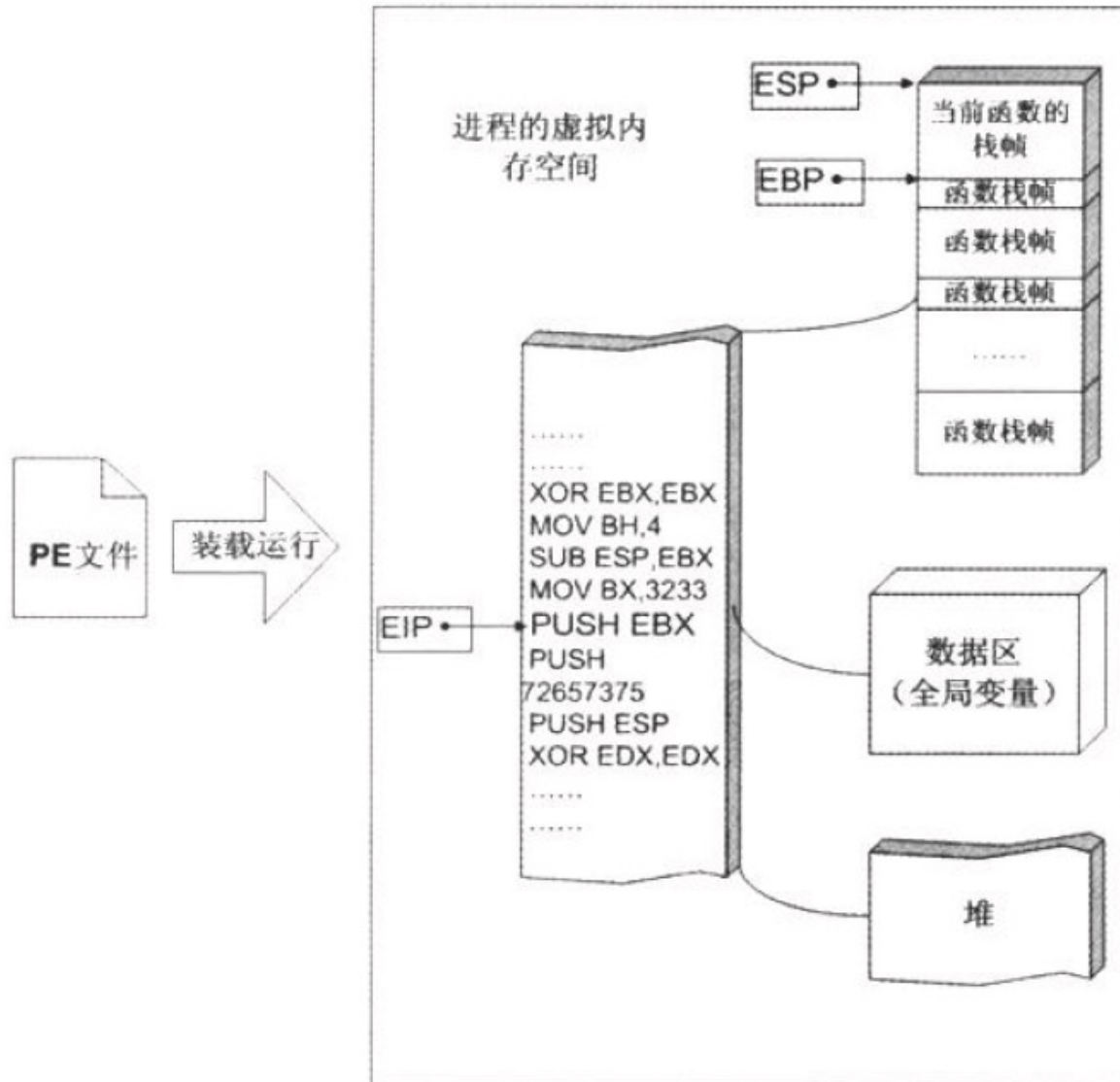
名称	用途
代码区	这个区域存储着被装入执行的二进制机器代码，处理器会到这个区域取指并执行。
数据区	用于存储全局变量等。
堆区	进程可以在堆区动态地请求一定大小的内存，并在用完之后归还给堆区。动态分配和回收是堆区的特点。
栈区	用于动态地存储函数之间的调用关系，以保证被调用函数在返回时恢复到父函数中继续执行。

内存的4个部分和相关用途

如果把计算机看成一个有条不紊的工厂，我们可以得到如下类比。

名称	类比
CPU	完成工作的工人
数据区、堆区、栈区等	存放原料、半成品、成品等各东西的场所。
存在代码区的指令	告诉CPU要做什么，怎么做，到哪里去领原料，用什么工具来做，做完以后把成品放到哪个货舱去。
栈区	栈除了扮演存放原料、半成品的仓库之外，它还是车间调度主任的办公室。

内存与工厂的类比关系



## 4.2 栈与系统栈

程序中所使用的缓冲区可以是堆区、栈区和存放静态变量的数据区。缓冲区溢出的利用方法和缓冲区到底属于上面哪个内存区域密不可分，本课主要介绍在系统栈发生溢出的情形。



从计算机科学的角度来看，栈指的是一种数据结构，是一种先进后出的数据表。

栈的最常见操作有两种：压栈(PUSH)、弹栈(POP)；用于标识栈的属性也有两个：栈顶(TOP)、栈底(BASE)。

为了便于理解，我们把栈和扑克牌进行一下类比

操作名称	类比效果
PUSH	为栈增加一个元素的操作叫做PUSH，这相当于在这摞扑克牌的最上面再放上一张牌。
POP	从栈中取出一个元素的操作叫做POP,相当于从这摞扑克牌取出最上面的一张。

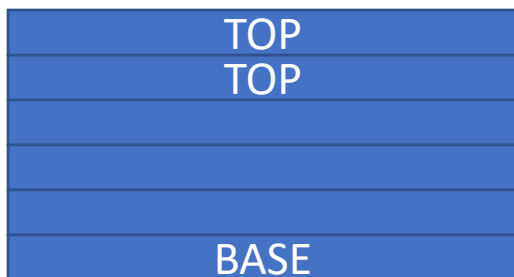
栈操作与扑克的类比

假如我们把栈想象成一摞扑克牌：

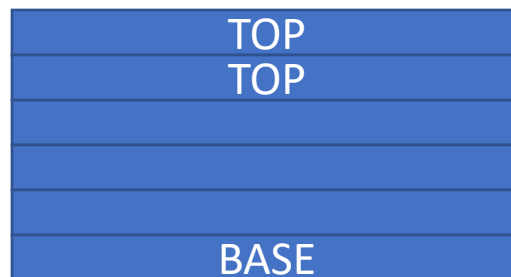
位置名称	类比效果
TOP	标识栈顶的位置，并且是动态变化的。每做一次PUSH操作，它都会自增1；相反，每做一次POP操作，它会自减1。栈顶元素相当于扑克牌最上面一张，只有这张牌的花色是当前可以看到的。
BASE	标识栈底的位置，它记录着扑克牌最下面一张的位置。BASE用于防止栈空后继续弹栈。一般情况下它的值是不会改变的。

栈位置属性与扑克牌的类比

假如我们把栈想象成一摞扑克牌：



PUSH操作



POP操作

栈位置属性与扑克牌的类比

内存的栈区实际上指的就是系统栈。系统栈由系统自动维护，它用于实现高级语言中函数的调用。对于类似C语言这样的高级语言，系统栈的PUSH/POP等堆栈平衡细节是透明的。

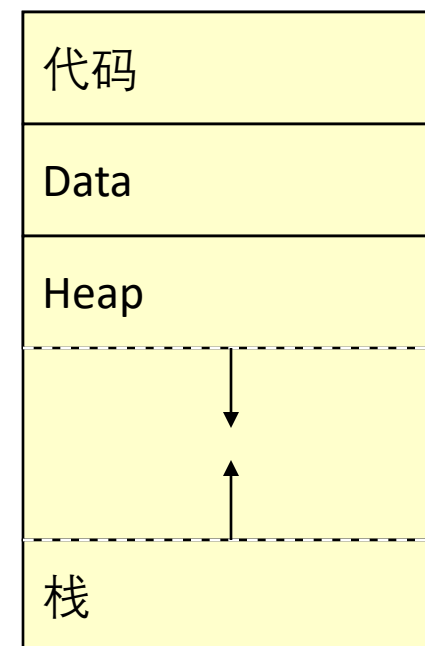
一般说来，只有在使用汇编语言开发程序的时候，才需要和它直接打交道。

# 内容安排

- 1 计算机引导
- 2 Win32内存体系
- 3 PE文件结构
- 4 进程空间分区
- 5 系统栈工作

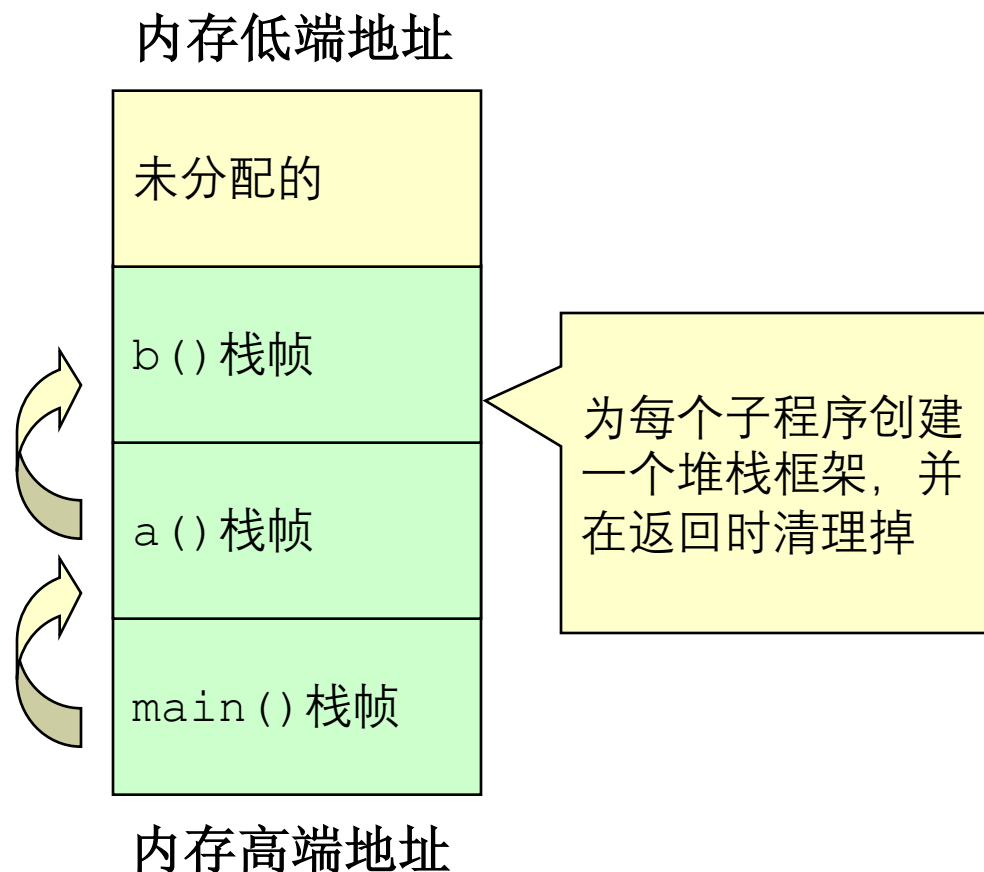
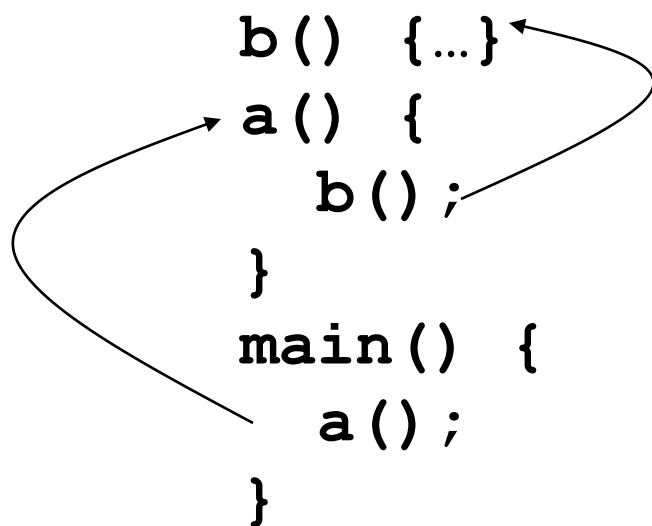
# 程序栈

- 栈通过存储下列内容来追踪程序的执行和状态。
  - 调用函数的返回地址
  - 函数参数
  - 局部 (临时) 变量
- 在下列情况下栈需要被修改
  - 在函数调用期间
  - 函数初始化期间
  - 从子例程返回时



# 程序栈

- 堆栈支持嵌套调用
- 帧指由函数调用引发的压入栈的数据。





# 程序栈

- 栈用于存储
  - 子例程的实际参数
  - 调用函数的返回地址
  - 局部（自动）变量
- 当前帧的地址被存储到帧或者基址寄存器中(英特尔架构中的EBP)
- 帧指针在栈中是一个定点的引用。
- 下列情况出现时，栈要被修改
  - 子例程调用
  - 子例程初始化
  - 从子例程返回

# 子例程调用

```
push 2  
push 4  
call function (411A29h)
```

把第2个参数压入栈

把第1个参数压入栈

把返回地址压入栈并跳到那个地址

• **function(4, 2);**

**EIP = 00411A80 ESP = 0012FE0C EBP = 0012FEDC**

**EIP:** 扩展指令指针    **ESP:** 扩展栈指针    **EBP:** 扩展基指针

# 子例程初始化

• `void function(int arg1, int arg2) {`

`push ebp`

存储帧指针

`mov ebp, esp`

子例程的帧指针被设置为当前栈指针

`sub esp, 44h`

为局部变量分配

**EIP = 00411A29 ESP = 0012FD40 EBP = 0012FE00**

EIP:扩展指令指针

ESP:扩展栈指针

EBP:扩展基指针

# 子例程初始化

- **function(4, 2);**

```
push 2
```

```
push 4
```

```
call function (411230h)
```

```
add esp, 8
```

存储栈指针

**EIP = 00411A8A ESP = 0012FE10 EBP = 0012FEDC**

**EIP:**扩展指令指针

**ESP:**扩展栈指针

**EBP:**扩展基指针

# Subroutine Return

- `return();`

```
mov esp, ebp
```

存储栈指针

```
pop ebp
```

存储帧指针

```
ret
```

将返回地址从栈弹出，并把控制移交给那个位置

**EIP = 00411A87** **ESP = 0012FE08** **EBP = 0012FEDC**

**EIP:**扩展指令指针

**ESP:**扩展栈指针

**EBP:**扩展基指针

# 程序案例

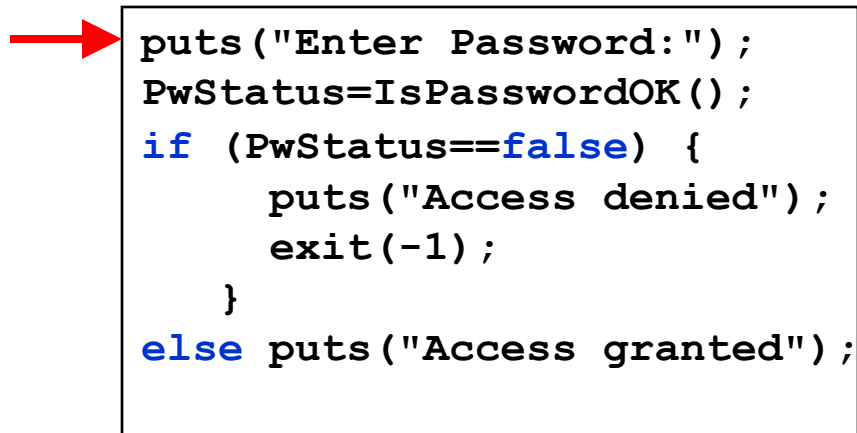
```
bool IsPasswordOK(void) {
    char Password[12]; // Memory 存储pwd
    gets(Password);    // Get input from keyboard
    if (!strcmp(Password, "goodpass")) return(true); //
    Password Good
    else return(false); // Password Invalid
}

void main(void) {
    bool PwStatus;           // Password Status
    puts("Enter Password:"); // Print
    PwStatus=IsPasswordOK(); // Get & Check Password
    if (PwStatus == false) {
        puts("Access denied"); // Print
        exit(-1);              // Terminate Program
    }
    else puts("Access granted");// Print
}
```

执行IsPasswordOK() 函数之前，栈中的信息

EIP

代码

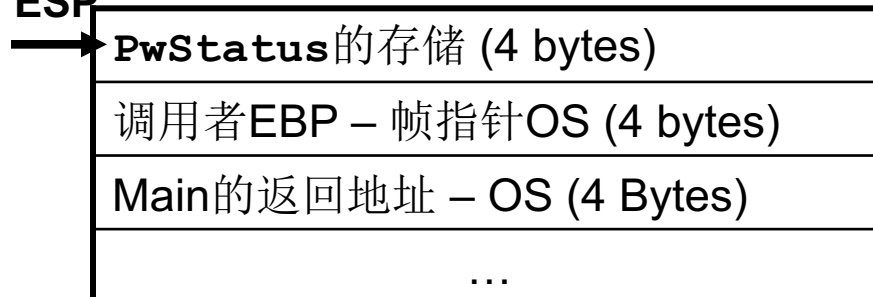


A red arrow points from the label 'EIP' to the first line of code in a box. The code is as follows:

```
puts("Enter Password:");  
PwStatus=IsPasswordOK();  
if (PwStatus==false) {  
    puts("Access denied");  
    exit(-1);  
}  
else puts("Access granted");
```

栈

ESP



# IsPasswordOK () 执行时栈中的信息

## 代码

EIP →

```
puts("Enter Password:");
PwStatus=IsPasswordOK();
if (PwStatus==false) {
    puts("Access denied");
    exit(-1);
}
else puts("Access granted");
```

```
bool IsPasswordOK(void) {
    char Password[12];

    gets(Password);
    if (!strcmp(Password, "goodpass"))
        return(true);
    else return(false)
}
```

## IsPasswordOK栈帧

ESP →	存储Password (12 Bytes)
	调用者EBP – 帧指针main (4 bytes)
EBP →	调用者的返回地址– main (4 Bytes)
	存储PwStatus (4 bytes)
	调用者EBP – 帧指针OS (4 bytes)
	Main的返回地址– OS (4 Bytes)
	...

注意: IsPasswordOK(void) 函数调用导致栈增长和收缩



# IsPasswordOK () 调用后栈中的信息

代码

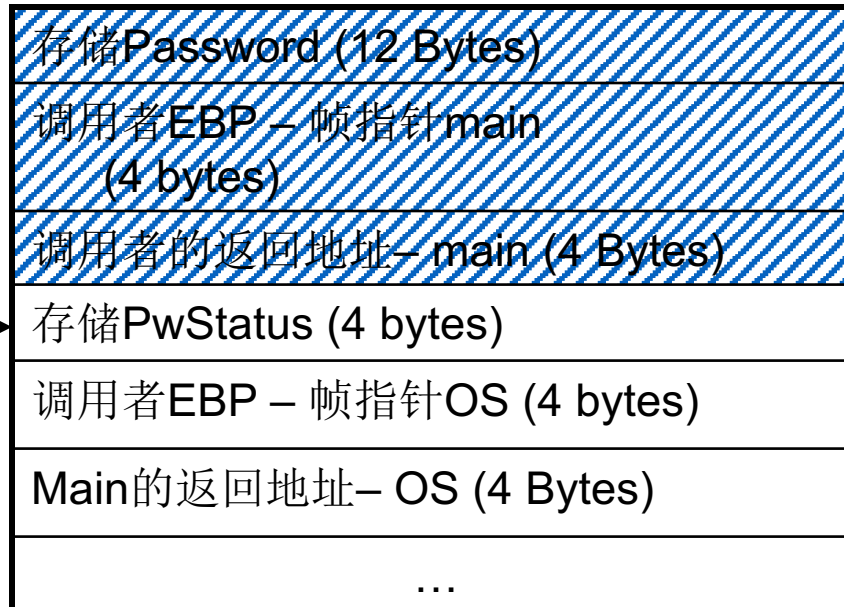
EIP



```
puts("Enter Password:");
PwStatus = IsPasswordOk();
if (PwStatus == false) {
    puts("Access denied");
    exit(-1);
}
else puts("Access granted");
```

栈

ESP



## 5.1 函数调用时发生了什么

我们假定有如下程序

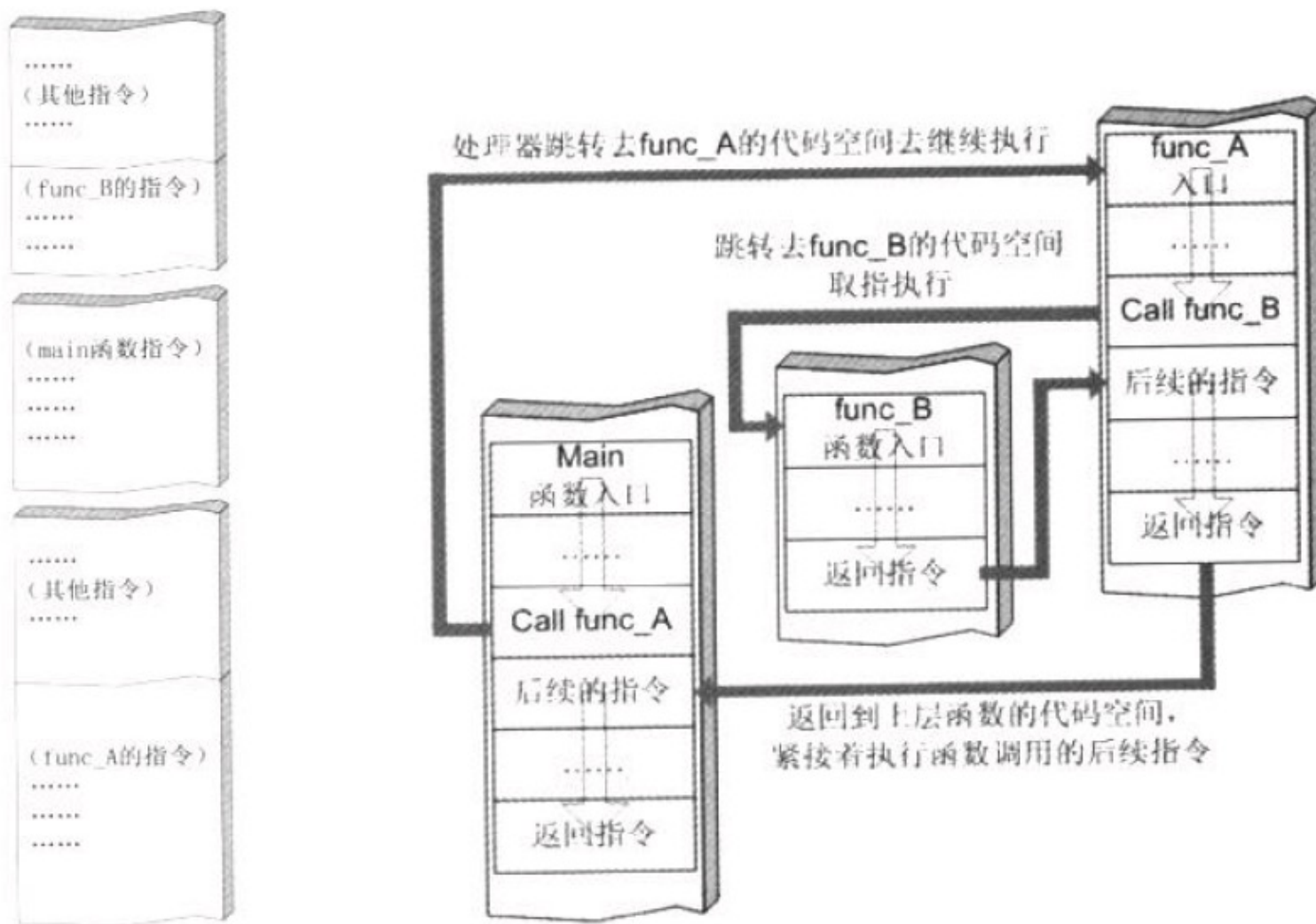
```
int funcb()  
{...  
}  
  
int funca()  
{...  
    funcb();  
...  
}
```

我们假定有如下程序

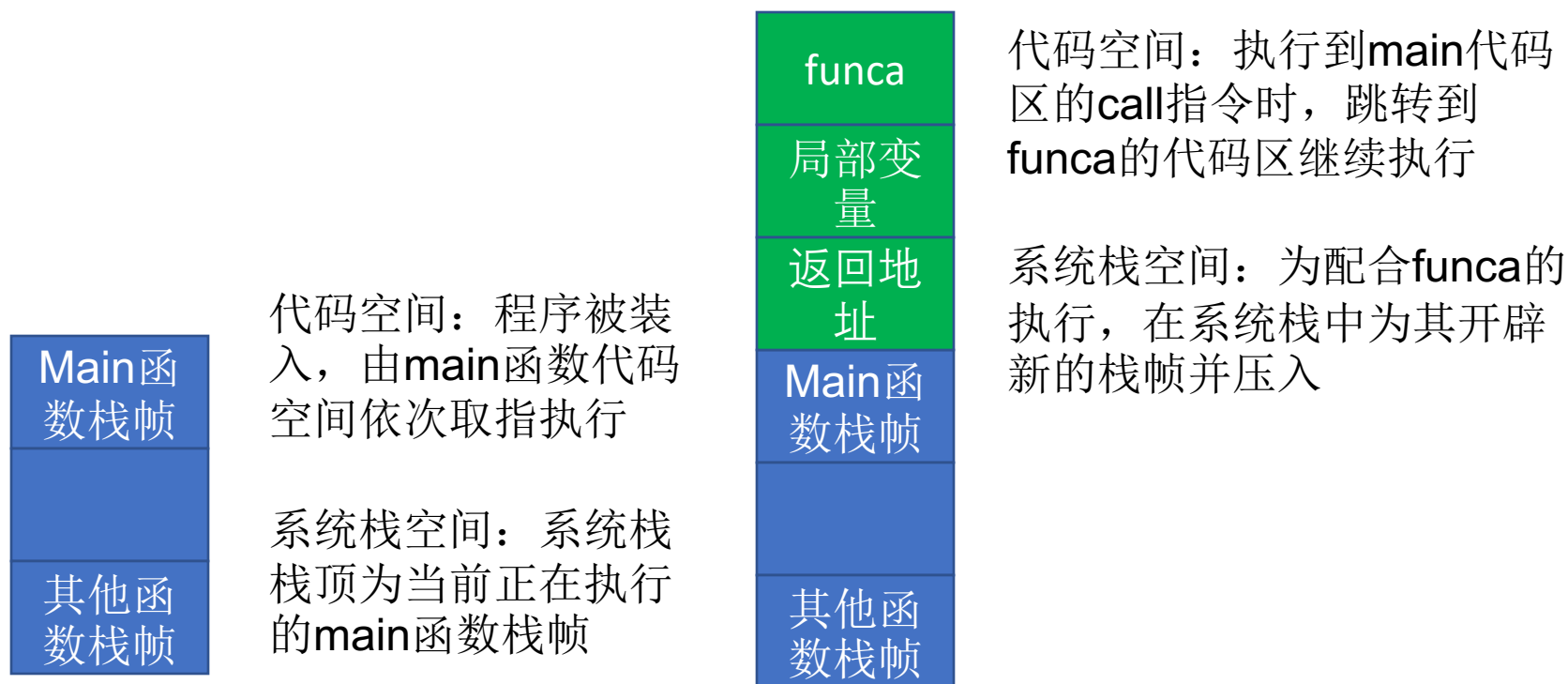
```
int main()  
{  
    funca();  
}
```

系统栈在函数调用时的变化是什么样的呢？

# 代码空间调用关系



# 系统栈在函数调用时的变化



# 系统栈在函数调用时的变化



代码空间：执行到 `funca` 代码区的 `call` 指令时，跳转到 `funcb` 的代码区继续执行

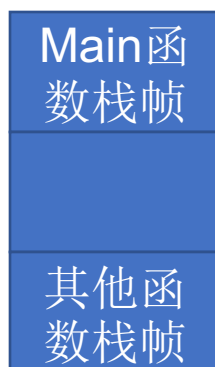
系统栈空间：为配合 `funcb` 的执行，在系统栈中为其开辟新的栈帧并压入



代码空间：`funcb` 代码执行完毕，弹出自己的栈帧并从中获得返回地址，跳回 `funca` 代码区继续执行

系统栈空间：弹出 `funcb` 的栈帧。对应于当前正在执行的函数，当前栈顶栈帧重新恢复成 `funca` 函数栈帧

# 系统栈在函数调用时的变化



代码空间：funca代码执行完毕，弹出自己的栈帧并从中获得返回地址，跳回main代码区继续执行

系统栈空间：弹出funca的栈帧。对应于当前正在执行的函数，当前栈顶栈帧重新恢复成main函数栈帧

## 5.2 寄存器与函数栈帧

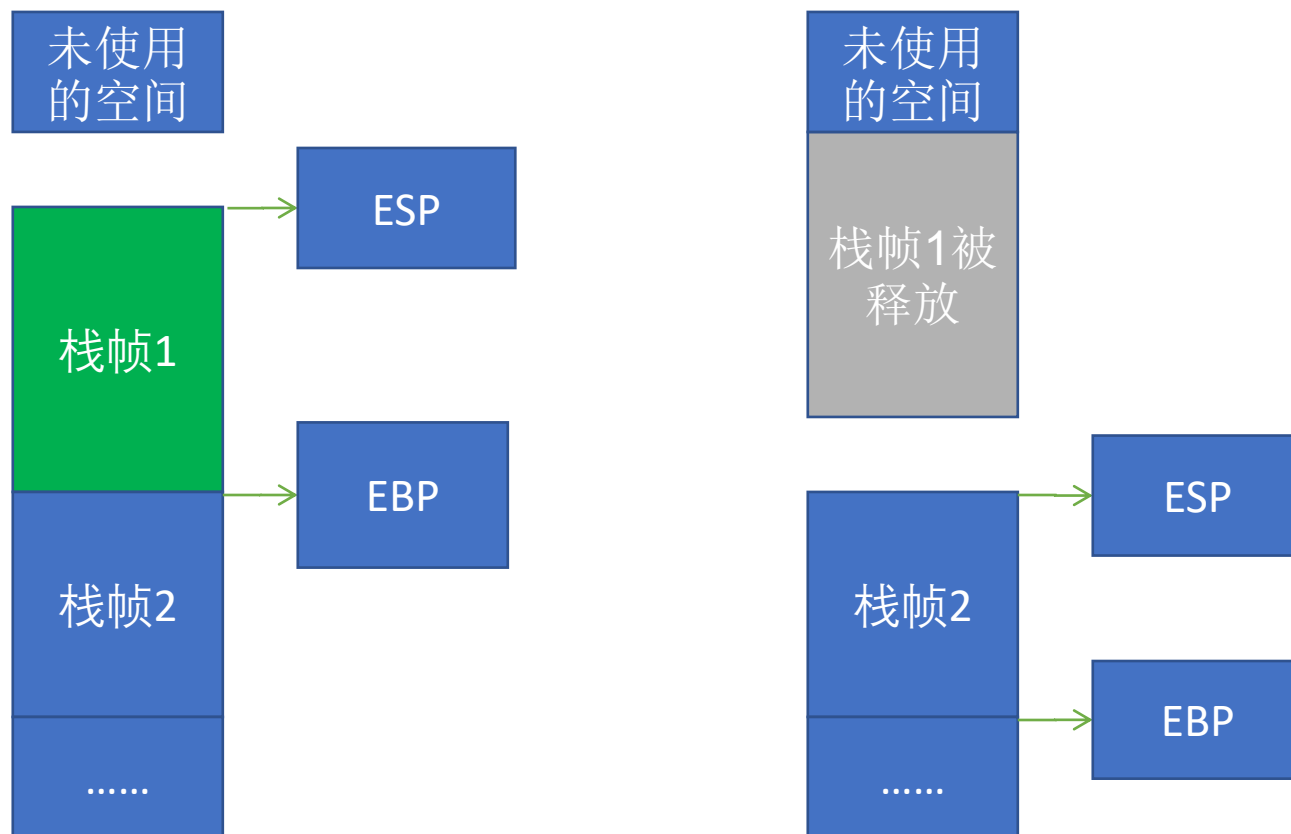
每一个函数独占自己的栈帧空间。当前正在运行的函数的栈帧总是在栈顶。Win32系统提供两个特殊的寄存器用于标识位于系统栈顶端的栈帧。

(1) ESP: 栈指针寄存器(extended stack pointer),其内存放着一个指针, 该指针永远指向系统栈最上面的一个栈帧的栈顶。

(2) EBP: 基址指针寄存器(extended base pointer),其内存放着一个指针, 该指针永远指向系统栈最上面的一个栈帧的底部。



# 栈帧寄存器ESP与EBP的作用



# 在函数栈帧中，一般包含以下几类重要信息

- (1) **局部变量**：为函数局部变量开辟的内存空间。
- (2) **栈帧状态值**：保存前栈帧的顶部和底部（实际上只保存前栈帧的底部，前栈帧的顶部可以通过堆栈平衡计算得到），用于在本帧被弹出后恢复出上一个栈帧。
- (3) **函数返回地址**：保存当前函数调用前的“断点”信息，也就是函数调用前的指令位置，以便在函数返回时能够恢复到函数被调用前的代码区中继续执行指令。

除了与栈相关的寄存器外，我们还需要记住另一个至关重要的寄存器。

**EIP**：指令寄存器(Extended Instruction Pointer),其内存放着一个指针，该指针永远指向一条等待执行的指令地址。

可以说如果控制了EIP寄存器的内容，就控制了进程---我们让EIP指向哪里，CPU就会去执行哪里指令。

## 5.3 函数调用约定与相关指令

函数调用大致包括以下几个步骤。

(1) 参数入栈：将参数从右向左一次压入系统栈中。



(2) 返回地址入栈：将当前代码区调用指令的下一跳指令地址压入栈中，供函数返回时继续执行。



(3) 代码区跳转：处理器从当前代码区跳转到被调用函数的入口处。



(4) 栈帧调整

# 第四步栈帧调整具体包括如下几个步骤

保存当前栈帧的状态值，以备后面恢复本栈帧时使用（EBP入栈）



将当前栈帧切换到新栈帧（将ESP值装入EBP，更新栈帧底部）



给新栈帧分配空间（把ESP减去所需空间的大小，抬高栈帧）

对于\_\_stdcall调用约定，函数调用时用到的指令序列大致如下

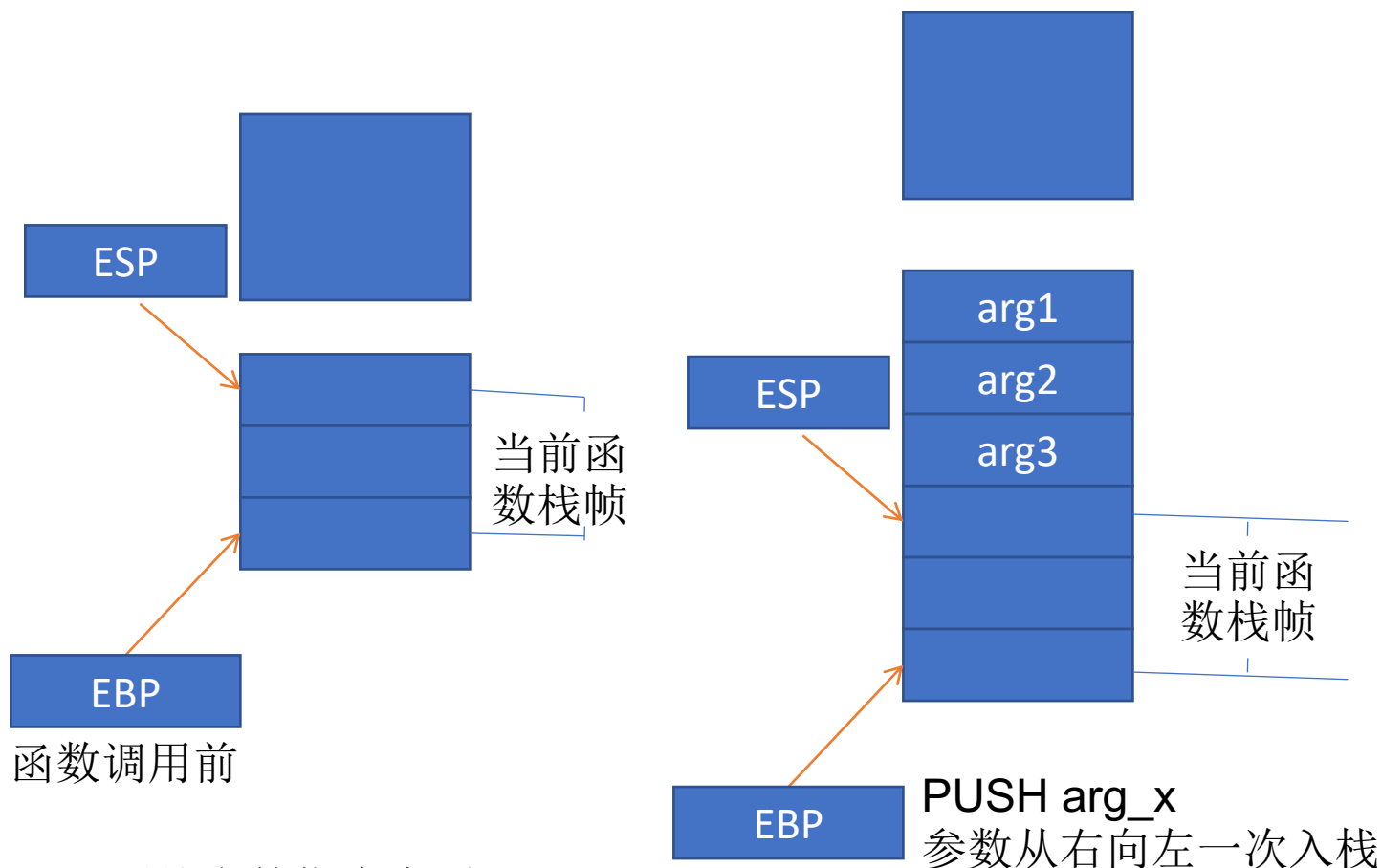
序列	备注
push 参数 3	假设该函数有3个参数，将从右向左依次入栈
push 参数 2	
push 参数 1	
call 函数地址	call指令将同时完成两项工作:a)向栈中压入当前指令在内存中的位置，即保存返回地址。 b)跳转到所调用函数的入口地址函数入口处：修改EIP
__stdcall调用约定的指令序列	

对于\_\_stdcall调用约定，函数调用时用到的指令序列大致如下

序列	备注
push ebp	保存旧栈帧的底部
mov ebp,esp	设置新栈帧的底部(栈帧切换)
sub esp,xxx	设置新栈帧的顶部(抬高栈顶，为新栈帧开辟空间)

\_\_stdcall调用约定的指令序列

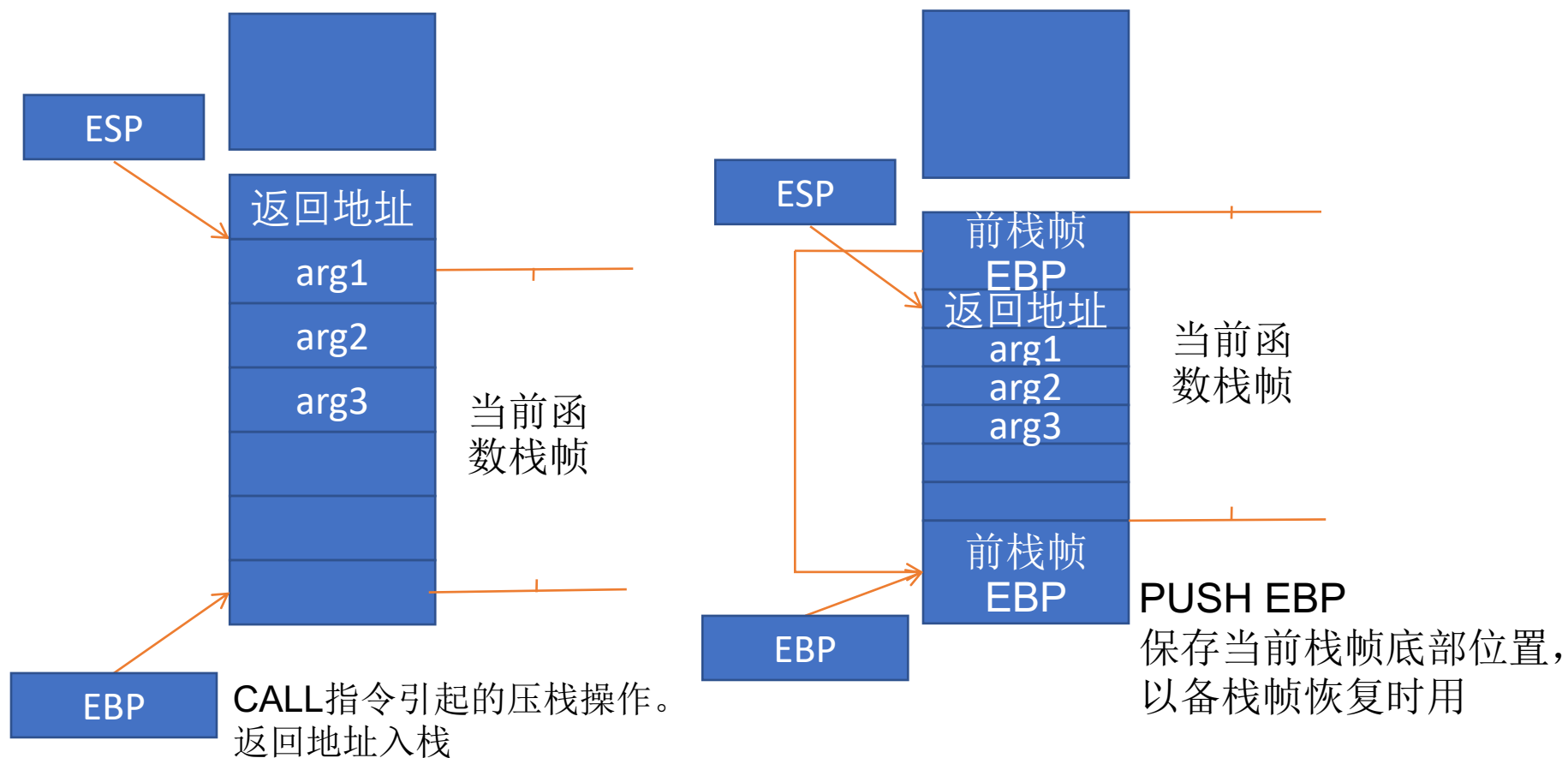
# 函数调用时系统栈的变化过程



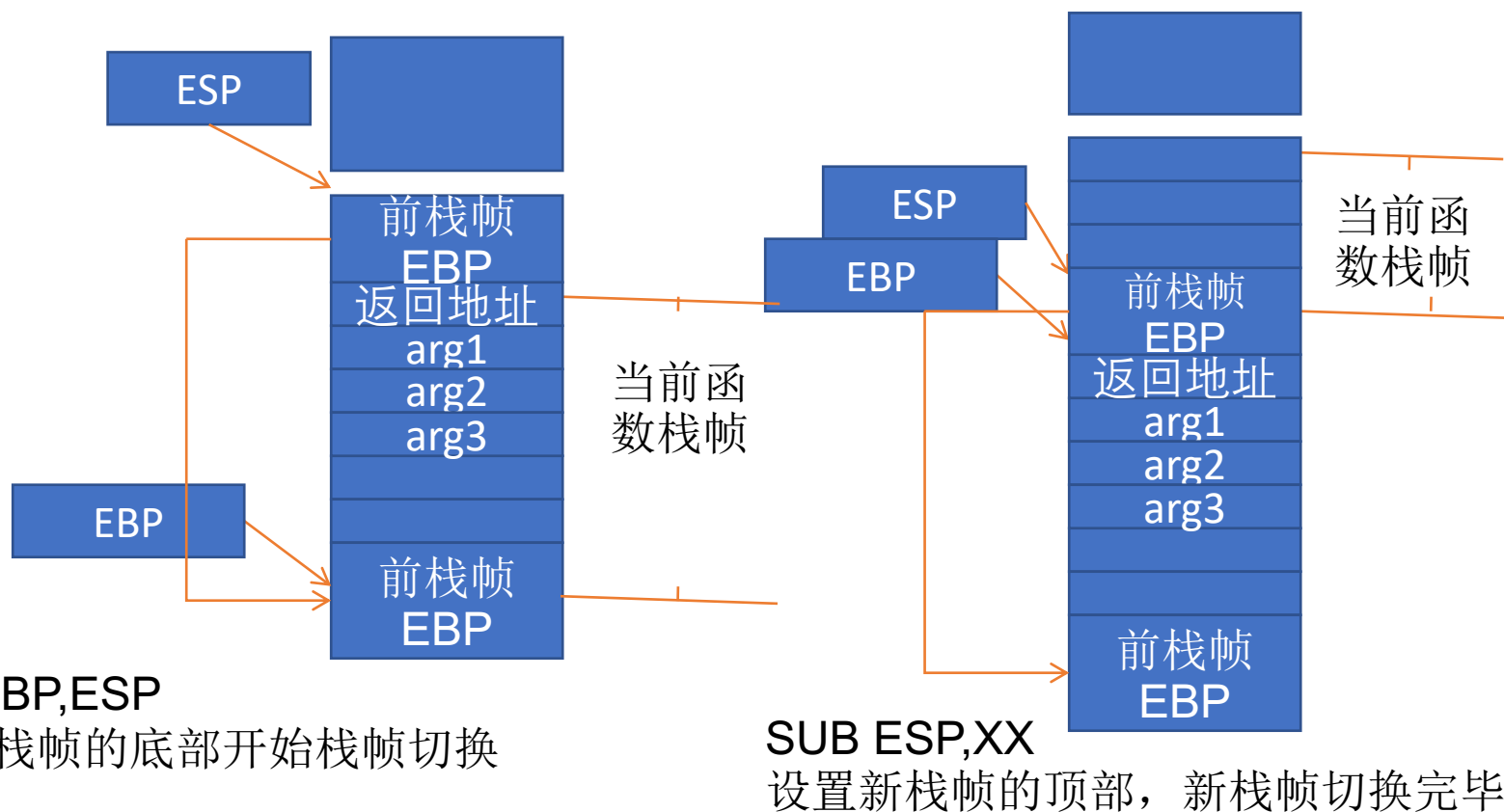
\_\_stdcall调用约定的指令序列



# 函数调用时系统栈的变化过程



# 函数调用时系统栈的变化过程



\_\_stdcall调用约定的指令序列

# 类似的，函数返回的步骤如下

- (1) 保存返回值：通常将函数的返回值保存在寄存器EAX中。
- (2) 弹出当前栈帧，恢复上一个栈帧，具体包括：
  - 在堆栈平衡的基础上，给ESP加上栈帧的大小，降低栈顶，回收当前栈帧的空间。
  - 将当前栈帧底部保存的前栈帧EBP值弹入EBP寄存器，恢复上一个栈帧。
  - 将函数返回地址弹给EIP寄存器。
- (3) 跳转：按照函数返回地址跳回母函数中继续执行。

# 以C语言和Win32平台为例，函数返回时的相关的指令序列

指令	备注
add esp,xxx	降低栈顶，回收当前的栈帧
pop ebp	将上一个栈帧底部位置恢复到ebp
retn	<p>这条指令有两个功能：</p> <p>a)弹出当前栈顶元素，即弹出栈帧中的返回地址。至此，栈帧恢复工作完成。</p> <p>b)让处理器跳转到弹出的返回地址，恢复调用前的代码区</p>

# 本讲小结

- 计算机引导
- Win32内存体系
- PE文件结构
- 进程空间分区
- 系统栈工作

谢谢大家

