東北大學秦皇島分校
Northeastern University at Qinhuangdao

Linux系统与内核分析

--内存寻址

于七龙

目录

Part 1 x86内存寻址

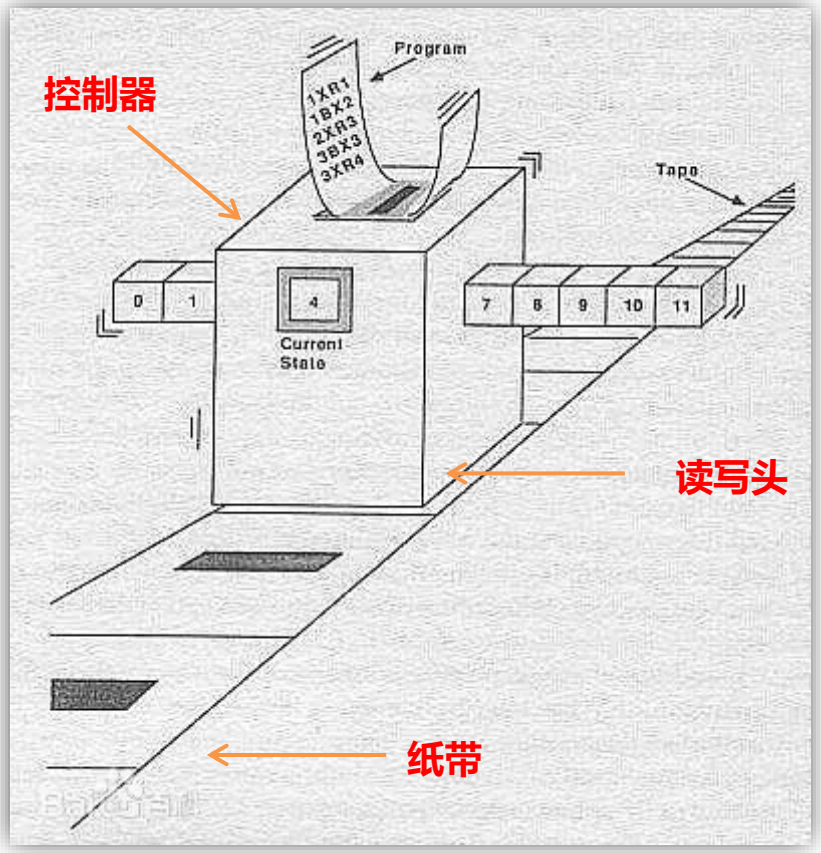
Part 2 Linux的分段与分页

Part 3 Linux中的汇编语言

Part 1 x86内存寻址

图灵机

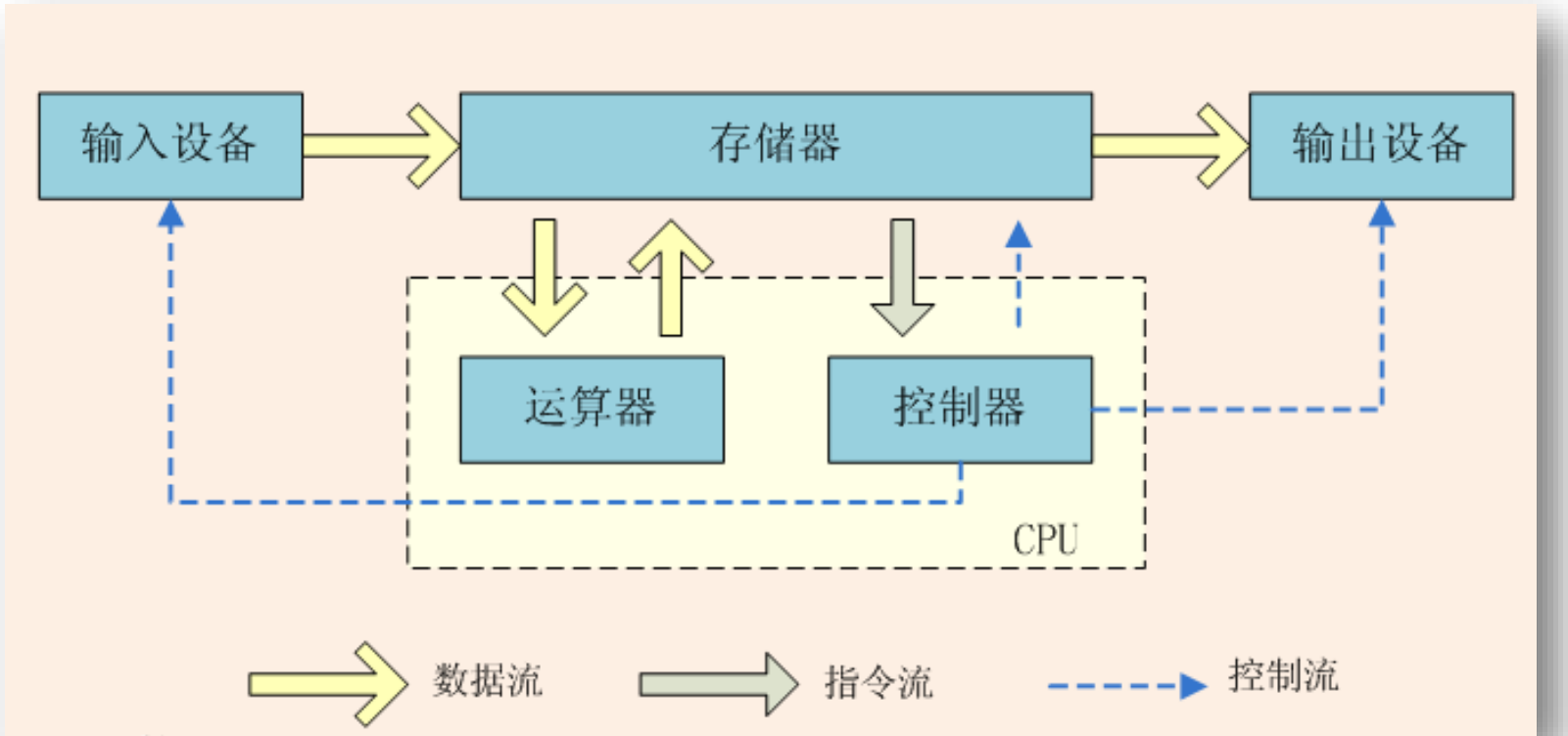
- 阿兰.图灵在论文《论数字计算在决断难题中的应用》（《On Computable Numbers, with an Application to the Entscheidungsproblem》）中提出
- 模型设想由一个控制器、一个读写头和一条假设两端无限长的工作带组成。读写头可以在工作带上随意移动



图灵机模型

冯.诺依曼体系结构

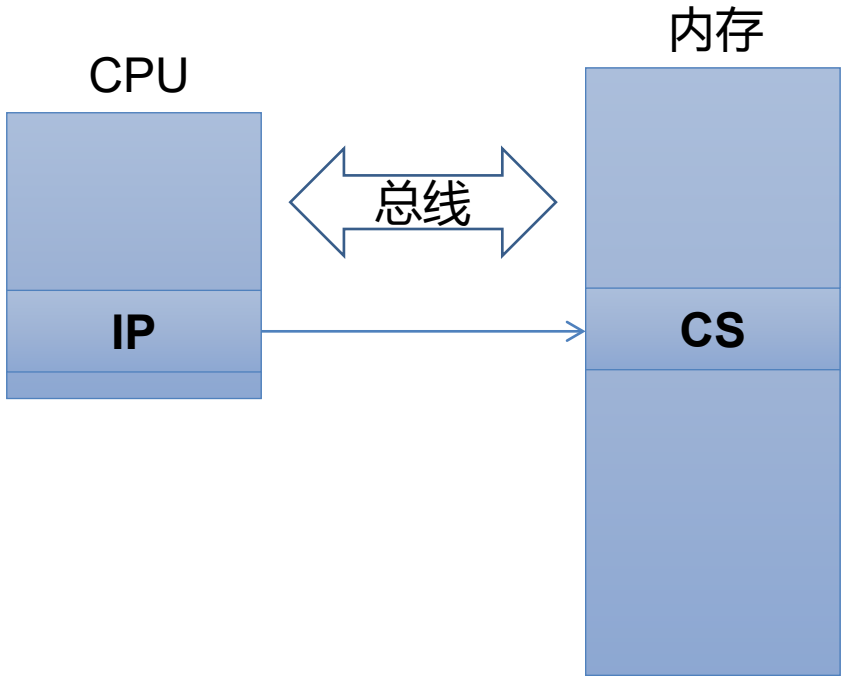
- 图灵机是冯.诺依曼计算机体系的鼻祖

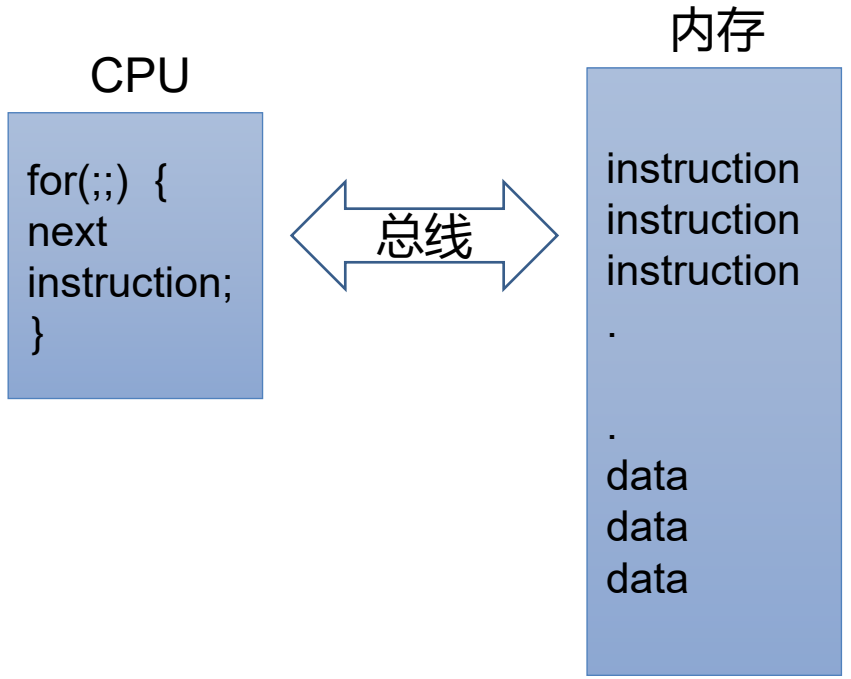


冯.诺依曼体系结构硬件视角

冯.诺依曼体系结构

- 图灵机是冯.诺依曼计算机体系的鼻祖





- 冯.诺依曼体系结构软件视角

操作系统的硬件相关代码

- 操作系统的设计者需在硬件相关代码与硬件无关的代码之间画出清楚的界限
  - Linux操作系统具有良好的可移植性
  - Linux把与硬件相关代码全部放在arch目录中

Intel x86 CPU

- Intel x86 CPU的演变

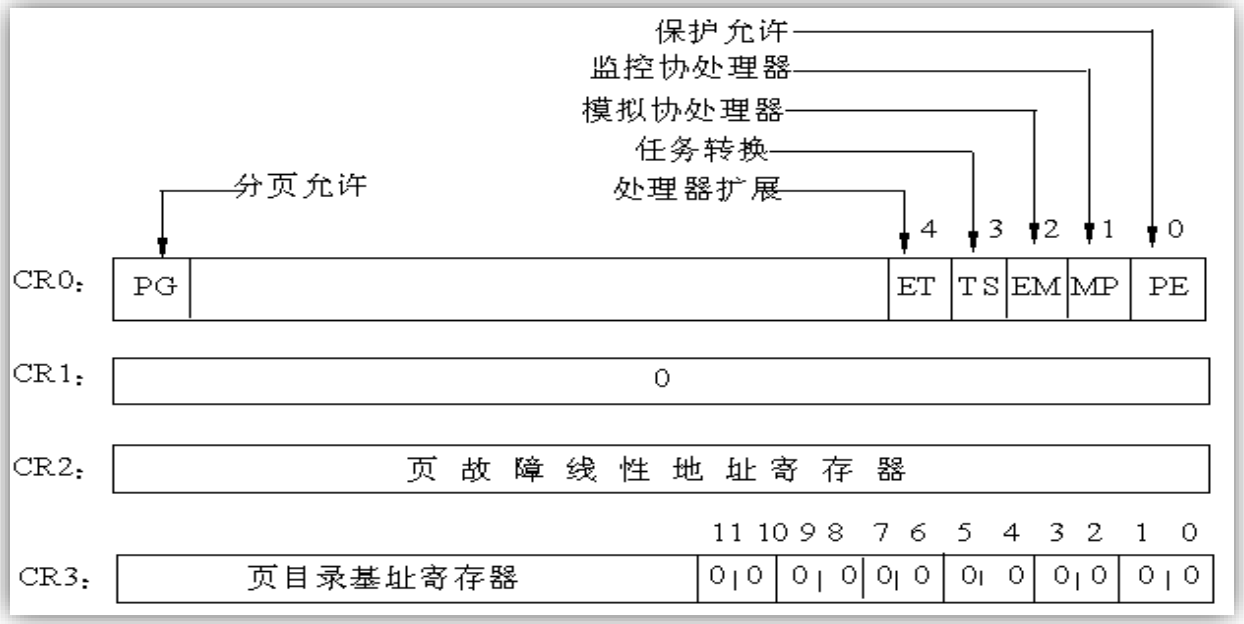
80x86寄存器

- 通用寄存器（32位）
  - EAX、EBX、ECX、EDX、EBP、ESP、 ESI、 EDI
- 段寄存器
  - 8086：4个16位的段寄存器：CS、DS、SS、ES，分别存放可执行代码的代码段、数据段、堆栈段和其他段的基地址
  - 80x86:6个16位的段寄存器，存放某个段的选择符
- 指令指针寄存器、标志寄存器
  - EIP：下一条将要执行指令的偏移量
  - 标志寄存器EFLAGS存放处理器的控制标志

80x86寄存器

控制寄存器

- 80x86有4个32位用于分页机制的控制寄存器(CR0、CR1、CR2、CR3)



控制寄存器组

虚拟地址、线性地址、物理地址

◆ 物理地址

- 物理内存条中的内存单元的实际地址

◆ 虚拟地址

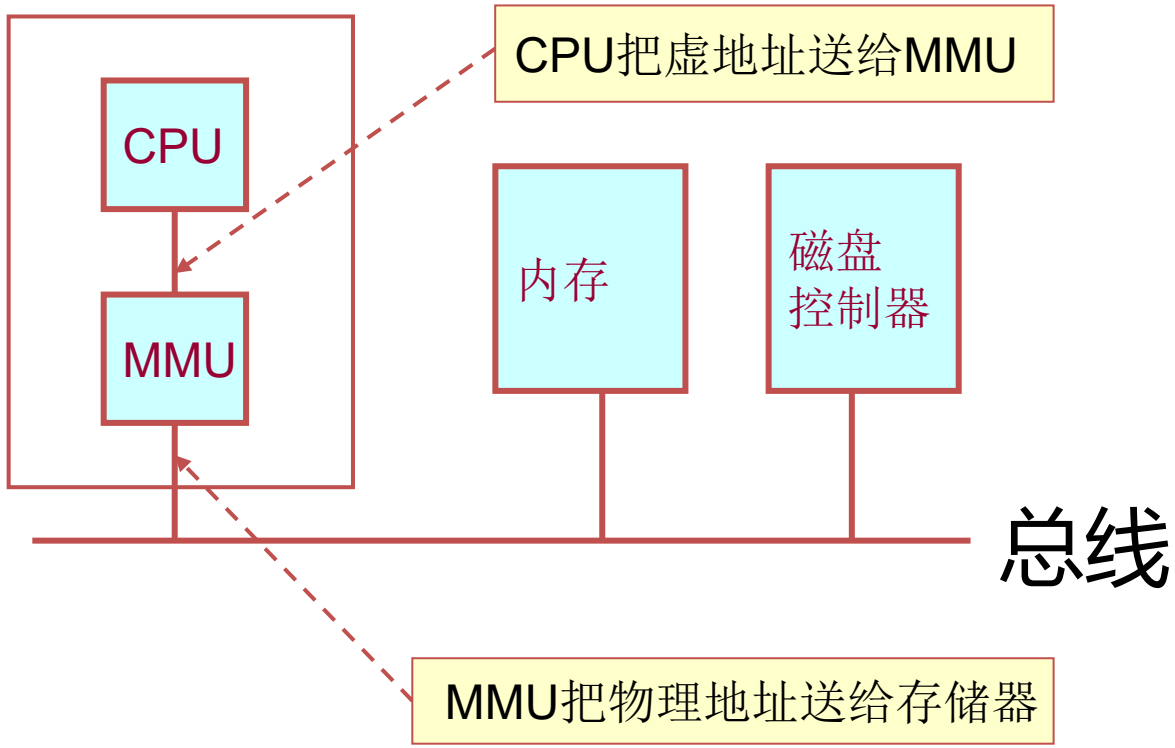
- 应用程序看到的内存空间定义为虚拟地址空间，其中的地址就叫虚拟地址(或逻辑地址)
- 虚拟地址一般用“段：偏移量”形式描述，如“A815：CF2D”表示段地址是“A815”，段内偏移量是“CF2D”

◆ 线性地址

- 一段连续的，不分段的，范围为0到4GB的地址空间（32位），一个线性地址就是线性地址空间的一个绝对地址

地址转换

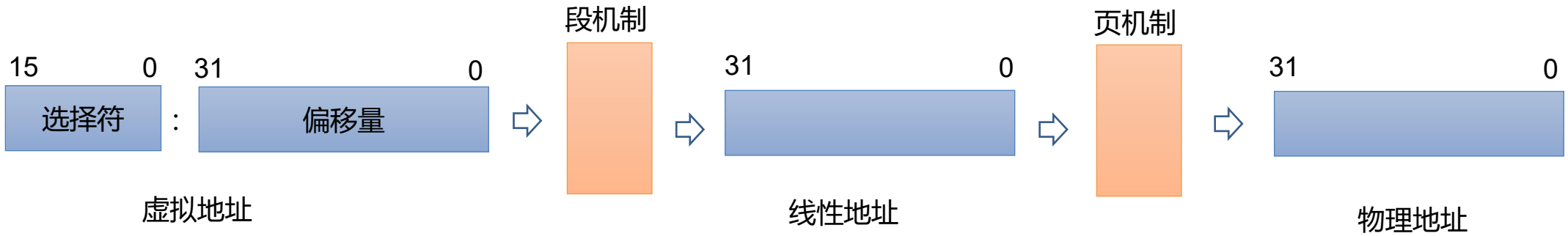
- 内存管理单元(MMU)用于将虚拟地址转换为物理地址



地址转换

地址转换

- 内存管理单元(MMU)用于将虚拟地址转换为物理地址



■ 地址转换

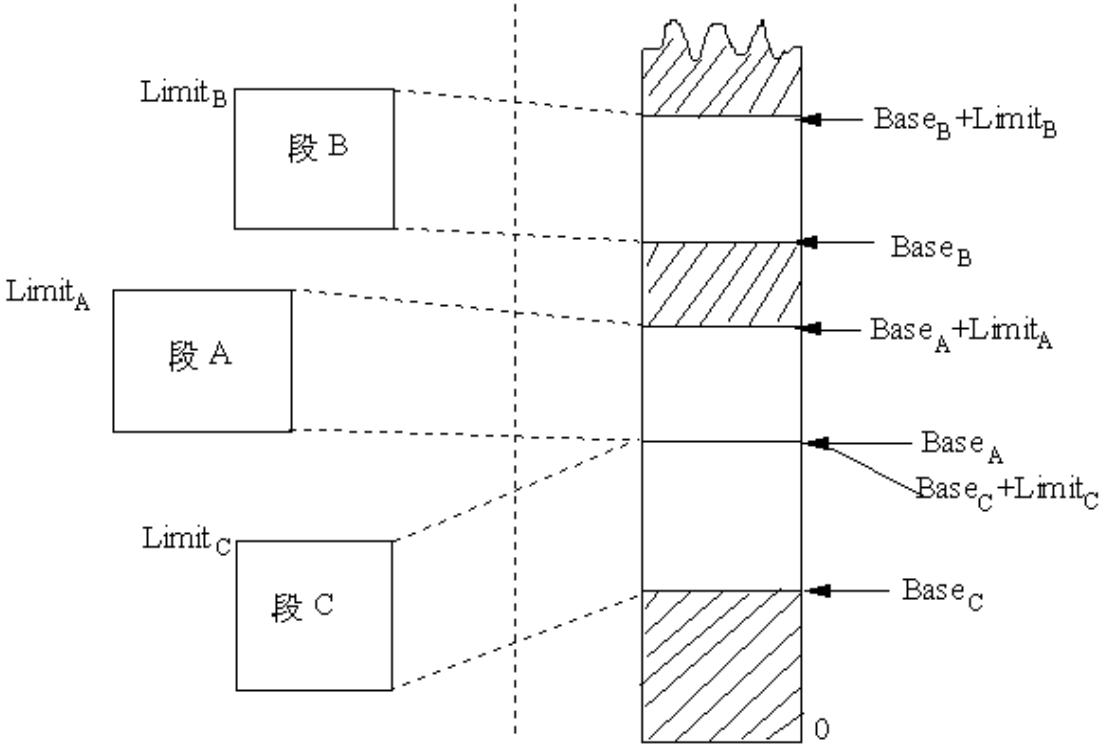
Part 2 Linux的分段与分页

分段机制

- 分段机制是x86架构的特有机制
  - 8086的CPU为16位，为达到1MB大的寻址空间，其地址总线扩展为20位，为填补该空隙，Intel设计了分段方法
  - 8086 CPU中设置4个16位的段寄存器（CS、DS、SS、ES），CPU内部地址在送往地址总线之前，CPU自动把内部地址与左移4位的相应段寄存器中的内容相加，实现16位内存地址到20位地址的转换

分段机制

- 分段机制通过段描述符表（或段表）描述分段属性



索引	基地址	界限	属性
0	Base-b	Limit-b	Attribute-b
1	Base-a	Limit-a	Attribute-a
2	Base-c	Limit-c	Attribute-c

- 虚拟地址-线性地址映射

Linux与分段机制

- 分段机制是x86架构的特有机制，为保持兼容性，x86延续了分段机制
- x86段机制要求
  - 任意给出的地址都是虚拟地址，即地址由“段地址：偏移量”组成
  - 必须为代码段与数据段创建不同的段
  - 段机制不可禁止
- 绝大多数硬件平台都不支持段机制

Linux与分段机制

- 为简化内核设计与保持可移植性，Linux有限度使用了分段机制
- Linux从两方面设计了段机制

1. Linux设置段基址为0，段界限设置为4G，实现虚拟地址直接映射到线性地址
   - “0 + 偏移量” = 线性地址
   - 段机制规定“偏移量<4GB”，即偏移量的范围为0H～FFFFFFFFH，这恰好是32位架构中线性地址空间范围
2. 分别创建特权级为0和3的代码段和数据段

Linux的分段机制

- 为简化内核设计与保持可移植性，Linux有限度使用了分段机制
- Linux从两方面设计了段机制

1. Linux设置段基址为0，段界限设置为4G，实现虚拟地址直接映射到线性地址
2. 分别创建特权级为0和3的代码段和数据段

   - x86段机制规定，必须为代码段和数据段创建不同的段，所以Linux必须为代码段和数据段分别创建一个基地址为0，段界限为4GB的段描述符。此外，由于Linux内核运行在特权级0，而用户程序运行在特权级别3，段保护机制规定，特权级3的程序是无法访问特权级为0的段，所以Linux必须为内核用户程序分别创建其代码段和数据段。这就意味着Linux必须创建4个段描述符：特权级0的代码段和数据段，特权级3的代码段和数据段

分页机制

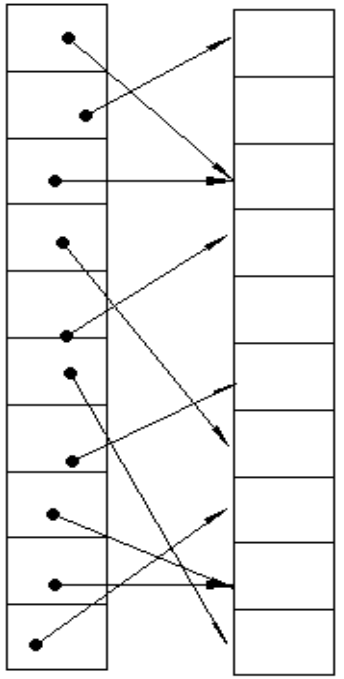
- 绝大多数操作系统均使用了分页机制
- X86中分页机制可选
- 分页原理
  - 将线性地址空间分成若干大小相等的片，成为页，并对每页编号
  - 将物理地址空间均分成若干与页大小相等的存储块并编号，称为页面
  - 32位系统标准页大小为4KB，64位系统为8KB

页表

- 页表用于描述线性地址到物理地址的映射关系
- 页表内容
  - 物理页面基地址：物理页面起始地址
  - 页属性：是否在内存中，是否可被读写等

页	页面基地址	属性
1	base1	*
2	base2	*
…	…	…

对于4G内存，共有4G/4KB=1M个页，即1M个页表项，每个页表项占用4字节，则页表需占用4M连续的空间



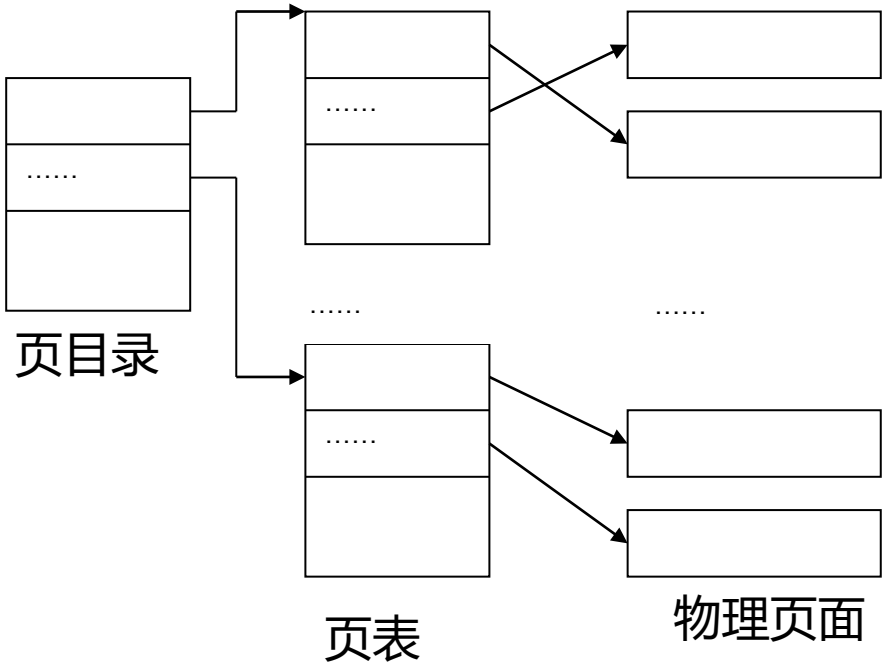
- 页-页面对应关系

两级页表

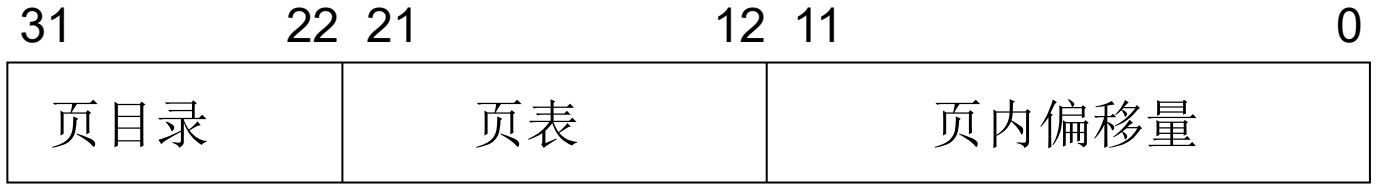
- 两级页表即对页表再进行分页
  - 第一级称为页目录，存放关于页表的信息
  - 第二级称为页表，即页与页面的映射关系

对于4MB空间的页表再分页，需可分为4MB/4KB=1K个页，每个页描述符4B，则页目录占用4KB，刚好一个页

对于1K个页，需占用10位表示，即页目录占用10位；每个页目录包含1K个页表，即页表占用10位；每个页面4KB，即占用12位

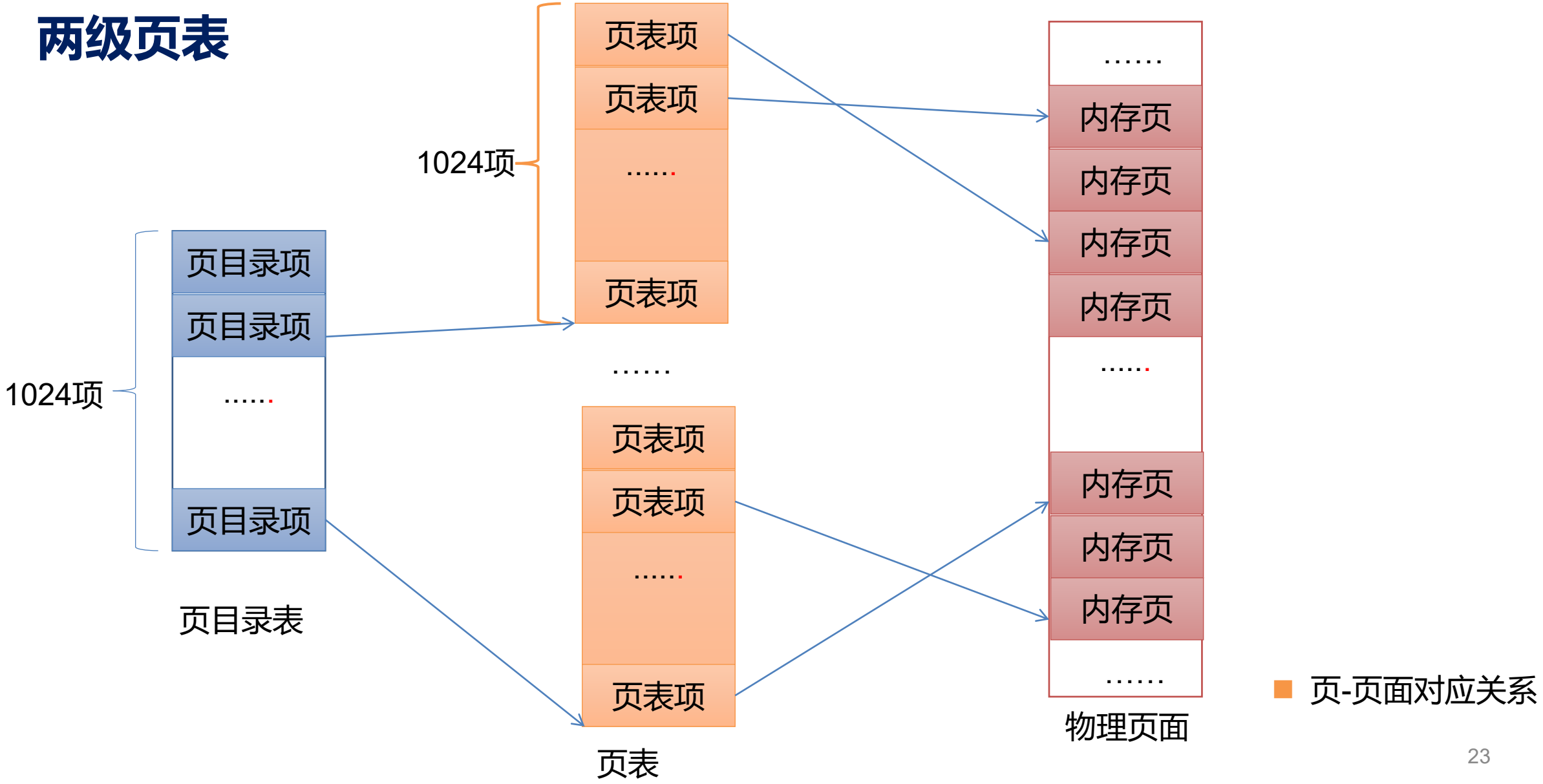


两级页表



两级页表线性地址结构

两级页表

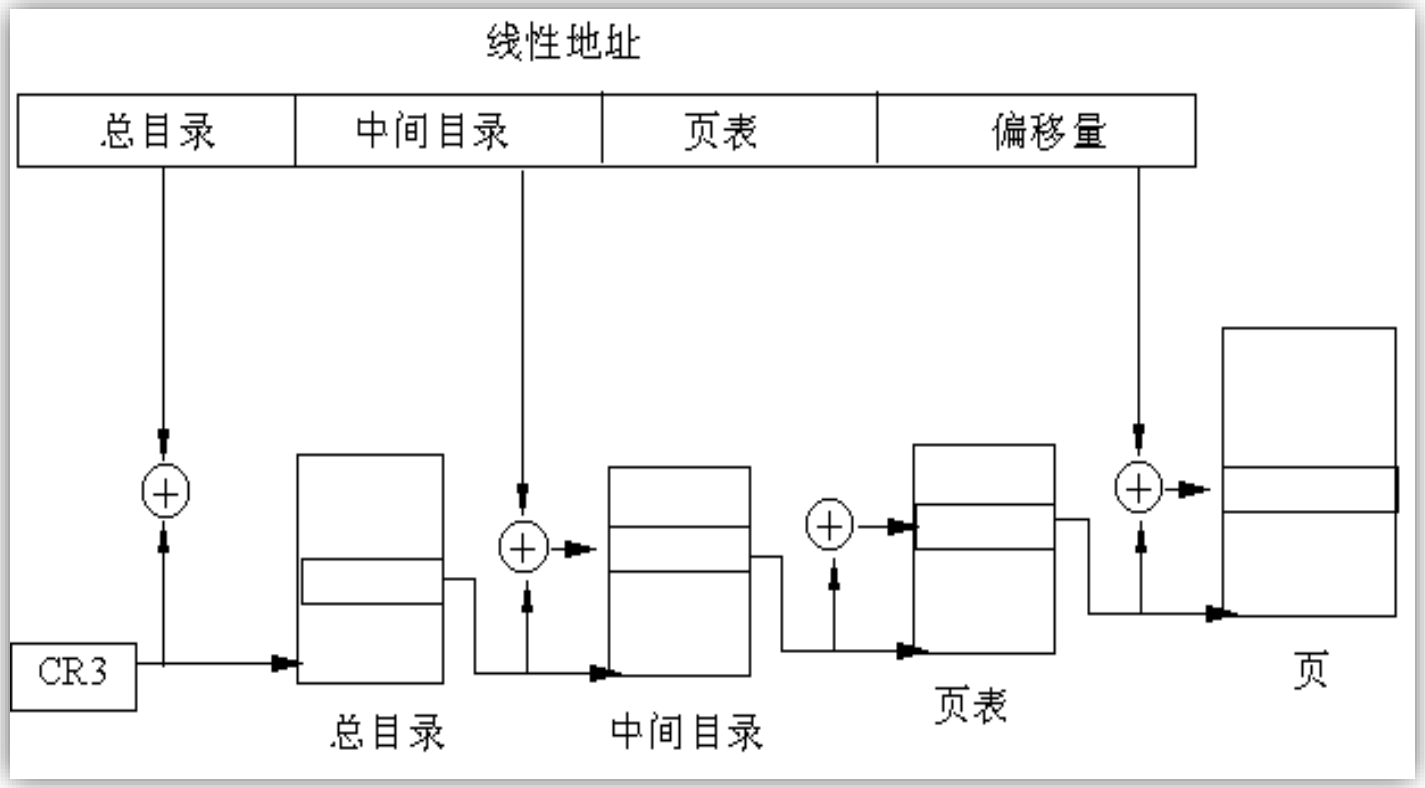

页表项
页表项
.....
页表项
1024项
.....
页目录项
页目录项
.....
页目录项
1024项
页目录表
页表项
页表项
.....
页表项
页表
.....
内存页
内存页
内存页
内存页
.....
内存页
内存页
内存页
.....
物理页面
页-页面对应关系

Linux的分页机制

- Linux主要采用分页机制来实现虚拟存储器管理
  - Linux的分段机制使得所有的进程都使用相同的段寄存器值，这就使得内存管理变得简单，32位系统中，所有的进程都使用同样的线性地址空间（0-4G）
  - Linux设计目标之一就是能够把自己移植到绝大多数流行的处理器平台，但是，许多RISC处理器支持的段功能非常有限
- 为了保持可移植性，Linux采用三级或四级分页模式而不是两级
  - 32位寻址空间采用三级分页
  - 64位寻址空间采用四级分页

Linux的分页机制

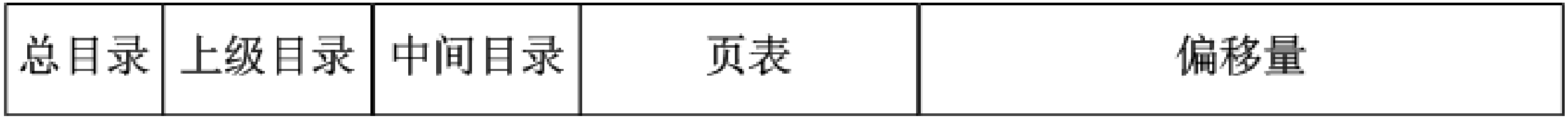
- 三级分页模式定义以下页表类型
  - 页总目录
  - 页中间目录
  - 页表



三级分页

Linux的分页机制

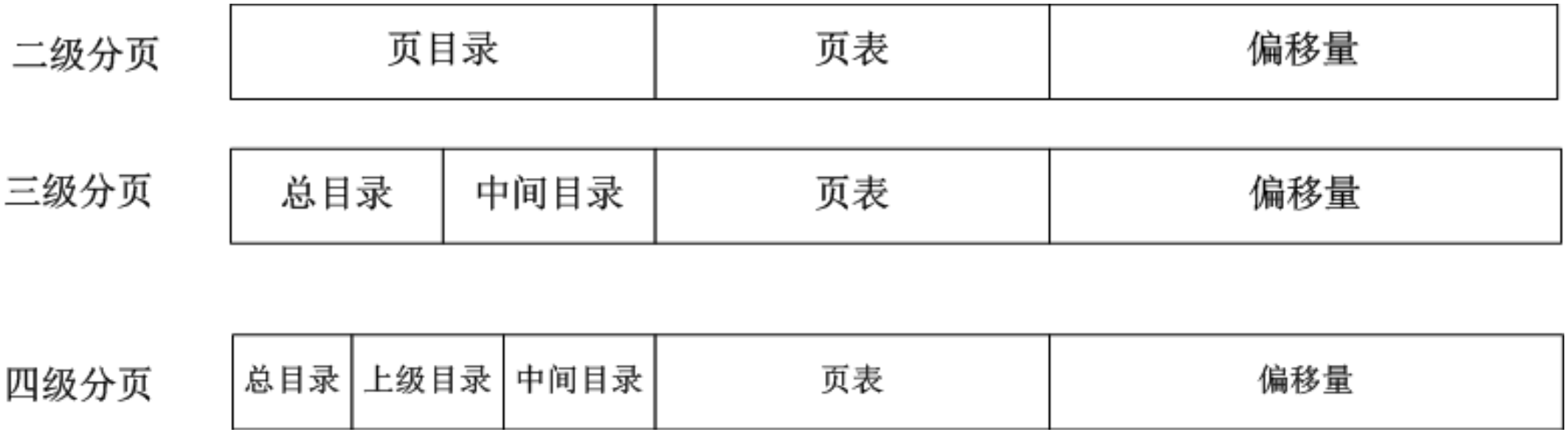
- 四级分页模式



四级分页

Linux的分页机制的兼容性

- Linux对不同分级的页面进行了兼容性设计
  - Linux使用一系列宏掩盖各种平台细节
  - Linux中页表的中间目录可进行折叠



- 多级分页兼容

Part 3 Linux中的汇编语言

Linux中的汇编

- Linux内核代码大部分由C语言编写，其余是汇编语言
  - 汇编语言出现在以.s位扩展名的汇编文件中
  - .c的C文件中也可嵌套汇编语言
- Linux源代码采用AT&T的i386汇编语言

Linux中的汇编

- Linux内核源代码很多代码采用C语言嵌入一段汇编语言程序段

```
static inline unsigned long get_limit(unsigned long segment)
{
        unsigned long __limit;
        asm("lsll %1,%0" : "=r" (__limit) : "r" (segment));
        return __limit + 1;
}
```

AT&T汇编语言基础

```
# include <stdio.h>
greeting ( )
{
    printf( “Hello,world!\n” );
}
main()
{
    greeting();
}
```

```
% objdump –d hello
08048568 <greeting>:
8048568:      pushl  %ebp
8048569:      movl  %esp, %ebp
804856b:      pushl  $0x809404
8048570:      call  8048474  <_init+0x84>
8048575:      addl   $0x4, %esp
8048578:      leave
8048579:      ret
804857a:      movl  %esi, %esi
0804857c <main>:
804857c:      pushl  %ebp
804857d:      movl  %esp, %ebp
804857f:      call    8048568  <greeting>
8048584:      leave
8048585:      ret
```

AT&T汇编语言基础

AT&T语法	含义
movl %edx, %eax	eax = edx
movl $0x123, %eax	eax = 0x123
movl 0x123, %eax	eax = *(int32_t*)0x123
movl (%edx), %eax	eax = *(int32_t*)edx
movl 4(%edx), %eax	eax = *(int32_t*) (edx+4)

AT&T汇编语言基础

AT&T语法	含义
pushl %eax	subl $4, %esp movl %eax, (%esp)
popl %eax	movl (%esp), %eax, addl $4, %esp
call 0x123	pushl %eip(*) movl $0x123, %eip(*)
leave	movl %ebp, %esp popl %ebp
ret	popl %eip(*)

系统地址映射举例

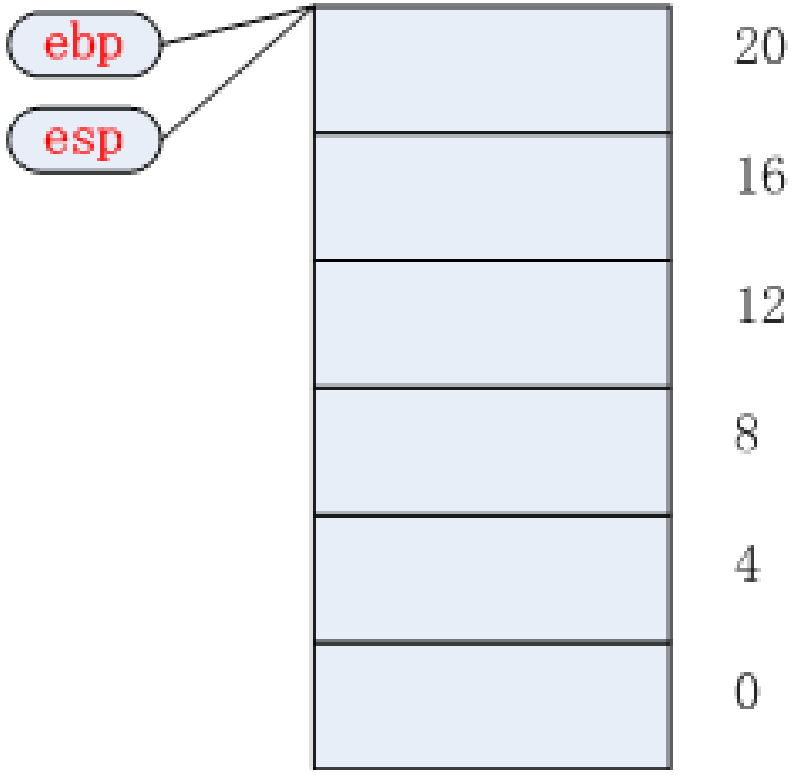
```
int f(int x)
{
  return x+1;
}

int main(void)
{
  return f(1)+1;
}
```

```
f:
    pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp), %eax
    addl $1, %eax
    popl %ebp
    ret
main:
    pushl %ebp
    movl %esp, %ebp
    subl $4, %esp
    movl $1, (%esp)
    call f
    addl $1, %eax
    leave
    ret
```

系统地址映射举例

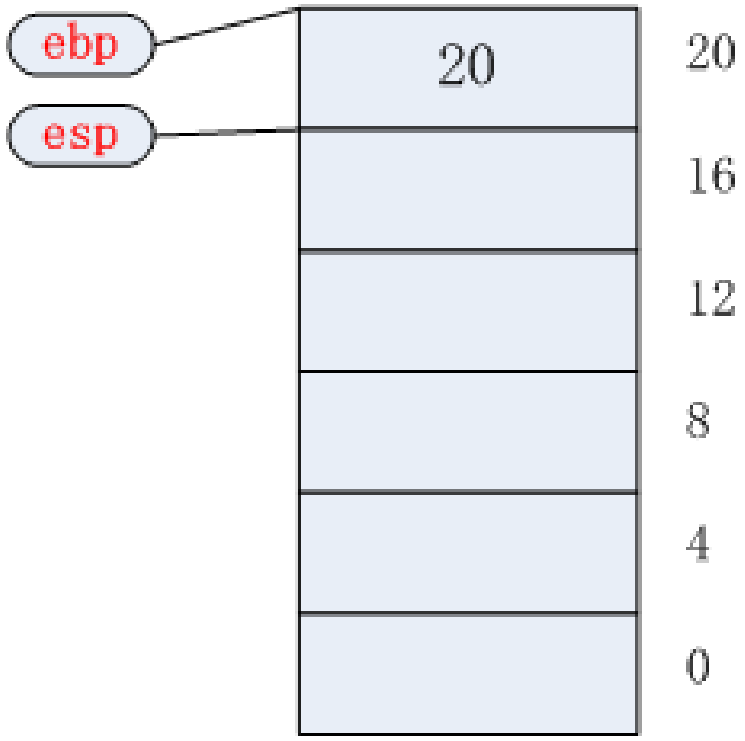
- 寄存器含义
  - ebp:栈底寄存器（Extend Base Pointer）
  - esp:栈顶存器（Extend Stack Pointer）
  - eip:指令寄存器（Extend Instruction Pointer)
  - eax:累积暂存器（Accumulator)

Linux中的汇编语言



pushl %ebp

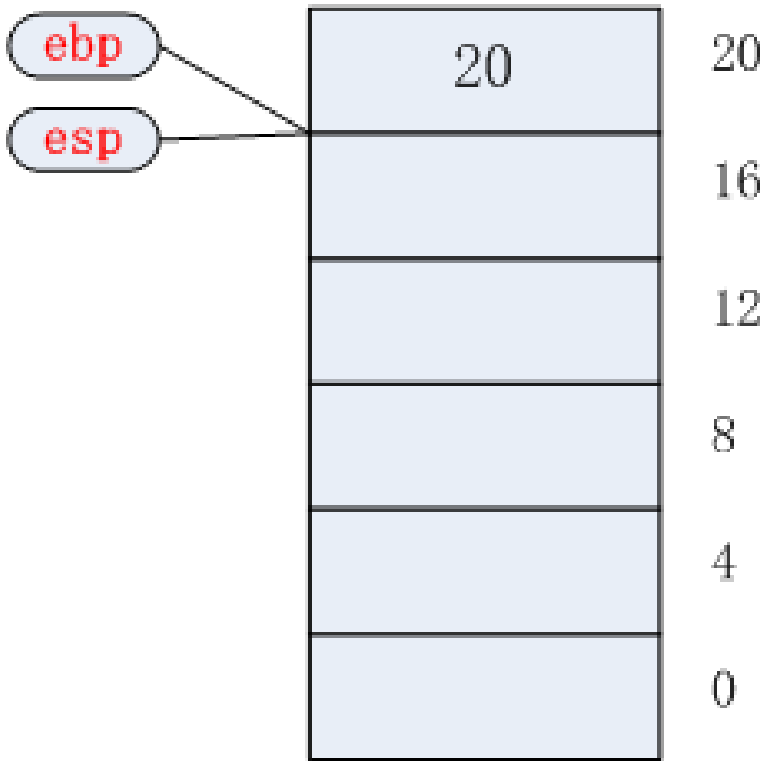
subl $4, %esp
movl %ebp, (%esp)



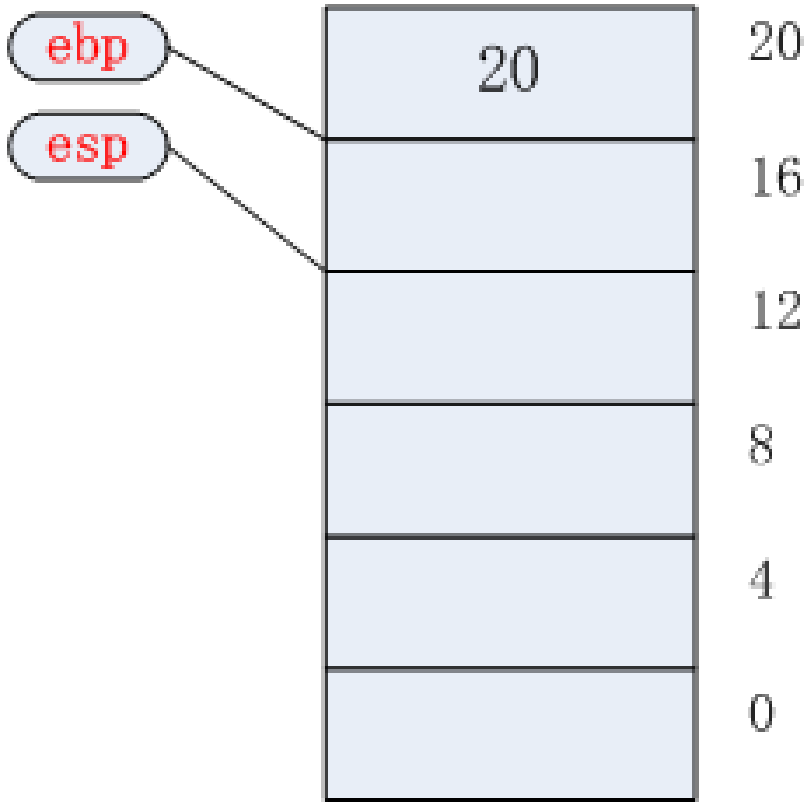

```
f:
   pushl %ebp
   movl %esp, %ebp
   movl 8(%ebp), %eax
   addl $1, %eax
   popl %ebp
   ret
main:
   pushl %ebp
   movl %esp, %ebp
   subl $4, %esp
   movl $1, (%esp)
   call f
   addl $1, %eax
   leave
   ret
```

Linux中的汇编语言

movl %esp, %ebp

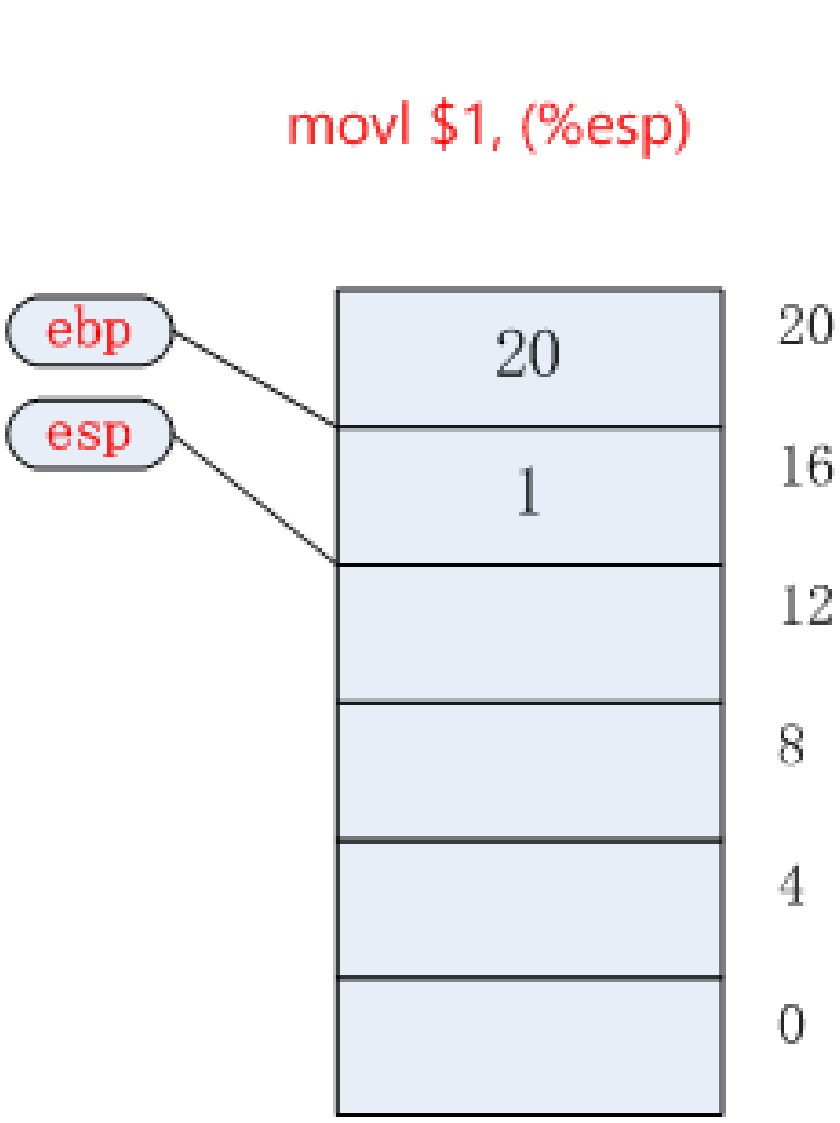


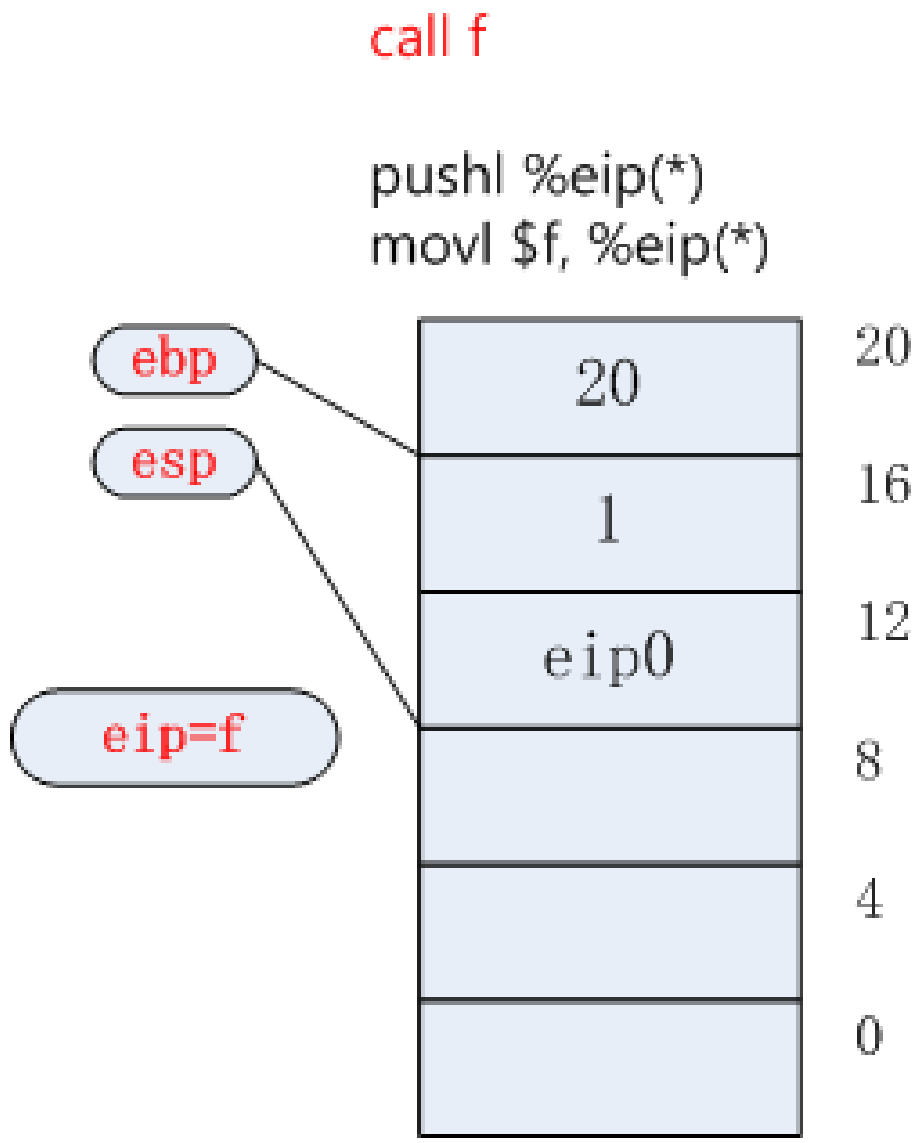
subl $4, %esp




```
f:
   pushl %ebp
   movl %esp, %ebp
   movl 8(%ebp), %eax
   addl $1, %eax
   popl %ebp
   ret
main:
   pushl %ebp
   movl %esp, %ebp
   subl $4, %esp
   movl $1, (%esp)
   call f
   addl $1, %eax
   leave
   ret
```

Linux中的汇编语言





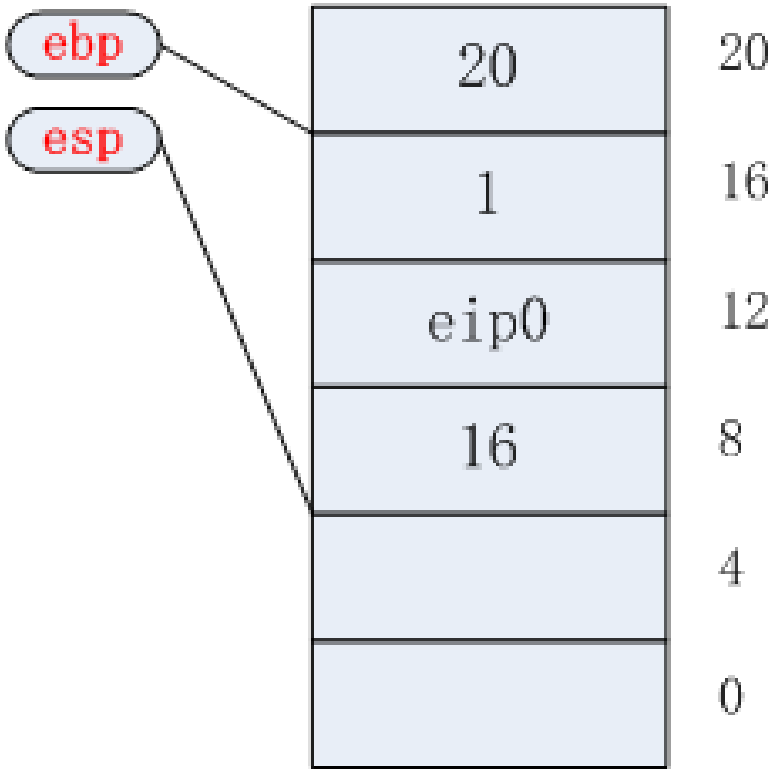

```
f:
   pushl %ebp
   movl %esp, %ebp
   movl 8(%ebp), %eax
   addl $1, %eax
   popl %ebp
   ret
main:
   pushl %ebp
   movl %esp, %ebp
   subl $4, %esp
   movl $1, (%esp)
   call f
   addl $1, %eax
   leave
   ret
```

Linux中的汇编语言

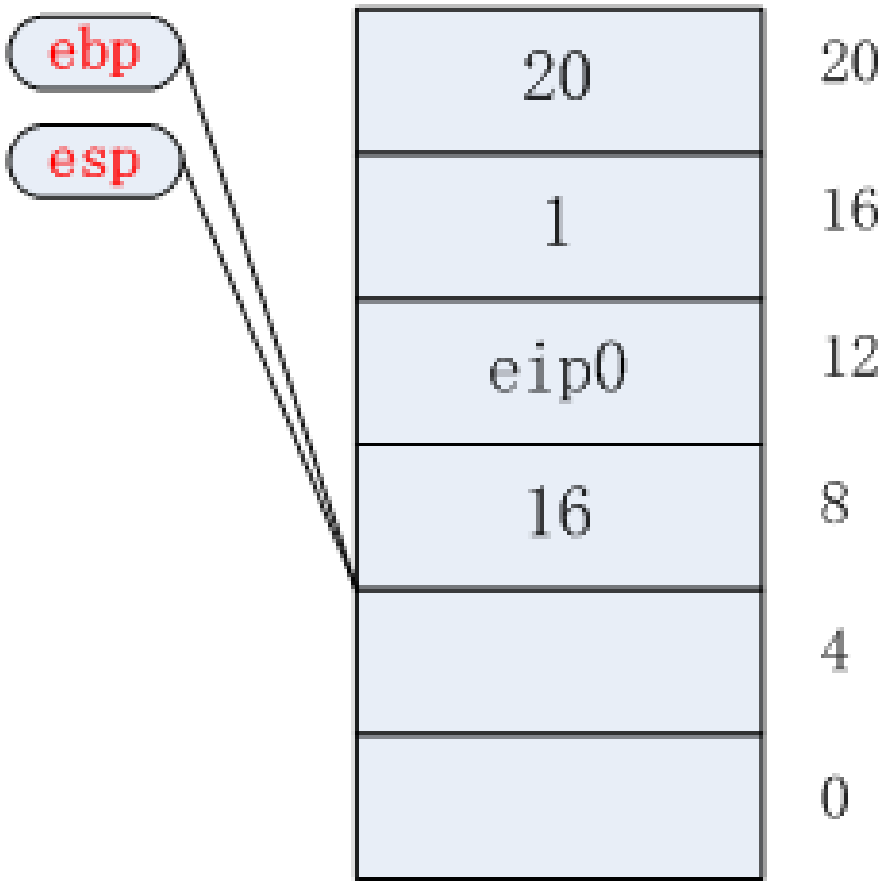
pushl %ebp

subl $4, %esp

movl %ebp, (%esp)



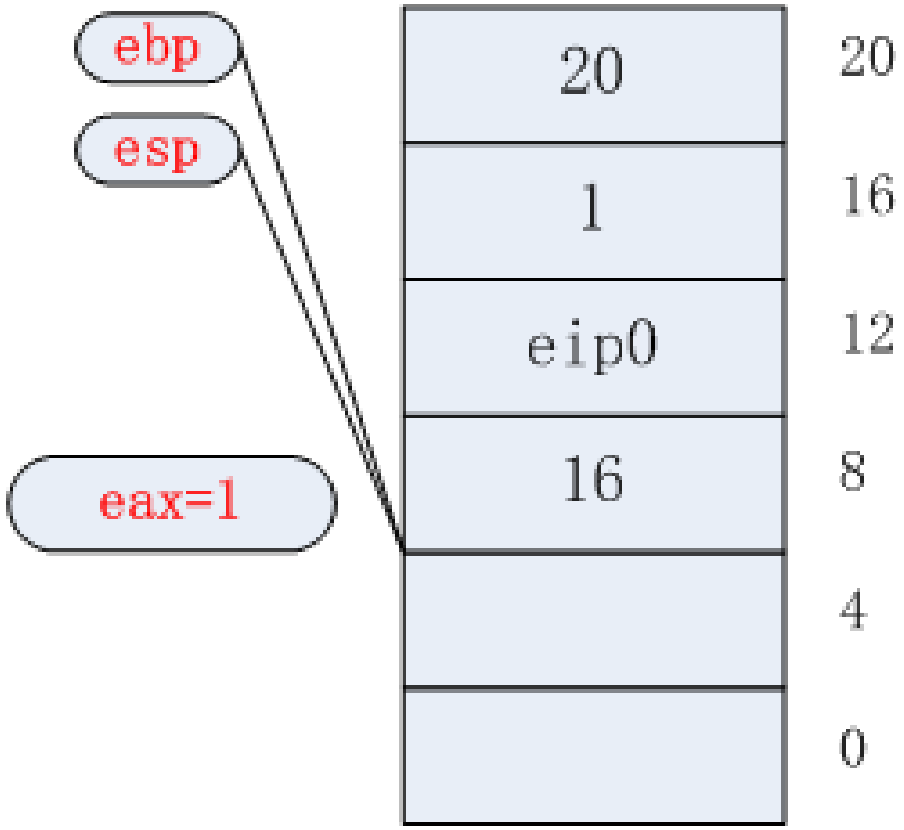
movl %esp, %ebp



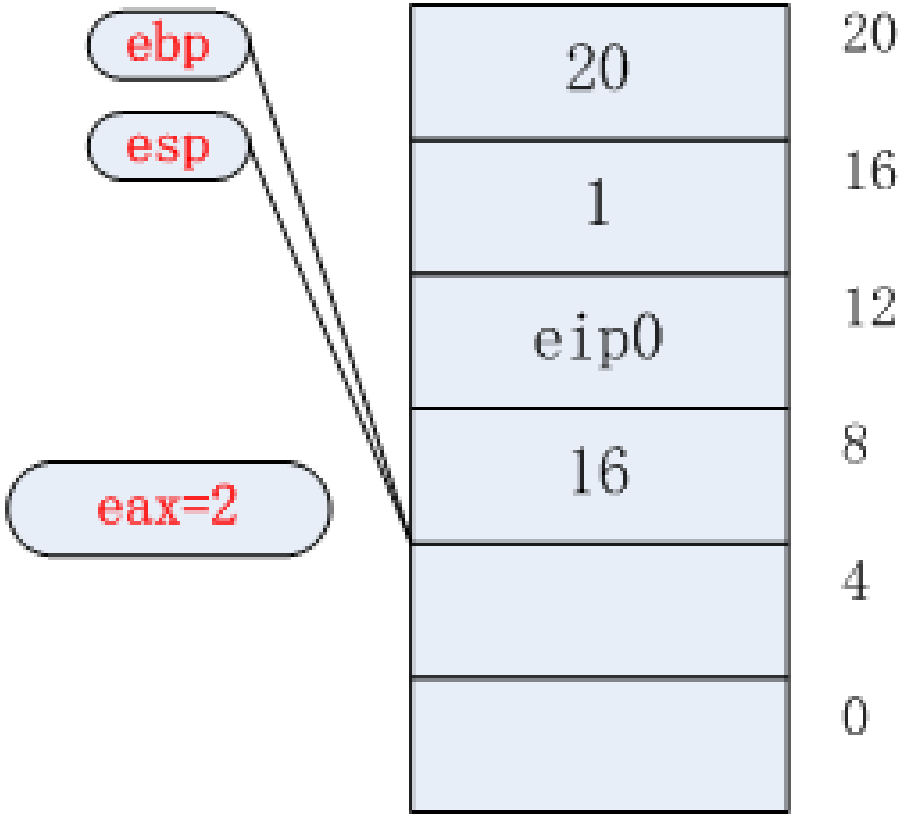

```
f:
   pushl %ebp
   movl %esp, %ebp
   movl 8(%ebp), %eax
   addl $1, %eax
   popl %ebp
   ret
main:
   pushl %ebp
   movl %esp, %ebp
   subl $4, %esp
   movl $1, (%esp)
   call f
   addl $1, %eax
   leave
   ret
```

Linux中的汇编语言

movl 8(%ebp), %eax



addl $1, %eax

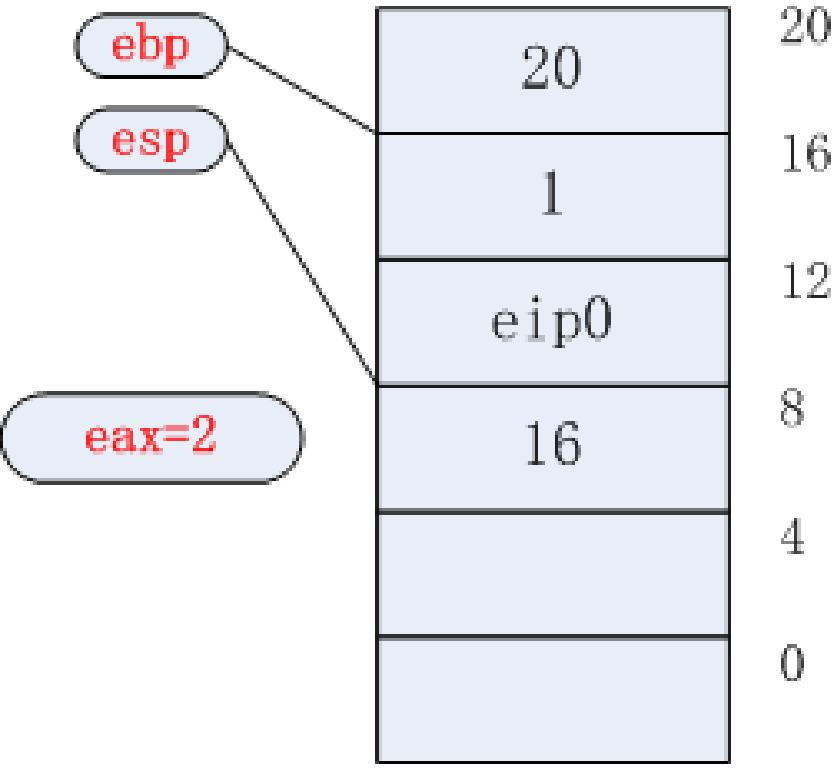



```
f:
   pushl %ebp
   movl %esp, %ebp
   movl 8(%ebp), %eax
   addl $1, %eax
   popl %ebp
   ret
main:
   pushl %ebp
   movl %esp, %ebp
   subl $4, %esp
   movl $1, (%esp)
   call f
   addl $1, %eax
   leave
   ret
```

Linux中的汇编语言

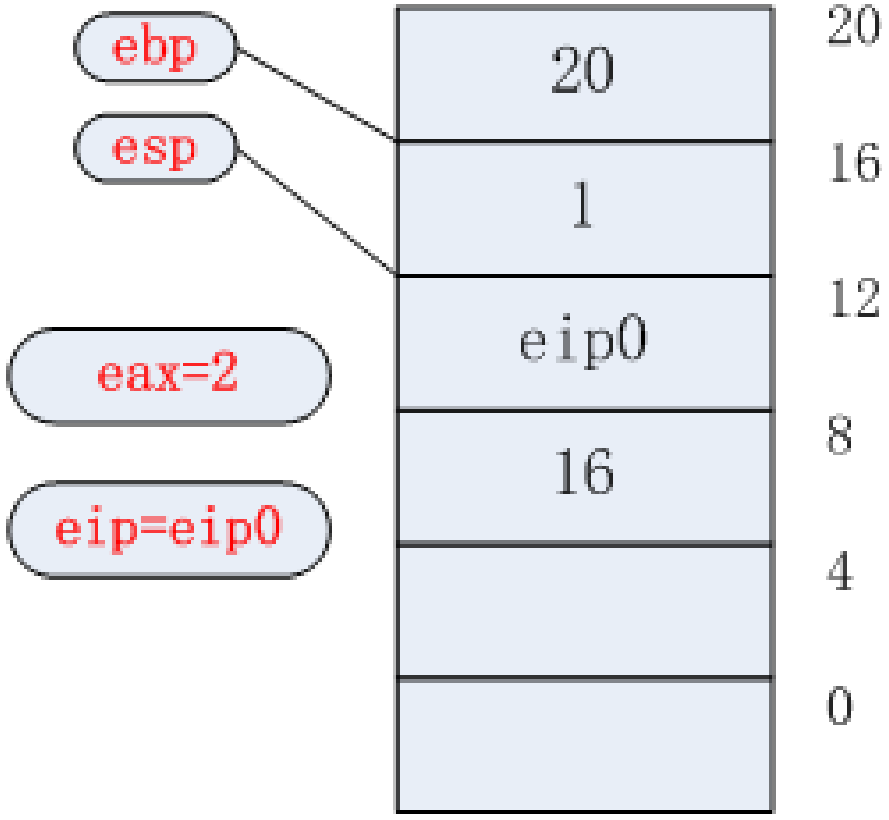
popl %ebp

movl (%esp), %ebp
addl $4, %esp



ret

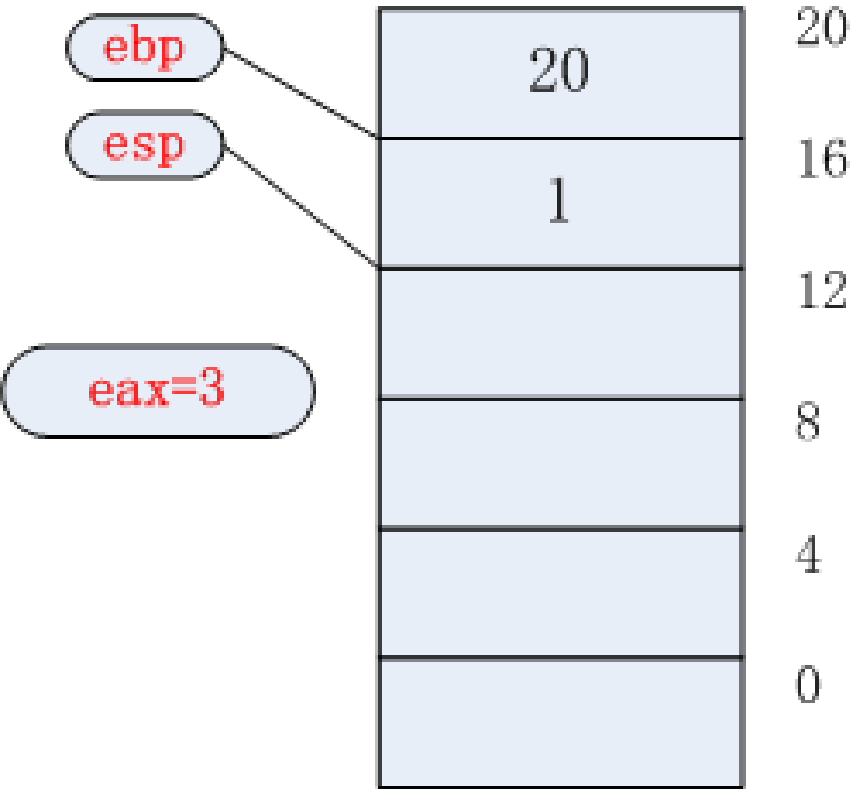
popl %eip(*)




```
f:
   pushl %ebp
   movl %esp, %ebp
   movl 8(%ebp), %eax
   addl $1, %eax
   popl %ebp
   ret
main:
   pushl %ebp
   movl %esp, %ebp
   subl $4, %esp
   movl $1, (%esp)
   call f
   addl $1, %eax
   leave
   ret
```

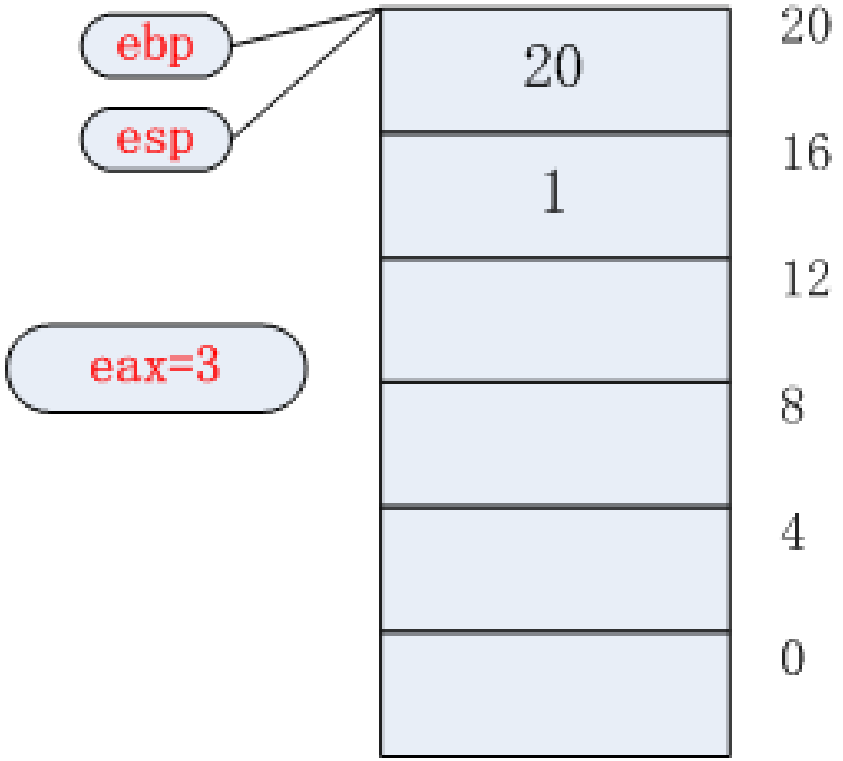
Linux中的汇编语言

addl $1, %eax



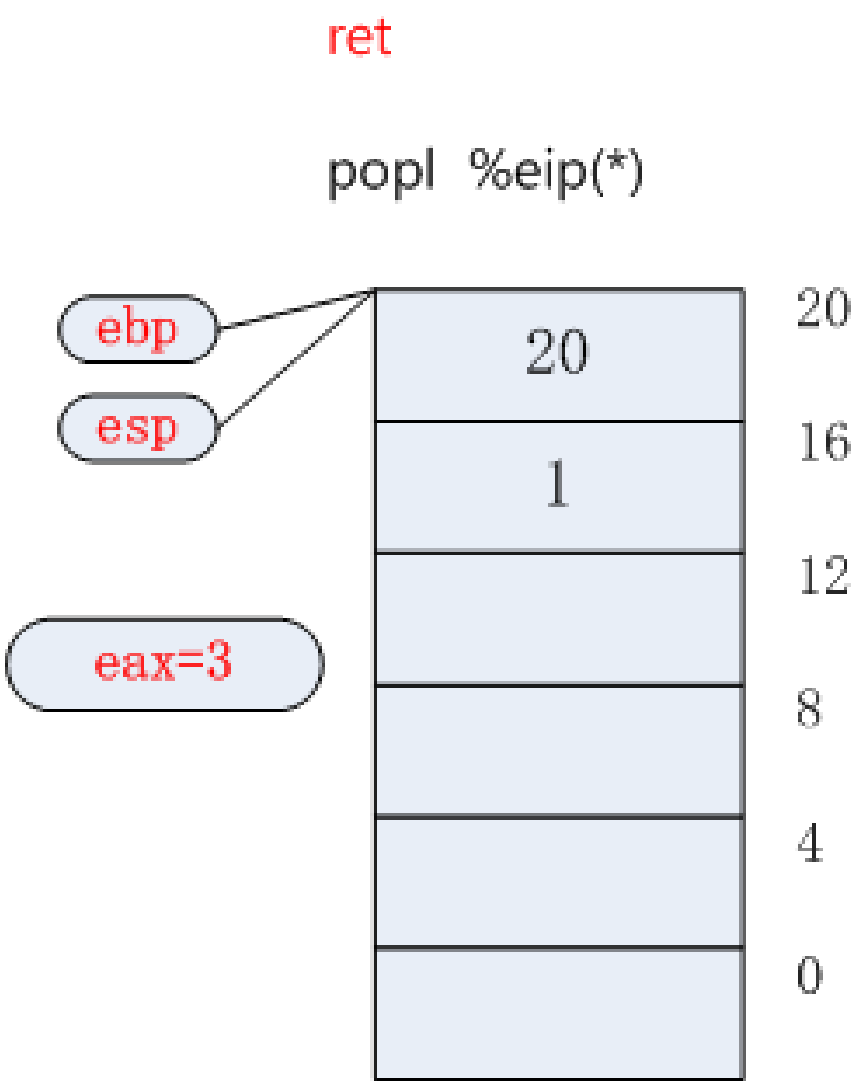
leave

movl %ebp, %esp
popl %ebp




```
f:
   pushl %ebp
   movl %esp, %ebp
   movl 8(%ebp), %eax
   addl $1, %eax
   popl %ebp
   ret
main:
   pushl %ebp
   movl %esp, %ebp
   subl $4, %esp
   movl $1, (%esp)
   call f
   addl $1, %eax
   leave
   ret
```

Linux中的汇编语言




```
f:
  pushl %ebp
  movl %esp, %ebp
  movl 8(%ebp), %eax
  addl $1, %eax
  popl %ebp
  ret
main:
  pushl %ebp
  movl %esp, %ebp
  subl $4, %esp
  movl $1, (%esp)
  call f
  addl $1, %eax
  leave
  ret
```