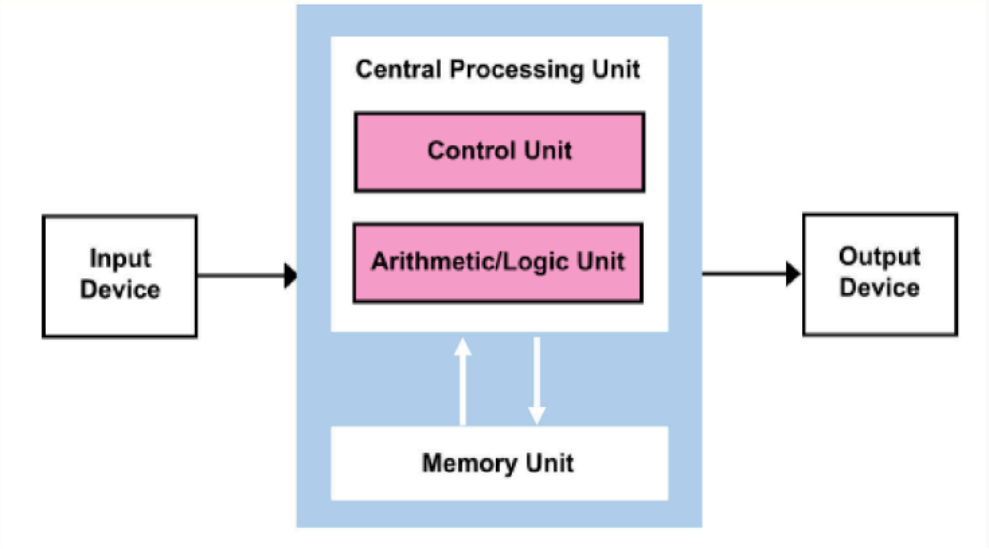
计算机系统概论 (ICS)

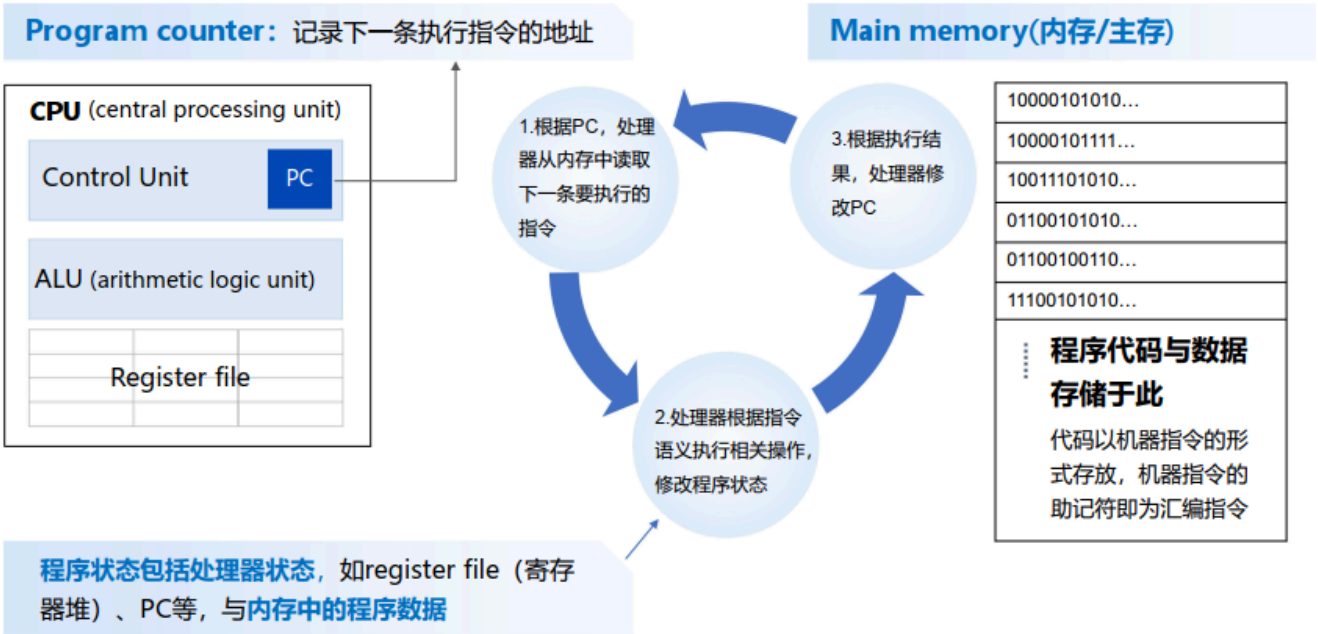
一些重要概念

1. **计算机体系结构 (Computer Architecture)**（狭义）：系统程序员所看到的计算机的属性，即计算机的逻辑结构和功能特征，包括其各个硬部件和软部件之间的相互关系。（example: IBM 360）

   冯·诺依曼架构 (Von Neumann)：① 采用二进制逻辑，② 程序存储执行， ③ 五部分组成（Input / Output Device, CPU (Central Processing Unit) = Control Unit + ALU (Arithmetic / Logic Unit) + registers, Memory Unit)

   - 以指令为中心，存算分离：**PC / Program Counter** in Control Unit 负责记录下一条执行指令的地址，操作对象为**Main Memory 主存/内存**（程序代码与数据存储于此）
   - 程序状态包括：① Control Unit状态（如register file、PC）， ② Main Memory中的程序数据





2. **计算机组成 (Computer Organization)**：计算机系统的逻辑实现，与“计算机体系结构”并称为广义概念上的计算机体系结构
3. **计算机实现 (Computer Implementation)**：计算机系统的物理实现

1 数据

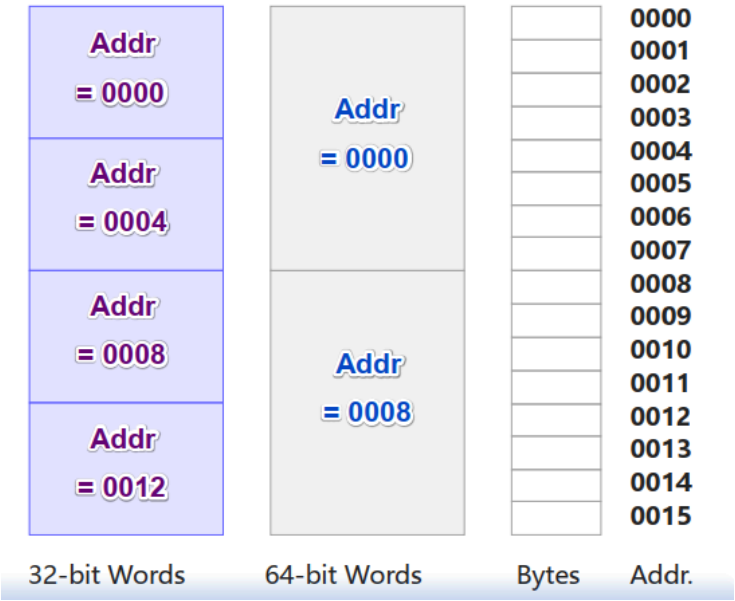
1.1 整数

预备知识

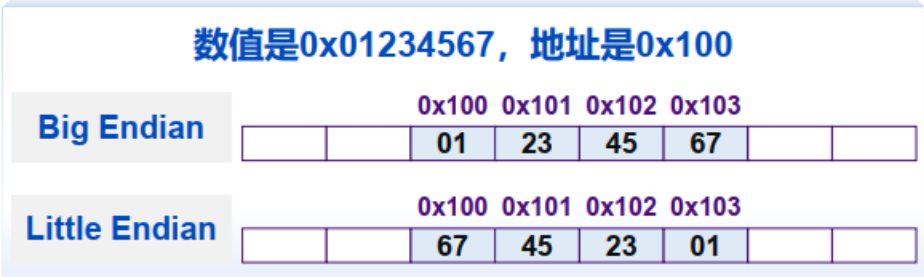
bit (比特)：1个二进制位
Byte (字节)：8个二进制位，$1Byte = 8bit$
Word (字)：2个字节，$1Word = 2Byte = 16bit$（X86架构；RISC-V架构下$1Word = 4Byte = 32bit$）

存储单位换算（kilo, Mega, Giga, Tera, Peta, Exa）
$1K = 2^{10}bit\ 1M = 2^{20}bit\ 1G = 2^{30}bit\ 1T = 2^{40}bit\ 1P = 2^{50}bit\ 1E = 2^{60}bit$

- **机器字 (Machine Word)**：计算机进行一次整数运算所能处理的二进制数据的位数，通常也指数据地址长度
  32位字长：地址的表示空间是4GB
  64位字长：地址的表示空间约为$1.8\times10^{19}bytes$，目前X86-64位机型实际支持48位宽的地址（256TB）
- **机器字在内存中的组织**：地址按照字节 (byte)来定位
  - 机器字中的第一个字节的地址
  - 相邻机器字的地址相差8



  - 字节序 (Byte Ordering)
    Big Endian：低位字节 (Least significant byte, LSB) 占据高地址
    Little Endian：高位字节 (Most significant byte, MSB) 占据高地址✓



1. 整数表示

C语言中基本数据类型的大小 (in Bytes)

C Data Type	32-bit	x86-32	x86-64	
char	1	1	1	
short	2	2	2	
int	4	4	4	
long	4	4	8	(Linux)
long long	8	8	8	
float	4	4	4	
double	8	8	8	
long double	8	10/12	10/16	
char *	4	4	8	
Or any other pointer	/	/	/	

重点记忆：

- int：4 Bytes，带符号整数$INT_MAX = 2147483647(= 2^{31} - 1)$，$INT_MIN = -2147483648(= -2^{31})$

2. 二进制编码 (w表示字长)

原码和补码

- **无符号整数（原码表示）**：$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$
- **带符号整数（补码 / Two's Complement）**：$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$ ($w-1$为符号位 / sign bit / MSB (Most Significant Bit)，表示整数的符号)

非负数：补码 = 原码

（反码是原码各位取反）

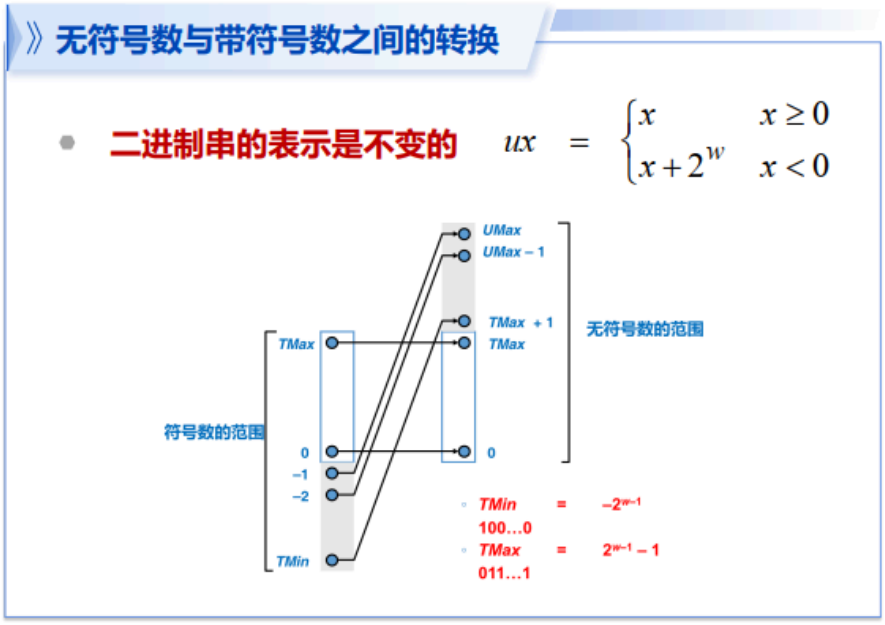
负数：补码 = 反码 + 1 = 2w + 该负数

一个数的补码表示 是 它的相反数的补码表示的按位取反加一

无符号数和带符号数

- 无符号数
  - $U_{min} = 0$ 000…0
  - $U_{max} = 2^w - 1$ 111…1
- 带符号数
  - $T_{min} = -2^{w-1}$ 100…0
  - $T_{max} = 2^{w-1} - 1$ 011…1
- Other Values
  - 负1 = 111…1

X	B2U(X)	B2T(X)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1



- **补码加法公式**：$[x]补 + [y]补 \equiv [x+y]补 (mod 2w)$
  意义：负整数用补码表示后，可以和正整数一样来处理，这样（处理器的）运算器只需要加法器
- **C语言**中的无符号数与带符号数
  - 常数 (Constants)：默认是带符号数，用`U`作为后缀表示**无符号数**
  - 无符号数与带符号数混合使用：**带符号数默认转换为无符号数**
  - 建议采用无符号数的场景：模运算、按位运算

无符号整数加法 & 补码加法

- 无符号数加法
  $$UAdd_w(u,v) = \begin{cases} u+v, u+v < 2^w \\ u+v-2^w, u+v \geq 2^w (溢出) \end{cases}$$
- 补码加法
  $$TAdd_w(u,v) = \begin{cases} u+v+2^w, u+v < TMin_w (溢出 NegOver) \\ u+v, TMin_w \leq u+v \leq TMax_w \\ u+v-2^w, Tmax_w < u+v (溢出 PosOver) \end{cases}$$

无符号整数 & 带符号整数 除以2的k次幂

- 无符号整数：采用**逻辑右移**
- 带符号整数：采用**算术右移**（高位补`1`或`0`），但是发生舍入错误（向下取整而非向零：-69.5 => -70）
  为了实现向零取整：Compute as $(x+2^k-1)/2^k$ => Incremented by 1

1.2 浮点数

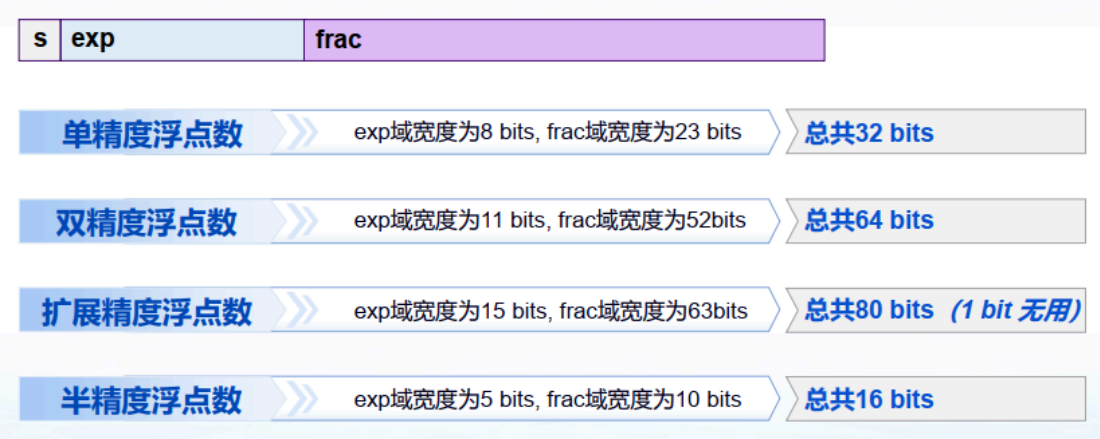
1. IEEE Floating Point (IEEE 754)

二进制表示方法：$\sum_{k=-j}^{i} b_k \cdot 2^k$

- 局限性：只能精确地表示$X/2^k$这类形式的数据
- `float`类型可以精确到小数点后6位，`double`类型可以精确到小数点后15位
- 计算方法：$b_k - 1$取小数部分不断乘2，进位保留1，不进位保留0

计算机中的浮点数二进制表示：

- 数字形式：$(-1)^s M \cdot 2^E$
  sign域：s（表示符号）
  exp域 (Exponent)：E（提示范围）
  frac域 (Mantissa)：M（尾数，位于$[1.0, 2.0)$内的小数，提示精度）



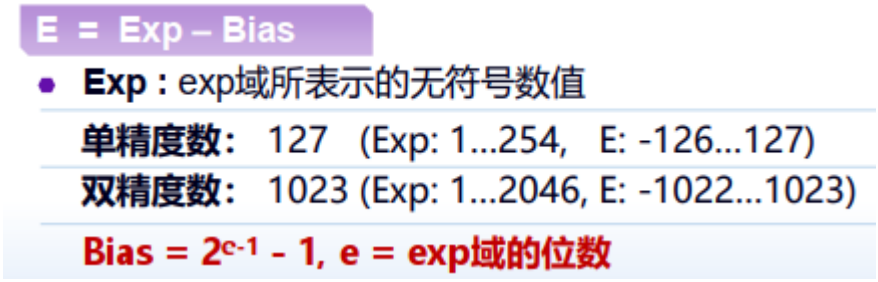
- 浮点数的类型：
  规格化浮点数（Normalized），非规格化浮点数（Denormalized），一些特殊值

> 先计算出$Bias = 2^{(e-1)} - 1$，其中e为exp域的位数
> 区分Normalized和Denormalized，算出$E = Exp - Bias$或$E - 1 - Bias$或$E = n/a$

规格化浮点数（Normalized）

满足条件：$exp \neq 000..0, exp \neq 111..1$

1. 真实的阶码值需要减去一个偏置（biased）量



2. frac域的第一位隐含为1（因此可以省去）

非规格化浮点数（Denormalized）

满足条件：$exp = 000..0$，$M = 0.xxx...x$ => $E = 1 - Bias, M = 0.xxx...x$

具体实例：

- $exp = 000..0, frac = 000..0$
  表示0（注意有+0和-0）
- $exp = 000..0, frac \neq 000..0$
  表示“非常接近”于0的浮点数
  会逐步丧失精度，称为“Gradual underflow”

一些特殊值

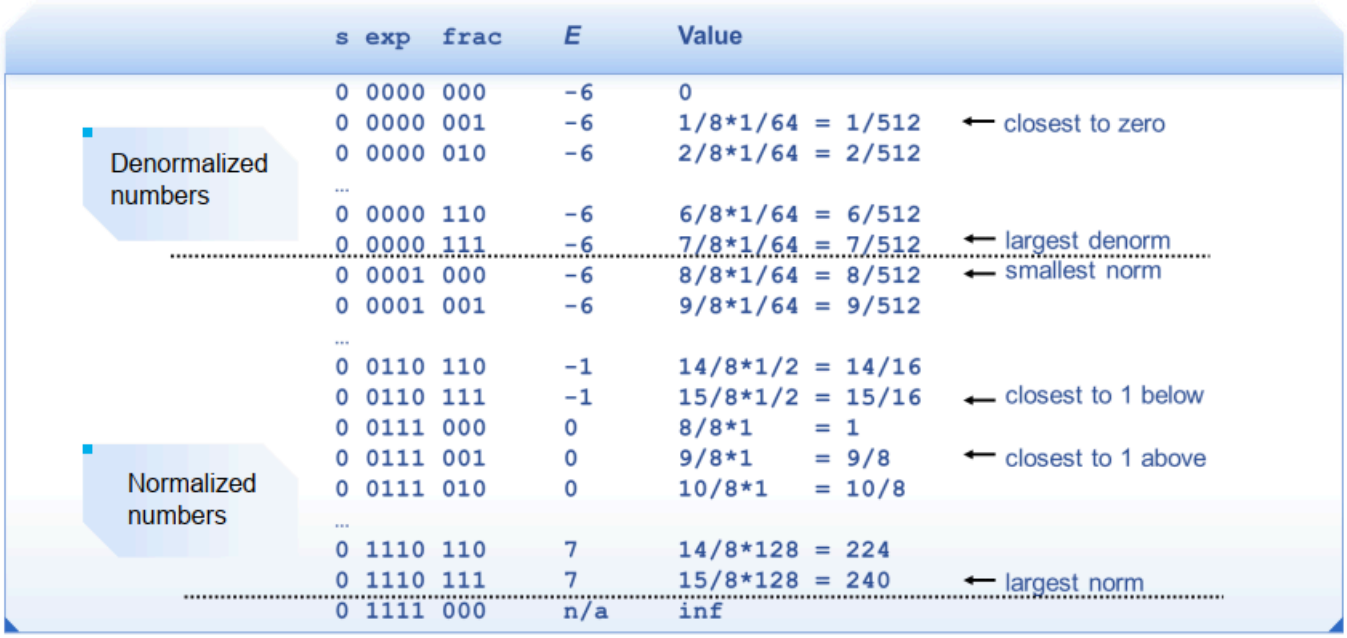
满足条件：$exp = 111..1$ => $E = n/a$

具体实例：

- $exp = 111...1, frac = 000...0$
  表示无穷
  可用于表示数值的溢出
  有正无穷与负无穷（EG/ 1.0/0.0 = +∞，EG/ -1.0/0.0 = -∞）
- $exp = 111...1, frac \neq 000...0$
  Not-a-Number (NaN)
  EG/ sqrt(-1), ∞, -∞, ∞*0

一种“小”浮点数实例

8位浮点数表示：exp域宽度为4 bits，frac域宽度为3 bits，bias=7

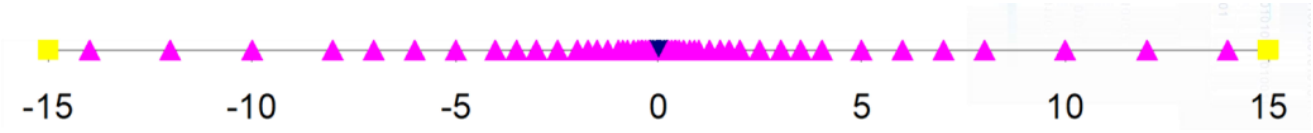


推广到单精度与双精度浮点数：

Description	exp	frac	Numeric Value
Zero	00…00	00…00	0.0
Smallest Pos. Denorm. Single ≈ 1.4×10^{-45} Double ≈ 4.9×10^{-324}	00…00	00…01	$2^{-\{23,52\}} \times 2^{-\{126,1022\}}$
Largest Denormalized Single ≈ 1.18×10^{-38} Double ≈ 2.2×10^{-308}	00…00	11…11	$(1.0 - \varepsilon) \times 2^{-\{126,1022\}}$
Smallest Pos. Normalized Just larger than largest denormalized	00…01	00…00	$1.0 \times 2^{-\{126,1022\}}$
One	01…11	00…00	1.0
Largest Normalized Single ≈ 3.4×10^{38} Double ≈ 1.8×10^{308}	11…10	11…11	$(2.0 - \varepsilon) \times 2^{\{127,1023\}}$

浮点数的编码特性

- （几乎）可以直接使用无符号整数的比较方式
  【反例】：必须先比较符号位
  考虑+0，-0的特例
  考虑NaN的问题（不考虑符号位的话，NaN比其他值都大，实际比较结果如何？——undefined behavior）
  其他情况都可以直接使用无符号整数的比较方式：规格化 vs 非规格化，规格化 vs 无穷



2. Rounding（舍入）

各种舍入模式：Zero, Round down, Round up, Nearest Even

向偶数舍入（Round-To-Even）
也称“向最接近值的舍入”，其他方式会产生统计误差（statistically biased）
关键的设计决策是确定两个可能结果的中间数值的舍入

确保舍入后的最低有效数字是偶数
E.g., round to nearest hundredth

1.2349999	1.23	(Less than half way)
1.2350001	1.24	(Greater than half way)
1.2350000	1.24	(Half way—round up)
1.2450000	1.24	(Half way—round down)

“Even”意味着**最低有效数字需为0**，而最低有效数字右侧的位串为100...

舍入到小数点后2位

Value	Binary	Rounded	Action	Rounded-Value
2 3/32	10.00011_2	10.00_2	(<1/2—down)	2
2 3/16	10.00110_2	10.01_2	(>1/2—up)	2 1/4
2 7/8	10.11100_2	11.00_2	(1/2—up)	3
2 5/8	10.10100_2	10.10_2	(1/2—down)	2 1/2

实例：将无符号数转换为8位浮点数并舍入

① 规格化 => ② 舍入 => ③ 调整/Postnormalize（舍入操作可能引起溢出）

Value	Binary	Fraction	Exponent
128	10000000	1.0000000	7
15	00001111	1.1110000	3
17	00010001	1.0001000	4
19	00010011	1.0011000	4
138	10001010	1.0001010	7
63	00111111	1.1111100	5

Value	Fraction	Incr?	Rounded
128	1.0000000	N	1.000
15	1.1110000	N	1.111
17	1.0001000	N	1.000
19	1.0011000	Y	1.010
138	1.0001010	Y	1.001
63	1.1111100	Y	10.000

Value	Rounded	E	Adjusted	Result
128	1.000	7		128
15	1.111	3		15
17	1.000	4		16
19	1.010	4		20
138	1.001	7		144
63	10.000	5	1.000/6	64

3. C语言中的浮点数（类型转换）

int（32位宽），float（1+8+23=32位宽），double（1+11+52=64位宽）

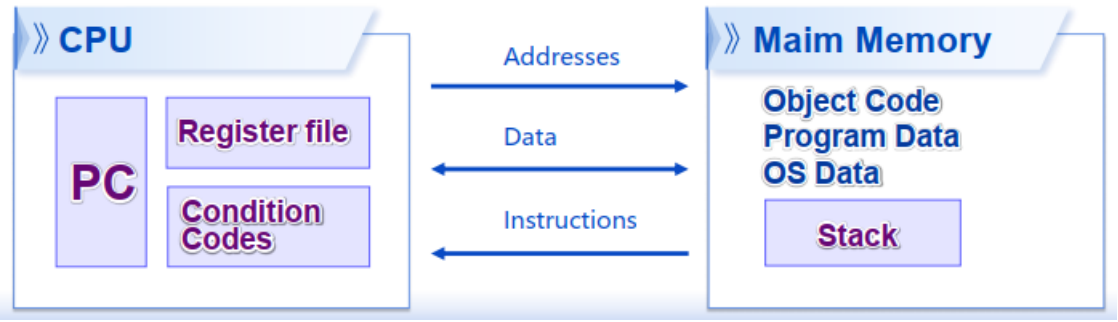
- double或float => int
  尾数部分截断；不足位补零
  如果溢出（考虑`double/float`可以表示的最大值要比`int`能表示的最大值要大，此时发生溢出）或者浮点数是NaN，则转换结果没有定义：通常置为T_{min}或T_{max}
- int => double
  能够精确转换
- int => float
  不会溢出，但是可能被舍入（因为随着`float`数值变大它会变“稀疏”）

2 代码

2.1 汇编语法

1. 汇编体系结构

（冯诺伊曼架构 / Control Flow）



- **指令寄存器 / Program Counter / PC**
  存储下一条指令的地址
  x86-64架构下的名字为RIP
  (8 bytes / 4 bytes(RISC-V))
- **寄存器与寄存器堆 / Register File**
  在处理器内部的以名字来访问的快速存储单元
  (≈16 × 8bytes)
- **条件码 / Condition Codes**
  用于存储最近执行指令的结果状态信息
  用于条件指令的判断执行
  (≈ bytes)
- **内存 / 存储器 / Main Memory**
  以字节编码的连续存储空间
  存放程序代码 (object code)、数据 (program data)、运行栈 (stack)以及操作系统数据 (OS data)
  (≈ GB)
  （简化理解）内存存放的三样东西：程序代码、运行栈、全局变量

寄存器 vs 内存:

项　目	寄存器	内　存
位　置	CPU内部	CPU外部
访问速度	快	慢
容　量	小	大
成　本	高	低
表示方式	用名字表示	用地址表示
地　址	没有	可用多种方式形成 (多种内存寻址方式)

2. 汇编代码

(web)Compiler Explorer

生成汇编代码的命令行：`gcc -Og -S sum.c #-fno-stack-protector`
其中：`-S`生成文件`sum.s`，`-c`的话生成`sum.o`（即对象文件，内含机器指令）

```
x

g++ -E main.cpp -o main.ii //Preprocess：预处理命令（处理#include和宏）

g++ -S main.ii -o main.s //Compile：编译为汇编代码

g++ -c main.s -o main.o //Assemble：编译成指令并保存在对象文件

//这三个合称“编译期”

g++ main.o -o main //Link：将对象文件和标准库链接为二进制可执行文件，“链接期”


//其他编译选项：

g++ test.cpp --std=c++11 -fno-elide-constructors -o test

    //禁止编译器进行返回值优化

g++ test.cpp -g -o test -fsanitize=undefined,address

    //查看非法访问、溢出、内存外泄的强大命令


//其它：

gcc编译C代码，g++编译C++代码
```

反汇编命令：`objdump -S/-d a.o`，`objdump -S/-d a`

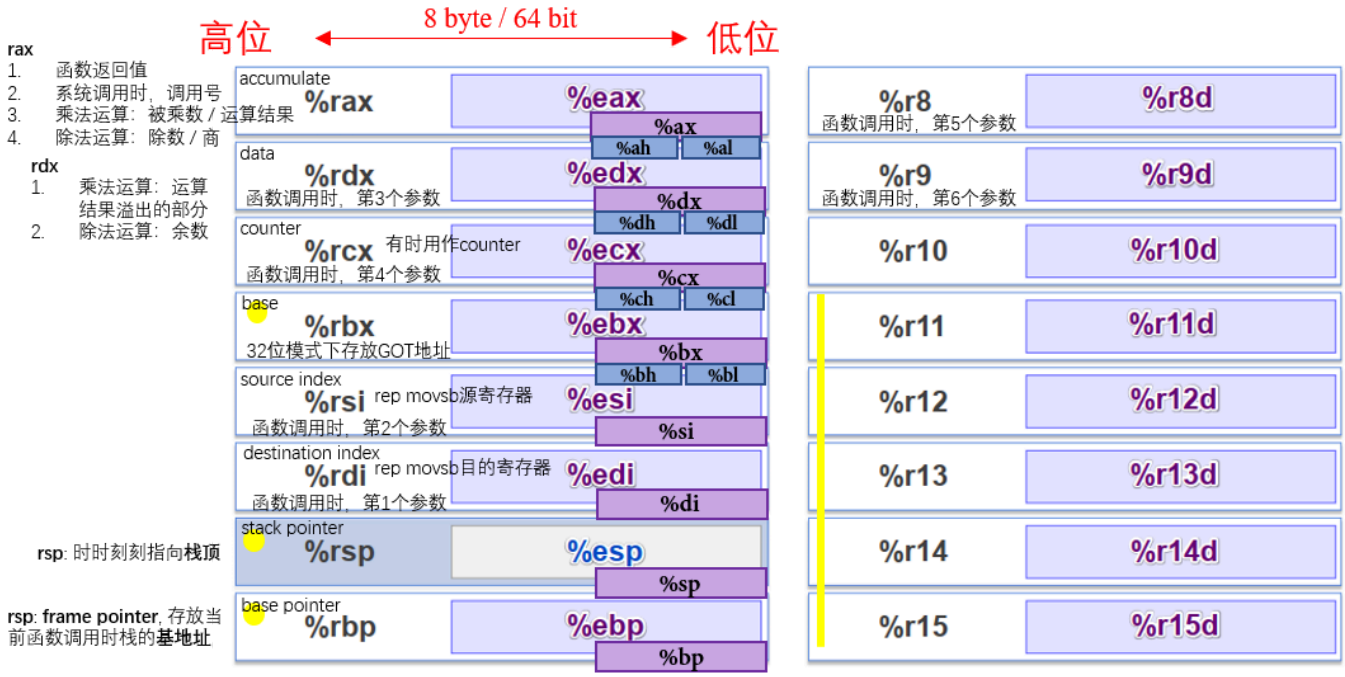
- 常见的C/C++编译器：GCC, Clang, MSVC, ICC
- *此部分见讲义*`assembly_tutorial`

2.0 操作类型

- 对寄存器数据或者内存数据进行算术/逻辑操作
- 内存与寄存器之间、或者寄存器/寄存器之间传递数据
- 程序执行顺序的转移（transfer control）

2.0 通用寄存器

通用寄存器 / General-Purpose Register



被调用者保存（可以直接读写），否则要存放在内存里/等函数调用结束再读取出来

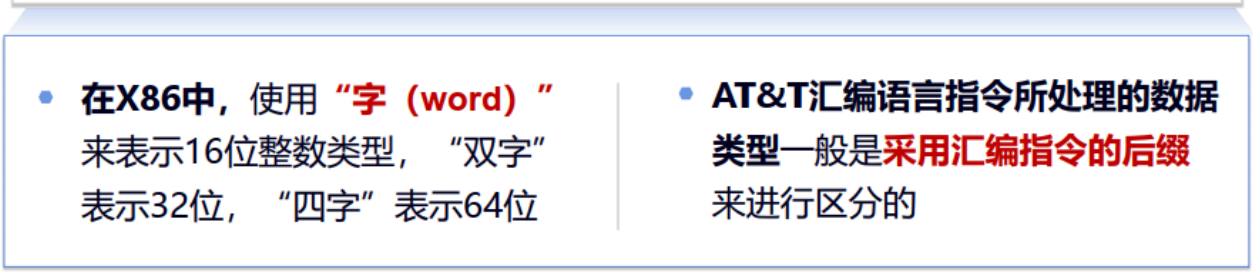
*当参数少于7个时，参数从左到右放入寄存器：rdi, rsi, rdx, rcx, r8, r9。**当参数为7个及以上时**，前6个传送方式不变，但后面的依次“右向左”放入栈中*

***被调用者保存**：汇编程序通常而言，在调用函数前，如果这些寄存器里存放了东西，需要把它们存到内存（栈）里*

2.1 基本概念（AT&T）

- 数据类型
  - 整型数据：数据宽度为1,2,4,8 bytes
  - 浮点数据：数据宽度为4,8,10 bytes
  - 无复杂数据类型：没有结构或者数组之类的
- 数据格式
  - 前缀（AT&T）
    寄存器前被冠以`%`
    立即数前被冠以`$`
    十六进制数前被冠以`0x`
  - 后缀
    分别对应的寄存器名称为：`al`/`ab`，`ax`，`eax`，`rax`

Suffix	Name	Size
B	BYTE	1byte (8bits)
W	WORD	2byte (16bits)
L	LONG	4byte (32bits)
Q	QUADWORD	8byte (64bits)



[EG]：`movl`表示操作32bit寄存器，`movb`表示操作8bit寄存器

- 操作数方向
  源操作在前，目的操作数在后
  [EG]：`movl $0x42, %ah`：把立即数`$0x42`放到寄存器`%ah`中
  对于**内存单元操作数**来说，在AT&T中是把寄存器用`()`括起来
  [EG]：`movl %ebx, 8(%si)`：将`%ebx`寄存器里的值放到内存地址是`8(%si)`的内存单元上，相当于intel汇编中的`[si + 8]`

2.2 常用指令（CSDN）

1. 运算指令

```
xxxxxxxxxx

addl %edx, %eax      //edx+eax，结果放到eax

add $5, %r10     //5+r10，结果放到r10

inc %r10     //r10加1

mul %r10     //将rax乘以r10，将结果放到rax中，溢出部分放到rdx

div %r10     //rax除以r10，商放到rax，余数放到rdx
```

2. 拷贝指令

```
xxxxxxxxxx

mov %r10,%r11    //将r10寄存器的值赋值给r11 ；

mov $99,%r10     //将立即数99赋值给r10寄存器；

mov %r10,(%r11) //将r10的值拷贝到r11寄存器中的数值指向的内存地址上；

mov (%r10),%r11 //将r10中数值指向的内存地址上的内容拷贝到r11；

push %r10        //将r10的值放到栈上 ；
```

```
pop %r10          //将栈顶的值pop到r10寄存器上。
```

3. 流程控制指令

```
xxxxxxxxxx

cmp %r10,%r11 //比较r10和r11，根据比较结果来设置CPU的状态寄存器，从而影响后面的jump语句;

cmp $99,%r11  //比较99和r11，根据比较结果来设置CPU的状态寄存器，从而影响后面的jump语句;

jmp label     //跳转到label

je label      //如果相等，跳转到label

jne label     //如果不相等，跳转到label

jl label      //如果小于，跳转到label

jg label      //如果大于，跳转到label

call label    //调用函数

ret           //从函数调用返回

syscall       //系统调用（32位模式下，使用"int $0x80" 软中断）
```

4. 汇编指示符（assembler directive）
   编译出的汇编文件的开始处，有一段由句号`.`开头的指令，是**AT&T 汇编指示符。**这里命令名的其余是字母，通常使用小写。这些不会被翻译成机器指令，而是给汇编器一些特殊指示，称为**汇编指示（Assembler Directive）**或**伪操作（Pseudo-operation）**
   - `.byte`表达式
     可不带参数或者带多个表达式参数，表达式之间由逗号分隔，每个表达式参数都被会变成下一个**字节**
   - `.word`表达式
     这个表达式表示任意一节中的一个或多个表达式，表达式占一个**字（两个字节）**
   - `.file`字符（string）
     通告编译器我们准备开启一个新的逻辑文件，string是新文件名。
   - `.text`小节（subsection）
     通知as编译器把后续语句汇编到编号为subsection的正文子段的末尾，subsection是一个纯粹的表达式
   - `.code16`：告诉编译器生成16位的指令
   - `.globl`：定义全局符号
   - `.fill repeat, size, value`

2.3 栈操作

每个汇编代码块的开始都有以下一样的汇编指令：

```
xxxxxxxxxx

pushl %ebp       //将调用者的栈底部地址保存起来

movl %esp, %ebp      //将调用者的栈顶部地址设置为本栈帧的底部
```

```
subq $18, %rsp      //为变量申请内存空间，开辟了24字节
```

可以简单的理解为：保存上一个函数的栈底指针，然后重新开始一个新的栈

3. 指令操作（上课PPT）

3.1 数据传送指令（mov）

mov指令集：`movb`，`movw`，`movl`，`movq`，`movslq`（将一个双字节符`l`拓展移动到四字地址`q`中），`movzwl`（读取一个字word然后将其拓展到双字节`l`）

> MOVS类指令：将较小的源复制到较大的目的时使用，通过**符号扩展**来填充
> MOVZ类指令：将较小的源复制到较大的目的时使用，通过剩余字节**补0**来填充

基本语法：`movq Source, Dest`

操作数类型

- **Imm**: 立即数 / 整型常数（[EG]: `$0x400`, `$-533`）：$prefix$=`$`
- **Reg**: 寄存器（16个通用处理器之一）：$prefix$=`%`
- **Mem**: 内存（8个连续字节，起始地址由相应寄存器内容指定）（[EG]: `(%rax)`）：$prefix$=`%`

	Source	Dest	Src,Dest	类似的C语言表示
movl	Imm	Reg	movq $0x4,%rax	temp = 0x4;
		Mem	movq $-147,(%rax)	*p = -147;
	Reg	Reg	movq %rax,%rdx	temp2 = temp1;
		Mem	movq %rax,(%rdx)	*p = temp;
	Mem	Reg	movq (%rax),%rdx	temp = *p;

但是不能两个操作数都为内存数据

3.2 寻址模式

1. **间接寻址**

   格式：(R) 寄存器R指定内存地址
   内存地址：$Mem[Reg[R]]$
   指令：`movq (%rcx), %rax`

2. **基址+偏移量寻址**

   格式：$D(R)$ 寄存器R指定内存起始地址，常数D给出偏移量
   内存地址：$Mem[Reg[R]+D]$
   指令：`movq 8(%rbp), %rdx`

3. **变址模式**

格式：$D(Rb, Ri, S)$
D：常量（地址偏移量）
S：比例因子（1, 2, 4 or 8）
Rb：基址寄存器，16个通用寄存器之一
Ri：索引寄存器，%rsp不作为索引寄存器

内存地址：$Mem[Reg[Rb] + S * Reg[Ri] + D]$
指令（三种常见变形）：
(Rb, Ri): $Mem[Reg[Rb] + Reg[Ri]]$
$(D(Rb, Ri))$: $Mem[Reg[Rb] + Reg[Ri] + D]$
(Rb, Ri, S): $Mem[Reg[Rb] + S * Reg[Ri]]$

[EG: 注意 常数不用加`$`，`%rdx == 0xf000, %rcx == 0x100]`

Expression	Address Computation	Address
0x8(%rdx)	0xf000 + 0x8	0xf008
(%rdx,%rcx)	0xf000 + 0x100	0xf100
(%rdx,%rcx,4)	0xf000 + 4*0x100	0xf400
0x80(,%rdx,2)	2*0xf000 + 0x80	0x1e080

X86-64：默认指针参数类型为64位，（如果使用`int`数据类型，被操作的数据仍是32位）

```
void swap(long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
    movq    (%rdi), %rax
    movq    (%rsi), %rdx
    movq    %rdx, (%rdi)
    movq    %rax, (%rsi)
    ret
```

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
    movl    (%rdi), %edx
    movl    (%rsi), %eax
    movl    %eax, (%rdi)
    movl    %edx, (%rsi)
    retq
```

3.3 地址计算指令

基本语法：`leaq Src, Dest`
其中，Src是**内存地址计算表达式**，计算出来的地址赋给$Dest$
使用实例：地址计算（无需访存），[EG]: `translation of p = &x[i];`
进行$x + k \times y$这一类型的整数计算，其中$k = 1, 2, 4, 8$
[EG]: `long m12(long x) { return x * 12; }` ==>

```
leaq(%rdi, %rdi, 2), %rax    # t <= x+x*2
salq $2, %rax    # return t<<2
```

3.4 整数计算指令

Format		Computation	
双操作数指令			
addq	Src,Dest	Dest = Dest + Src	
subq	Src,Dest	Dest = Dest – Src	
imulq	Src,Dest	Dest = Dest * Src	Multiply （取低64位）
salq	Src,Dest	Dest = Dest << Src	与shll等价
sarq	Src,Dest	Dest = Dest >> Src	算术右移
shrq	Src,Dest	Dest = Dest >> Src	逻辑右移
xorq	Src,Dest	Dest = Dest ^ Src	
andq	Src,Dest	Dest = Dest & Src	
orq	Src,Dest	Dest = Dest | Src	

Format	Computation
单操作数指令	
incq Dest	Dest = Dest + 1
decq Dest	Dest = Dest - 1
negq Dest	Dest = - Dest
notq Dest	Dest = ~ Dest

- [EG/一个编译器优化的例子：省略了`t3`]

```
long arith
(long x, long y, long z)
{
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
arith:
  leaq    (%rdi,%rsi), %rax        # t1
  addq    %rdx, %rax               # t2
  leaq    (%rsi,%rsi,2), %rdx      # t4
  salq    $4, %rdx
  leaq    4(%rdi,%rdx), %rcx       # t4+x+4
  imulq   %rcx, %rax               # rval
  ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	t1, t2, rval
%rdx	t4
%rcx	t5

- [EG/又一个编译器优化的例子]

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

$2^{13} = 8192, 2^{13} – 7 = 8185$

```
logical:
  movl   %edi, %eax
  xorl   %esi, %eax
  sarl   $17, %eax
  andl   $8185, %eax
  ret
```

Register	Use(s)
%edi	Argument **x**
%esi	Argument **y**
%eax	**t1, t2, rval**

4. X86汇编格式

At&T Format 和 **Intel / Microsoft Format**：别名关系

GAS / GNU Assembler：汇编器

X86-64指令的特点

- 支持**多种类型**的指令操作数
  立即数，寄存器，内存数据
- 支持**多种内存地址**计算模式
  $Rb + S * R_i + D$，也可用于整数计算（如`leal`/`leaq`指令）
- 算逻指令**可以以内存数据**为操作数
- 变长指令
  from 1 to 15 bytes

2.2 指令结构

1. 控制流（control flow）

处理器：从启动到关闭，CPU读取和执行（解释）一个指令序列，一次一条——这个执行顺序被称为**CPU的控制流**
当前执行程序的信息：
临时数据：%rax...
栈顶地址：%rsp...
当前指令地址/下一条：%rip...（X86架构下的PC）
条件码：CF, ZF, SF, OF
改变控制流的两种机制：**Jump and branch**, **Call and return**

- **条件码**（由算术指令隐式设置）

  CF: Carry Flag（进位）
  ZF: Zero Flag
  SF: Sign Flag (negative if true)
  OF: Overflow Flag

2. 算术指令

- add

addq Src, Dest

> CF 进位标志：可用于检测无符号整数运算的溢出
> SF set if t < 0
> ZF set if t == 0
> OF set if 补码运算溢出（即带符号整数运算）

leaq指令不设置条件码

- cmp

cmpq Src2, Src1 (≈计算Src1-Src2)

> CF set if carry out from most significant bit：可用于无符号数的比较
> ZF set if a == b
> SF set if (a-b) < 0
> OF set if two's complement overflow：补码计算溢出

- Test

Testq Src2, Src1

> 计算Src1 & Src2并设置相应的条件码，但是不改变目的操作数
>
> ZF set when a&b == 0
> SF set when a&b < 0

CF = 0, OF = 0

- SetX

SetX指令
读取当前的条件码（或者某些条件码的组合），并存入目的**字节寄存器**（余下的7个字节不会被修改）

> | SetX | Condition | Description |
> |---|---|---|
> | sete | ZF | Equal / Zero |
> | setne | ~ZF | Not Equal / Not Zero |
> | sets | SF | Negative |
> | setns | ~SF | Nonnegative |
> | setg | ~(SF^OF)&~ZF | Greater (Signed) |
> | setge | ~(SF^OF) | Greater or Equal (Signed) |
> | setl | (SF^OF) | Less (Signed) |
> | setle | (SF^OF)\|ZF | Less or Equal (Signed) |
> | seta | ~CF&~ZF | Above (unsigned) |
> | setb | CF | Below (unsigned) |
>
> 》 **如果补码计算溢出，则OF置1** (a>=0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)

[EG/通常使用`movzbl`指令对目的寄存器进行“0”扩展]

```
int gt (long x, long y)
{
  return x > y;
}
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	Return value

```
cmpq    %rsi, %rdi    # Compare x:y
setg    %al           # Set when >
movzbl  %al, %eax     # Zero rest of %rax
ret
```

"64-bit operands generate a 64-bit result in the destination general-purpose register. 32-bit operands generate a 32-bit result, zero-extended to a 64-bit result in the destination general-purpose register."

摘自"Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture"

3. 跳转指令

- jX

依赖当前的条件码选择下一条执行语句（是否顺序执行）

jX	Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	~ZF	Not Equal / Not Zero
js	SF	Negative
jns	~SF	Nonnegative
jg	~(SF^OF)&~ZF	Greater (Signed)
jge	~(SF^OF)	Greater or Equal (Signed)
jl	(SF^OF)	Less (Signed)
jle	(SF^OF)|ZF	Less or Equal (Signed)
ja	~CF&~ZF	Above (unsigned)
jb	CF	Below (unsigned)

[EG/编译指令：`gcc -Og -S -fno-if-conversion control.c`]

Generation
gcc –Og -S –fno-if-conversion control.c

```
long absdiff
  (long x, long y)
{
  long result;
  if (x > y)
    result = x-y;
  else
    result = y-x;
  return result;
}
```

```
absdiff:
  cmpq   %rsi, %rdi   # x-y
  jle    .L4
  movq   %rdi, %rax
  subq   %rsi, %rax
  ret
.L4:                  # x <= y
  movq   %rsi, %rax
  subq   %rdi, %rax
  ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	Return value

- Goto

C allows goto statement.
Jump to position designated by label.

```
xxxxxxxxxx

long absdiff_j(long x, long y) {

    long result;

    int ntest = (x <= y);

    if (ntest) goto Else;

    result = x - y;

    goto Done;

  Else:

    result = y - x;

  Done:

    return result;

}
```

4. 条件移动

语义：$if\ (Test)\ Dest <= Src$

> 条件跳转指令对于现代流水线处理器的执行效率有很大的负面影响
> **条件移动指令**可以避免这一现象

[EG]

```
long absdiff (long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	Return value

```
absdiff:
    movq    %rdi, %rax      # x
    subq    %rsi, %rax      # result = x-y
    movq    %rsi, %rdx
    subq    %rdi, %rdx      # eval = y-x
    cmpq    %rsi, %rdi      # x:y
    cmovle  %rdx, %rax      # if <=, result = eval
    ret
```

> **条件转移指令的局限性**

> 1. Then-Expr或Else-Expr表达式有“副作用”
> 2. Then-Expr或Else-Expr表达式的计算量较大

微体系结构背景

处理器流水线（五级流水示例）：

> Instruction Fetch (IF)
> Read Registers (RD)
> Arithmetic Operation (ALU)
> Memory Access (MEM)
> Write Back (WB)

现代的通用处理器，支持深度流水线以及多发射结构
条件跳转指令往往会引起一定的性能损失，因此需要尽量消除

5. 循环/分支指令

- `do while` & `while`
- `for` 循环
- `switch` 语句：跳转表

> **Direct**: `jmp .L8` / jump target is denoted by label .L8
> **Indirect**: `jmp *.L4(, %rdi, 8)`
> start of jump table: .L4
> must scale by factor of 8 (addresses are by 8 bytes)
> fetch target from effective Address .L4 + x * 8 (only for $0 \leq x \leq 6$)
> 间接跳转解决方案不一样，需要linker

```
long switch_eg(long x, long y, long z)
{
        long w = 1;
        switch(x) {
          . . .
        }
        return w;
}
```

Setup:

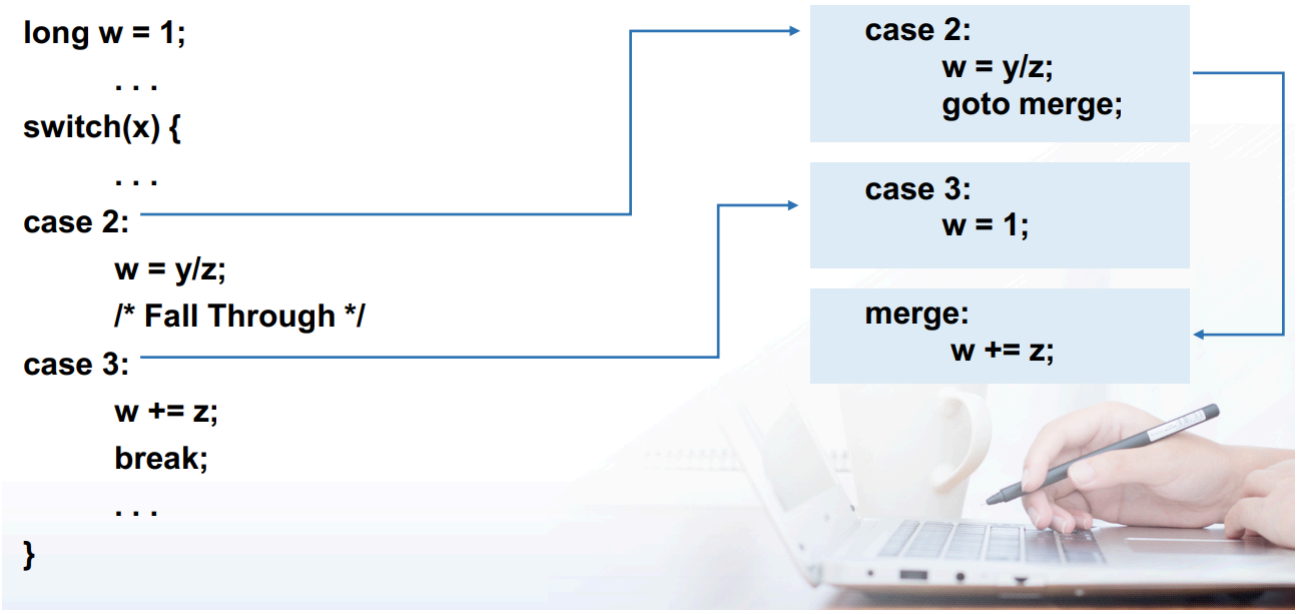
```
switch_eg:
   movq     %rdx, %rcx
   cmpq     $6, %rdi          # x:6
   ja       .L8               # Use default
   jmp      *.L4(,%rdi,8)     # goto *JTab[x]
```

Indirect jump

Jump table (跳转表)

```
.section    .rodata
  .align 8
.L4:
  .quad     .L8   # x = 0
  .quad     .L3   # x = 1
  .quad     .L5   # x = 2
  .quad     .L9   # x = 3
  .quad     .L8   # x = 4
  .quad     .L7   # x = 5
  .quad     .L7   # x = 6
```

> 处理 Fall-Through



> 跳转表的适用范围：其它情况下会使用平衡二叉树的结构，提升性能

2.3 函数与栈帧

基于栈的编程语言

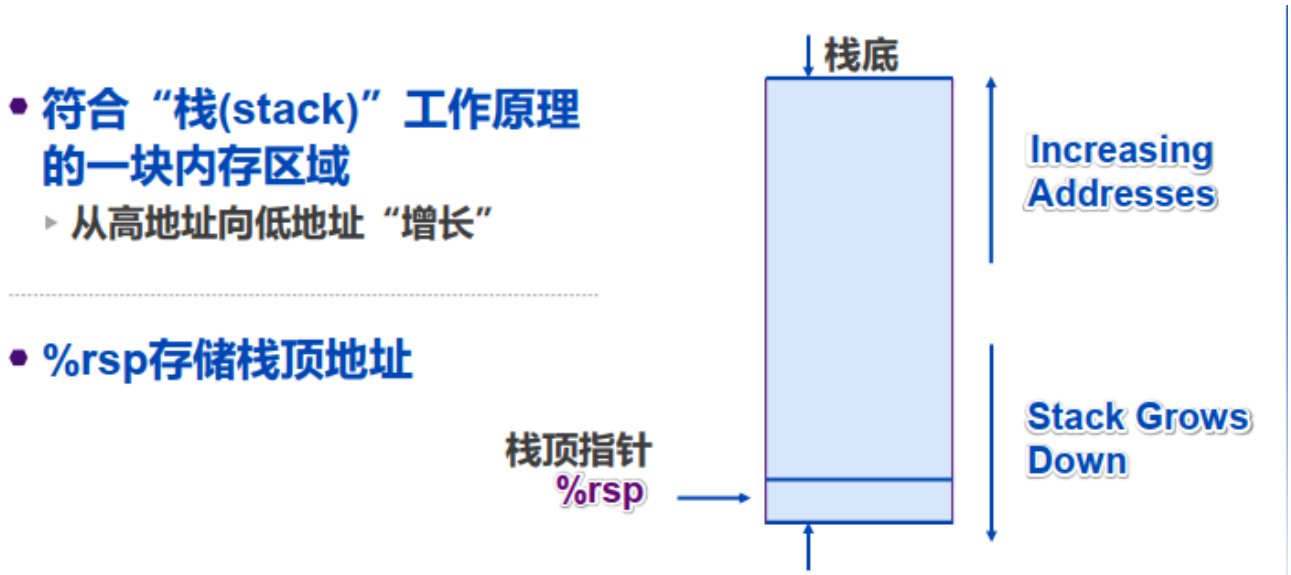
(memory management相关)

1. 支持递归
   [EG] C, Pascal, Java
   代码是**可重入**的（Reentrant）——同时有同一个函数的多个实例在运行
   因此需要有一块区域来存储每个函数实例的数据——参数/局部变量/返回地址
2. 栈的工作规律
   每个实例的运行时间是有限的，即**栈的有效时间有限**——From when called to when return
   被调用者**先于**调用者返回（一般情况下）
3. 每个实例在栈中维护一个**栈帧**（stack frame）

- 函数调用的作用与工作机理

  > Passing control: 代码调用与返回
  > Passing data: 函数参数与返回值
  > Memory management: 调用所需的内存分配/返回时释放
  >
  > 上述机制都是用机器指令实现的

- X86-64的程序栈



> `Push Src`
> 从`Src`取得操作数
> $\%rsp = \%rsp - 8$
> 写入栈顶地址（$\%rsp$）
>
> `popq Dest`
> 读取栈顶数据（$\%rsp$）
> $\%rsp = \%rsp + 8$
> 写入`Dest`

1. 调用与返回（Passing control）

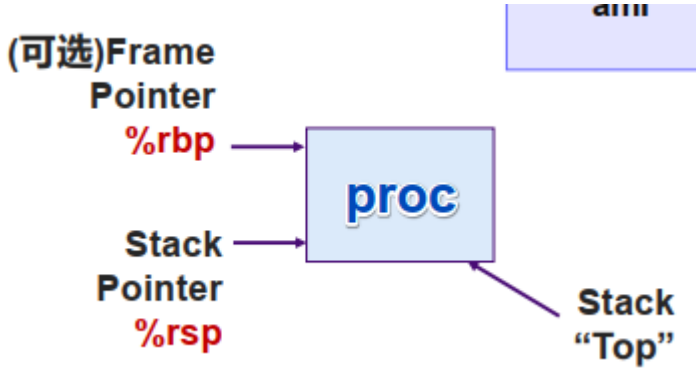
- 函数调用指令：`call label`， 将**返回地址**压入栈，跳转至label
- **返回地址**：Call指令的下一条指定地址
- 函数返回指令：`ret`，跳转至栈顶的返回地址

2. 传参（Passing data）

- 前六个参数存在寄存器内：$\%rdi, \%rsi, \%rdx, \%rcx, \%r8, \%r9$
  返回值：$\%rax$
- 大于六个参数使用栈传递，**从右往左**压栈

3. 栈帧（stack frame）

- 存储的内容：局部变量、返回地址、临时空间
- 栈帧的分配与释放
  进入函数后先**分配**栈帧空间：set-up code
  函数返回时**释放**：finish code
  寄存器%rbp指向当前栈帧的起始地址

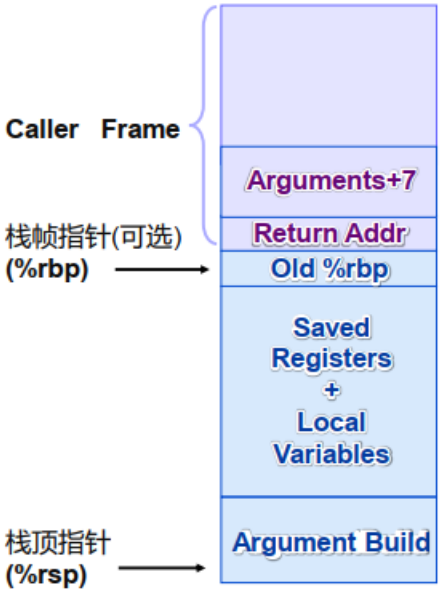


- 当前栈帧的内容（x86-64/Linux）

> 子函数参数（Argument build）
> 局部变量（因为通用寄存器个数有限）
> 被保存的寄存器值
> 父函数的栈帧起始地址（可选）

- 父函数的栈帧中与当前函数相关的内容

> 返回地址：由`call`指令存入
> 当前函数的输入参数



- 操作特性：一次性分配整个帧；释放简单

> 1. 将$\%rsp$减去某个值（栈帧的大小）
> 2. 对于栈帧内容的访问都是基于$\%rsp$完成的
> 3. 可以延迟分配：直接访问不超过当前栈指针（$\%rsp$）128字节的栈上空间
> 4. $\%rsp$直接加上某个值（栈帧大小）即可释放

4. 寄存器使用惯例

通用寄存器分为两类：
调用者负责保存：Caller在调用子函数之前将这些寄存器内容存储在它的栈帧内，调用后恢复
被调用者负责保存：Callee在使用寄存器之前将其原有内容存储在它的栈帧内，退出前恢复

%rax	Return Value	%r8	Argument #5
%rbx	Callee Saved	%r9	Argument #6
%rcx	Argument #4	%r10	Caller Saved
%rdx	Argument #3	%r11	Caller Saved
%rsi	Argument #2	%r12	Callee Saved
%rdi	Argument #1	%r13	Callee Saved
%rsp	Stack Pointer	%r14	Callee Saved
%rbp	Callee Saved	%r15	Callee Saved

5. 实例：递归

volatile变量：强制使用栈空间（直接存取原始内存地址）；但在实际使用中**没有修改栈顶寄存器%rsp**

> 通常而言，局部变量一般：① 被优化 ② 寄存器 ③ 运行栈，**volatile**相当于保证了③

Red Zone: The 128-byte area beyond the location pointed to by %rsp is considered to be reserved.

- RIP-relative addressing

  [EG] 全局变量的内存地址位置采用相对于$\%rip$的地址记录（这是因为$\%rip$是静态的，对比$\%rsp$的地址不断在变化）

```
long scount = 0;
/* Swap a[i] & a[i+1] */
void swap_ele_se
        (long a[], int i)
{
                swap(&a[i], &a[i+1]);
                scount++;
}
```

```
swap_ele_se:
        movslq %esi,%rsi                # Sign extend i
        leaq   (%rdi,%rsi,8), %rdi      # &a[i]
        leaq   8(%rdi), %rsi            # &a[i+1]
        call   swap                     # swap()
        incq   scount(%rip)             # scount++;
        ret
```

2.4 X86-32的函数调用

编译选项： `-m32`

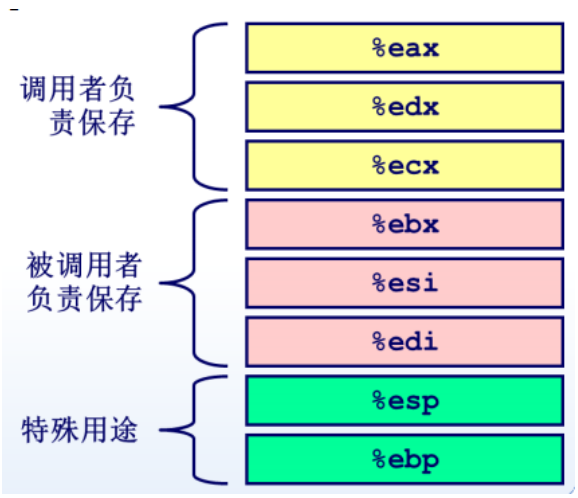
1. 传参直接压栈（对比X86-64会采用寄存器存储）

```
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;

    *xp = t1;
    *yp = t0;
}
```

```
swap:
    pushl %ebp                  Set
    movl %esp,%ebp              Up
    pushl %ebx
    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx           Body
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)
    movl -4(%ebp),%ebx          Finish
    movl %ebp,%esp
    popl %ebp                   leave指令
    ret
```

2. 只有8个通用寄存器，caller-callee关系如下：



思考："打破"函数调用与返回顺序

一般情况下"返回的顺序与调用顺序相反"

但是可以利用汇编以及对运行栈的理解来编写代码以打破这一惯例

- 有如下汇编代码（x86-32架构），其中GET唯一的输入参数是一个用于存储当前处理器以及栈信息的内存块地址（假设该内存块的空间足够大），而由SET恢复被GET所保存的处理器及栈信息，其唯一的输入参数也是该内存块地址

```
GET:
movl 4(%esp), %eax   #(A)
...
movl %edi, 20(%eax)
movl %esi, 24(%eax)
movl %ebp, 28(%eax)
movl %ebx, 36(%eax)
movl %edx, 40(%eax)
movl %ecx, 44(%eax)
movl $1, 48(%eax)
movl (%esp), %ecx      #(B)
movl %ecx, 60(%eax)
leal 4(%esp), %ecx     #(C)
movl %ecx, 72(%eax)
movl 44(%eax), %ecx
movl $0, %eax
ret
```

```
SET:
movl 4(%esp), %eax
...
movl 20(%eax), %edi
movl 24(%eax), %esi
movl 28(%eax), %ebp
movl 36(%eax), %ebx
movl 40(%eax), %edx
movl 44(%eax), %ecx
movl ___ (%eax), %esp      #(D)
pushl 60(%eax)             #(E)
movl 48(%eax), %eax
ret
```

1. SET过程的返回地址是什么，其返回值是多少？
2. 代码段中的（A）指令执行后，eax中存放的是什么？（B）指令执行后，ecx中存放的是什么？（C）指令的作用是什么？（E）指令的作用是什么？并将（D）指令补充完整

A 指令执行后，eax 存放的是**内存块的地址**（用于保存处理器和栈信息的内存块）。

B 指令执行后，ecx 存放的是**返回地址**（调用 GET 函数后下一句要执行的地址）。

C 指令的作用是**记录 esp 栈顶指针**。（用于在 D 中将返回地址压入栈）。

D 指令完整的是 `movl 72(%eax), %esp`。

E 指令的作用是将**返回地址压入栈**（调用 GET 函数后下一句要执行的地址）。

2.5 复合数据类型的汇编表示

基本数据类型

整型 如何区分无符号数与符号数?

Intel	GAS	Bytes	C
byte	b	1	[unsigned] char
word	w	2	[unsigned] short
double word	l	4	[unsigned] int
quad word	q	8	[unsigned] long int (x86-64)

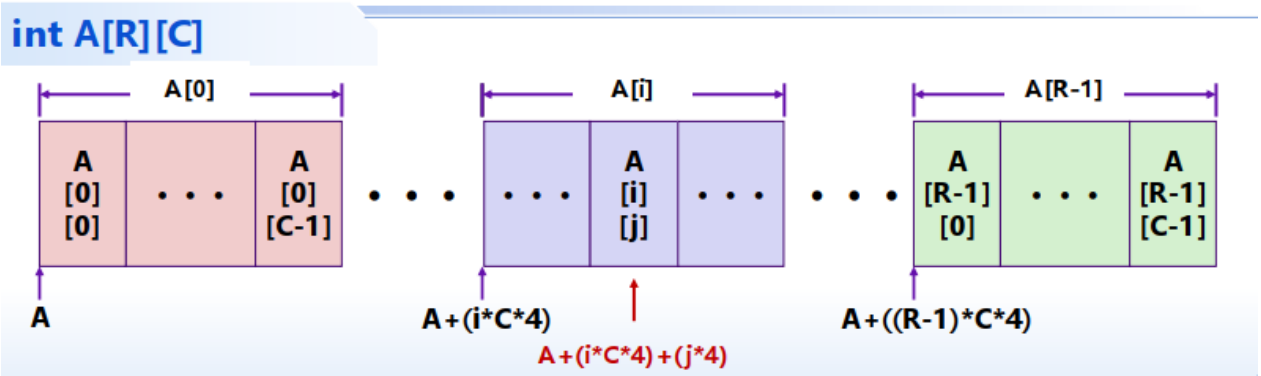
浮点数

Intel	GAS	Bytes	C
Single	s	4	float
Double	l	8	double
Extended	t	10/12/16	long double

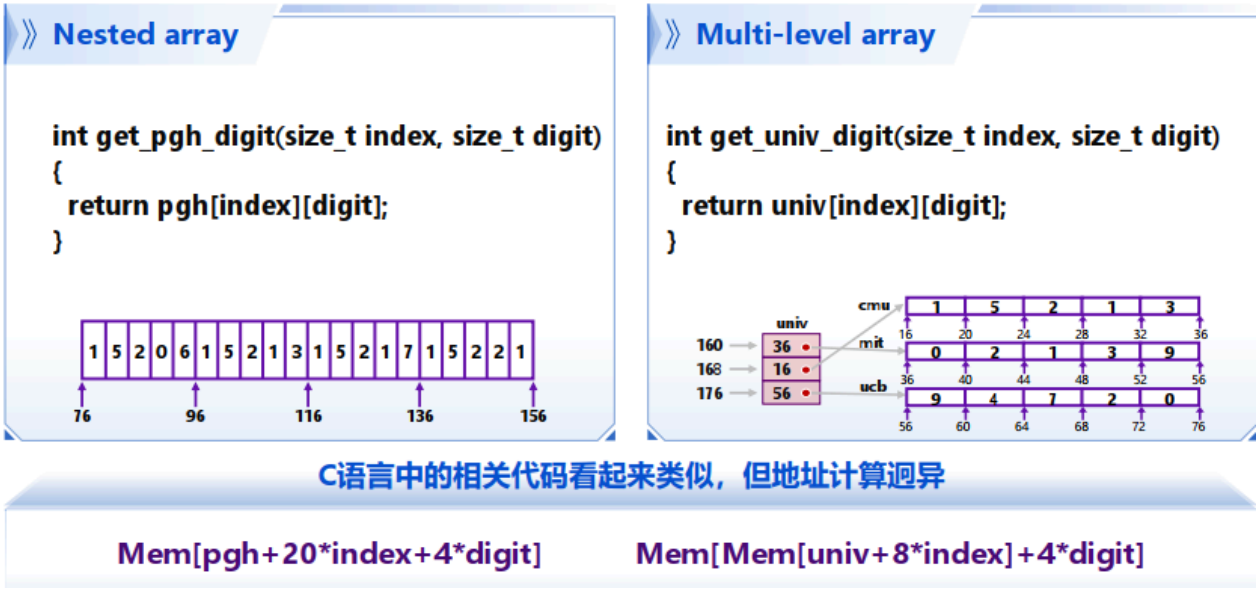
1. 数组与数组访问

> **一维数组**：`T A[L]`
> 基本数据类型：T， 数组长度：L
> 连续存储在大小为`L * sizeof(T)`字节的空间内

> **二维数组**
>
> - **嵌套数组**：`T A[R][C]` (R行C列)
>   数组所占据的空间大小：$R \times C \times K$ $bytes$ （T元素占据K字节）
>   行优先存储（Row-Major Ordering）



- **Multi-Level Array / Nested Array**
  变量univ是一个指针数组，数组长度为3，数组元素长度为8 bytes（`int* univ[3] = {a, b, c}`）
  每个指针指向一个整数数组
  与嵌套数组的区别：



- **数组维度大小的确定**
  - 固定大小

    > 编译时就知道数组的维度，也被称为**静态数组**

  - 可变大小

    > 运行时才知道数组的维度，也被称为**动态数组**

    - 显示索引计算：Traditional way to implement dynamic arrays
    - 隐式索引计算：supported by gcc (C99标准)

静态数组与动态数组访问优化的对比

2. 结构与结构访问

> **结构**的内存存储分布（Memory Layout）
> 连续分配的内存区域，内部各个域通过名字访问
> 各个域依声明顺序存放
> 结构的整体大小与各个域的位置布局由编译器决定（汇编层面不了解结构/域等信息）

- **对齐的一般原则**
  已知某种基本数据类型的大小为K字节，那么其存储地址必须是K的整数倍（X86-64）

> 计算机访问内存一般是以内存块为单位的，块的大小是地址对齐的，如4/8/16字节对齐等（如果数据访问地址跨越“块”边界会引起额外的内存访问）
>
> - **编译器的工作**
>
> 在结构的各个元素间插入额外空间来满足不同元素的对齐要求

- **对齐要求**
  - 基本数据类型

    `1 byte` (e.g. `char`)：N/A

    `2 bytes` (e.g. `short`)：地址最后一位为0

    `4 bytes` (e.g. `int`, `float`)：地址最后二位为0

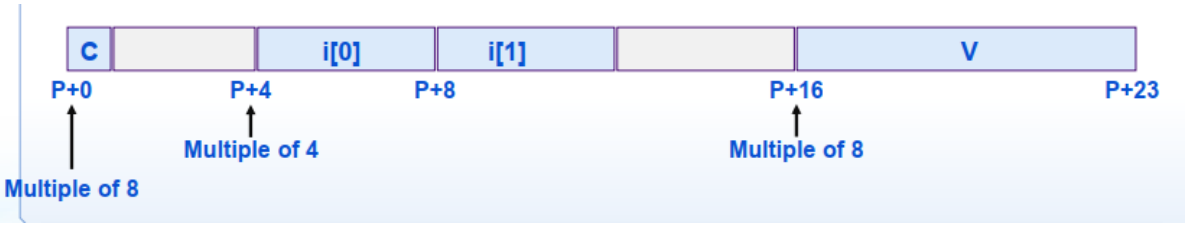
    `8 bytes` (e.g. `double`, `char*`)：地址最后三位为0

    `16 bytes` (e.g. `long double`)：地址最后四位为0
  - 结构的存储对齐要求

    结构起始地址的对齐要求等同于该结构各个元素中对齐要求最高的那个

    结构中元素的对齐要求必须满足该元素自身的对齐要求

    结构的长度必须是该结构各个元素中对齐要求最高的那个元素长度的整数倍

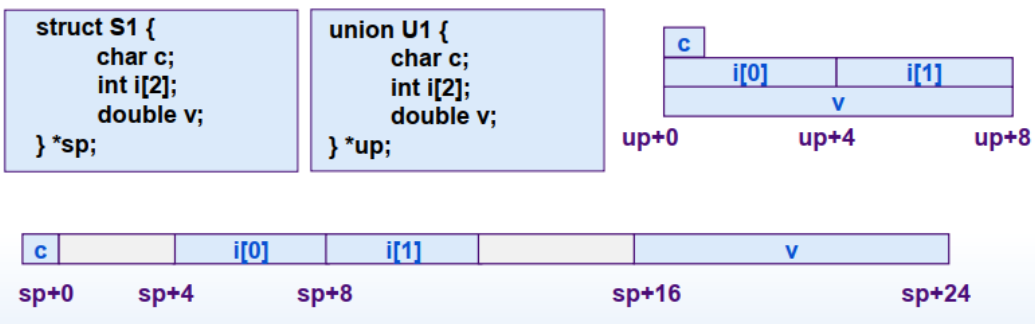
    > example：`struct S1 {char c; int i[2]; double v;}`——[K =8]
    >
    > 
    >
    >
    > S1如果是一个数组，那么数组中的每个元素（结构体）的地址是8的整数倍
    >
    > i必须是4对齐
  - 数组的每个元素在结构中的相对地址在编译时就已确定
  - 结构数组：整体结构长度乘以数组长度，满足每个元素的对齐要求

    > 访问结构数组中的元素：
    >
    > 首先计算数组元素（即结构）的地址
    >
    > 然后访问该结构中的原元素

    结构内元素不同，字节浪费情况也不相同

- **union / 联合**

> union中可以定义多个成员，union的大小由最大的成员的大小决定
>
> union成员共享同一块大小的内存，一次只能使用其中的一个成员



2.6 链接

数据、函数、指令的“定位”：

	编译	链接（静态）	说明
局部变量(非静态)	√		存于栈内 or 寄存器 or 被优化掉
全局变量		√	各个模块(.o文件)的数据段、代码段合并后，方能确定绝对地址
函数参数	√		通过指定的寄存器与栈传递
函数返回值	√		%rdx + %rax
跳转指令（直接）	√		相对于PC的offset
函数返回地址	√		存于栈顶
静态函数入口地址	√		相对于PC的offset
全局函数入口地址		√	各个模块(.o文件)的数据段、代码段合并后，方能确定绝对地址

非静态全局函数/变量是可以被多个模块引用的，而静态的不会，这是实现“定位”时需要考虑的

```
xxxxxxxxxx

g++ -E main.cpp -o main.ii //Preprocess：预处理命令（处理#include和宏）

g++ -S main.ii -o main.s //Compile：编译为汇编代码

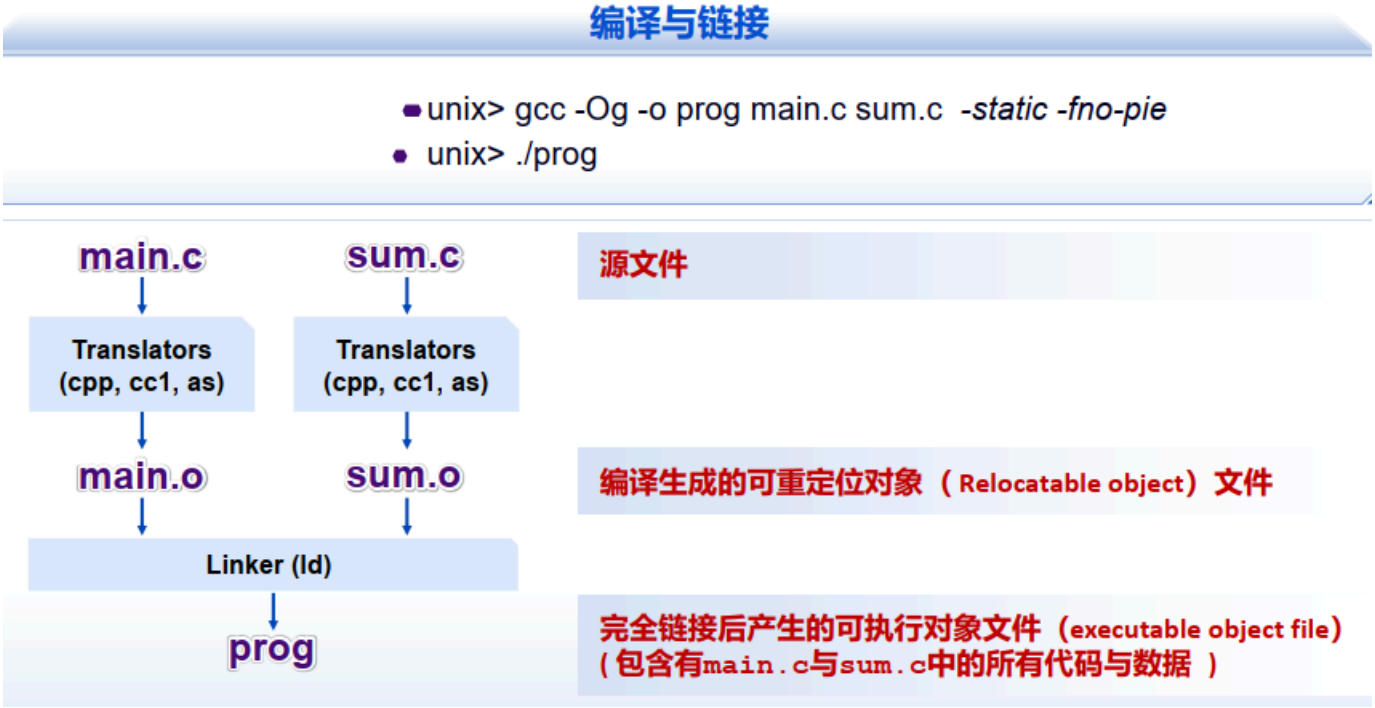
g++ -c main.s -o main.o //Assemble：编译成指令并保存在对象文件

//这三个合称“编译期”

g++ main.o -o main //Link：将对象文件和标准库链接为二进制可执行文件，“链接期”
```

1. 静态链接（Static Linking）

> 源文件 **=>** 编译生成**可重定位对象**（Relocatable Object）文件 **=>** 完全链接后产生**可执行对象**文件（executable object file）



程序链接的作用：

1. 模块化好
   多个小的源文件可以链接成一个程序，而不是一个巨大的单一源文件
   可以将多个通用函数链接成库文件
2. 工作效率高
   省时间：独立编译
   某个原文件被修改后，可以独立编译并重链接，而不需要编译所有文件
   省空间：库文件
   多个通用函数可以被集成到一个文件中
   可执行文件及其运行时的**内存镜像（memory image）**内只包含有实际使用到的函数

符号解析 / Symbol Resolution

- 程序定义以及引用了一系列**符号**（symbols，包括变量和函数）

> define symbol: `void swap() {...}`
> reference symbol: `swap();`
> define symbol and reference: `int *xp = &x;`

- 编译器将符号定义存储在**符号表**（symbol table）中

> **Symbol table** is an array of structs
> Each entry includes name, size and location of symbol

- 连接器将每一个**符号引用**（reference）与**符号定义**联系起来

重定位 / Relocation

1. 将多个文件的数据 / 代码段集成为单一的数据段和代码段
2. 将.o文件中的符号解析为绝对地址
3. 然后将所有的符号引用更新为这些新的地址

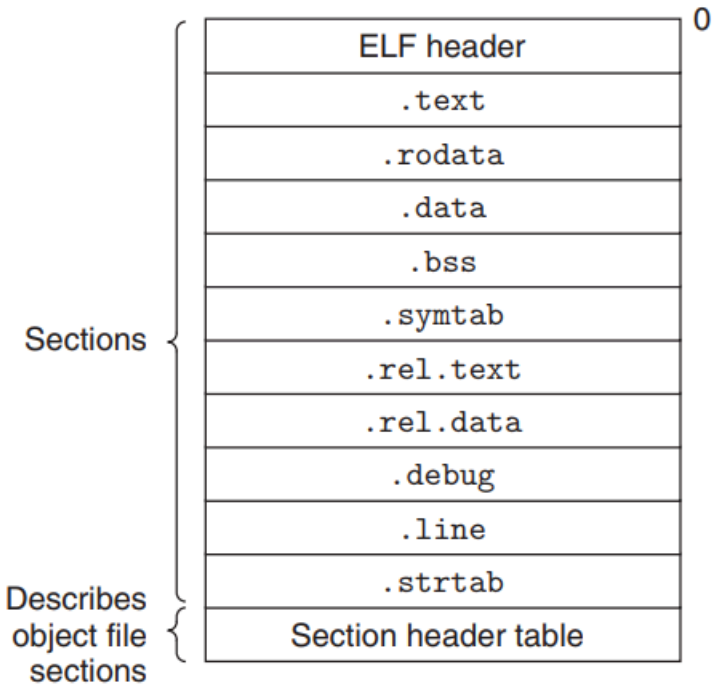
2. 对象文件

- **重定向对象文件**（`.o`文件）/ Relocatable object file
  含有一定格式的代码与数据内容，可以与其它重定向对象文件一起集成为执行文件
  一个`.o`文件由唯一的一个源文件生成
- **执行文件**（`.out`文件）/ Executable object file
  含有一定格式的代码与数据内容，可以直接被装载入内存并执行
- **共享对象文件**（`.so`文件）/ Shared object file
  特殊类型的重定向对象文件，可以被装载入内存后进行**动态链接**；链接可以在装载时或者运行时完成
  Windows系统下被称为`DLL`文件

> **标准二进制格式**：Executable and Linkable Format (ELF)
> 最初由AT&T System V Unix系统采用，后来被广泛采用
>
> Windows uses the Portable Executable (PE) format
> Mac OS-X uses the Mach-O format
> Modern x86-64 Linux and Unix systems use Executable and Linkable Format

ELF 文件的格式

`readelf main.o -h`：显示ELF header信息
`readelf main.o -S`：显示section headers信息
`readelf main.o -e`：相当于`-h, -l, -S`
`readelf main.o -s`：显示`.symtab`段的信息
`readelf main.o -r`：显示重定位段信息
`readelf main.o -a`：显示所有信息



Each symbol is assigned to some section of the object file, denoted by the `section` field, which is an index into the section header table. There are three special pseudosections that don't have entries in the section header table: ABS is for symbols that should not be relocated. UNDEF is for undefined symbols—that is, symbols that are referenced in this object module but defined elsewhere. COMMON is for uninitialized data objects that are not yet allocated. For COMMON symbols, the `value` field gives the alignment requirement, and `size` gives the minimum size. Note that these pseudosections exist only in relocatable object files; they do not exist in executable object files.

The distinction between COMMON and `.bss` is subtle. Modern versions of GCC assign symbols in relocatable object files to COMMON and `.bss` using the following convention:

COMMON	Uninitialized global variables
`.bss`	Uninitialized static variables, and global or static variables that are initialized to zero

> In Section 7.5, we saw how the compiler assigns symbols to COMMON and .bss using a seemingly arbitrary convention. Actually, this convention is due to the fact that in some cases the linker allows multiple modules to define global symbols with the same name. When the compiler is translating some module and encounters a weak global symbol, say, x, it does not know if other modules also define x, and if so, it cannot predict which of the multiple instances of x the linker might choose. So the compiler defers the decision to the linker by assigning x to COMMON. On the other hand, if x is initialized to zero, then it is a strong symbol (and thus must be unique by rule 2), so the compiler can confidently assign it to .bss. Similarly, static symbols are unique by construction, so the compiler can confidently assign them to either .data or .bss.

- **ELF header**
  Word size, byte ordering, file type (`.o`, `exec`, `.so`), machine type, etc.

  > begin with: a 16-byte sequence that describes the **word size** and **byte ordering** of the system that generated the file
  >
  > rest: **information** that allows a linker to parse and interpret the object file (**size** of the ELF header, object file **type**, **machine type**, **file offset** of the section header table, **size** of various sections described by the *section header table*)

- **Segment header table**
  Page size, virtual addresses memory segments (sections), segment sizes.
- **.text section**
  Code (functions)
- **.rodata section**
  Read only data: format strings, jump tables, ...
- **.data section**
  Initialized global variables, initialized static C variables (not zero)

  > Local C variables are maintained at run time on the **stack** and do *not* appear in either the `.data` or `.bss` sections

- **.bss section** (better save space)
  Uninitialized static C variables, global or static variables that are initialized to zero
  Has section header but occupies no space (merely a placeholder). At run time, they are allocated in memory with an initial value of zero.

  > Uninitialized global variable is classified as COMMON

- **.symtab section**
  Symbol table with information about functions and global variables defined and referenced in the program (no local variables)

```
Num:    Value          Size Type    Bind   Vis      Ndx Name
  8: 0000000000000000    24 FUNC    GLOBAL DEFAULT    1 main
  9: 0000000000000000     8 OBJECT  GLOBAL DEFAULT    3 array
 10: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND sum
```

In this example, we see an entry for the definition of global symbol `main`, a 24-byte function located at an offset (i.e., `value`) of zero in the `.text` section. This is followed by the definition of the global symbol `array`, an 8-byte object located at an offset of zero in the `.data` section. The last entry comes from the reference to the external symbol `sum`. READELF identifies each section by an integer index. `Ndx=1` denotes the `.text` section, and `Ndx=3` denotes the `.data` section.

符号表 / Symbol Tables

1. 全局符号 / Global symbols, defined by module `m`
   e.g. nonstatic C 函数，nonstatic 全局变量
2. 外部符号 / External symbols, referenced by module `m`
   e.g. nonstatic C 函数，nonstatic 全局变量
3. 局部符号 / Local symbols
   e.g. static 全局变量（不是local variables）
   这部分变量不保存在栈上，编译器在`.data`或`.bss`分配空间并且在symbol tables里分配独一无二的别名（e.g. 两个函数分别定义`static int x = 0`，则在汇编器中定义的别名不同）

Interestingly, local procedure variables that are defined with the C `static` attribute are not managed on the stack. Instead, the compiler allocates space in `.data` or `.bss` for each definition and creates a local linker symbol in the symbol table with a unique name. For example, suppose a pair of functions in the same module define a static local variable `x`:

> Symbol tables are built by **assemblers**, using symbols exported by the **compiler** into the assembly-language `.s` file.

code/link/elfstructs.c

```
1  typedef struct {
2      int   name;      /* String table offset */
3      char  type:4,    /* Function or data (4 bits) */
4            binding:4; /* Local or global (4 bits) */
5      char  reserved;  /* Unused */
6      short section;   /* Section header index */
7      long  value;     /* Section offset or absolute address */
8      long  size;      /* Object size in bytes */
9  } Elf64_Symbol;
```

code/link/elfstructs.c

Figure 7.4 ELF symbol table entry. The `type` and `binding` fields are 4 bits each.

- **.rel.text section**
  Relocation info for `.text` section
  Addresses of instructions that will need to be modified in the executable Instructions for modifying.
  usually ommited in executable object files
- **.rel.data section**
  Relocation info for `.data` section
  Addresses of pointer data that will need to be modified in the merged executable
- **.debug section**
  Info for symbolic debugging (`-g` option)
- **.line section**
  a mapping between line numbers in the original C source program and machine code instructions in the `.text` section (`-g` option)
- **.strtab section**
  a *string table* for the symbol tables in the `.symtab` and `.debug` sections and for the section names in the `section headers`
- **Section header table**
  Offsets and sizes of each section

全局变量重复定义

Strong: procedures and initialized globals
Weak: unintialized globals

Rule.1 Multiple strong symbols with the same name are not allowed.
Otherwise: Linker error

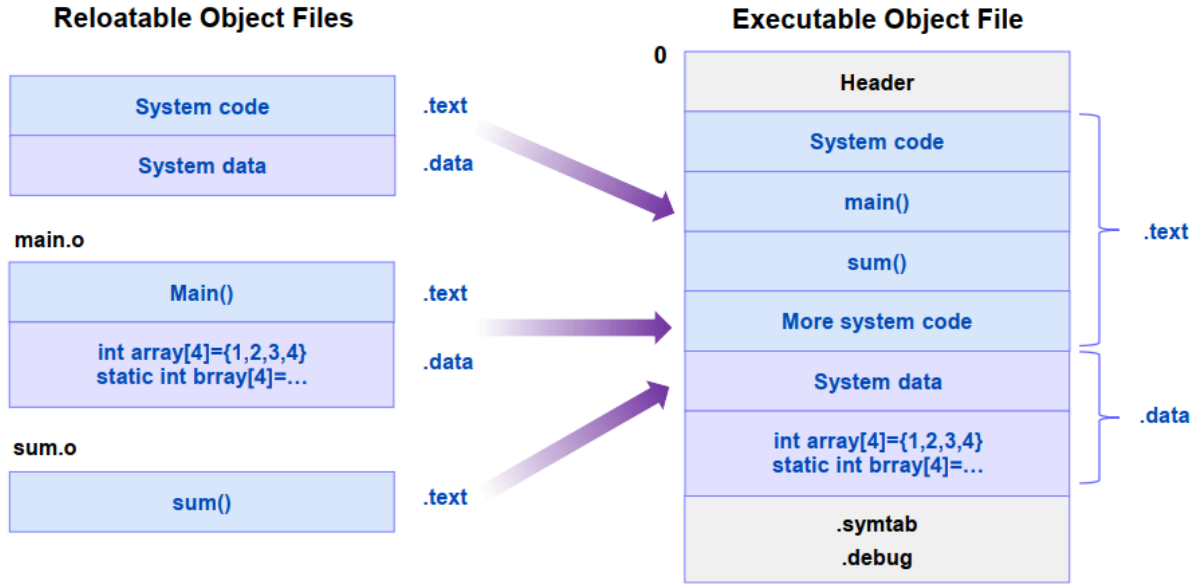
Rule.2 Given a strong symbol and multiple weak symbols with the same name, choose the strong symbol.

Rule.3 Given multiple weak symbols with the same name, choose any of the weak symbols.
can override this with `gcc -fno-common`

3. 代码与数据重定位

1. **Relocating sections and symbol definitions**
   merge sections from relocatable object files to executable object files
   assign run-time memory addresses to the new aggregate sections, to each section defined by the input modules, and to each symbol defined by the input modules

2. **Relocating symbol references within sections**
   modify every symbol reference in the bodies of the code and data sections so that they point to the correct run-time addresses



Relocation Entry

When an assembler generates an object module, it does not know where the code and data will ultimately be stored in memory. Nor does it know the locations of any externally defined functions or global variables that are referenced by the module. So whenever the assembler encounters a reference to an object whose ultimate location is unknown, it generates a *relocation entry* that tells the linker how to modify the reference when it merges the object file into an executable. Relocation entries for code are placed in `.rel.text`. Relocation entries for data are placed in `.rel.data`.

```
Relocation section '.rela.text' at offset 0x310 contains 5 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
00000000000d  000c00000004 R_X86_64_PLT32    0000000000000000 add - 4
000000000038  000300000002 R_X86_64_PC32     0000000000000000 .data - 4
000000000042  000e00000002 R_X86_64_PC32     0000000000000000 b - 4
00000000004c  000f00000002 R_X86_64_PC32     0000000000000020 a - 4
000000000054  000a00000004 R_X86_64_PLT32    0000000000000000 process - 4
```

- **Addend**
  The addend is a signed constant that is used by some types of relocations to bias the value of the modified reference
- **Type: R_X86_64_PC32**
  Relocate a reference that uses a 32-bit PC-relative address.
  R_X86_64_PLT52：计算方法同**PC32**，可以理解为新版本的**PC32**
- **Type: R_X86_64_32**
  Relocate a reference that uses a 32-bit absolute address.

these two relocation types support the x86-64 *small code model*
programs larger than 2GB can be compiled using the `-mcmodel=medium/large` flags

计算过程

`objdump -dx main.o`: *disassembled code*

重定位算法:

```
foreach section s {
    foreach relocation entry r {
        refptr = s + r.offset;  /* ptr to reference to be relocated */

        /* Relocate a PC-relative reference */
        if (r.type == R_X86_64_PC32) {
            refaddr = ADDR(s) + r.offset; /* ref's run-time address */
            *refptr = (unsigned) (ADDR(r.symbol) + r.addend - refaddr);
        }

        /* Relocate an absolute reference */
        if (r.type == R_X86_64_32)
            *refptr = (unsigned) (ADDR(r.symbol) + r.addend);
    }
}
```

- **重定位 PC-relative references**

```
0000000000000000 <main>:
   0:   48 83 ec 08             sub    $0x8,%rsp
   4:   be 02 00 00 00          mov    $0x2,%esi
   9:   bf 00 00 00 00          mov    $0x0,%edi       %edi = &array
                    a: R_X86_64_32 array            Relocation entry

   e:   e8 00 00 00 00          callq  13 <main+0x13>  sum()
                    f: R_X86_64_PC32 sum-0x4        Relocation entry
  13:   48 83 c4 08             add    $0x8,%rsp
  17:   c3                      retq
```

code/link/main-relo.d

Figure 7.11 **Code and relocation entries from** `main.o`. The original C code is in Figure 7.1.

Let's see how the linker uses this algorithm to relocate the references in our example program in Figure 7.1. Figure 7.11 shows the disassembled code from `main.o`, as generated by the GNU OBJDUMP tool (`objdump -dx main.o`).

The `main` function references two global symbols, `array` and `sum`. For each reference, the assembler has generated a relocation entry, which is displayed on the following line.[2] The relocation entries tell the linker that the reference to `sum` should be relocated using a 32-bit PC-relative address, and the reference to `array` should be relocated using a 32-bit absolute address. The next two sections detail how the linker relocates these references.

Relocating PC-Relative References

In line 6 in Figure 7.11, function `main` calls the `sum` function, which is defined in module `sum.o`. The `call` instruction begins at section offset `0xe` and consists of the 1-byte opcode `0xe8`, followed by a placeholder for the 32-bit PC-relative reference to the target `sum`.

The corresponding relocation entry `r` consists of four fields:

```
r.offset = 0xf
r.symbol = sum
r.type   = R_X86_64_PC32
r.addend = -4
```

These fields tell the linker to modify the 32-bit PC-relative reference starting at offset `0xf` so that it will point to the `sum` routine at run time. Now, suppose that the linker has determined that

```
ADDR(s) = ADDR(.text) = 0x4004d0
```

and

```
ADDR(r.symbol) = ADDR(sum) = 0x4004e8
```

Using the algorithm in Figure 7.10, the linker first computes the run-time address of the reference (line 7):

```
refaddr = ADDR(s)  + r.offset
        = 0x4004d0 + 0xf
        = 0x4004df
```

It then updates the reference so that it will point to the `sum` routine at run time (line 8):

```
*refptr = (unsigned) (ADDR(r.symbol) + r.addend - refaddr)
        = (unsigned) (0x4004e8       + (-4)     - 0x4004df)
        = (unsigned) (0x5)
```

In the resulting executable object file, the `call` instruction has the following relocated form:

```
4004de:  e8 05 00 00 00          callq  4004e8 <sum>      sum()
```

At run time, the `call` instruction will be located at address `0x4004de`. When the CPU executes the `call` instruction, the PC has a value of `0x4004e3`, which is the address of the instruction immediately following the `call` instruction. To execute the call instruction, the CPU performs the following steps:

1. Push PC onto stack
2. PC $\leftarrow$ PC + 0x5 = 0x4004e3 + 0x5 = 0x4004e8

Thus, the next instruction to execute is the first instruction of the `sum` routine, which of course is what we want!

- **重定位 Abosolute references**

Relocating Absolute References

Relocating absolute references is straightforward. For example, in line 4 in Figure 7.11, the `mov` instruction copies the address of `array` (a 32-bit immediate value) into register `%edi`. The `mov` instruction begins at section offset `0x9` and consists of the 1-byte opcode `0xbf`, followed by a placeholder for the 32-bit absolute reference to `array`.

The corresponding relocation entry `r` consists of four fields:

```
r.offset = 0xa
r.symbol = array
r.type   = R_X86_64_32
r.addend = 0
```

These fields tell the linker to modify the absolute reference starting at offset `0xa` so that it will point to the first byte of `array` at run time. Now, suppose that the linker has determined that

```
ADDR(r.symbol) = ADDR(array) = 0x601018
```

The linker updates the reference using line 13 of the algorithm in Figure 7.10:

```
*refptr = (unsigned) (ADDR(r.symbol) + r.addend)
        = (unsigned) (0x601018       + 0)
        = (unsigned) (0x601018)
```

In the resulting executable object file, the reference has the following relocated form:

```
4004d9:  bf 18 10 60 00          mov    $0x601018,%edi   %edi = &array
```

Putting it all together, Figure 7.12 shows the relocated `.text` and `.data` sections in the final executable object file. At load time, the loader can copy the bytes from these sections directly into memory and execute the instructions without any further modifications.

Result:

```
0000000000000000 <main>:
   0:   48 83 ec 08             sub    $0x8,%rsp
   4:   be 02 00 00 00          mov    $0x2,%esi
   9:   bf 00 00 00 00          mov    $0x0,%edi        %edi = &array
                        a: R_X86_64_32 array           Relocation entry

   e:   e8 00 00 00 00          callq  13 <main+0x13>   sum()
                        f: R_X86_64_PC32 sum-0x4       Relocation entry
  13:   48 83 c4 08             add    $0x8,%rsp
  17:   c3                      retq
```

(a) Relocated .text section

```
00000000004004d0 <main>:
  4004d0:   48 83 ec 08             sub    $0x8,%rsp
  4004d4:   be 02 00 00 00          mov    $0x2,%esi
  4004d9:   bf 18 10 60 00          mov    $0x601018,%edi    %edi = &array
  4004de:   e8 05 00 00 00          callq  4004e8 <sum>      sum()
  4004e3:   48 83 c4 08             add    $0x8,%rsp
  4004e7:   c3                      retq

00000000004004e8 <sum>:
  4004e8:   b8 00 00 00 00          mov    $0x0,%eax
  4004ed:   ba 00 00 00 00          mov    $0x0,%edx
  4004f2:   eb 09                   jmp    4004fd <sum+0x15>
  4004f4:   48 63 ca                movslq %edx,%rcx
  4004f7:   03 04 8f                add    (%rdi,%rcx,4),%eax
  4004fa:   83 c2 01                add    $0x1,%edx
  4004fd:   39 f2                   cmp    %esi,%edx
  4004ff:   7c f3                   jl     4004f4 <sum+0xc>
  400501:   f3 c3                   repz retq
```

(b) Relocated .data section

```
0000000000601018 <array>:
  601018:   01 00 00 00 02 00 00 00
```

e.g. 重定位信息（`main.o`, `sum.o`）

```
......
0000000000000008 <main>:                                        main.o
  8:    53                  push   %rbx
  9:    be 03 00 00 00      mov    $0x3,%esi
  e:    bf 00 00 00 00      mov    $0x0,%edi
                            f: R_X86_64_32      array  #relocation entry
 13:    e8 e8 ff ff ff      callq  0 <intra_sum>
 18:    89 c3               mov    %eax,%ebx
 1a:    be 03 00 00 00      mov    $0x3,%esi
 1f:    bf 00 00 00 00      mov    $0x0,%edi
                            20: R_X86_64_32   brray  #relocation entry
 24:    b8 00 00 00 00      mov    $0x0,%eax
 29:    e8 00 00 00 00      callq  2e <main+0x26>
                            2a: R_X86_64_PC32   inter_sum - 0x4 # ...
 2e:    01 d8               add    %ebx,%eax
 30:    5b                  pop    %rbx
 31:    c3                  retq
```

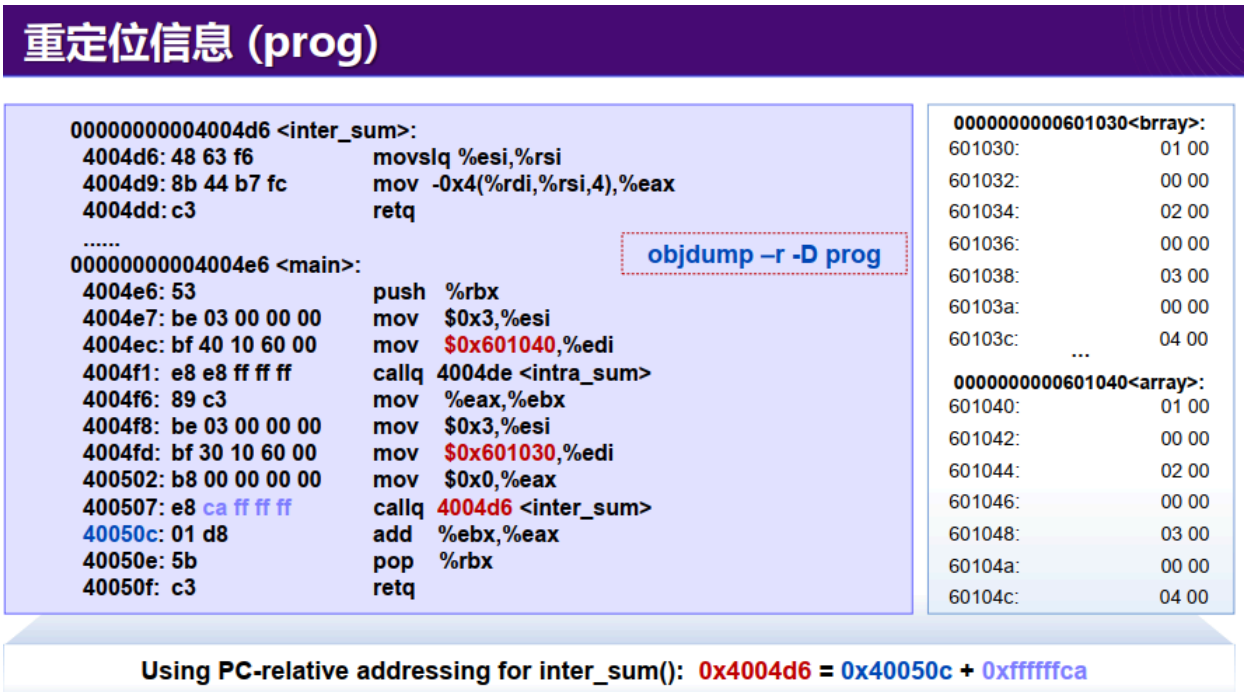
```
int array[4] = {1,2,3,4};
static int brray[4] = {1,2,3,4};

static int intra_sum (int x[4],int y)
{
  return x[y-1];
}

int main()
{
   int val = intra_sum(array, 3) +
             inter_sum(brray,3);
   return val;
}
```

Why "-4"?

```
objdump –r –D   0000000000000000 <inter_sum>:
                  0:    48 63 f6          movslq %esi,%rsi
                  3:    8b 44 b7 fc       mov    -0x4(%rdi,%rsi,4),%eax
                  7:    c3                retq                          sum.o
```



> **此处，为什么“inter_sum - 0x4"**:
>
> 1. 机器码`e8`对应的含义是`call`
> 2. 该行占据的指令数为5bytes，因此执行到改行时，PC = 0x400507 + 0x5 = 0x40050c
> 3. 后续的4组数字/4bytes为跳转函数的地址与当前地址的offset。且由于LSB (Least Significant Byte, 低位字节占据高地址)，此处实际表示的offset = 0xffffffca
> 4. 因此计算得到函数的地址为PC + offset = 0x40050c + 0xffffffca = 0x4004d6
> 5. **-4**：按照$400508+?=4004d6$计算的结果需要减去4才是正确结果
>
> **这里，400508为400507跳过callq指令的机器码**

4. 重定位文件

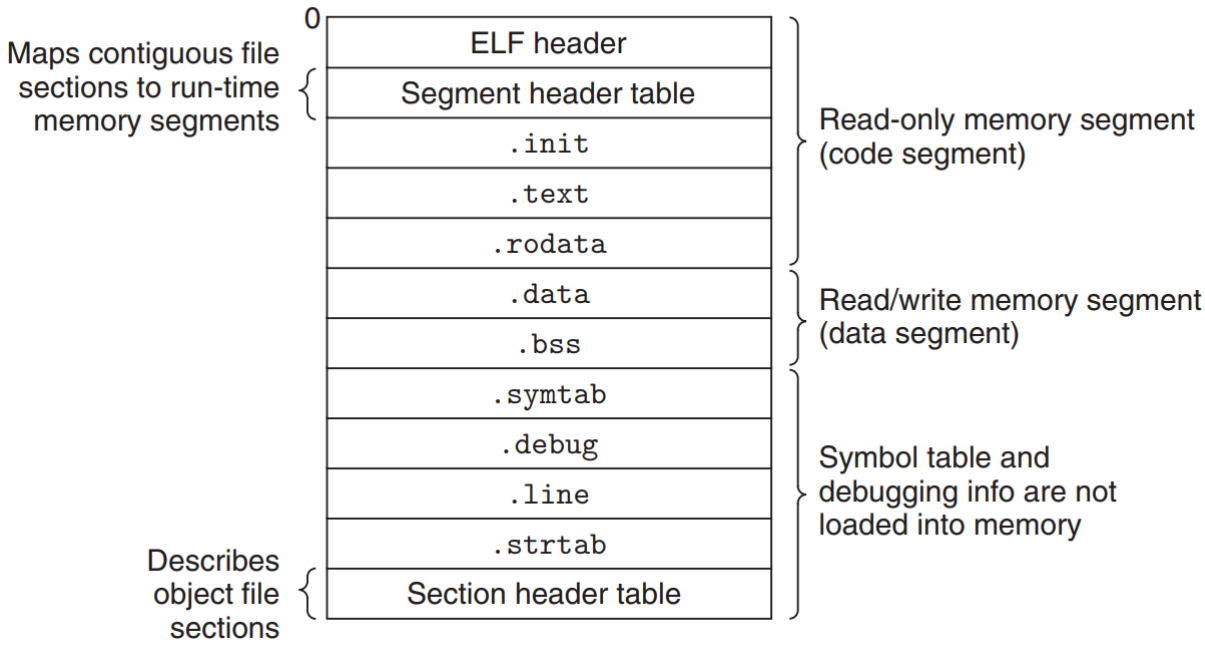
a typical executable file:

> - the **ELF header** includes the program’s **entry point**, which is the address of the first instruction to execute when the program runs
> - **Segment header table**
>
> *code/link/prog-exe.d*
>
> ```
> Read-only code segment
> 1 LOAD off    0x0000000000000000 vaddr 0x0000000000400000 paddr 0x0000000000400000 align 2**21
> 2       filesz 0x000000000000069c memsz 0x000000000000069c flags r-x
>
> Read/write data segment
> 3 LOAD off    0x0000000000000df8 vaddr 0x0000000000600df8 paddr 0x0000000000600df8 align 2**21
> 4       filesz 0x0000000000000228 memsz 0x0000000000000230 flags rw-
> ```
>
> *code/link/prog-exe.d*
>
> **Figure 7.14 Program header table for the example executable** `prog`. `off`: offset in object file; `vaddr`/`paddr`: memory address; `align`: alignment requirement; `filesz`: segment size in object file; `memsz`: segment size in memory; `flags`: run-time permissions.
>
> From the program header table, we see that two memory segments will be initialized with the contents of the executable object file. Lines 1 and 2 tell us that the **first segment (the code segment)** has read/execute permissions, starts at memory address 0x400000, has a total size in memory of 0x69c bytes, and is initialized with the first 0x69c bytes of the executable object file, which includes the ELF

> header, the program header table, and the .init, .text, and .rodata sections.
>
> Lines 3 and 4 tell us that the **second segment (the data segment)** has read/write permissions, starts at memory address 0x600df8, has a total memory size of 0x230 bytes, and is initialized with the 0x228 bytes in the .data section starting at offset 0xdf8 in the object file. **The remaining 8 bytes in the segment correspond to .bss data that will be initialized to zero at run time.**

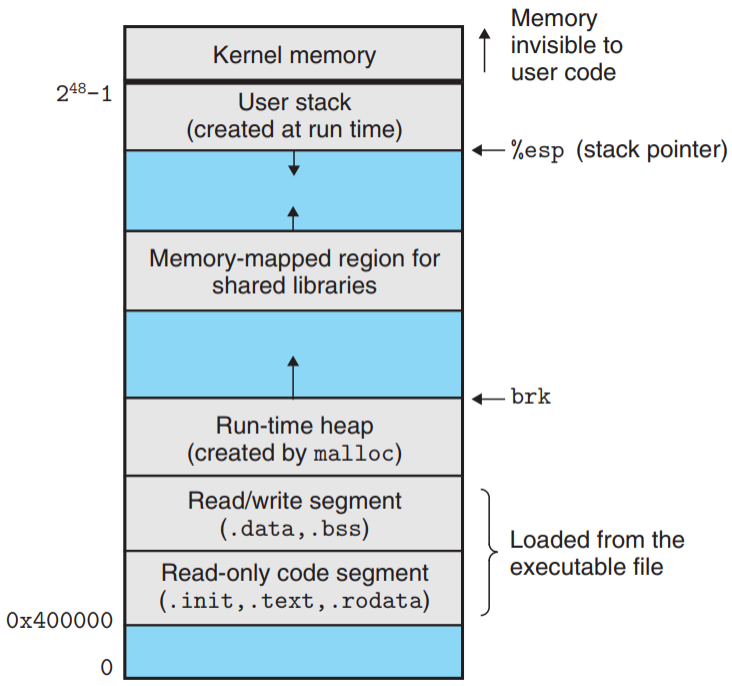
- the **.init** section defines a small function, called **_init**, that will be called by the program's initialization code
- the **.text**, **.rodata** and **.data** sections are similar to those in a relocatable object file, except that these sections have been relocated to their eventual run-time memory addresses
- since the executable is fully linked (relocated), it needs no **.rel** sections

For any **segment**, the linker must choose a starting address, **vaddr**, such that $vaddr \ mod \ align \ = \ off \ mod \ align$ where **off** is the offset of the segment's first section in the object file, and **align** is the alignment specified in the program header (2^{21} = 0x200000).



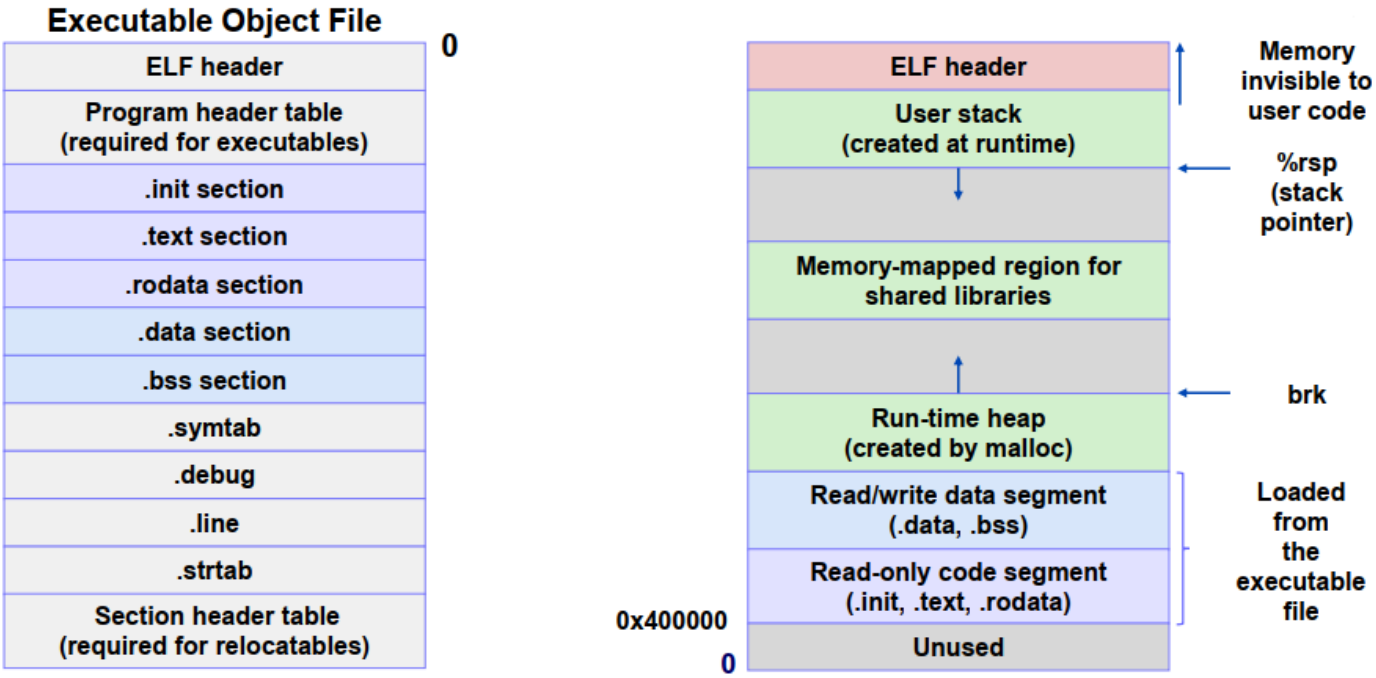
Loading Executable Object Files

> Since `./main` does not correspond to a built-in shell command, the shell assumes that `main` is an executable object file, which it runs for us by invoking some memory-resident operating system code known as the **loader**. Any Linux program can invoke the loader by calling the *execve* function.
> The loader copies the code and data in the executable object file from disk into memory and then runs the program by jumping to its first instruction, or **entry point**. This process of copying the program into memory and then running it is known as **loading**.



Every running Linux program has a run-time memory image similar to the one in Figure 7.15. On Linux x86-64 systems, the code segment starts at address 0x400000, followed by the data segment. The run-time *heap* follows the data segment and grows upward via calls to the `malloc` library. (We will describe `malloc` and the heap in detail in Section 9.9.) This is followed by a region that is reserved for shared modules. The user stack starts below the largest legal user address ($2^{48} - 1$) and grows down, toward smaller memory addresses. The region above the stack, starting at address 2^{48}, is reserved for the code and data in the *kernel*, which is the memory-resident part of the operating system.

Next, the loader jumps to the program's entry point, which is always the address of the **_start function**. This function is defined in the system object file `crt1.o` and is the same for all C programs. The **_start function** calls the system startup function, ***_libc_start* main**, which is defined in `libc.so`. It initializes the execution environment, calls the user-level main function, handles its return value, and if necessary returns control to the kernel.

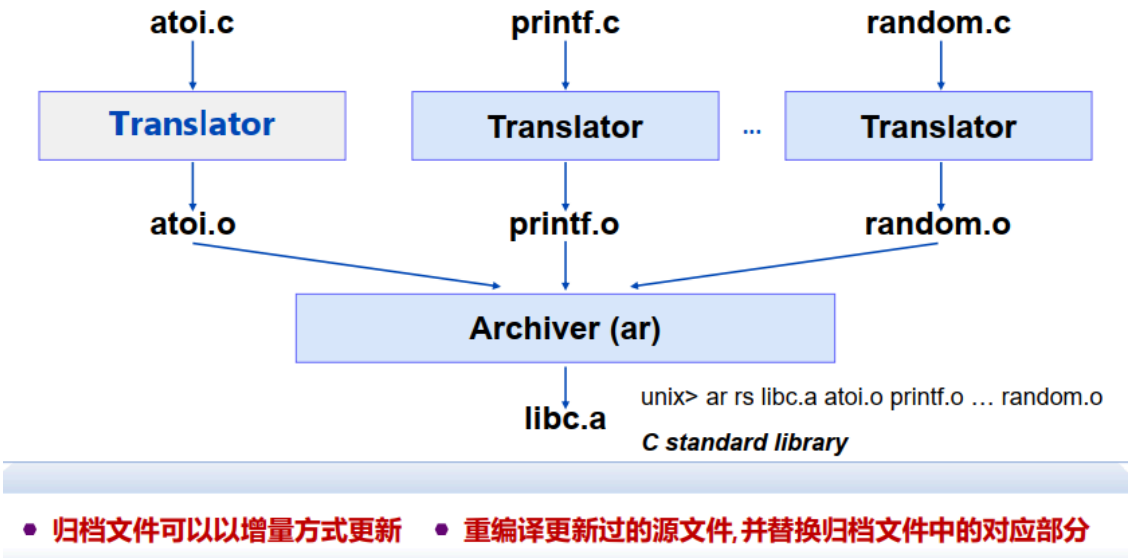


5. 静态库文件 vs 共享库文件

static library vs shared library

静态库文件（`.a`文件）

> 1. 将多个相关的重定位对象文件集成为一个单一的带索引的文件 (称为归档文件, archive file)
> 2. 增强链接器的功能使之能够在归档文件中解析外部符号
> 3. 如果归档文件中的某个成员解析了外部符号，就将其链接入执行文件



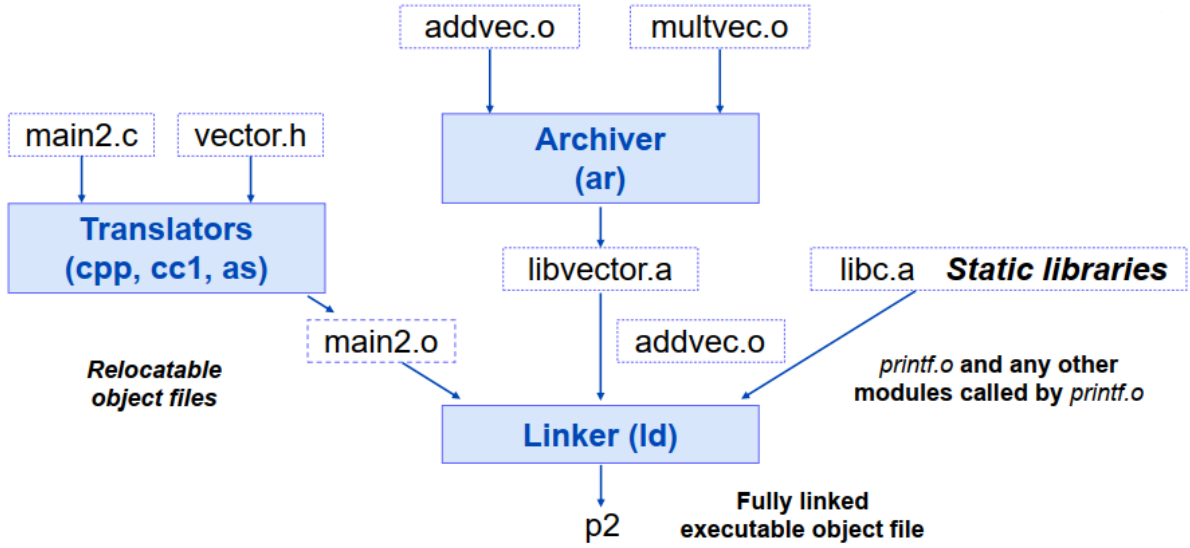
- `libc.a` **(the C standard library)**

  > 8MB archive of 1392 object files
  > I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math

- `libm.a` **(the C math library)**

  > 1MB archive of 401 object files
  > floating point math (sin, cos, tan, log, exp, sqrt, ...)

静态库连接



共享库文件

- 静态库文件有其劣势

> 执行文件中会重复包含有所需的库文件函数或者数据
> 运行时内存中也会有重复部分
> 库文件的细微变动需要所有相关执行文件进行重链接

- 更好的方案：共享库文件方式

> 特殊类型的重定向对象文件，可以被装载入内存后进行动态链接；链接可以在装载时或者运行时完成
> Windows系统下被称为DLL文件

> A **shared library** is an object module that, at **either run time or load time**, can be loaded at an arbitrary memory address and linked with a program in memory. This process is known as **dynamic linking** and is performed by a program called a **dynamic linker**. Shared libraries are also referred to as shared objects, and on Linux systems they are indicated by the `.so` suffix. Microsoft operating systems make heavy use of shared libraries, which they refer to as **DLLs (dynamic link libraries)**
>
> - one `.so` file for a particular library
> - a `.text` section in memory shared by different running processes

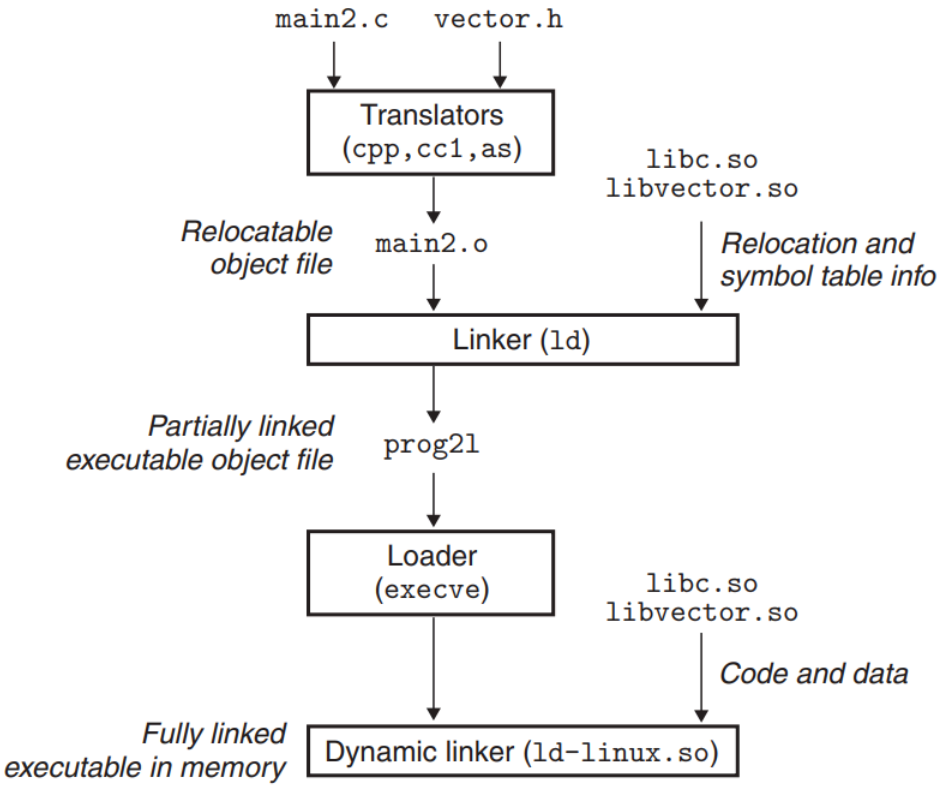
1. build a shared library `libvector.so`
   `linux> gcc -shared -fpic -o libvector.so addvec.c multvec.c`
   (`-shared`: direct the loader to create a shared object file; `-fpic`: directs the compiler to generate position-independent code)
2. link to program
   `linux> gcc main2.c ./libvector.so -o prog2l`
   This creates an executable object file `prog2l` in a form that can be linked with `libvector.so` at run time.

> None of the code or data sections from `libvector.so` are actually copied into the executable `prog2l` at this point.
> Instead, the linker copies some **relocation and symbol table information** that will allow references to code and data in `libvector.so` to be resolved at load time

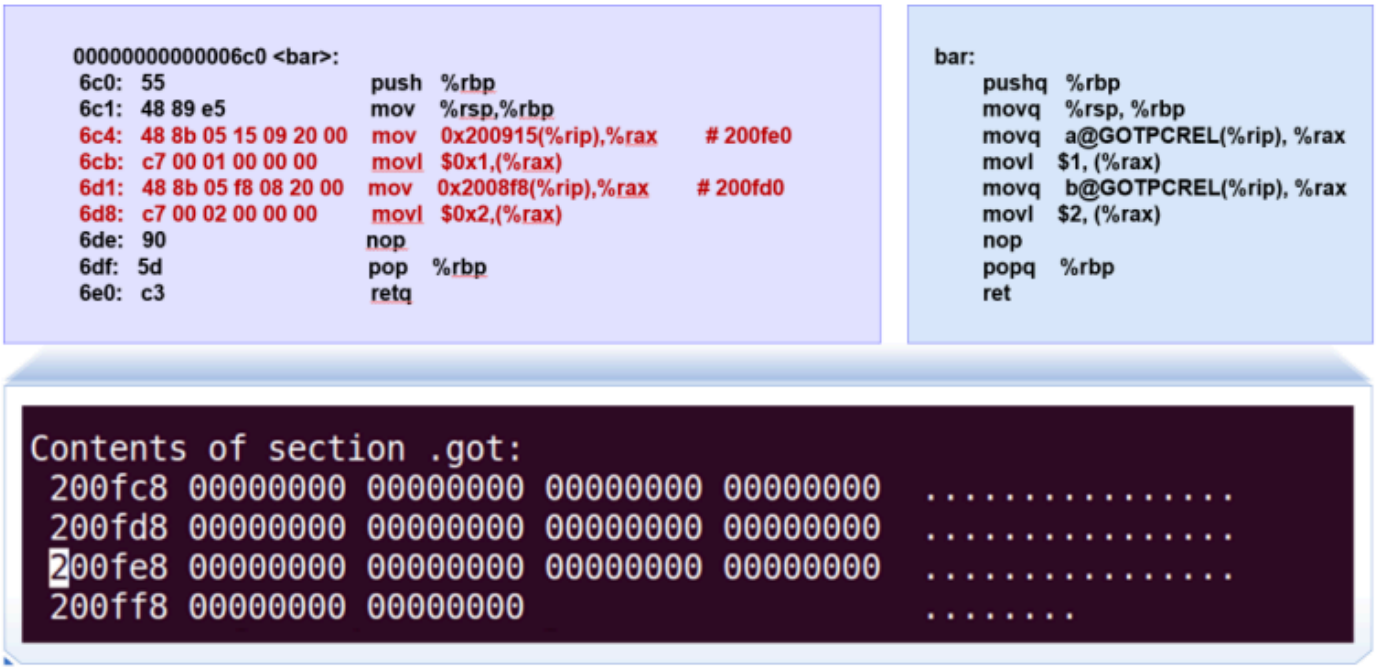
> `prog2l` contains a `.interp` section, which contains the path name of the dynamic linker, which is itself a shared object (e.g., ld-linux.so on Linux systems). The loader loads and runs the dynamic linker

3. dynamic linker: fully linked executable in memory

> relocate the text and data of `libc.so` into some memory segment.
> relocate the text and data of `libvector.so` into another memory segment.
> relocate any references in `prog2l` to symbols defined by `libc`.so and `libvector.so`







3 内存

3.1 内存布局 memory layout

Linux进程的内存布局（X86-64）

- **Stack**
  Runtime stack (8MB limit)
  栈顶地址：0x 0000 7fff ffff ffff
  e.g. local variables
- **Heap**
  按需动态分配
  when call `malloc()`, `calloc()`, `new()` ——**libC**库提供

  > 操作系统只能管理Heap的堆顶调度
  > LibC库提供管理整个Heap的方法

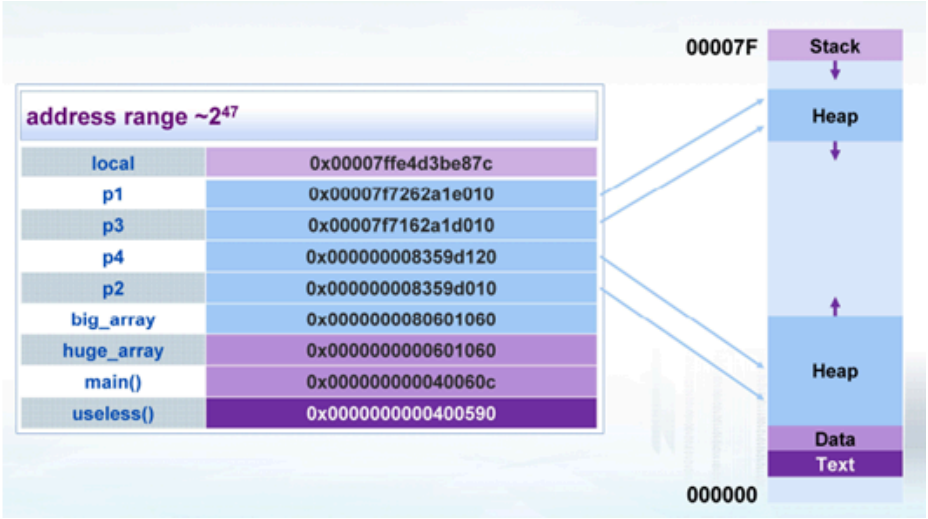
  - 大的内存块在高地址端，小的内存块在低地址端（可能的布局形式）
- **Data**
  静态分配的数据空间
  e.g. global vars, static vars, string constants
- **Text / Shared Libraries**
  指令（代码）
  Read-only

```
char big_array[1L<<24];  /* 16 MB */
char huge_array[1L<<31]; /*  2 GB */

int global = 0;

int useless() { return 0; }

int main ()
{
   void *p1, *p2, *p3, *p4;
   int local = 0;
   p1 = malloc(1L << 28); /* 256 MB */
   p2 = malloc(1L << 8);  /* 256  B */
   p3 = malloc(1L << 32); /*   4 GB */
   p4 = malloc(1L << 8);  /* 256  B */
 /* Some print statements ... */
}
```



其实，Heap和Data没有一条泾渭分明的分界线

3.2 缓冲区溢出 buffer overflow

易受攻击的缓冲区相关代码

```
xxxxxxxxxx

/* Echo Line */

void echo() {

    char buf[4]; /* Way too small! */

    gets(buf);

    puts(buf);

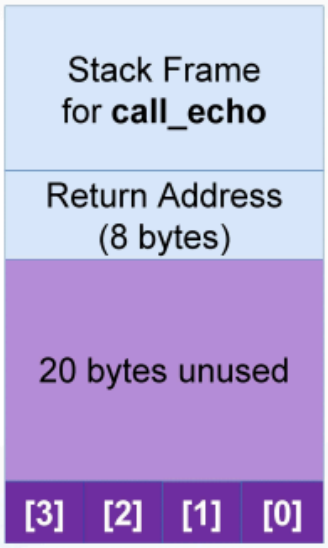
}

void call_echo() {

    echo();

}20221107101421023
```

Before call to *gets*

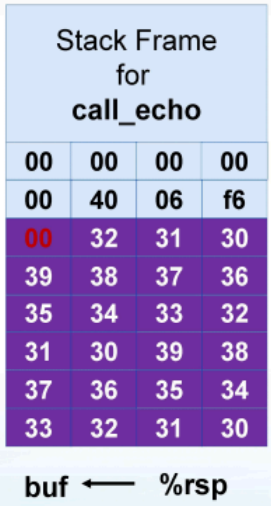



```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

echo:

```
subq  $24, %rsp
movq  %rsp, %rdi
call  gets
. . .
```

After call to *gets*




```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
  subq  $24, %rsp
  movq  %rsp, %rdi
  call  gets
  . . .
```

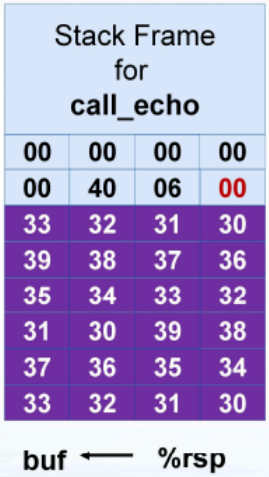
call_echo:

```
. . .
4006f1:callq  4006cf <echo>
4006f6:add    $0x8,%rsp
. . .
```

```
unix>./bufdemo-nsp
Type a string:0123456789012345678901 2
0123456789012345678901
```

Overflowed buffer, but did not corrupt state

After call to *gets*




```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

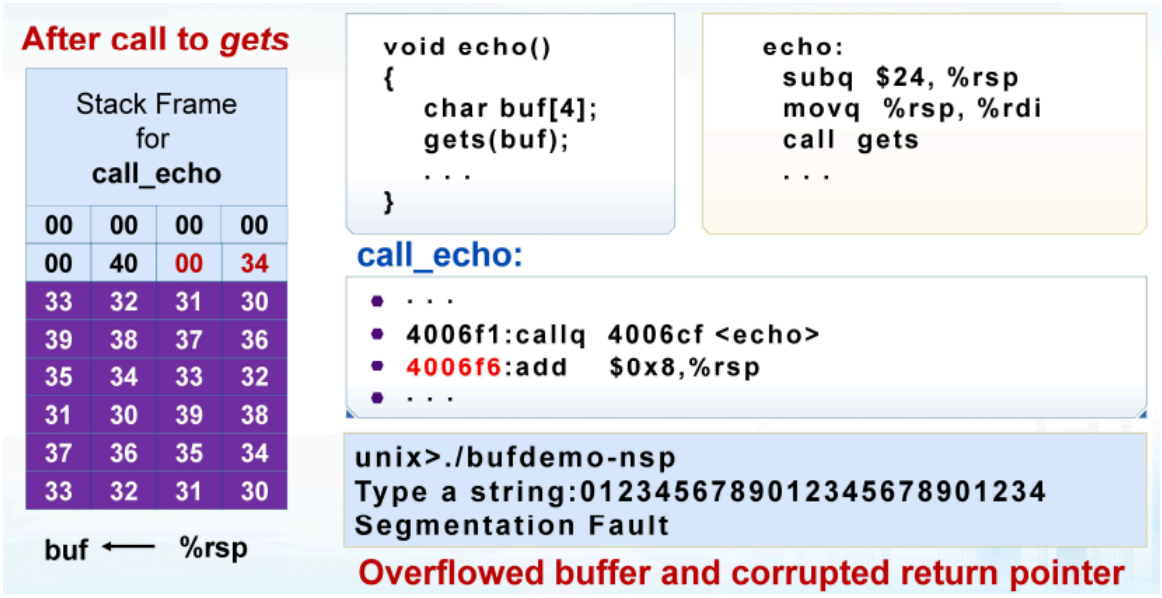
```
echo:
  subq  $24, %rsp
  movq  %rsp, %rdi
  call  gets
  . . .
```

call_echo:

```
. . .
4006f1:callq  4006cf <echo>
4006f6:add    $0x8,%rsp
. . .
```

```
unix>./bufdemo-nsp
Type a string:0123456789012345678901 23
0123456789012345678901 23
```

Overflowed buffer, corrupted return pointer, but program seems to work!



1. 代码注入攻击

> 输入字符串包含可执行代码的字节表示
> 用缓冲区B的地址**覆盖函数返回地址A**
> 当程序执行ret时，将跳转到**“注入代码”**（绝对地址）

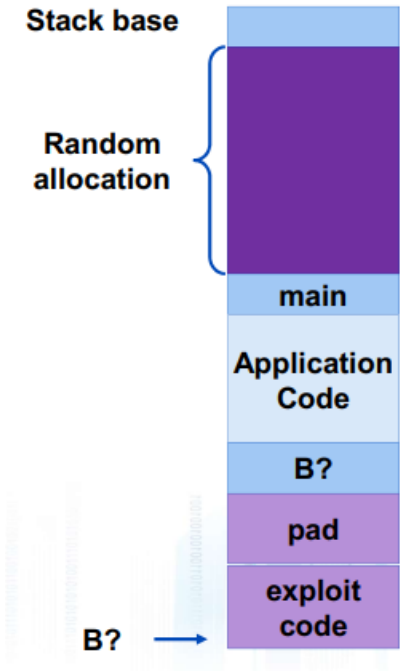
预防手段

1. avoid overflow vulnerabilities
   不要用`gets()`，`puts()`这种函数

   > **使用限制字符串长度的库函数**
   > **fgets** instead of **gets**
   > **strncpy** instead of **strcpy**
   > Don’t use **scanf** with **%s** conversion specification
   > use **fgets** to read the string
   > or use **%ns** where **n** is a suitable integer

2. employ system-level protections

   > **随机的stack base地址**



（处理器支持的）不可执行段

mark region of memory as either "read-only" or "writeable"

X86-64 added explicit "execute" permission

stack marked as non-executable

——Any attempt to execute this code (exploit code) will fail!

3. have compiler use "stack canaries"

Stack Canaries (金丝雀) can help

Idea

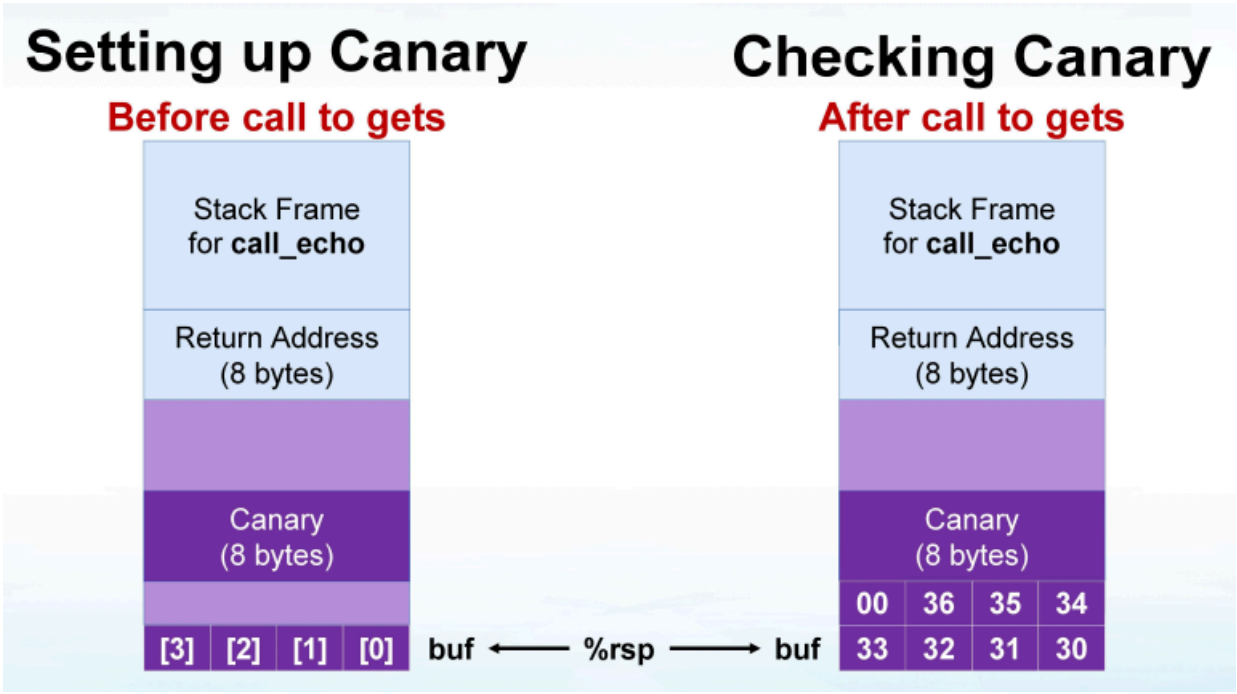
Place special value (“canary”) on stack just beyond buffer

Check for corruption before exiting function

GCC Implementation

-fstack-protector

Now the default (disabled earlier)



fs:0x28 on Linux is storing a special sentinel stack-guard value. This address is defined as *stack_chk_guard* in glibc, and the related code might look like this:

echo:

```
40072f:  sub     $0x18,%rsp
400733:  mov     %fs:0x28,%rax
40073c:  mov     %rax,0x8(%rsp)
400741:  xor     %eax,%eax
400743:  mov     %rsp,%rdi
400746:  callq   4006e0 <gets>
40074b:  mov     %rsp,%rdi
40074e:  callq   400570 <puts@plt>
400753:  mov     0x8(%rsp),%rax
400758:  xor     %fs:0x28,%rax
400761:  je      400768 <echo+0x39>
400763:  callq   400580 <__stack_chk_fail@plt>
400768:  add     $0x18,%rsp
40076c:  retq
```

```
unsigned long __stack_chk_guard;
void __stack_chk_guard_setup(void)
{
     __stack_chk_guard = 0xBAAAAAAD;//provide some magic numbers
}

void __stack_chk_fail(void)
{
 /* Error message */
}// will be called when guard variable is corrupted
```

2. Return-Oriented Programming (ROP) Attacks

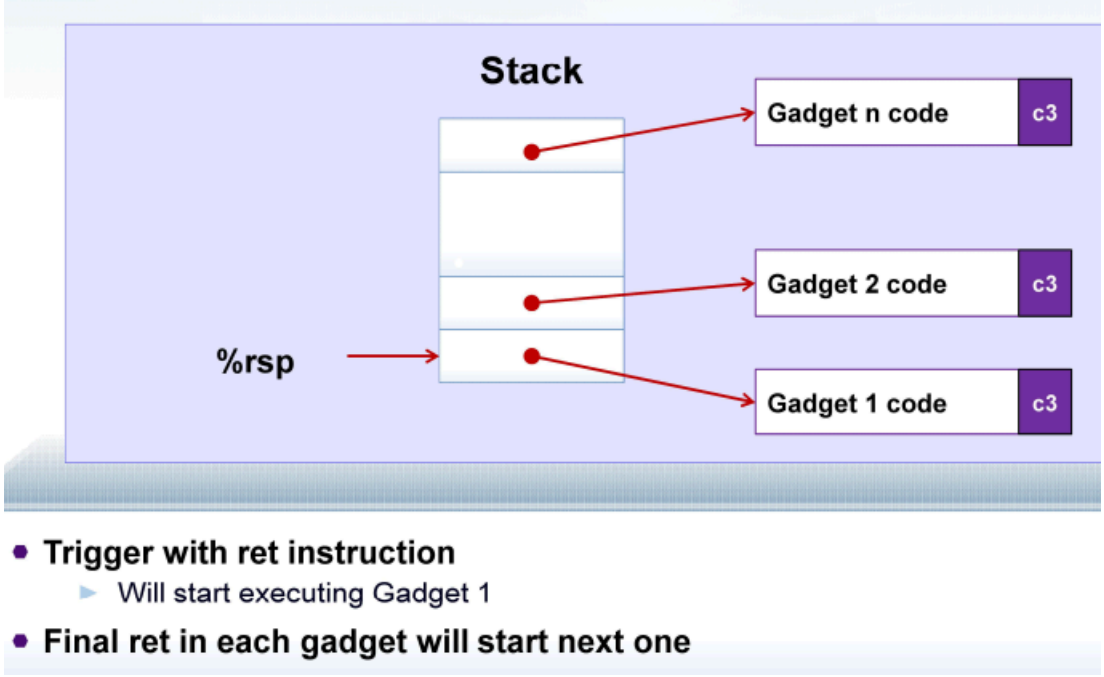
> **Challenges for hackers**
> **Stack randomization** makes it hard to predict buffer location
> Marking stack **non-executable** makes it hard to insert binary code

Alternative Strategy:

- Using existing code (e.g. library code from stdlib)
  String together fragments to achieve overall desired outcome

> **Construct program from gadgets**
> Gadget: Sequence of instructions ending in **ret**
> encoding by single byte **0xc3**
> Code positions fixed from run to run
> Code is executable

- However, does not overcome stack canaries



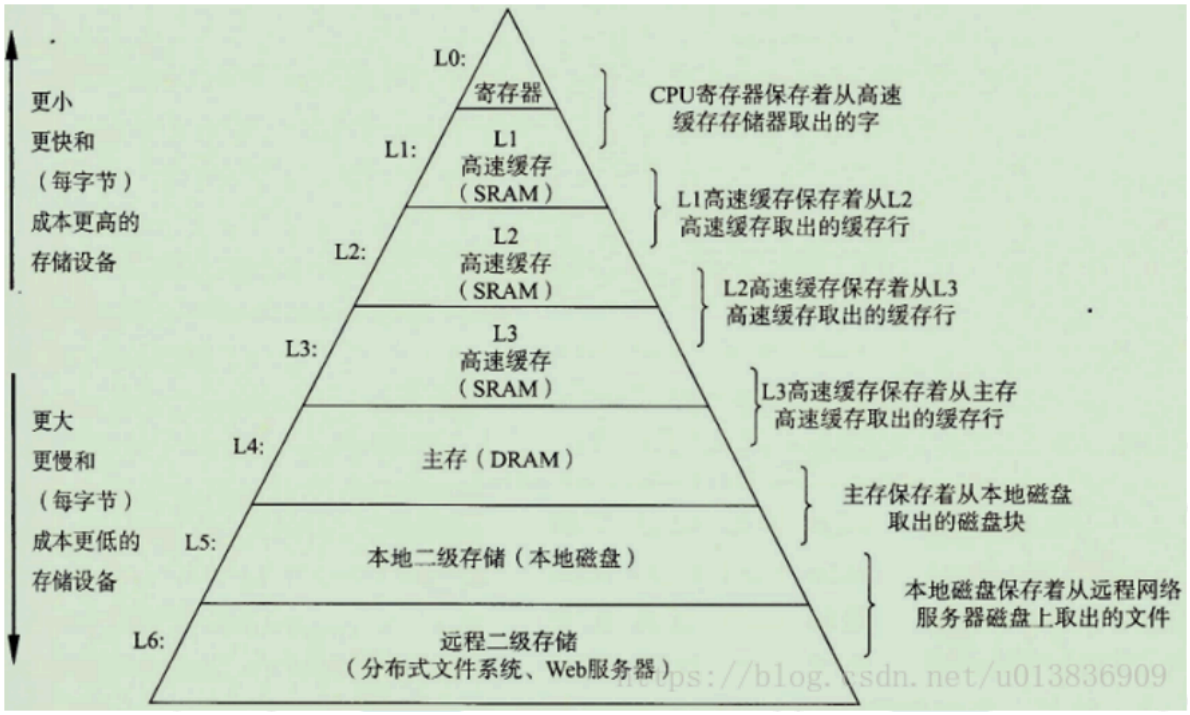
3. Supplement: COP / JOP attack

补充：IO操作的响应速度与协程

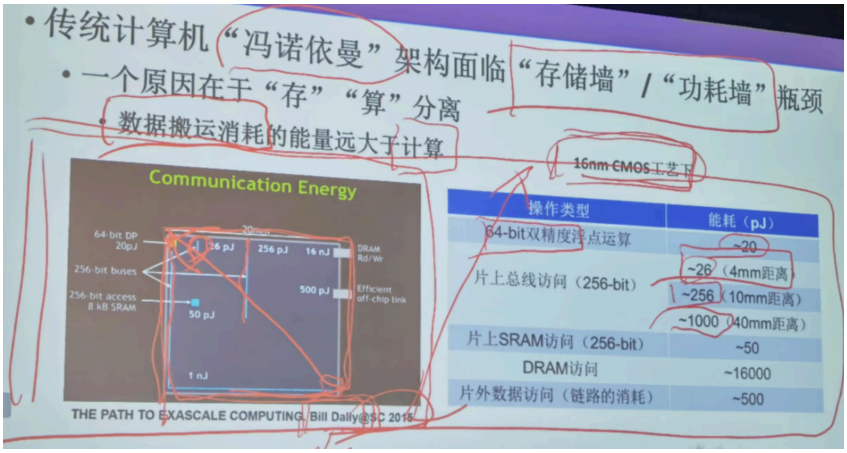
协程：可以主动切出，并在之后可恢复运行的**函数运行模式**
*协程的用途：在CPU进行IO操作时，切换到其它Ready函数的**执行现场**，并在合适时间点切换回来*

线程/进程/协程 - 知乎

	线程	协程
切换代价	内核态下，> 1000 cycle	用户态下，≈100 cycle
调度方式	操作系统负责	用户代码负责
状态太小	TCB：KB级别	执行现场



冯诺依曼架构所面临的“存储墙”和“功耗墙”——存算分离

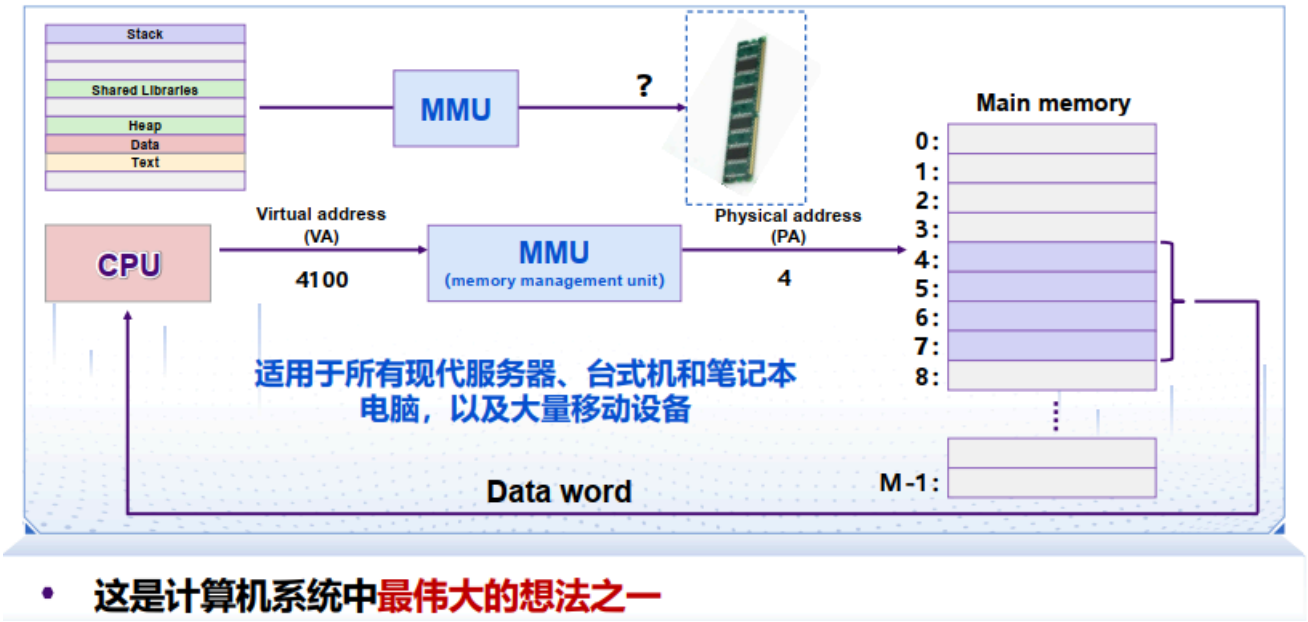


3.3 虚存

笔记

虚存是一个概念（而非实体）：一个连续的地址空间，进程的数据、代码、运行栈、堆等存于其中

> **物理地址**：**Physical** Addressing
> **虚拟地址**：**Virtual** Addressing (VM)

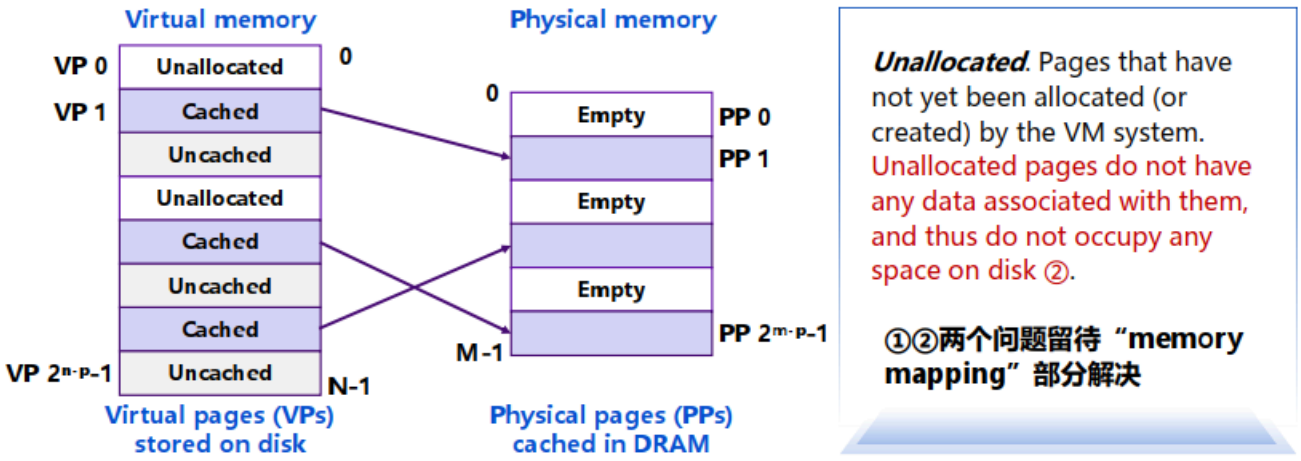


Advantages:

> 1. uses main memory efficiently
>   use **DRAM** as a **cache** for the parts of a **virtual address space**
> 2. simplifies memory management
>   each process gets the same **uniform linear address space**
> 3. isolates address spaces
>   one process can't interfere with another's memory
>   user program cannot access privileged kernel information

1. VM用于缓存

Virtual memory is an array of N contiguous bytes stored on **disk**.
The contents of the array on disk are cached in **physical memory (DRAM cache)**
These cache blocks are called **pages** (size is $P = 2^p$ bytes)



Consequences

- **Large page (block) size: typically 4-8 KB, sometimes 4 MB or larger**
- **Fully associative（全相联）**
  - Any virtual page (VP) can be placed in any physical page (PP)
  - Requires a "large" mapping function
- **Highly sophisticated, expensive replacement algorithms （较复杂的替换算法不适宜用硬件完成）**
  - Too complicated and open-ended to be implemented in hardware

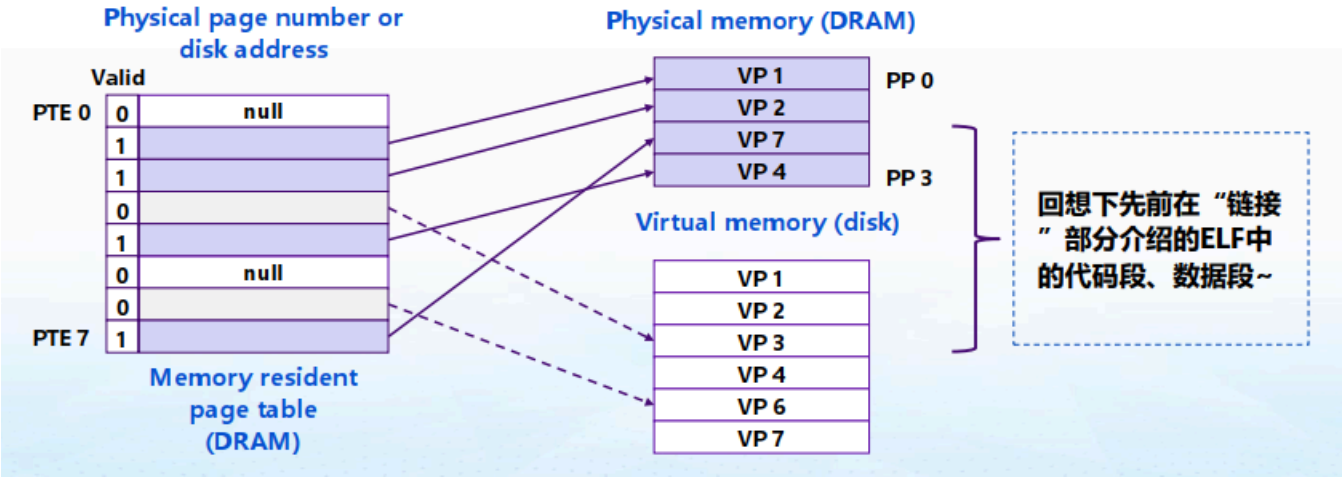
> large page (block) size: typically 4-8 KB, sometimes 4 MB or larger

Page Tables / 页表: an array of **page table entries (PTEs)** that maps virtual pages to physical pages
per-process **kernel** data structure in DRAM

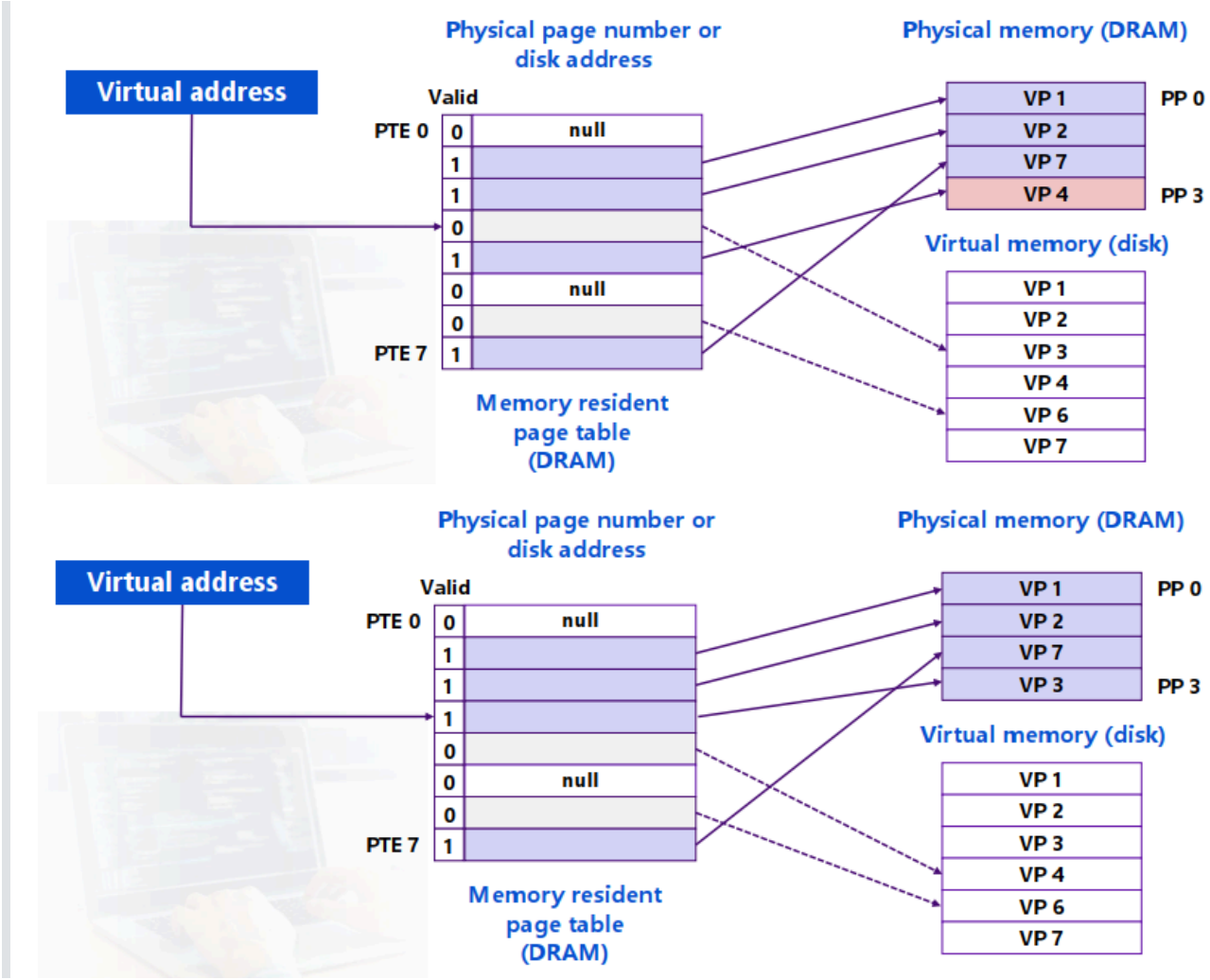
> $valid\ bit + nbit\ address\ field$
> "whether cached" + "DRAM offset"

A *page table* is an array of page table entries (PTEs) that maps virtual pages to physical pages. （页表：映射虚拟地址到物理地址）

- Per-process kernel data structure in DRAM （每一个进程拥有这样一个关键数据结构）



- **Page Hit / 命中**: reference to VM word that is in physical memory
- **Page Fault / 页缺失**: reference to VM word that is **not** in physical memory
  page miss causes **page fault** (an exception)
  Page fault handler selects a victim to be evicted.



- **Allocating Pages**: 一个Physical Page可能对应任意一个Virtual Page（全相联 fully associative）

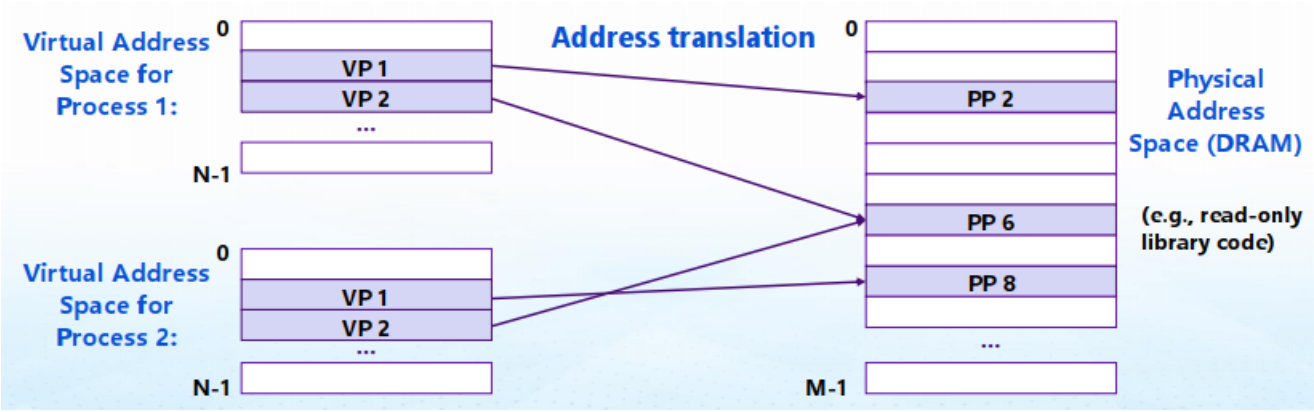
工作原理:

> 1. Virtual memory works because of locality（数据/指令的局部性）
> 2. At any point in time, programs tend to access a set of active virtual pages called the **working set** （活跃页面的集合称为工作集）
> 3. If (working set size < main memory size): Good performance for one process after compulsory misses
> 4. If (SUM(working set sizes) > main memory size) : (**Thrashing**) Performance meltdown where pages are swapped (copied) in and out continuously

2. VM 作为内存管理工具

每个进程都有自己的私有虚拟空间

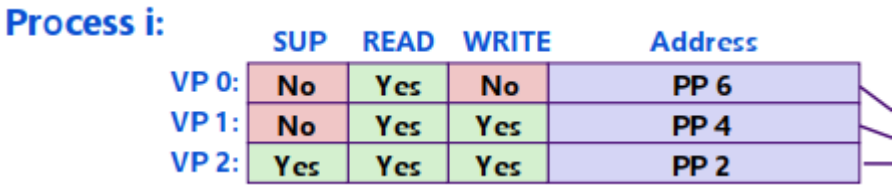
> Mapping function scatters addresses through physical memmory
> 同一块物理地址可以被不同的virtual address映射——节省空间 / sharing code and data among processes



3. VM 作为内存保护工具

> Extend PTEs with **permission bits** （PTE: page table entry）
> Page fault handler checks these before remapping
> If violated, send process SIGSEGV (segmentation fault)

不同的权限分配：



- SUP bit: whether processes must be running in kernel (supervisor) mode to access a page
- READ bit: read access to the page
- WRITE bit: write access to the page

> violating consequence: the kernel exception handler sends a SIGSEGV signal to the offending process (Linux reports this as a *segmentation fault*)

3.4 虚存地址翻译/转换

Notation

PTE: Page table entry (value, *PTEA), 存在内存里
$PTEA$: Page table entry address (key, 地址的概念), 内存地址
$PTBR$: Page table base register
TLB(快表): Translation lookaside buffer

1. Basic Parameters

   $N = 2^n$: Number of addresses in VAS (virtual address space), Must be a power of 2
   $M = 2^m$: Number of addresses in PAS (physical address space), Need not be a power of 2
   $P = 2^p$: Page size (bytes)
   $T = 2^t$: the sets that TLB has

2. Components of VA

VPO: Virtual page offset (bytes)
VPN: Virtual page number
$TLBL$: TLB index
$TLBT$: TLB tag

3. Components of PA

PPO: Physical page offset (same as VPO)
PPN: Physical page number
CO: Byte offset within cache block
CI: Cache index
CT: Cache tag

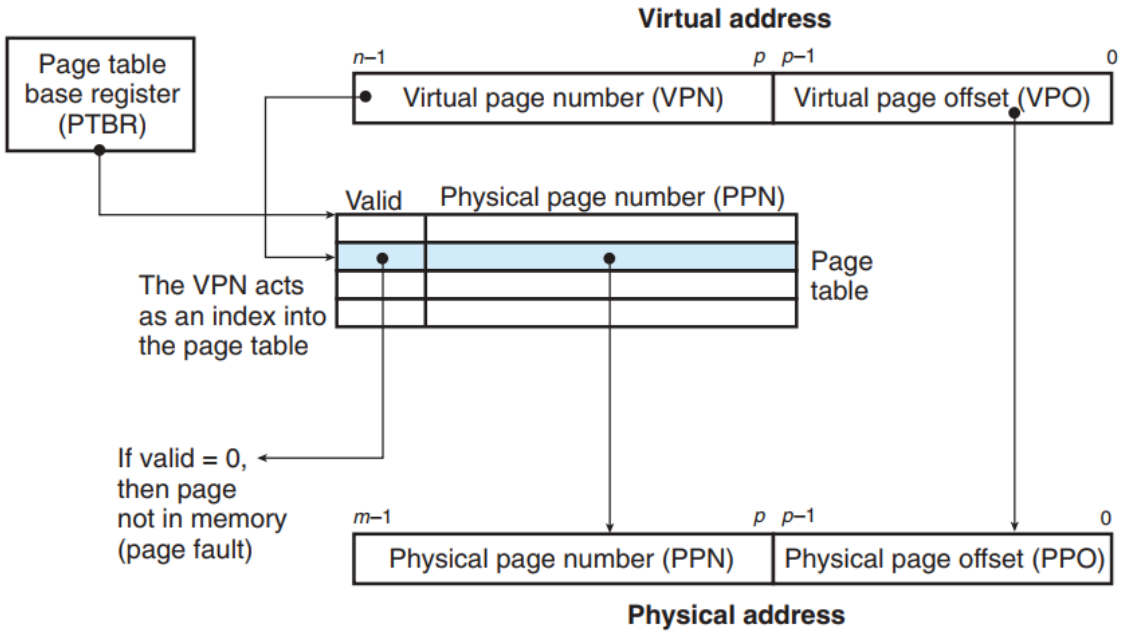
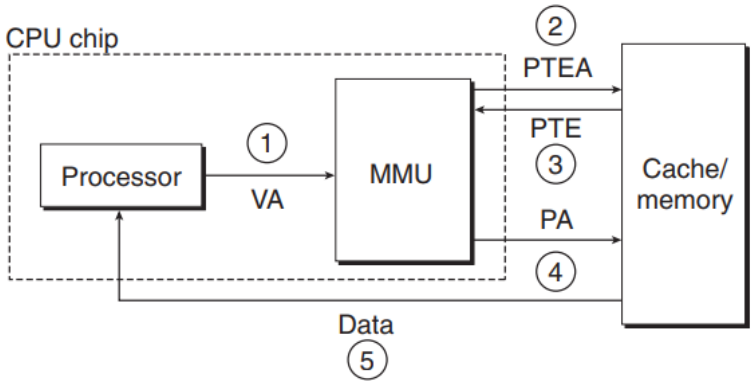


Figure 9.12 Address translation with a page table.

1. Page Hit

page hit is handled entirely by hardware

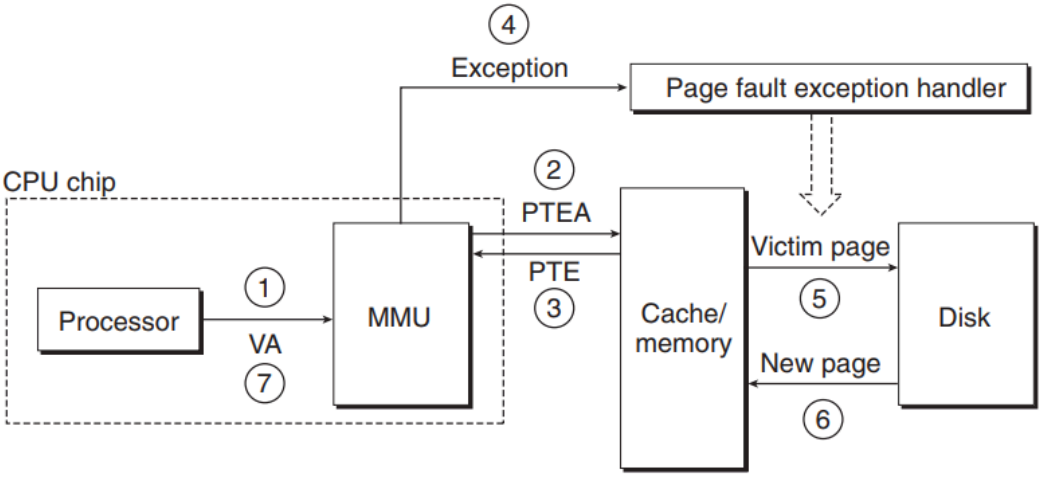


(a) Page hit

2. Page Fault

page fault requires cooperation between hardware and the operating system kernel

- MMU triggers an exception which transfers control in the CPU to a **page fault exception handler** in the operating system kernel
- the fault handler identifies a **victim page** in physical memory and pages it out to disk
- and pages in the new page and updates the PTE in memory
- returns to the orginial process and results in **page hit**



(b) Page fault

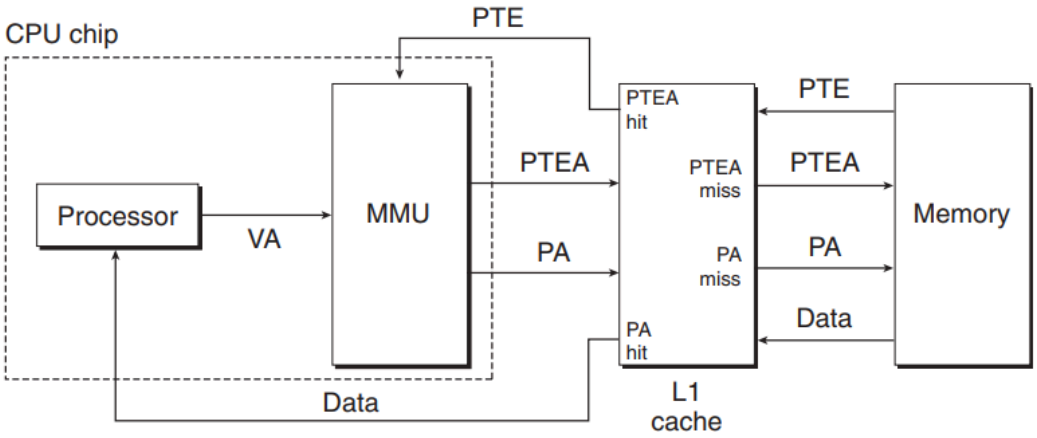
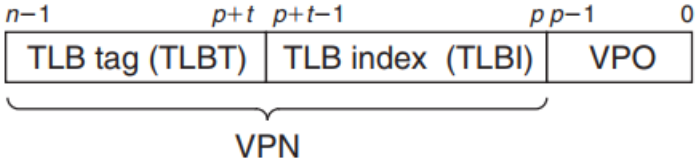


Figure 9.14 **Integrating VM with a physically addressed cache.** VA: virtual address. PTEA: page table entry address. PTE: page table entry. PA: physical address.

3. TLB (Translation Lookaside Buffer)

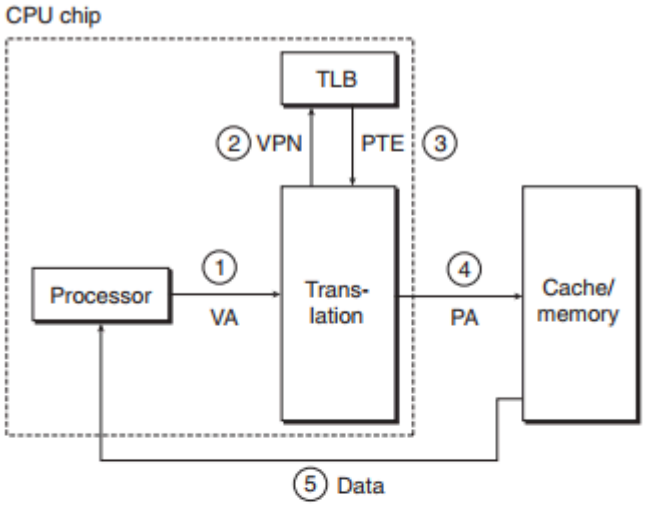
> Small hardware cache in MMU
> Contains complete page table entries for **small number** of pages
> Maps virtual page numbers to physical page numbers

- Components of a VPN that are used to access the TLB

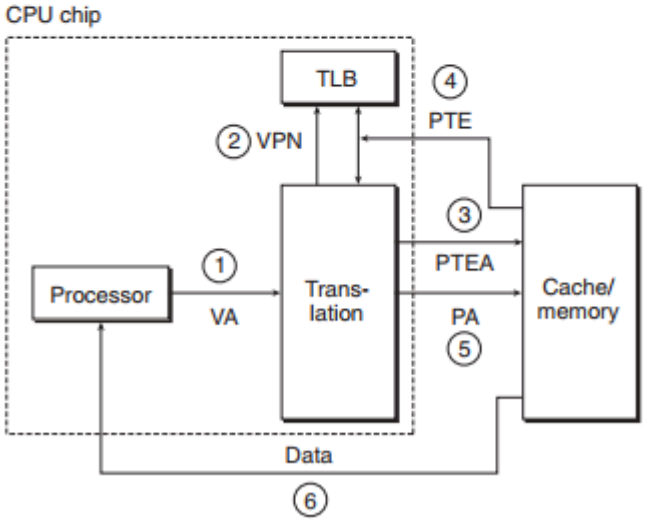


- TLB hit and TLB miss

  TLB is **4-set**（唯一的） and **4-way**（任意的） set associative with 16 total entries.

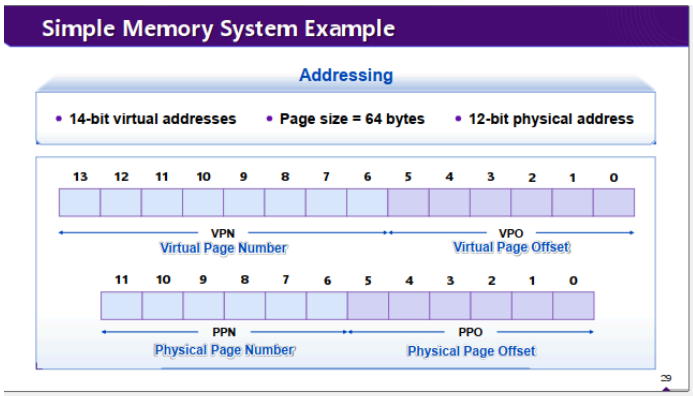


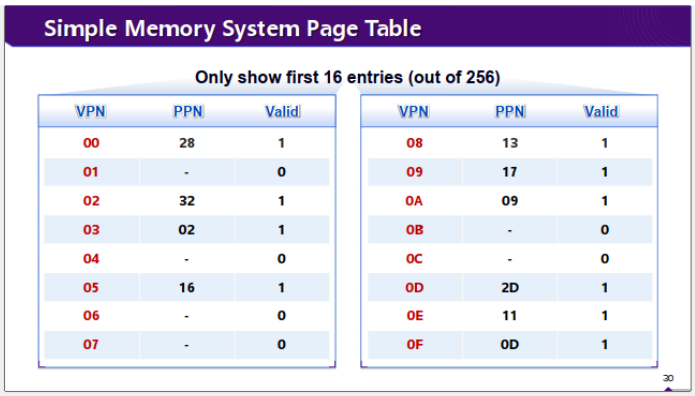
(a) TLB hit

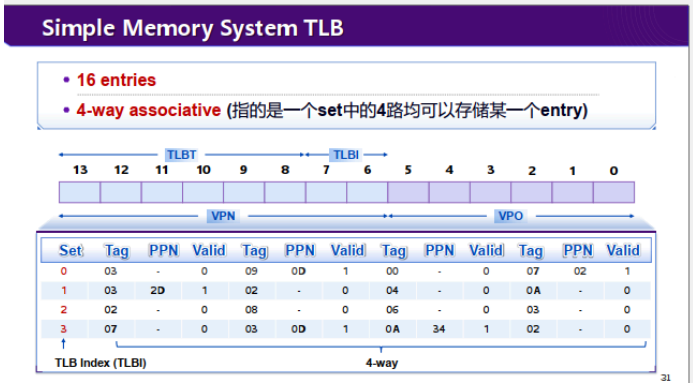


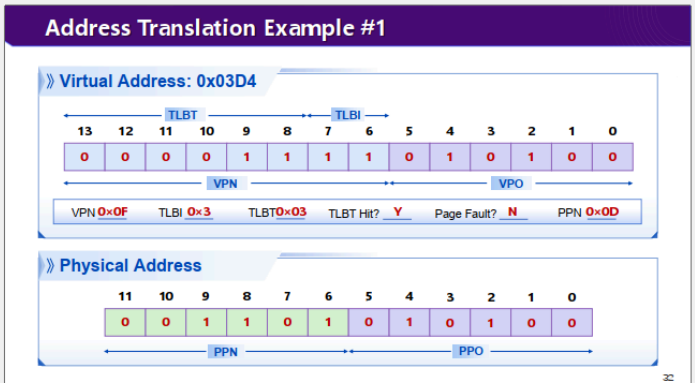
(b) TLB miss

[EG]:



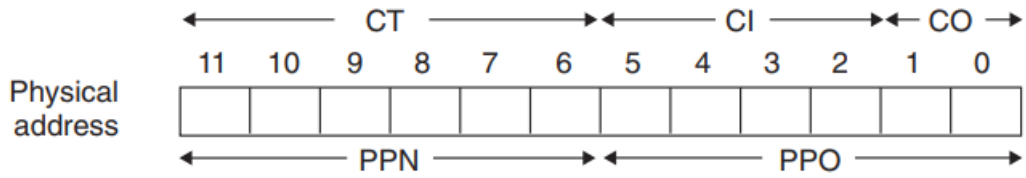






4. Multilevel page table

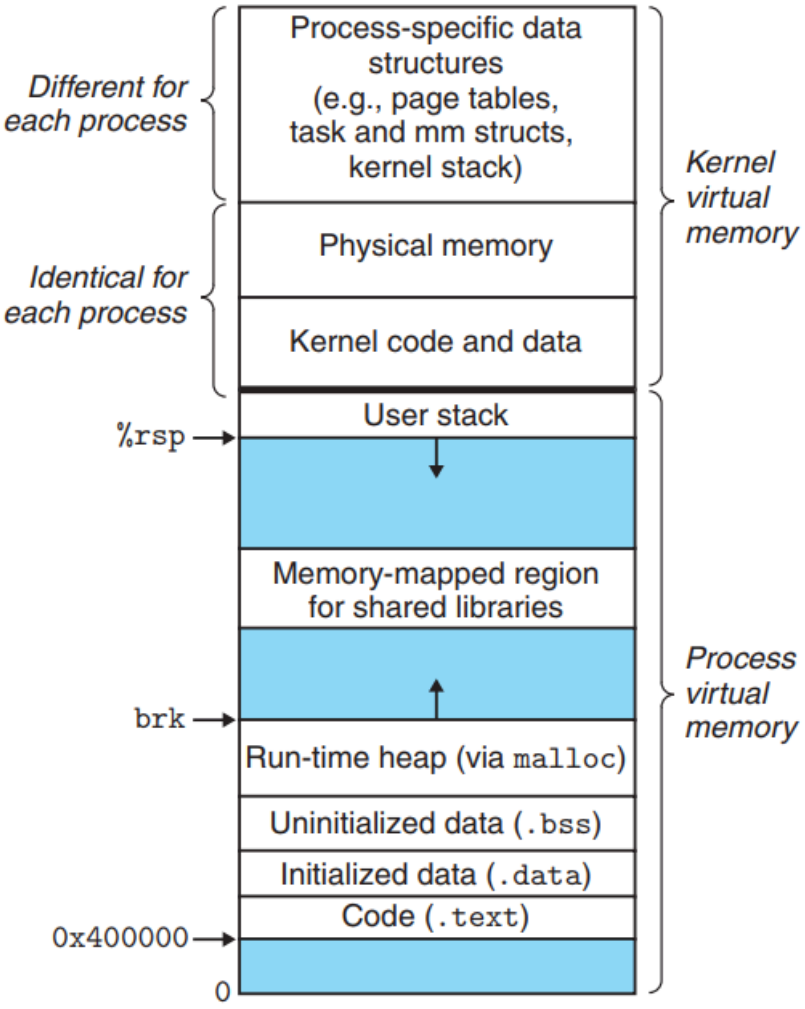
5. Cache



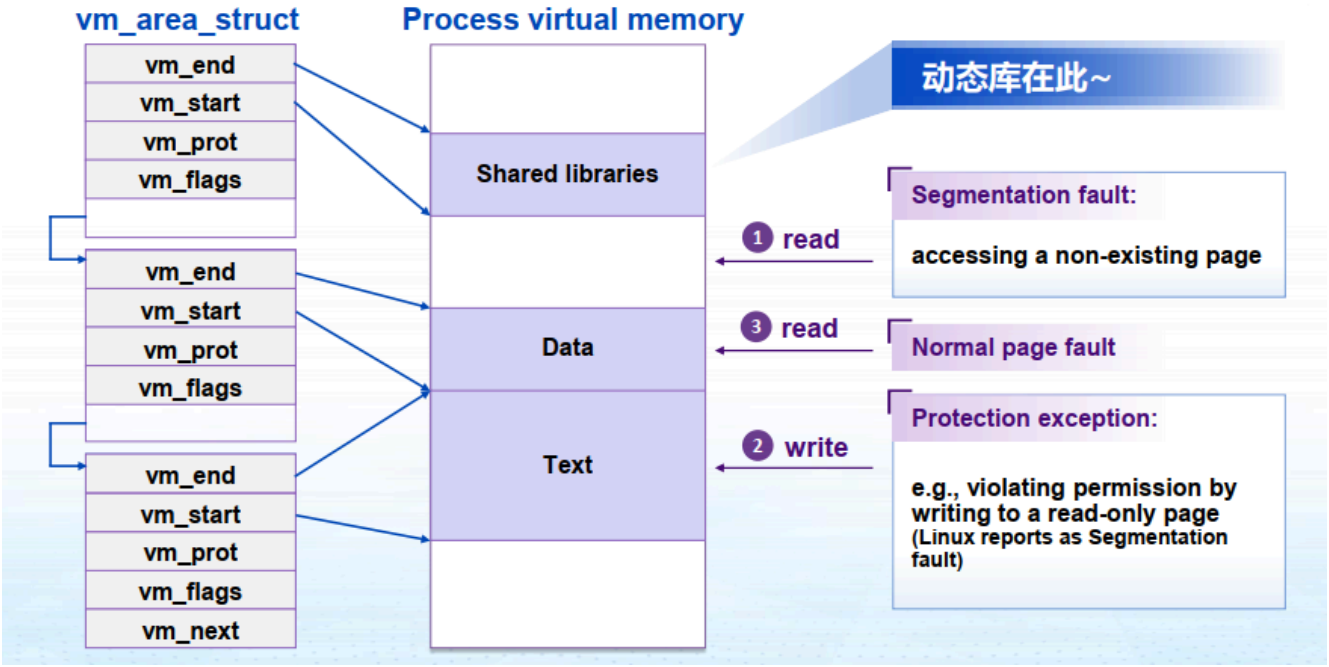
3.5 内存映射 memory mapping

Linux Virtual Memory System

1. The virtual memory of a Linux process



2. Linux Page Fault Handling



Memory Mapping

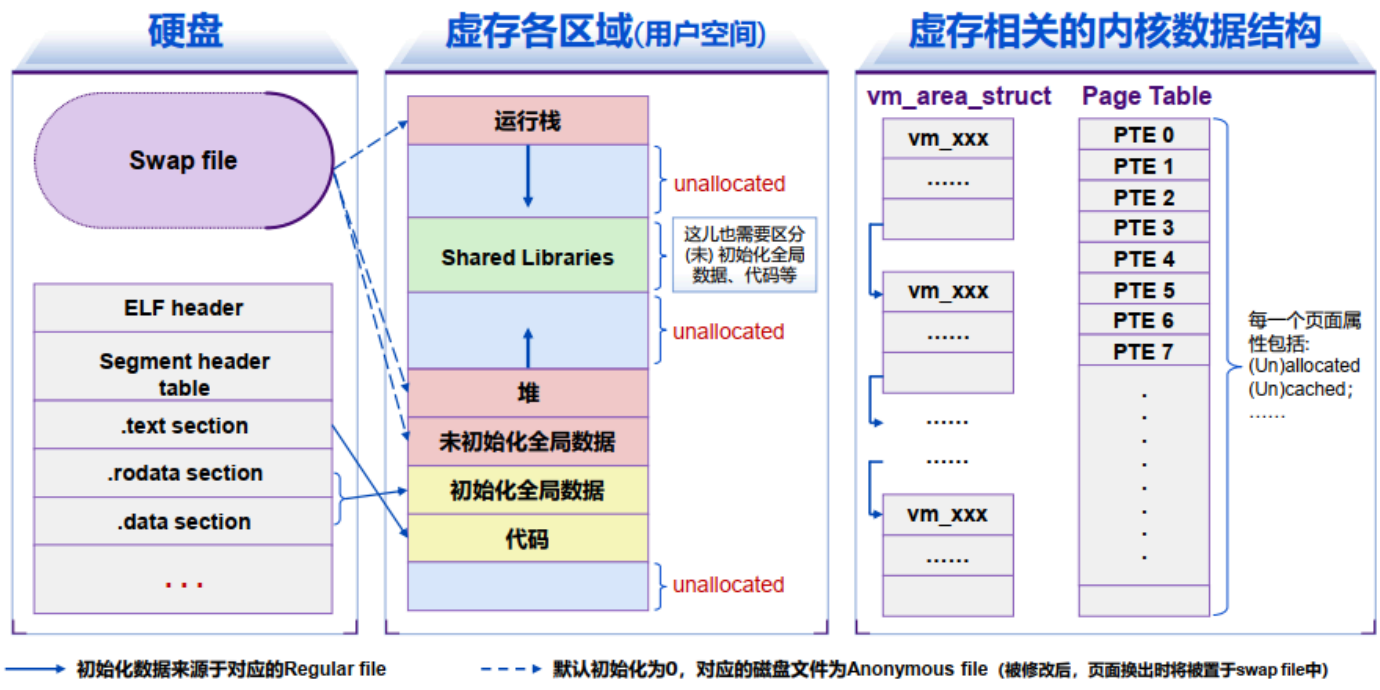
Area can be backed by...

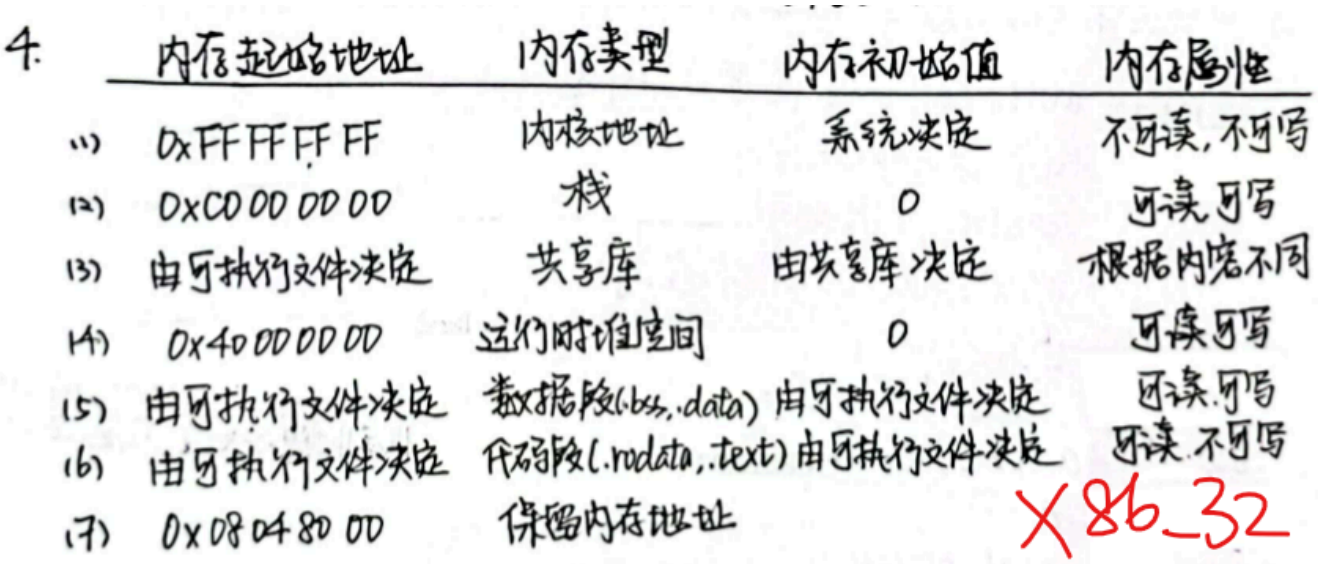
- **Regular File** on disk: initial page bytes come from a section of a file
- **Anonymous File** (nothing): first fault will allocate a physical page full of 0's (**demand-zero page**), once the page is written to (**dirtied**), it is like any other page
  Dirty pages are copied back and forth between memory and a special **swap file**

Linux initializes the contents of a virtual memory area by associating it with an *object* on disk, a process known as *memory mapping*. Areas can be mapped to one of two types of objects:

1. *Regular file in the Linux file system:* An area can be mapped to a contiguous section of a regular disk file, such as an executable object file. The file section is divided into page-size pieces, with each piece containing the initial contents of a virtual page. Because of demand paging, none of these virtual pages is actually swapped into physical memory until the CPU first *touches* the page (i.e., issues a virtual address that falls within that page's region of the address space). If the area is larger than the file section, then the area is padded with zeros.
2. *Anonymous file:* An area can also be mapped to an anonymous file, created by the kernel, that contains all binary zeros. The first time the CPU touches a virtual page in such an area, the kernel finds an appropriate victim page in physical memory, swaps out the victim page if it is dirty, overwrites the victim page with binary zeros, and updates the page table to mark the page as resident. Notice that no data are actually transferred between disk and memory. For this reason, pages in areas that are mapped to anonymous files are sometimes called *demand-zero pages*.

In either case, once a virtual page is initialized, it is swapped back and forth between a special *swap file* maintained by the kernel. The swap file is also known as the *swap space* or the *swap area*. An important point to realize is that at any point in time, the swap space bounds the total amount of virtual pages that can be allocated by the currently running processes.





Shared Objects

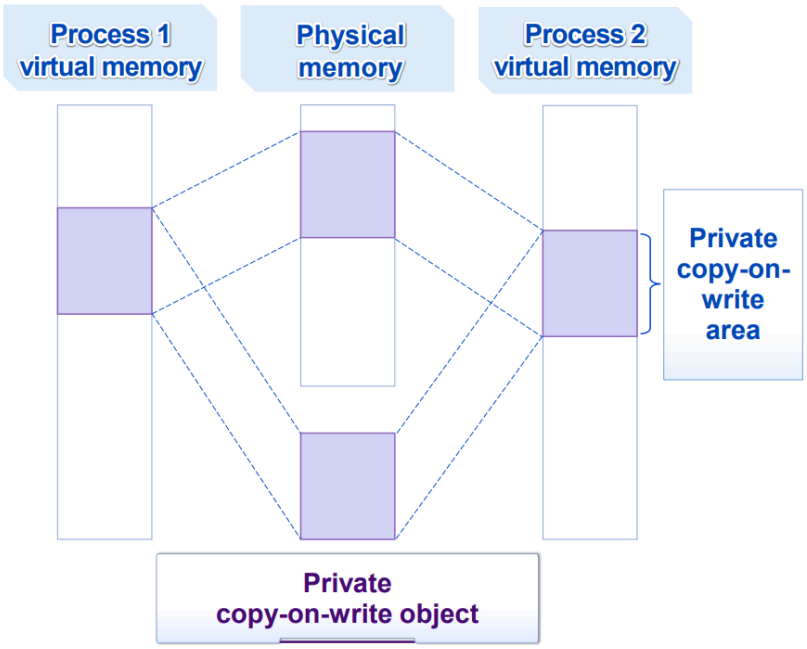
- **Shared Object**: two virtual memory Processes can share the same Physical memory object

> Process 1 maps the shared object.
> Process 2 maps the same shared object. (Note that the physical pages are not necessarily contiguous.)
>
> Since each object has a unique filename, the kernel can quickly determine that process 1 has already mapped this object and can point the page table entries in process 2 to the appropriate physical pages. The key point is that only a single copy of the shared object needs to be stored in physical memory, even though the object is mapped into multiple shared areas. For convenience, we have shown the physical pages as being contiguous, but of course this is not true in general.

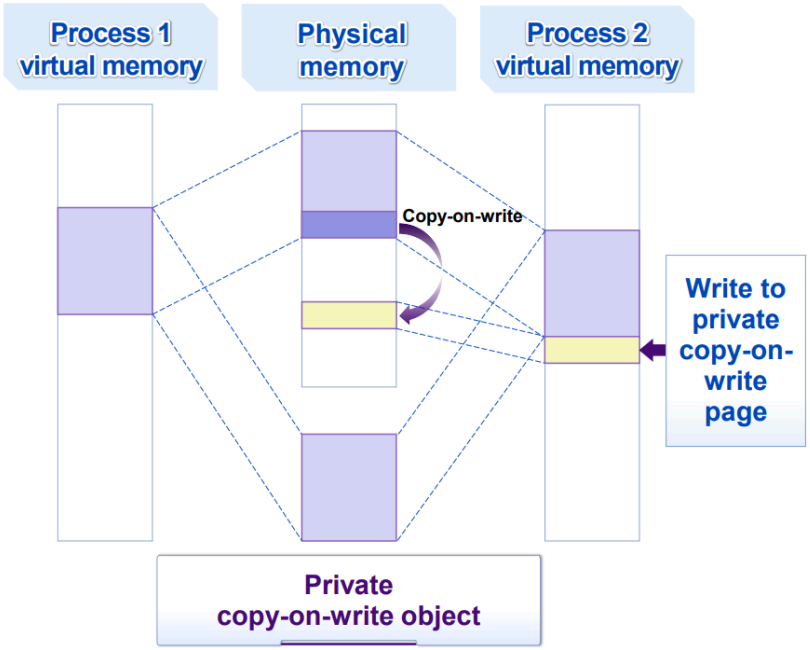
An object can be mapped into an area of virtual memory as either a *shared object* or a *private object*. If a process maps a shared object into an area of its virtual address space, then any writes that the process makes to that area are visible to any other processes that have also mapped the shared object into their virtual memory. Further, the changes are also reflected in the original object on disk.

Changes made to an area mapped to a private object, on the other hand, are not visible to other processes, and any writes that the process makes to the area are *not* reflected back to the object on disk. A virtual memory area into which a shared object is mapped is often called a *shared area*. Similarly for a *private area*.

- **Private Copy-on-Write (COW) Objects**:



- Two processes mapping a *private copy-on-write (COW)* object.
- Area flagged as private copy-on-write
- PTEs in private areas are flagged as read-only



- Instruction writing to private page triggers protection fault.
- Handler creates new R/W page.
- Instruction restarts upon handler return.
- Copying deferred as long as possible!

在虚存机制下

地址转换指的是处理器将__A__（A. 虚拟地址 B.物理地址）转换为__B__（A. 虚拟地址 B.物理地址），这一转换过程中，地址部分的__A__（A. 页内地址 B.页号）不变；在这一转换过程中先查找__B__（A.内存页表 B.快表），如果这一查找缺失，则再查找__A__（A.内存页表 B.快表）

TLB可以认为是处理器的内存管理模块（MMU）的一部分

对于虚存，它的主要作用是__A__（A.加速 B.扩容）。对一个8-set，4-way的TLB而言，一个页表项可以存于其__A__（A.唯一的 B.任一个）set中，一旦set确定后，则可以存于其__B__（A.唯一的 B.任一个）way中

一个C语言编写的执行文件刚被启动

它的memory layout中正文段所对应的磁盘文件的相应部分是__D__，初始化全局数据段所对应的磁盘文件的相应内容是__A__，栈（未被写入前）所对应的磁盘文件是__B__。如果堆上某块虚存空间被分配且被写入，那么其可能对应的磁盘文件是__B/C__

A. 该执行文件的数据段 B. Anonymous file C. swap file(或分区) D. 该执行文件的代码段 E.不知道

execve

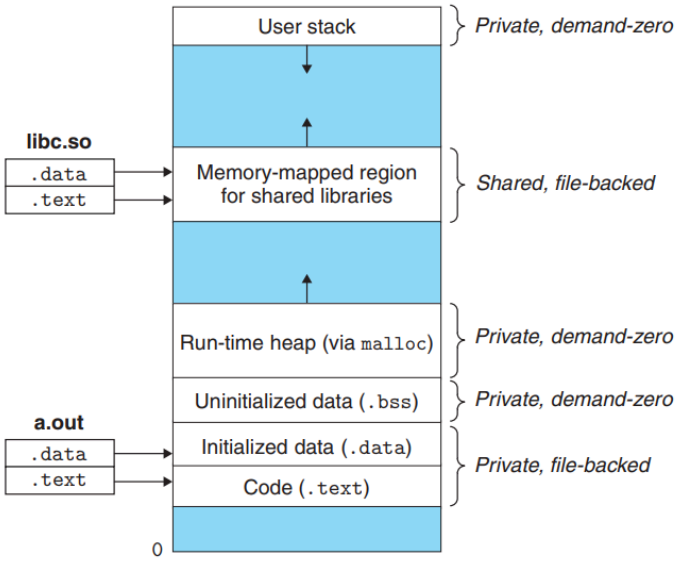


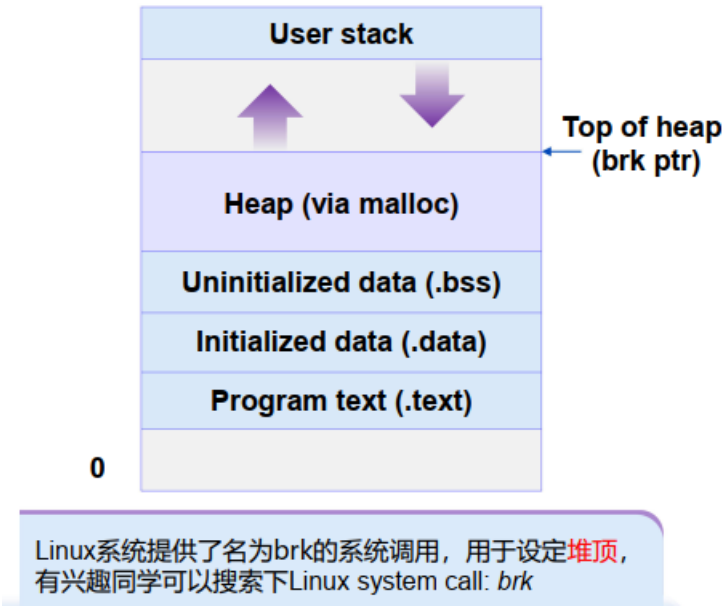
Figure 9.31
How the loader maps the areas of the user address space.

1. *Delete existing user areas.* Delete the existing area structs in the user portion of the current process's virtual address.
2. *Map private areas.* Create new area structs for the code, data, bss, and stack areas of the new program. All of these new areas are private copy-on-write. The code and data areas are mapped to the `.text` and `.data` sections of the `a.out` file. The bss area is demand-zero, mapped to an anonymous file whose size is contained in `a.out`. The stack and heap area are also demand-zero, initially of zero length. Figure 9.31 summarizes the different mappings of the private areas.
3. *Map shared areas.* If the `a.out` program was linked with shared objects, such as the standard C library `libc.so`, then these objects are dynamically linked into the program, and then mapped into the shared region of the user's virtual address space.
4. *Set the program counter (PC).* The last thing that `execve` does is to set the program counter in the current process's context to point to the entry point in the code area.

3.6 内存分配

用户层的动态内存分配

- **动态内存分配** dynamic memory allocators (such as *malloc*)，用来获取VM空间
- Dynamic memory allocators manage an area of process VM known as the **heap** (堆)



- Allocator maintains heap as collection of variable sized **blocks**, which are either **allocated** or **free**

> Types of allocators:
>
> - **Explicit allocator**: `malloc` and `free` in C（库通过`brk`等系统调用开辟heap空间，在其中初始化了相应地数据机构来实现动态内存分配）
>
> ```
> xxxxxxxxxx
> int *p = (int*) malloc(n * sizeof(int)); // allocate a block
> if (p == nullptr) {
>     perror("malloc");
>     exit(0);
> }
> for (int i = 0; i < n; ++i) p[i] = i;
> free(p); // return p to the heap
> ```
>
> - **Implicit allocator**: garbage collection in Java, ML, and Lisp

- **内存分配器**——一些要求

任意大小的block，高速，无冲突，对齐要求（8 bytes alignment for GNU `malloc`），返回的是指针（一旦分配不能移动）

Performance Goal

- **吞吐率 / Throughput**：单位时间内完成的请求数
- **峰值内存利用率 / Peak memory utilization**

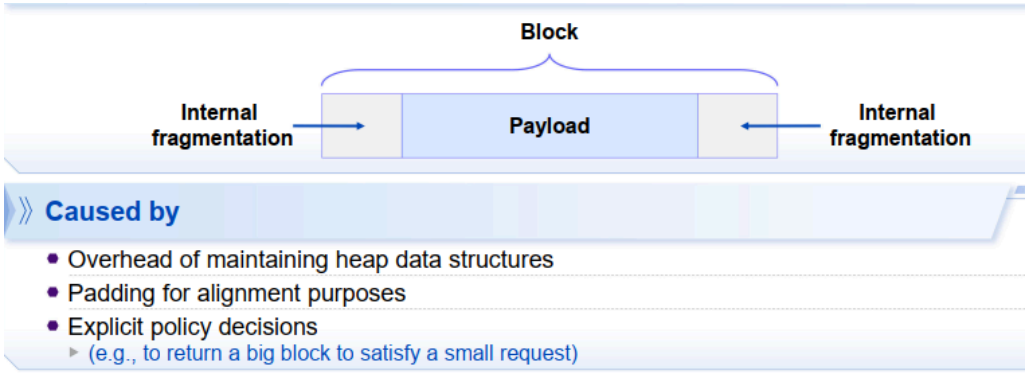
> **Aggregate payload** P_k is the sum of currently allocated payloads
> **Current heap size** H_k, assumed monotonically nondecreasing
> **Peak memory utilization** (after k requests) $U_k = max_{i<k} P_i / H_k$

碎片化问题

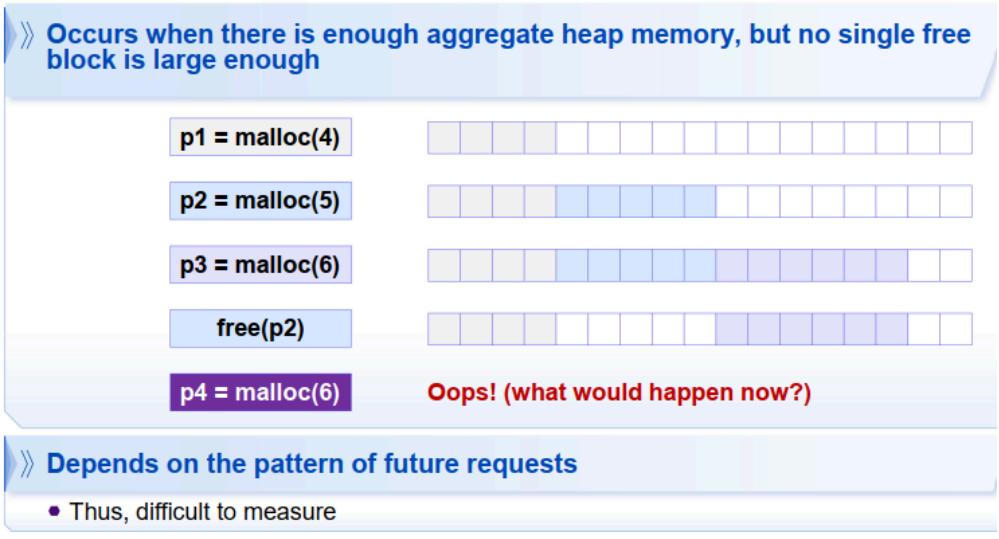
*Poor memory utilization caused by **fragmentation***

1. 内部碎片化 internal fragmentation

> payload is smaller than block size

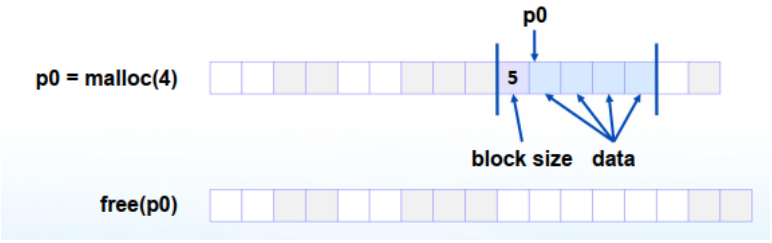


2. 外部碎片化 external fragmentation

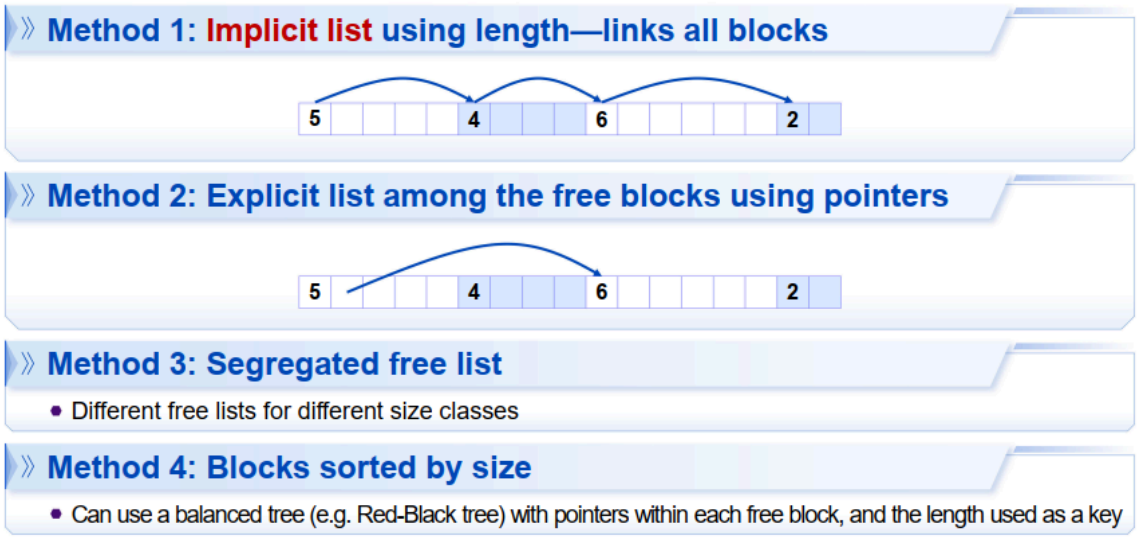


3. header field / header

> Thus we know how many memory to free given just a pointer.

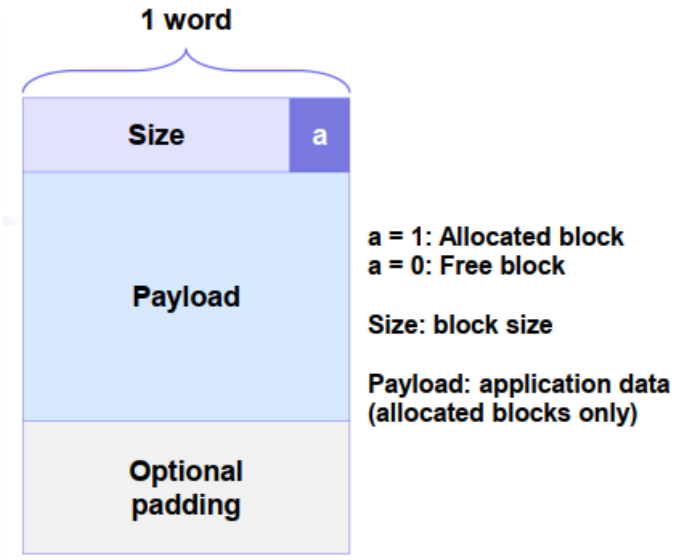


- Implicit list 链接所有的blocks
- Explicit list 连接free blocks using 指针



4. Implicit List 实现

1) Finding a Free Block



> First Fit

```
xxxxxxxxxx

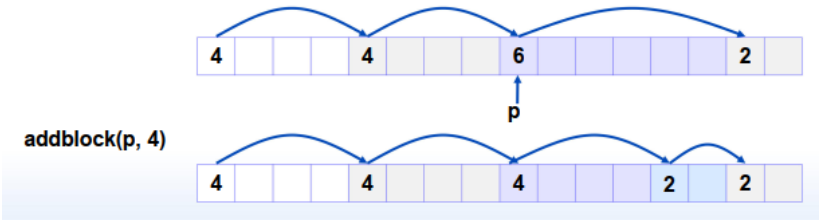
p = start;

while ((p < end)  // not passed end
```

```
    && (*p & 1)  // already allocated
    || (*p <= len)) // too small
p = p + (*p & -2) // goto next block (word addressed)
```

Next Fit: search list starting where previous search finished
Best Fit: choose the best free block (fits, with fewest bytes left over)

2) Splitting

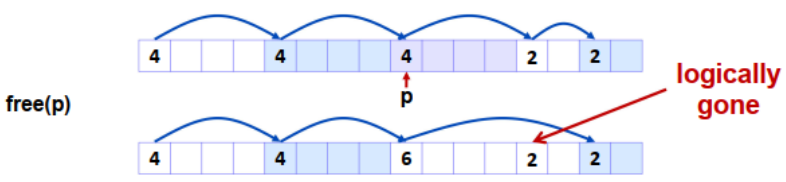



```
xxxxxxxxxx
void addblock(ptr p, int len) {
    int newsize = ((len + 1) >> 1) << 1; // round up to even
    int oldsize = *p & -2; // mask out low bit
    *p = newsize | 1; // set new length
    if (newsize < oldsize)
        *(p + newsize) = oldsize - newsize; // set length in remaining part of block
}
```

3) Freeing a Block

```
xxxxxxxxxx
void free_block(ptr p) {
    *p = *p & -2; // need only clear the "allocated" flag
}
```

Join with next/previous blocks:




```
xxxxxxxxxx
```

```
void free_block(ptr p) {

    *p = *p & -2; // clear allocated flag

    next = p + *p;

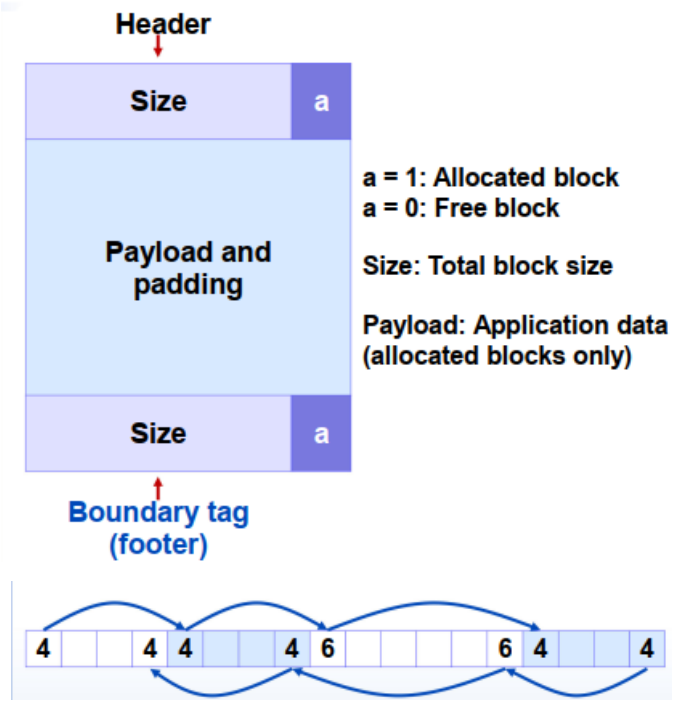
    if ((*next & 1) == 0) // if next block isn't allocated

        *p = *p + *next; // add to this block

}
```

Bidirectional Coalescing / 双向

> **Boundary Tags [Knuth73]**



> 然而Boundary Tages会产生内部碎片。

4) Other Policies

- Immediate coalescing / Deferred coalescing

4 异常控制流 ECF

Code Example

异常：各个任务切换的重要机制

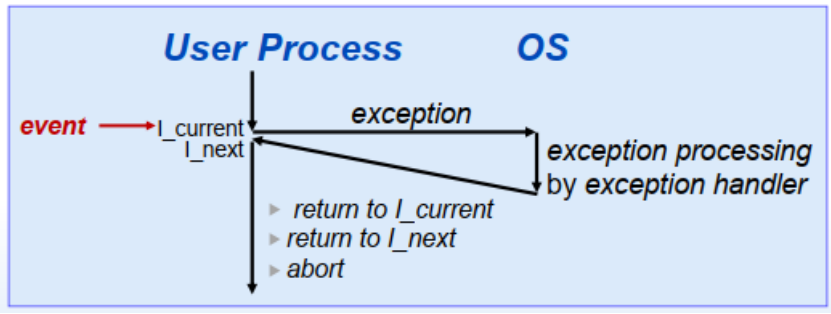
Changing Control Flow:

- **program state**: jumps and branches, call and return
- **system state**: exceptional control flow
  low level mechanisms: hardware & OS software
  higher level mechanisms: OS software (process context switch, signals, nonlocal jumps...)

Exceptions

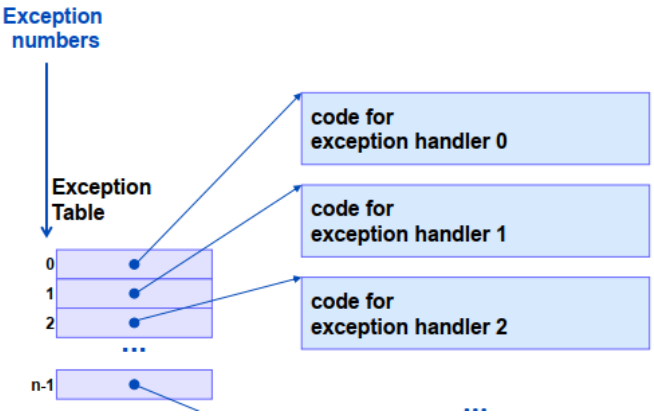
1. **Exceptions**: transfer of control to the OS in response to some event
   (从用户态到核心态 user => kernel)

> Examples: div by 0, arithmetic overflow, page fault, I/O request completes, Ctrl-C...
>
> processor pushses additional processor state onto the stack necessary to restart the interrupted program when the handler returns
>
> Control transferred from user program to the kernel: items pushed on the **kernel's stack** rather than onto the user's stack



2. **Interupt Vectors**（中断向量/异常处理向量）

> each type of event has unique exception number k
> handler k is called each time exception k occurs
>
> **exception table**: used to form the address of the appropriate exception handler
> **exception table base register**: contains the starting address of the exception table



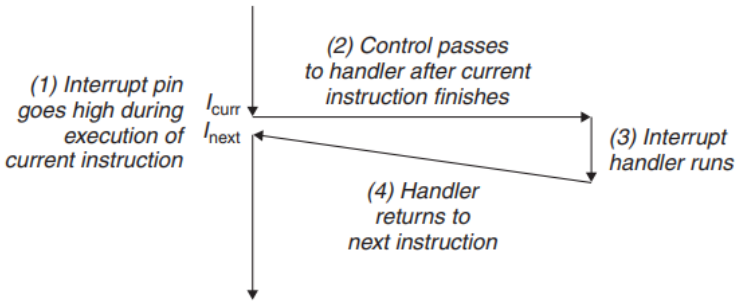
Class	Cause	Async/sync	Return behavior
Interrupt	Signal from I/O device	Async	Always returns to next instruction
Trap	Intentional exception	Sync	Always returns to next instruction
Fault	Potentially recoverable error	Sync	Might return to current instruction
Abort	Nonrecoverable error	Sync	Never returns

Exception number	Description	Exception class
0	Divide error	Fault
13	General protection fault	Fault
14	Page fault	Fault
18	Machine check	Abort
32–255	OS-defined exceptions	Interrupt or trap

Figure 8.9 Examples of exceptions in x86-64 systems.

3. **Asynchronous Exceptions**（异步异常/中断/Interrputs）

> Caused by events external to the processor（外设触发）
> indicated by setting the processor's **interrupt pin**
> handler returns to "next" instruction



Examples:

- I/O interrputs
  hitting Ctrl-C at the keyboard
  arrival of a packet from a network
  arrival of data from a disk
- Hard reset interrupt
  hitting the reset button
- Soft rest interrupt
  hitting Ctrl-Alt-Delete on a PC

4. **Synchronous Exceptions**（同步异常）

> Caused by events that occur as a result of executing an instruction（由指令执行引起）. This instruction is referred as the **faulting instruction**

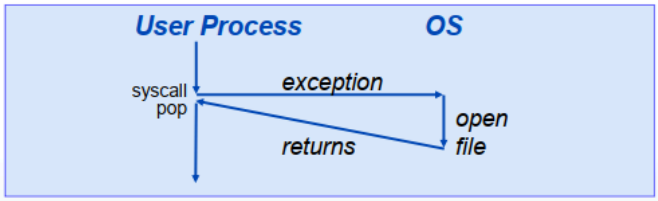
- **Traps**
  Intentional（程序故意引起的）: system calls, breakpoint traps, special instructions
  Returns control to "next" instruction

  > reading a file *(read)*
  > creating a new process *(fork)*
  > loading a new program *(execve)*
  > terminating the current process *(exit)*

User calls: **open(filename, options)**

Function **open** executes system call instruction, *syscall*

```
0804d070 <__libc_open>:
 . . .
804d082:       0f 05                   syscall
804d084:       5b                      pop    %ebx
 . . .
```



OS must find or create file, get it ready for reading or writing

Returns integer file descriptor

- **Faults**

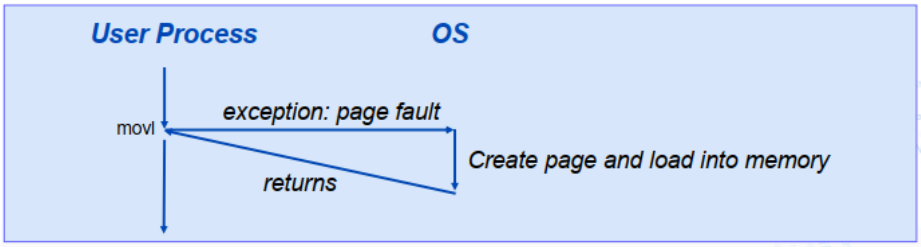
  Unintentional but possibly recoverable（程序无意引起的）: page faults (recoverable), protection faults (unrecoverable), floating point excetions

  Either **re-executes** faulting ("current") instruction or **aborts**

- User writes to memory location
- That portion (page) of user's memory is currently on disk

```
int a[1000];
main ()
{
   a[500] = 13;
}
```

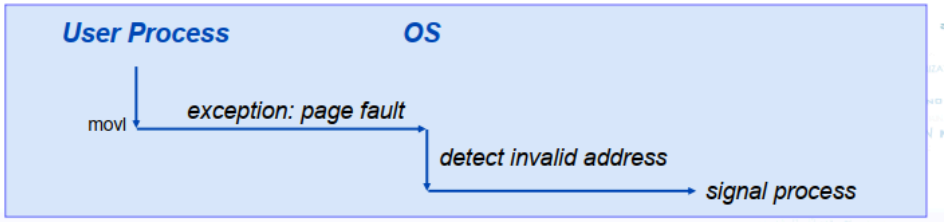
```
80483b7:    c7 05 10 9d 04 08 0d    movl   $0xd,0x8049d10
```



- Page handler must load page into physical memory
- Returns to faulting instruction
- Successful on second try

```
int a[1000];
main ()
{
   a[5000] = 13;
}
```

```
80483b7:    c7 05 60 e3 04 08 0d    movl   $0xd,0x804e360
```



- Page handler detects invalid address
- Sends SIGSEGV signal to user process
- User process exits with "segmentation fault"

- **Aborts**

  Unintentional and unrecoverable: parity error, machine check

  Aborts current program

syscall

> System calls are provided on x86-64 systems via a trapping instruction called syscall. It is quite interesting to study how programs can use this instruction to invoke Linux system calls directly. All arguments to Linux system calls are passed through general-purpose registers rather than the stack. By convention, register %rax contains the syscall number, with up to six arguments in %rdi, %rsi, %rdx, %r10, %r8, and %r9. The first argument is in %rdi, the second in %rsi, and so on. On return from the system call, registers %rcx and %r11 are destroyed, and %rax contains the return value. A negative return value between −4,095 and −1 indicates an error corresponding to negative errno.

Number	Name	Description	Number	Name	Description
0	`read`	Read file	33	`pause`	Suspend process until signal arrives
1	`write`	Write file	37	`alarm`	Schedule delivery of alarm signal
2	`open`	Open file	39	`getpid`	Get process ID
3	`close`	Close file	57	`fork`	Create process
4	`stat`	Get info about file	59	`execve`	Execute a program
9	`mmap`	Map memory page to file	60	`_exit`	Terminate process
12	`brk`	Reset the top of the heap	61	`wait4`	Wait for a process to terminate
32	`dup2`	Copy file descriptor	62	`kill`	Send signal to a process

Figure 8.10 Examples of popular system calls in Linux x86-64 systems.

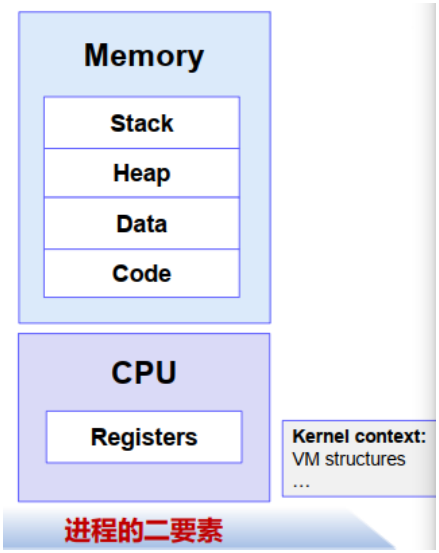
4.1 Processes（进程）

1. 基本概念

> A process is an instance of a running program（程序的一个运行实例）

Process provides each program with two **key abstractions**:

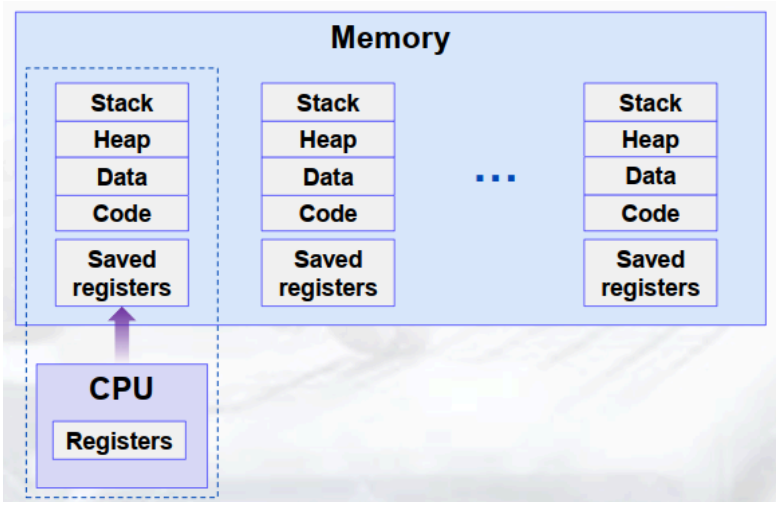
- Logical control flow: each program seems to have exclusive use of the CPU
- Private virtual address space: each program seems to have exclusive use of main memory

> maintained by... 多进程交错运行、快速切换 and 由虚拟内存系统管理地址空间
>
> 
>
>
> 1. Save current registers in memory

2. Schedule next process for execution

3. Load saved registers and switch address space (context switch)

(with one CPU core)

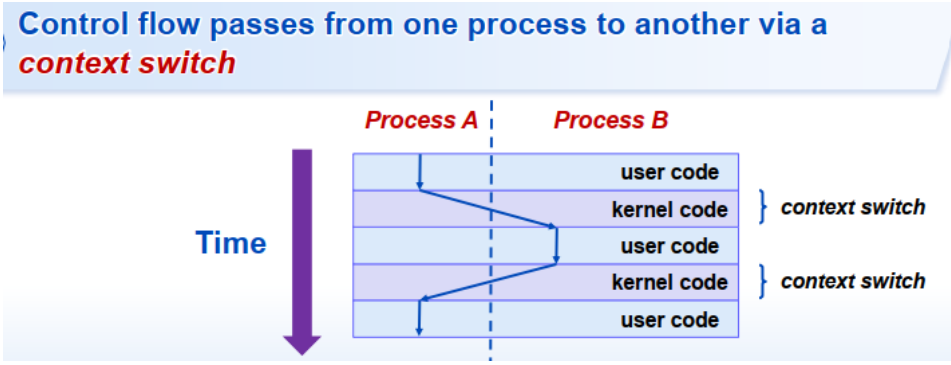


2. 并发进程 Concurrent Processes

Two processes run **concurrently** if they flows overlap in time
——Otherwise, they are **sequential**

3. 进程切换 Context Switching

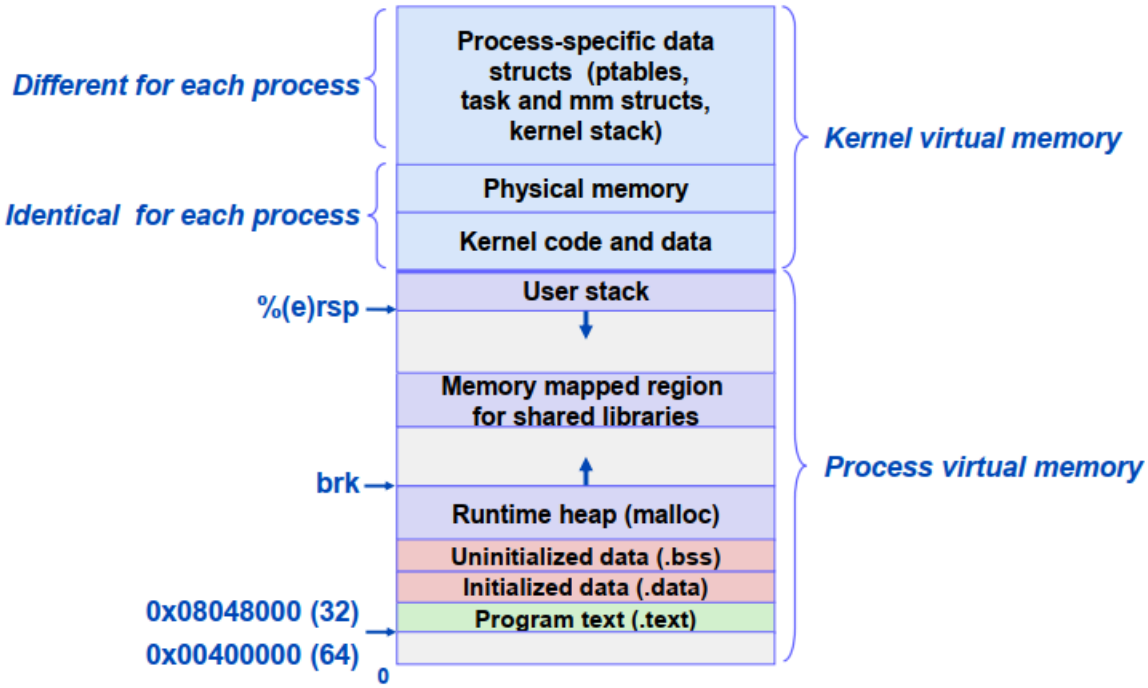
Processes are managed by a shared chunk of OS code called the **kernel**
[Important]: The kernel **is NOT** a separate process, but rather runs as part of some user process

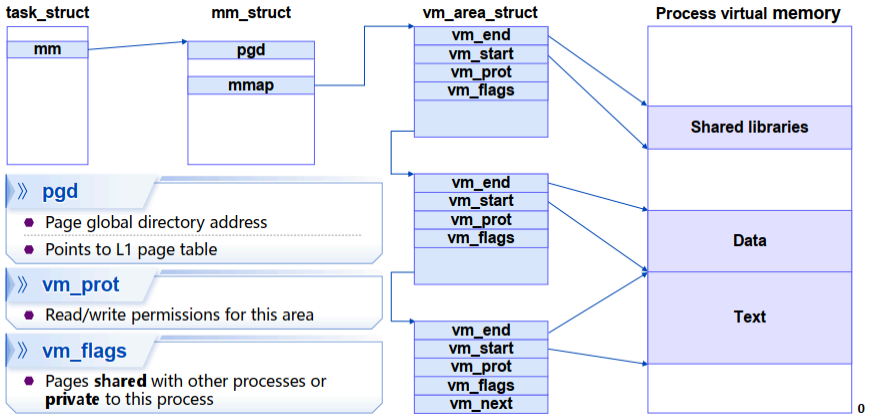


4. fork: 创建新进程

creates a new process (**child process**) that is identical to the calling process (**parent process**)
return 0 to the **child process**
return child's **pid** to the parent process

fork is called **once** but returns **twice**





1. create VM for new process

   create exact **copies** of current `mm_struct`, `vm_area_struct` and `page tables`

   flag each page in both processes as **read-only**

   flag each `vm_area_struct` in both processes as **private COW**

2. fork的机制

3. exit：终止当前进程

   `atexit()`: registers functions to be executed upon exit

```
xxxxxxxxxx

void cleanup(void) {

    printf("cleaning up\n");

}

void fork() {

    atexit(cleanup);

    fork();
```

```
    exit(0);
}
```

5. Zombies：僵尸进程

> 子进程比父进程先结束，进程终止后仍然会消耗系统资源：various tables maintained by OS

Reaping / 回收：父进程对终止的子进程进行回收（父进程获取子进程的退出状态信息），内核释放子进程的剩余资源

> 如果父进程没有回收子进程，那么子进程会被**init process**回收

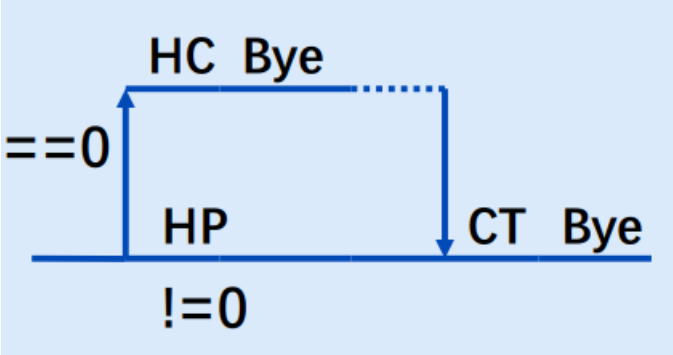
- **挂起 / wait**：synchronizing with children

```
xxxxxxxxxx

#include <sys/wait.h>

int wait(int* child_status)
```

挂起当前进程直到它的一个子进程终止（如果没有子进程，直接返回-1）
返回值是终止的子进程的pid
if child_status != NULL, then the object it points to will be set to a status indicating why the child process terminated（获得子进程终止的原因，正常终止的话是子进程的退出状态值）——`void exit(int status)`




```
xxxxxxxxxx

pid_t wait(int& status); // random pid

// == waitpid(-1, status, 0)

pid_t waitpid(pid_t pid, int& status, int options) // specific pid

    // pid == -1 means the wait set consists all of the parent's child processes, pid > 0 then wait set is the singleton child process whose process ID is equal to pid

    // return PID of child if OK, 0 if WNOHANG, -1 on error (signal / has no children)
```

> 宏：**WIFEXITED**, **WEXITSTATUS**：

```
xxxxxxxxxx
```

```
pid_t wpid = wait(&child_status);

if (WIFEXITED(child_status))

    printf("Child %d terminated with exit status %d\n", wpid,
WEXITSTATUS(child_status));

else printf("Child %d terminate abnormally\n", wpid);
```

> $status = 256 \times exit_value_of_the_child$

6. execve

loading and running programs in the context of the current process

```
xxxxxxxxxx

#include <unsitd.h>

int execve(const char* filename, const char* argv[], const char* envp[]);
```

> **execve** is called once and never return
> The execve function loads and runs the **executable object file** filename with the **argument list** argv and the **environment variable list** envp (`name = value` **strings**). Execve returns to the calling program only if there is an error, such as not being able to find filename.

Figure 8.20
Organization of an argument list.

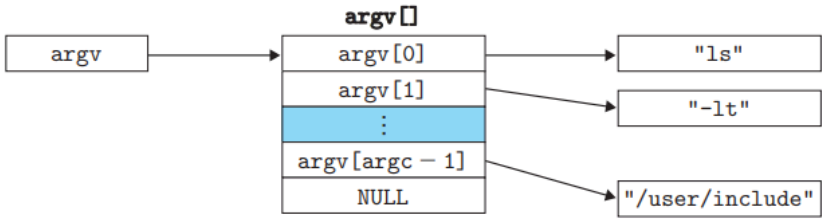
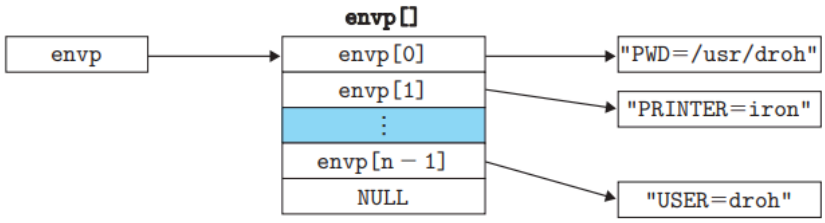
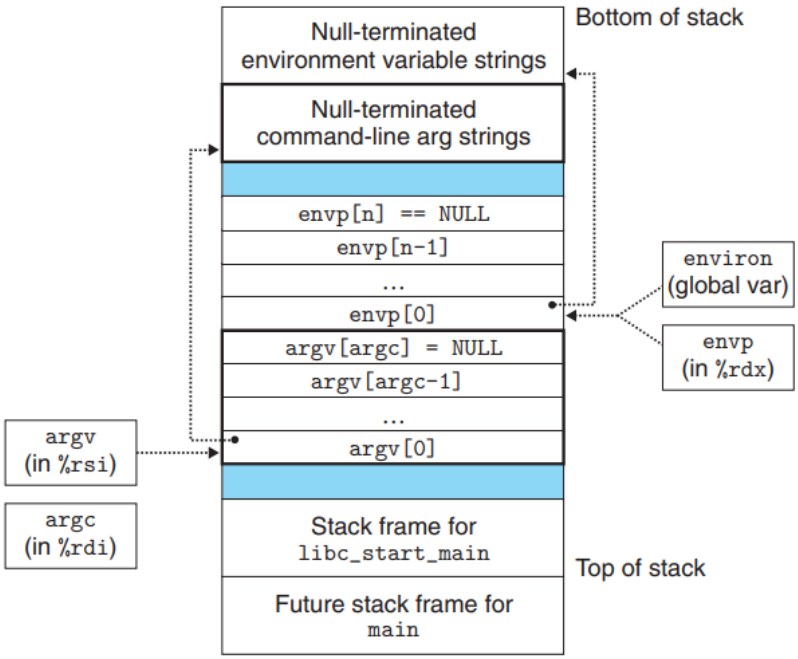


Figure 8.21
Organization of an environment variable list.





- `getenv()`, `setenv()`, `unsetenv()`

```
xxxxxxxxxx

#include <stdlib.h>

char* getenv(const char* name); // return pointer to name if exists

int setenv(const char* name, const char* newvalue, int overwrite);

// return 0 on success, -1 on error

void unsetenv(const char* name);
```

```c
#include "csapp.h"

int main(int agrc, char* argv[], char* envp[]) {
    int i;
    printf("Command-line arguments:\n");
    for (i = 0; argv[i] != NULL; i++)
        printf("    argv[%2d]: %s\n", i, argv[i]);
    printf("\n");
    printf("Environment variables:\n");
    for (i = 0; envp[i] != NULL; i++)
        printf("    envp[%2d]: %s\n", i, envp[i]);
    exit(0);
}
```

PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL JUPYTER

```
catherine@Catherine:/mnt/c/Users/Catherine/Desktop/code$ ./test
Command-line arguments:
    argv[ 0]: ./test

Environment variables:
    envp[ 0]: SHELL=/bin/bash
    envp[ 1]: WSL_DISTRO_NAME=Ubuntu
    envp[ 2]: NAME=Catherine
    envp[ 3]: PWD=/mnt/c/Users/Catherine/Desktop/code
    envp[ 4]: LOGNAME=catherine
    envp[ 5]: HOME=/home/catherine
    envp[ 6]: LANG=C.UTF-8
    envp[ 7]: LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35
:*.tar=01;31:*.tgz=01;31:*.arc=01;31:*.arj=01;31:*.taz=01;31:*.lha=01;31:*.lz4=
=01;31:*.gz=01;31:*.lrz=01;31:*.lz=01;31:*.lzo=01;31:*.xz=01;31:*.zst=01;31:*.t
*.war=01;31:*.ear=01;31:*.sar=01;31:*.rar=01;31:*.alz=01;31:*.ace=01;31:*.zoo=0
=01;35:*.jpeg=01;35:*.mjpg=01;35:*.mjpeg=01;35:*.gif=01;35:*.bmp=01;35:*.pbm=01
g=01;35:*.svgz=01;35:*.mng=01;35:*.pcx=01;35:*.mov=01;35:*.mpg=01;35:*.mpeg=01;
=01;35:*.nuv=01;35:*.wmv=01;35:*.asf=01;35:*.rm=01;35:*.rmvb=01;35:*.flc=01;35:
:*.emf=01;35:*.ogv=01;35:*.ogx=01;35:*.aac=00;36:*.au=00;36:*.flac=00;36:*.m4a=
ga=00;36:*.opus=00;36:*.spx=00;36:*.xspf=00;36:
    envp[ 8]: LESSCLOSE=/usr/bin/lesspipe %s %s
    envp[ 9]: TERM=xterm-256color
    envp[10]: LESSOPEN=| /usr/bin/lesspipe %s
    envp[11]: USER=catherine
    envp[12]: SHLVL=1
    envp[13]: WSLENV=
    envp[14]: XDG_DATA_DIRS=/usr/local/share:/usr/share:/var/lib/snapd/desktop
    envp[15]: PATH=/home/catherine/.local/bin:/usr/local/sbin:/usr/local/bin:/u
```

4.2 壳程序

shell: an interactive application-level program that runs other programs on behalf of the user.（“提供用户使用界面”的程序）

> the original shell was the `sh` program, followed by `csh`, `tcsh`, `ksh` and `bash`
> perform a sequence of *read/evaluate* steps and then terminates
>
> main routine:

code/ecf/shellex.c

```c
#include "csapp.h"
#define MAXARGS   128

/* Function prototypes */
void eval(char *cmdline);
int parseline(char *buf, char **argv);
int builtin_command(char **argv);

int main()
{
    char cmdline[MAXLINE]; /* Command line */

    while (1) {
        /* Read */
        printf("> ");
        Fgets(cmdline, MAXLINE, stdin);
        if (feof(stdin))
            exit(0);

        /* Evaluate */
        eval(cmdline);
    }
}
```

code/ecf/shellex.c

code/ecf/shellex.c

```c
/* eval - Evaluate a command line */
void eval(char *cmdline)
{
    char *argv[MAXARGS]; /* Argument list execve() */
    char buf[MAXLINE];   /* Holds modified command line */
    int bg;              /* Should the job run in bg or fg? */
    pid_t pid;           /* Process id */

    strcpy(buf, cmdline);
    bg = parseline(buf, argv);
    if (argv[0] == NULL)
        return;   /* Ignore empty lines */

    if (!builtin_command(argv)) {
        if ((pid = Fork()) == 0) {   /* Child runs user job */
            if (execve(argv[0], argv, environ) < 0) {
                printf("%s: Command not found.\n", argv[0]);
                exit(0);
            }
        }

        /* Parent waits for foreground job to terminate */
        if (!bg) {
            int status;
            if (waitpid(pid, &status, 0) < 0)
                unix_error("waitfg: waitpid error");
        }
        else
            printf("%d %s", pid, cmdline);
    }
    return;
}

/* If first arg is a builtin command, run it and return true */
int builtin_command(char **argv)
{
    if (!strcmp(argv[0], "quit")) /* quit command */
        exit(0);
    if (!strcmp(argv[0], "&"))    /* Ignore singleton & */
        return 1;
    return 0;                     /* Not a builtin command */
}
```

code/ecf/shellex.c

code/ecf/shellex.c

```c
/* parseline - Parse the command line and build the argv array */
int parseline(char *buf, char **argv)
{
    char *delim;         /* Points to first space delimiter */
    int argc;            /* Number of args */
    int bg;              /* Background job? */

    buf[strlen(buf)-1] = ' ';  /* Replace trailing '\n' with space */
    while (*buf && (*buf == ' ')) /* Ignore leading spaces */
        buf++;

    /* Build the argv list */
    argc = 0;
    while ((delim = strchr(buf, ' '))) {
        argv[argc++] = buf;
        *delim = '\0';
        buf = delim + 1;
        while (*buf && (*buf == ' ')) /* Ignore spaces */
                buf++;
    }
    argv[argc] = NULL;

    if (argc == 0)  /* Ignore blank line */
        return 1;

    /* Should the job run in the background? */
    if ((bg = (*argv[argc-1] == '&')) != 0)
        argv[--argc] = NULL;

    return bg;
}
```

code/ecf/shellex.c

Figure 8.25 `parseline` **parses a line of input for the shell.**

BASH ps 命令

```
ps -l
```

```
catherine@Catherine:/mnt/c/Users/Catherine/Desktop/code$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000     8     7  0  80   0 -  4525 -      tty1     00:00:00 bash
0 T  1000   476     8  0  80   0 -  2634 -      tty1     00:00:00 test
0 R  1000   479     8  0  80   0 -  4645 -      tty1     00:00:00 ps
```

F：进程的权限标志
S：进程的状态（stat），主要状态有：**R** (running), **S** (sleep), **T** (stop), **D** (不可被唤醒), **Z** (zombie)
UID / PID / PPID：此进程被UID所拥有 / 进程的PID号 / 此进程的父进程的PID
PRI / NI：priority/nice的缩写，表示该进程被CPU所执行的优先顺序（越小越优先）
ADDR / SZ / WCHAN：该进程在内存的部分 / 此进程用掉多少内存 / 目前进程是否在工作
TTY：登入这的终端机位置，若为远程登入则使用动态终端介面（pts/n）
TIME：使用掉的CPU时间
CMD：指令

`ps -t`

```
catherine@Catherine:/mnt/c/Users/Catherine/Desktop/code$ ps -t
  PID TTY      STAT   TIME COMMAND
    7 tty1     Ss     0:00 /init
    8 tty1     S      0:00 -bash
  476 tty1     T      0:00 ./test
  480 tty1     R      0:00 ps -t
```

`jobs`
查看挂起的进程信息

前 / 后台程序

background job: `[command] &` 不需要等待进程结束

> 前台程序 correctly **waits** for and **reaps** foreground jobs
> 但是，后台程序终止时会变成“僵尸”（因为可能永远不会被回收）
> 这会导致**内存泄漏**，导致OS内核资源不足（一旦超过了进程配额 process quota），shell就不能运行任何新命令，即`fork()`返回-1
>
> ```
> xxxxxxxxxx
>
> ulimit -u // bash syntax 查询进程配额
> ```

当一个后台进程完成时，OS内核将采取一种类似“异常”(ECF)的机制来“提醒“ shell。这一机制被称为*信号 signal*。

4.3 信号处理 Signals

1. 信号

Definition: A **signal** is a small message that notifies a process that an event of some type has occurred in the system

> 类似于“异常”
> 其类型由小整数ID表示（1-30）`kill -s -id`
> 由内核（有时在另一个进程的请求下）发送给进程
> 传递的唯一的信息是其自身（即ID和它到达这一事实）

ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	Interrupt (e.g., ctl-c from keyboard)
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate & Dump	Segmentation violation
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated

> Default Action 还包括：暂停（直至恢复）

Number	Name	Default action	Corresponding event
1	SIGHUP	Terminate	Terminal line hangup
2	SIGINT	Terminate	Interrupt from keyboard
3	SIGQUIT	Terminate	Quit from keyboard
4	SIGILL	Terminate	Illegal instruction
5	SIGTRAP	Terminate and dump core [a]	Trace trap
6	SIGABRT	Terminate and dump core [a]	Abort signal from `abort` function
7	SIGBUS	Terminate	Bus error
8	SIGFPE	Terminate and dump core [a]	Floating-point exception
9	SIGKILL	Terminate [b]	Kill program
10	SIGUSR1	Terminate	User-defined signal 1
11	SIGSEGV	Terminate and dump core [a]	Invalid memory reference (seg fault)
12	SIGUSR2	Terminate	User-defined signal 2
13	SIGPIPE	Terminate	Wrote to a pipe with no reader
14	SIGALRM	Terminate	Timer signal from `alarm` function
15	SIGTERM	Terminate	Software termination signal
16	SIGSTKFLT	Terminate	Stack fault on coprocessor
17	SIGCHLD	Ignore	A child process has stopped or terminated
18	SIGCONT	Ignore	Continue process if stopped
19	SIGSTOP	Stop until next SIGCONT [b]	Stop signal not from terminal
20	SIGTSTP	Stop until next SIGCONT	Stop signal from terminal
21	SIGTTIN	Stop until next SIGCONT	Background process read from terminal
22	SIGTTOU	Stop until next SIGCONT	Background process wrote to terminal
23	SIGURG	Ignore	Urgent condition on socket
24	SIGXCPU	Terminate	CPU time limit exceeded
25	SIGXFSZ	Terminate	File size limit exceeded
26	SIGVTALRM	Terminate	Virtual timer expired
27	SIGPROF	Terminate	Profiling timer expired
28	SIGWINCH	Ignore	Window size changed
29	SIGIO	Terminate	I/O now possible on a descriptor
30	SIGPWR	Terminate	Power failure

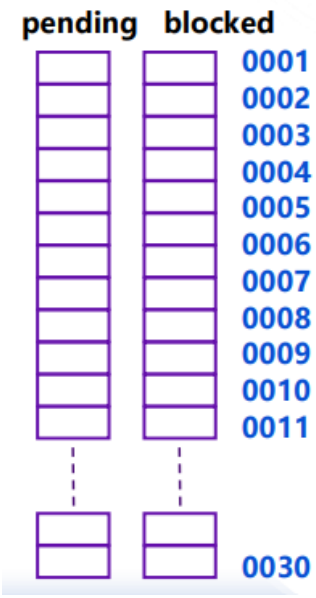
- **Sending a Signal**

  > Kernel sends (delivers) a signal to a **destination process** by **updating some state in the context** of the destination process

- **Receiving a Signal**

> A destination process **receives** a signal **when it is forced by the kernel to react** in some way to the delivery of the signal

  - Ignore: 忽略信号
  - Terminate：终止进程
  - Catch：响应函数（通过注册一个信号处理函数 signal handler 来响应）
- **Pending and Blocked Signals**



pending：信号已发送但暂未被接收

> 每一**种**信号同时最多存在一个pending signal
> signals并非队列排序：若某进程当前已有pending signal，则后续向该进程发送的该种类的信号将被**discarded**
>
> Kernel maintains pending and blocked **bit vectors** in the context of each process
> 当type k信号在pending时set bit k
> 当type k信号接收时clear bit k

blocked：一个被目标进程阻塞的信号可以被递送（delivered），但是在信号被解除阻塞之前不会被接收（received）

> 通过**sigprocmask方程**进行set和clear操作（Explicit blocking mechanism）
>
> ```
> xxxxxxxxxx
> #include <signal.h>
> int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
> int sigemptyset(sigset_t *set);
> int sigfillset(sigset_t *set);
> int sigaddset(sigset_t *set, int signum);
> int sigdelset(sigset_t *set, int signum);
> // Returns: 0 if OK, -1 on error
> ```

```
int sigismember(const sigset_t *set, int signum);

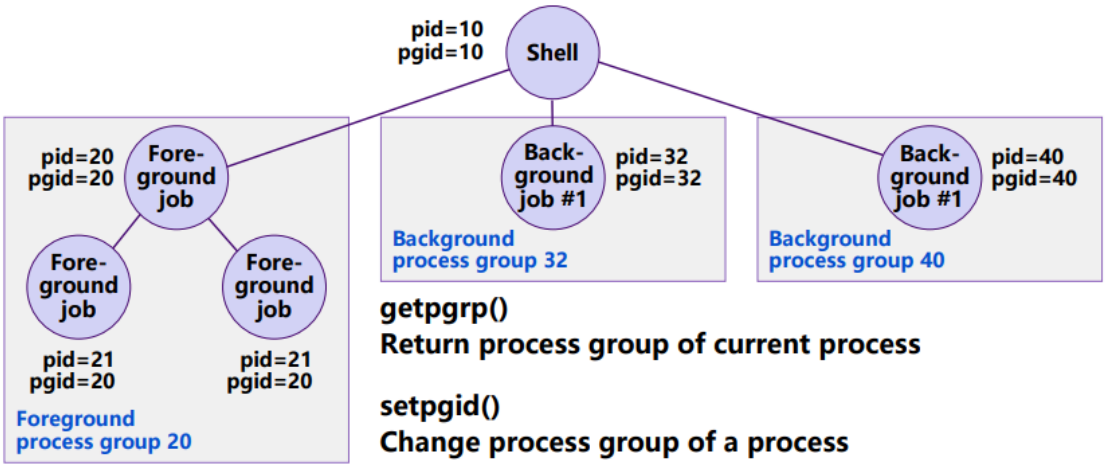
// Returns: 1 if member, 0 if not, -1 on error
```

- Process Groups 进程组
  每一个进程属于一个进程组，用`pgid`表示group id

```
xxxxxxxxxx

#include <unistd.h>
```

`pid_t getpgrp(void)`: return process group of current process
`int setpgid(pid_t pid, pid_t pgid)`: change process group of a process



- **Sending Signals**
  - **with /bin/kill Program**

    /bin/kill program sends arbitrary signal to a process or process group

    Examples:
    `/bin/kill -9 24818`: send SIGKILL to process 24818
    `/bin/kill -9 -24817`: send SIGKILL to every process in **process group** 24817

  - **from the Keyboard**

    `ctrl-c` / `ctrl-z` sends a SIGINT / SIGTSTP to every job in the foreground process group (`fg num` / `bg num`恢复进程到前/后台执行，其中num表示`jobs`中查询到的数字)
    SIGINT: default action is to terminate each process
    SIGTSTP: default action is to stop / suspend each process

  - **with kill Function**

    `int kill(pid_t pid, int SIGINT)`: 则子进程为“不正常退出” / WIFEXITED == False
    `pid == 0`: sends signal to every process in the process group of the calling process
    `pid > 0`: sends signal to process pid
    `pid < 0`: sends signal to every process in process group |pid|

- **Receiving Signals**
  信息的接收时间：进程被调度恢复执行的时候
  `pnb = pending & ~blocked` 为真

2. Installing Signal Handlers

```
xxxxxxxxxx

typedef void (*handler_t)(int);

handler_t signal(int signum, handler_t handler)

    //handler的两种取值：（其他情况handler是signal handler的地址）

    // SIG_IGN：忽略类型为signum的信号

    // SIG_DFL：恢复ID为signum的信号的默认处理

    // Return: pointer to previous handler if OK

    //        SIG_ERR on error
```

```
xxxxxxxxxx

void int_handler(int sig) {

    safe_printf("Process %d received signal %d\n", getpid(), sig);

    //Async-Signal-Safety的printf，稍后会讲到~

    exit(0);

}

void fork13() {

    pid_t pid[N];

    int i, child_status;

    signal(SIGINT, int_handler);

    for (i = 0; i < N; i++)

        if ((pid[i] = fork()) == 0) {

            while(1); // child infinite loop

        }

    for (i = 0; i < N; i++) {

        printf("Killing process %d\n", pid[i]);

        kill(pid[i], SIGINT);

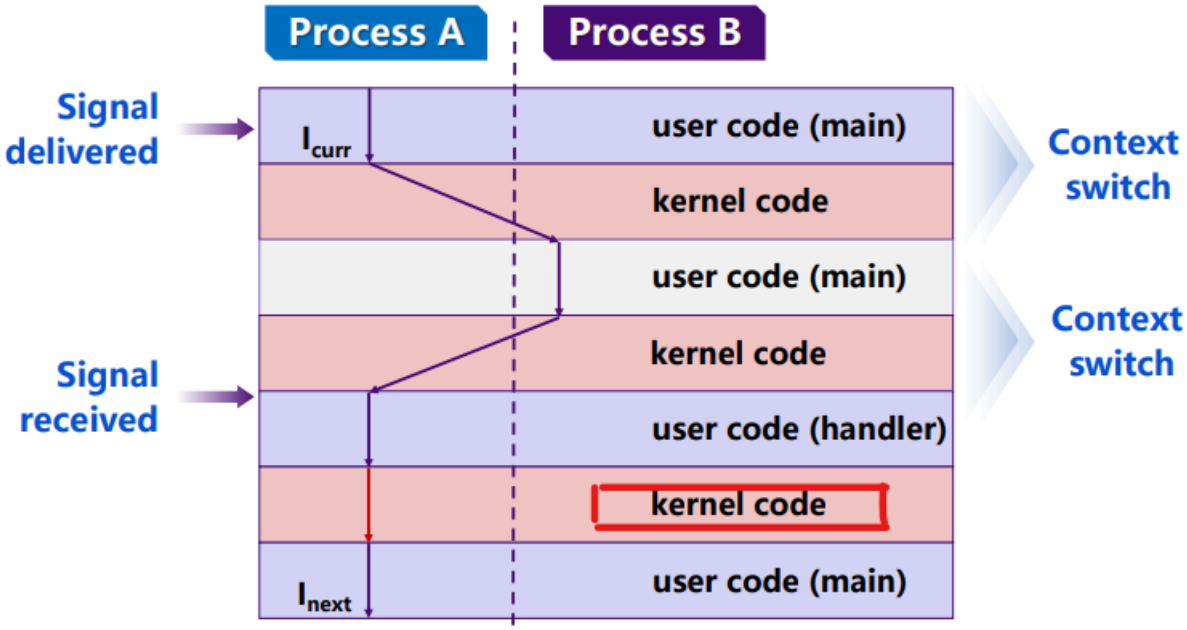
    }

    for (i = 0; i < N; i++) {
```

```
        pid_t wpid = wait(&child_status);
    if (WIFEXITED(child_status))
        printf("Child %d terminated with exit status %d\n", wpid,
WEXITSTATUS(child_status)); //正常退出
    else
        printf("Child %d terminated abnormally\n", wpid);
    }
}
```

1. Signals Handlers as Concurrent Flows

> A signal handler is a **separate logical flow** (not process) that runs concurrently with the main program
>
> 具有独立的stack与处理器上下文（CPU context / control flow）
> 内存是共享的



2. pending signals are not queued
   因信号无法缓存，多个同类信号同时到达的话会丢失若干
3. living with nonqueuing Signals: must check for all terminate jobs
   检查所有的终止子进程：`wait`

```
xxxxxxxxxx
while ((pid = waitpid(-1, &child_status, WNOHANG)) > 0) {
    safe_printf("Received signal %d from process %d\n", sig, pid);
}
//对比:
```

```
// pid_t pid = wait(&child_status);
```

4. signal arrival during long system calls (say a read)
   系统调用执行过程中信号到达了，signal handler会中断read call，signal handler结束后read call会继续自动进行
5. signal handler reacts to externally generated events

```
xxxxxxxxxx

#include <stdlib.h>

#include <stdio.h>

#include <signal.h>


void handler(int sig) {

    safe_printf("You think hitting ctrl-c will stop the bomb?\n");

    sleep(2);

    safe_printf("Well...");

    sleep(1);

    printf("OK\n");

    exit(0);

}

int main() {

    signal(SIGINT, handler); /* installs ctrl-c handler */

    while(1) {}

}
```

> Function is **async-signal-safe** if either reentrant（仅有局部变量） or non-interruptible by signals
> 即当前的函数不会被signal所屏蔽掉
> async-signal-safe wrapper for **printf**: **safe_print**

```
void safe_printf(const char *format, ...) {
    char buf[MAXS];
    va_list args;

    va_start(args, format);                    /* reentrant */
    vsnprintf(buf, sizeof(buf), format, args); /* reentrant */
    va_end(args);                              /* reentrant */
    write(1, buf, strlen(buf));                /* async-signal-safe */
}
```

```
#include <stdarg.h>
```

4.4 非本地跳转 Nonlocal jumps

1. `setjmp` **&** `longjmp`

```
xxxxxxxxxx

#include <setjmp.h>
```

> Powerful (but dangerous) user-level mechanism for transferring control to an arbitrary location
> Controlled to way to **break the procedure call / return discipline**
> Useful for error recovery and signal handling
>
> efficient in deeply nested functions

- `int setjmp(jmp_buf env, int i)`
  must be called before `longjmp`
  called once, returns one or more times
  identifies a return site for a subsequent longjmp
  implementation: remember where you are by storing the current **register context, stack pointer and PC value** in `jmp_buf` and return 0

  > `setjmp` 的返回值不能赋给一个变量！！！
  > 但可以用于switch

- `void longjmp(jmp_buf env, int i)`
  return from the `setjmp` remembered by jump buffer
  return i instead of 0
  called after `setjmp`
  called once, never return
  implementation: restore register context, set %eax to i, jump to the location indicated by the PC stored in `jmp_buf`

一般情况下“返回的顺序与调用顺序相反”

但是可以利用汇编以及对运行栈的理解来编写代码以打破这一惯例

- 有如下汇编代码（x86-32架构），其中GET唯一的输入参数是一个用于存储当前处理器以及栈信息的内存块地址（假设该内存块的空间足够大），而由SET恢复被GET所保存的处理器及栈信息，其唯一的输入参数也是该内存块地址

```
GET:
movl 4(%esp), %eax   #(A)
...
movl %edi, 20(%eax)
movl %esi, 24(%eax)
movl %ebp, 28(%eax)
movl %ebx, 36(%eax)
movl %edx, 40(%eax)
movl %ecx, 44(%eax)
movl $1, 48(%eax)
movl (%esp), %ecx      #(B)
movl %ecx, 60(%eax)
leal 4(%esp), %ecx     #(C)
movl %ecx, 72(%eax)
movl 44(%eax), %ecx
movl $0, %eax
ret
```

```
SET:
movl 4(%esp), %eax
...
movl 20(%eax), %edi
movl 24(%eax), %esi
movl 28(%eax), %ebp
movl 36(%eax), %ebx
movl 40(%eax), %edx
movl 44(%eax), %ecx
movl ___ (%eax), %esp     #(D)
pushl 60(%eax)            #(E)
movl  48(%eax), %eax
ret
```

1. SET过程的返回地址是什么，其返回值是多少？
2. 代码段中的（A）指令执行后，eax中存放的是什么？（B）执行后，ecx中存放的是什么？（C）指令的作用是什么？指令的作用是什么？并将（D）指令补充完整

A 指令执行后，eax 存放的是**内存块的地址**（用于保存处理器和栈信息的内存块）。

B 指令执行后，ecx 存放的是**返回地址**（调用 GET 函数后下一句要执行的地址）。

C 指令的作用是**记录 esp 栈顶指针**。（用于在 D 中将返回地址压入栈）。

D 指令完整的是 `movl 72(%eax), %esp`。

E 指令的作用是将**返回地址压入栈**（调用 GET 函数后下一句要执行的地址）。

2. Limitations of Nonlocal Jumps

Work with stack discipline: can only long jump to environment of function that has been called but not yet completed

```
#include <stdio.h>
#include <signal.h>
#include <setjmp.h>

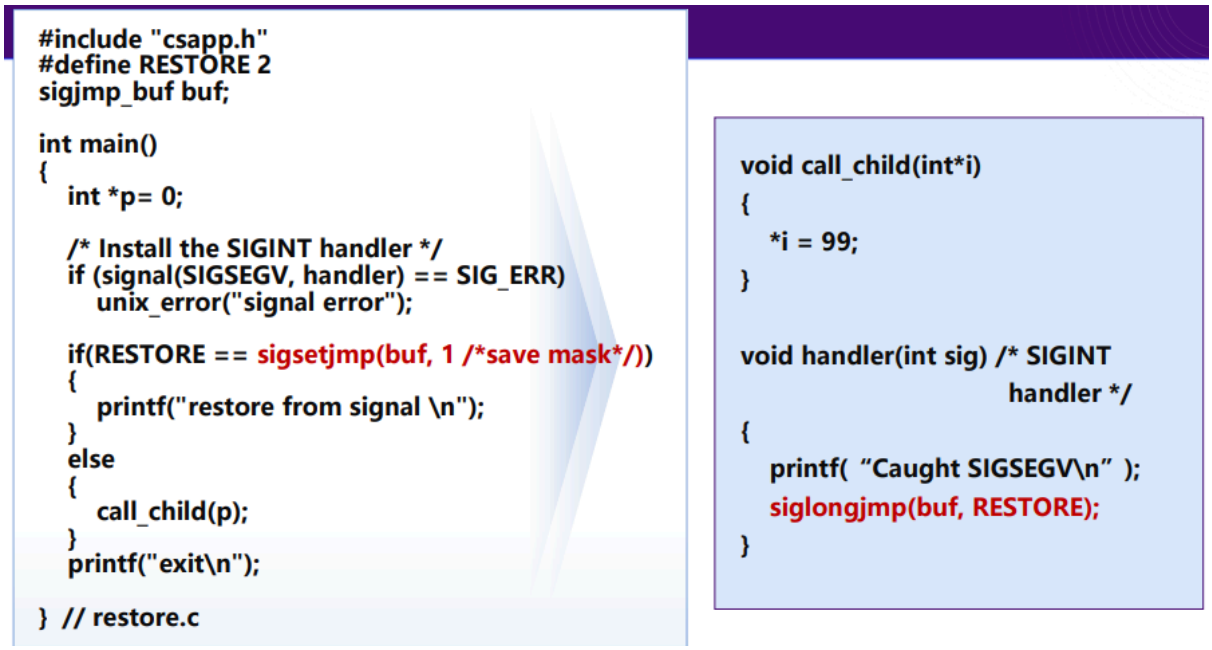
sigjmp_buf buf;
void handler(int sig) {
  siglongjmp(buf, 1);
}

main() {
        if (!sigsetjmp(buf, 1))
        {
           signal(SIGINT, handler);
           printf("starting\n");
        }
        else
           printf("restarting\n");

        while(1) {
           sleep(1);
           printf("processing...\n");
        }
}
```

restart.c

```
greatwhite> ./restart
starting
processing...
processing...
processing...
restarting          <-- Ctrl-c
processing...
processing...
restarting          <-- Ctrl-c
processing...
processing...
processing...
```



```
#include "csapp.h"
#define RESTORE 2
sigjmp_buf buf;

int main()
{
   int *p= 0;

   /* Install the SIGINT handler */
   if (signal(SIGSEGV, handler) == SIG_ERR)
      unix_error("signal error");

   if(RESTORE == sigsetjmp(buf, 1 /*save mask*/))
   {
      printf("restore from signal \n");
   }
   else
   {
      call_child(p);
   }
   printf("exit\n");

} // restore.c
```

```
void call_child(int*i)
{
   *i = 99;
}

void handler(int sig) /* SIGINT
                         handler */
{
   printf( "Caught SIGSEGV\n" );
   siglongjmp(buf, RESTORE);
}
```

> `sigsetjmp()` and `siglongjmp()` are versions of `setjmp()` and `longjmp()` that can be used by signal handlers

小结

- **Signals provide process-level exception handling**
  can generate from user program
  can define effect by declaring signal handler
- **Some caveats**
  very high overhead: >10,000 clock cycles
  don't have queues
  only use for exceptional conditions
  just one bit for each pending signal type
- **Nonlocal jumps provide exceptional control flow within process**
  within constraints of stack discipline

4.5 ECF小结

1. `fork()`

```
xxxxxxxxxx

#include <sys/types.h>

#include <unistd.h>

pid_t fork(void);

// return 0 to child, PID of child to parent, -1 on error


#include <sys/wait.h>
```

```
pid_t waitpid(pid_t pid, int *statusp, int options);

// return PID of child if OK, 0 (if WNOHANG), or -1 on error

pid_t wait(int *statusp);

// return PID of child if OK or -1 on error
```

2. `sleep()`

```
xxxxxxxxxx

#include <unistd.h>

unsigned int sleep(unsigned int secs);

// return seconds left to sleep

// The latter case (rtval > 0) is possible if the sleep function returns prematurely
because it was interrupted by a signal
```

3. `pause()`

```
xxxxxxxxxx

#include <unistd.h>

int pause(void);

// return -1
```

4. `execve()`

```
xxxxxxxxxx

#include <unistd.h>

int execve(const char *filename, const char *argv[], const char *envp[]);

// does not return if OK; return -1 on error


#include <stdlib.h>

char *getenv(const char *name);

// return pointer to name if it exists, NULL if no match

int setenv(const char *name, const char *newvalue, int overwrite);

// return 0 on success, -1 on error

void unsetenv(const char *name);
```

```
// return nothing
```

5. `kill()`

```
xxxxxxxxxx

#include <sys/types.h>

#include <signal.h>

int kill(pid_t pid, int sig);

// return 0 if OK, -1 on error
```

6. `signal()`

```
xxxxxxxxxx

#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);

// return pointer to previous handler if OK, SIG_ERR on error (does not set errno)
```

7. `jump()`

```
xxxxxxxxxx

#include <setjmp.h>

int setjmp(jmp_buf env);

int sigsetjmp(sigjmp_buf env, int savesigs);

// return 0 from setjmp, nonzero from longjmps


void longjmp(jmp_buf env, int retval);

void siglongjmp(sigjmp_buf env, int retval);

// never returns
```

5 IO处理

5.1 Unix I/O

1. Unix Files

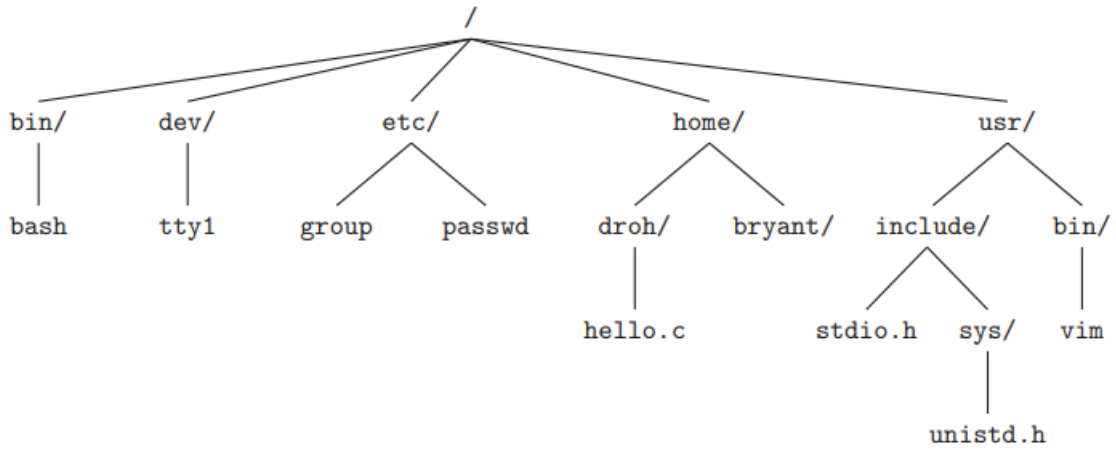
Definition:

> A Unix **file** is a sequence of m bytes
> All I/O devices are represented as files（IO设备都视作文件）
> `/dev/sda2` (/usr disk partition)
> `/dev/tty2` (terminal)
> Even the kernel is represented as a file
> `/dev/kmem` (kernel memory image)
> `/proc` (kernel data structures)
>
> `vim limits`
> `vim maps`

- **Unix File Types**
  - Regular File

    > often classified into *binary files* and *ASCII / Unicode characters files*
    >
    > `\n`, ASCII = `LF`, numeric_value = `0x0a`

    File containing user/app data (binary, text, whatever)
    OS does not know anything about the format（文件格式 / 内容对OS透明）
  - Directory File
    A file that contains the names and locations of other files
  - Character special and block special files
    Terminals (character special) and disks (block special)
  - FIFO (named pipe)
    A file type used for interprocess communication
  - Socket
    A file type used for network communication between processes
- **directory hierarchy**



> absolute pathname: `/home/droh/hello.c`
> relative pathname: (current working directory) `/home/droh`, (hello.c) `./hello.c`

- **summary**

> Key Features:
> Elegant mapping of files to devices allows kernel to export simple interface called **Unix I/O**
> Important idea: All input and output is handled in a consistent and uniform way
>
> **Basic Unix I/O operations / system calls**:
>
> - Opening and closing files: `open()` and `close()`
> - Reading and writing a file: `read()` and `write()`
> - Changing the **current file offset/position** (*seek*)
> indicates next offset into file to read or write
> `lseek()`
>
> ```
> xxxxxxxxxx
> #include <unistd.h>
> #include <sys/types.h>
>
> off_t lseek(int filedes, off_t offset, int whence);
> // return: new offset if OK, -1 if error
> // whence: SEEK_SET: set to offset
> //         SEEK_CUR: set to cur + offset
> //         SEEK_END: file_size() + offset
> ```
>
> - access file metadata: `stat()`

Opening Files

```
xxxxxxxxxx

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

int open(char* filename, int flags, mode_t mode);

// return: new file descriptor if OK, -1 on error

// flags: O_RDONLY, O_WRONLY, O_RDWR, O_CREAT, O_TRUNC, O_APPEND

// mode: specifies access permission


int fd;

#define DEF_MODE S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH

#define DEF_UMASK S_IWGRP|S_IWOTH

umask(DEF_UMASK);

if ((fd = open("/etc/hosts", O_RDONLLY, DEF_MODE)) < 0) {

    // owner: read and write permissions, others: read permissions

    perror("open");

    exit(1);

}
```

Mask	Description
S_IRUSR	User (owner) can read this file
S_IWUSR	User (owner) can write this file
S_IXUSR	User (owner) can execute this file
S_IRGRP	Members of the owner's group can read this file
S_IWGRP	Members of the owner's group can write this file
S_IXGRP	Members of the owner's group can execute this file
S_IROTH	Others (anyone) can read this file
S_IWOTH	Others (anyone) can write this file
S_IXOTH	Others (anyone) can execute this file

Figure 10.2 Access permission bits. Defined in `sys/stat.h`.

> Return a small identifying integer **file descriptor**（文件描述符）
> the descriptor returned is always the smallest descriptor that is not currently open in the process
> `fd == -1`: an error occurred

> Each process created by a Unix shell begins life with **three open files** associated with a terminal (files are recorded in a FD_TABLE)
> 0: standard input
> 1: standard output
> 2: standard error (diagnostic) output

```
xxxxxxxxxx

#include <unistd.h>

const: STDIN_FILENO, STDOUT_FILENO, STDERR_FILENO
```

Closing Files

```
xxxxxxxxxx

#include <unistd.h>

int close(int fd);

// Returns 0 if OK, -1 on error


int fd; /*file descriptor*/

int retval; /*return value*/


if ((retval = close(fd)) < 0) {

    perror("close");

    exit(1);

}
```

> Closing an already closed file is an error
> Moral: always check return codes, even for seemingly benign functions such as `close()`

- **Tips-1**:
  当一个进程终止时，内核对该进程所有尚未关闭的文件描述符调用close关闭。
- **Tips-2**:
  成功的关闭并不确保数据已经成功地保存到磁盘，因为内核会延迟写操作。如果需要确保，请在关闭前使用fsync。

Reading Files

读取文件内容并更新文件访问位置

```
xxxxxxxxxx
```

```
#include <unistd.h>

ssize_t read(int fd, void* buf, size_t n);

// Returns: number of bytes read if OK, 0 on EOF, -1 on error


char buf[512];

int fd; /*file descriptor*/

int nbytes; /*number of bytes read*/

    /* Open file fd ... */

    /* Then read up to 512 bytes from file fd */

if ( (nbytes = read(fd, buf, sizeof(buf))) < 0 ) {

    perror("read"); //output to stderr

    exit(1);

}
```

returns number of bytes read from file fd into buf

> Return type ssize_t is signed integer
> nbytes < 0 indicates that an error occurred
> Short counts (nbytes < sizeof(buf)) are possible and are not errors

Writing Files

写入文件内容并更新文件访问位置

```
xxxxxxxxxx

#include <unistd.h>

ssize_t write(int fd, const void* buf, size_t n);

// Returns: number of bytes written if OK, -1 on error


char buf[512];

int fd; /*file descriptor*/

int nbytes; /*number of bytes read*/

    /* Open the file fd ... */

    /* Then write up to 512 bytes from buf to file fd*/

if ((nbytes = write(fd, buf, sizeof(buf)) < 0) {

    perror("write");
```

```
    exit(1);
}
```

returns number of bytes written from buf to file fd

> nbytes < 0 indicates that an error occurred
> As with reads, short counts are possible and are not errors

dealing with Short Counts



5.2 RIO (robust I/O) package

5.3 Metadata, sharing and redirection

1. File Metadata（元数据）

Definition: metadata is data about data, in this case file data

> Per-file metadata maintained by kernel
> Accessed by users with the *stat* and *fstat* functions

```
xxxxxxxxxx
#include <unistd.h>
#include <sys/stat.h>
int stat(const char* filename, struct stat* buf);
int fstat(int fd, struct stat* buf);
// Returns: 0 if OK, -1 on error
```

```
/* Metadata returned by the stat and fstat functions */
struct stat
 {
   dev_t     st_dev;     /* device */
   ino_t     st_ino;     /* inode */
   mode_t    st_mode;    /* protection and file type */
   nlink_t   st_nlink;   /* number of hard links */
   uid_t     st_uid;     /* user ID of owner */
   gid_t     st_gid;     /* group ID of owner */
   dev_t     st_rdev;    /* device type (if inode device) */
   off_t     st_size;    /* total size, in bytes */
   unsigned long  st_blksize; /* blocksize for filesystem I/O */
   unsigned long  st_blocks;  /* number of blocks allocated */
   time_t    st_atime;   /* time of last access */
   time_t    st_mtime;   /* time of last modification */
   time_t    st_ctime;   /* time of last change  (权限修改)  */
};
```

2. Accessing File Metadata

```
xxxxxxxxxx

#include "csapp.h"

int main (int argc, char **argv) {

    struct stat buf;

    char *type, *readok;


    stat(argv[1], &buf);

    if (S_ISREG(buf.st_mode)) type = "regular";

    else if (S_ISDIR(buf.st_mode)) type = "directory";

    else type = "other";


    if ((buf.st_mode & S_IRUSR)) /* OK to read?*/ readok = "yes";

    else readok = "no";


    printf("type: %s, read: %s\n", type, readok);

    exit(0);

}
```

参数	说明	参数	说明
S_IRUSR	所有者拥有读权限	S_IXGRP	群组拥有执行权限
S_IWUSR	所有者拥有写权限	S_IROTH	其他用户拥有读权限
S_IXUSR	所有者拥有执行权限	S_IWOTH	其他用户拥有写权限
S_IRGRP	群组拥有读权限	S_IXOTH	其他用户拥有执行权限
S_IWGRP	群组拥有写权限		

- accessing directories

```
xxxxxxxxxx

#include <sys/types.h>

#include <dirent.h>

DIR* opendir(const char* name);

//Returns: pointer to handle if OK, NULL on error

struct dirent* readdir(DIR* dirp);

//Returns: pointer to next directory entry if OK, NULL is no more entries or error


{

    DIR *directory;

    struct dirent *de;

    ...

    if (!(directory = opendir(dir_name)))

        error("Failed to open directory");

    ...

    while (0 != (de = readdir(directory))) {

        printf("Found file: %s\n", de->d_name);

    }

    ...

    closedir(directory);

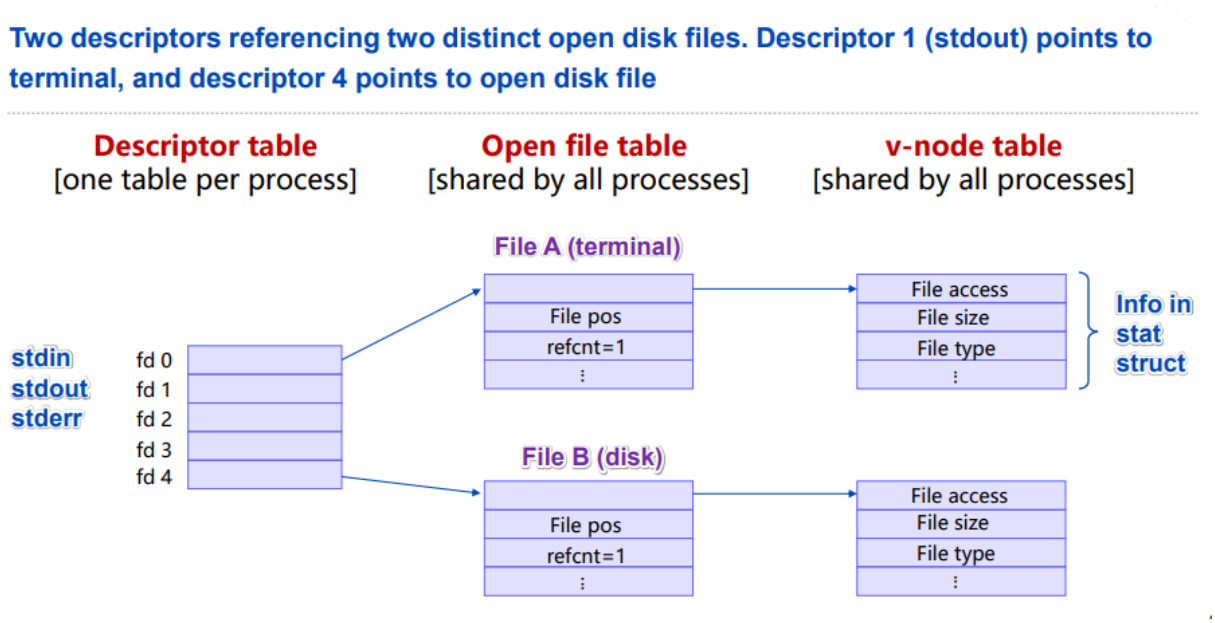
}


struct dirent {

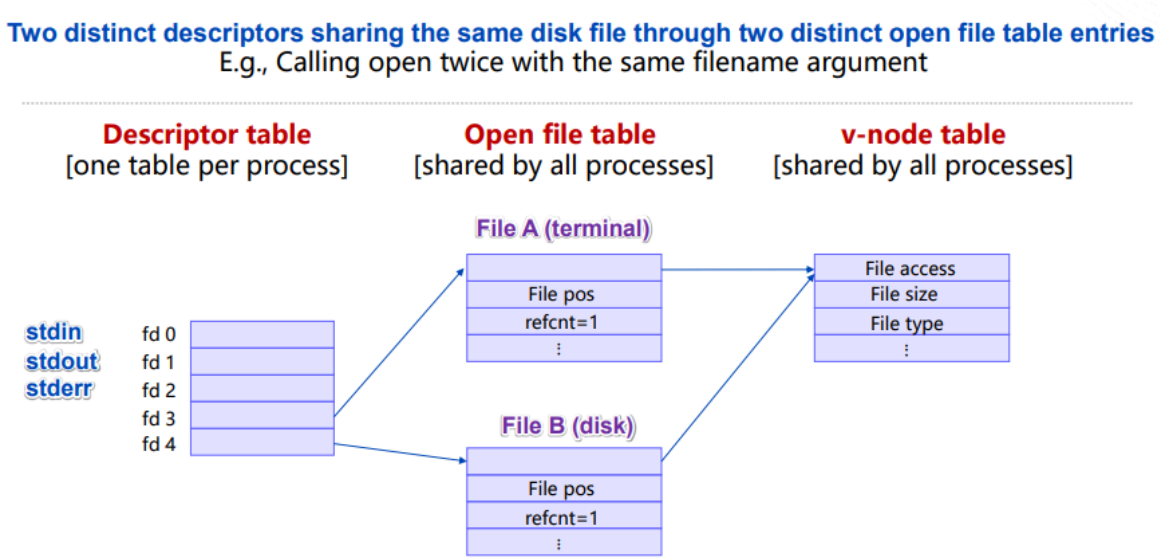
    long d_ino;  /*inode number 索引节点号 */
```

```
    off_t d_off;  /* offset to this dirent 在目录文件中的偏移 */
    unsigned short d_reclen;  /* length of this d_name 文件名长 */
    unsigned char d_type;  /* the type of d_name 文件类型 */
    char d_name [NAME_MAX+1];  /* file name(null结尾)文件名最长255字符*/
}
```

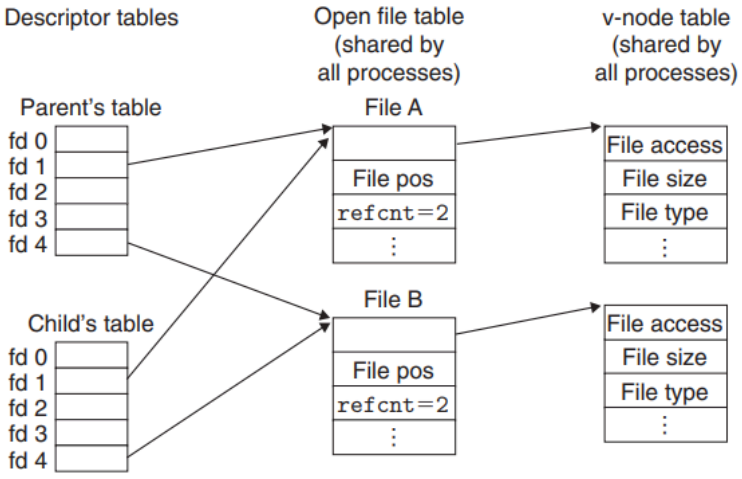
3. Unix Kernel Representation of Open Files



File sharing:



After call to `fork`:



4. I/O Redirection（重定向）

```
xxxxxxxxxx

#include <cstdio>

int main() {

    freopen("in.txt", "r", stdin); //输入重定向，输入数据将从in.txt中读取

    freopen("out.txt", "w", stdout); //输出重定向，输出数据讲保存到out.txt

}
```

```
xxxxxxxxxx

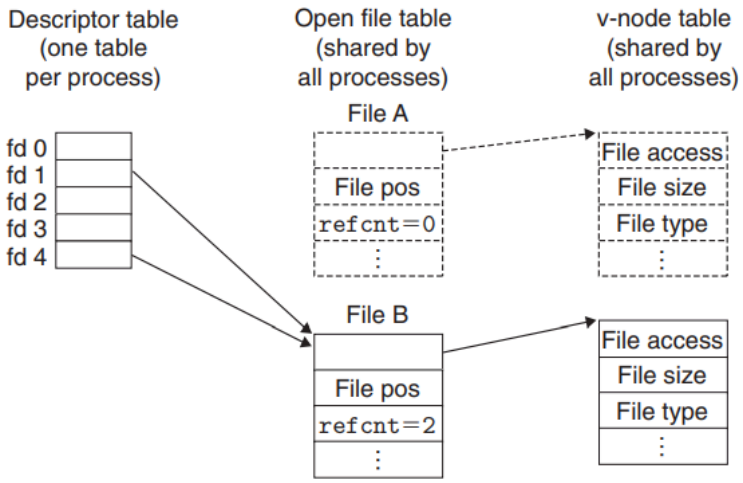
#include <unistd.h>

int dup2(int oldfd, int newfd);

//Returns: nonnegative descriptor if OK, -1 on error

int dup(int fd); // fd1 = dup(fd4);
```

> Example: call `dup(4, 1)`



Remarks: 此时fd1和fd4连续调用`read(fd, &buffer, 1)`得到的`&buffer`将不同（因为其中一个调用完`read`之后`file pos`的值也改变了）

Remarks：几种可能的Descriptor table和Open file table的关系

> - 一个进程的多个descriptor table entry一一对应Open file table中的不同项
>   这是最一般的情况（多次打开同一个文件也是如此）
> - 不同进程的同一descriptor table entry对应Open file table中的同一项
>   子进程继承父进程
> - 同一进程的不同descriptor table entry对应Open file table中的同一项
>   dup / dup2

5.4 Standard I/O

The C standard library (`libc.so`) contains a collection of higher-level standard I/O functions

> Opening and closing files: `fopen`, `fclose`
> Reading and writing bytes: `fread`, `fwrite`
> Reading and writing text lines: `fgets`, `fputs`
> Formatted reading and writing: `fscanf`, `fprintf`

特点：流式文件处理（a stream is a pointer to a structure of type FILE）
ANSI C program begins with three open streams: *stdin*, *stdout*, *stderr*

```
xxxxxxxxxx

#include <cstdio>

extern FILE* stdin; // descriptor 0

extern FILE* stdout; // descriptor 1

extern FILE* stderr; // descriptor 2
```

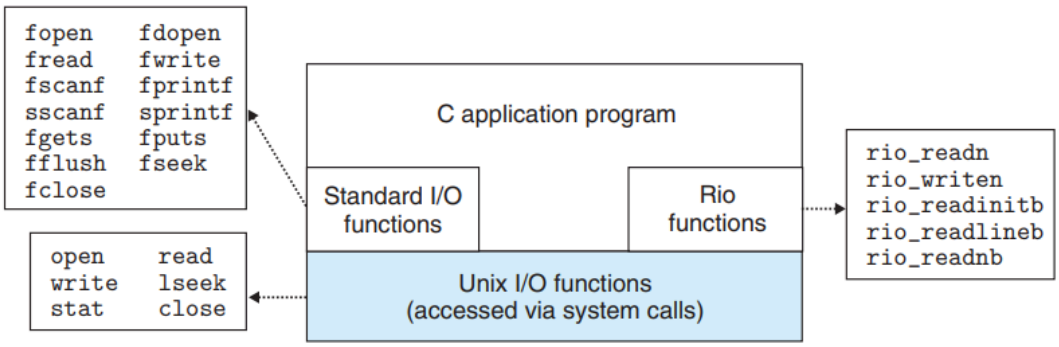


Figure 10.16 Relationship between Unix I/O, standard I/O, and RIO.

1. 通过`fflush()`调用显示地flush到输出文件 / 检测到换行符输出
2. `strace ./main`显示进程的系统调用信息
3. 不能用在二进制文件上的函数：
   基于行处理的IO：`fgets`，`scanf`，`printf`，`rio_readlineb`

   > 不同系统解释换行符的方式不同0x0a (`\n`)：
   > Linux和Mac OS X：LF(0x0a), [`\n`]
   > HTTP服务器 & Windows：CR+LF(0x0d 0x0a), [`\r\n`]
   > 使用`rio_readn`或`rio_readnb`替代

4. 字符串处理函数：`strlen`，`strcpy`——解释字节0为串结尾

Pros and Cons: Unix I/O

- **Pros**
  Unix IO是最通用而且额外开销最小的IO形式
  所有其他IO包都是使用UNIX IO来实现的
  Unix IO提供了访问文件元数据的功能
- **Cons**
  对不足值的处理不够，容易出错
  有效读取文本行需要某种形式的缓冲，容易出错
  这些问题在标准IO和RIO包中都得到解决

Pros and Cons: Standard I/O

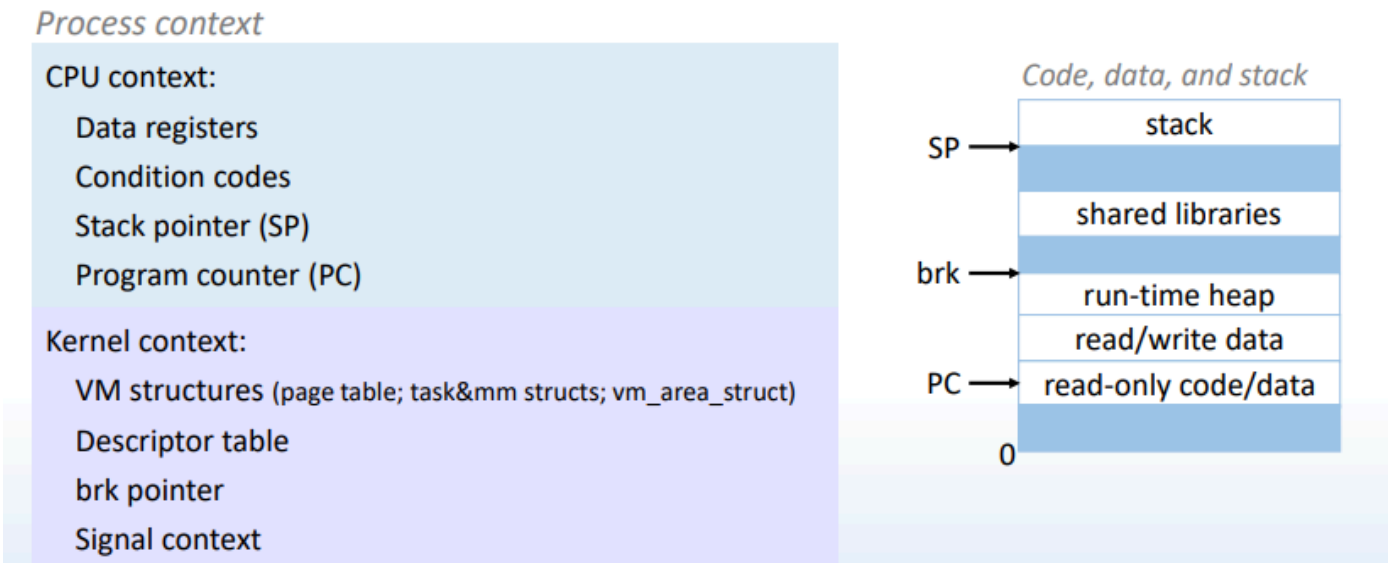
- **Pros**
  通过减小read和write调用的数目，缓冲增加了效率
  不足值可以自动处理
- **Cons**
  没有提供访问文件元数据的功能
  标准IO程序并非同步信号安全的，也不适合在信号处理程序中采用
  标准IO不适合对网络socket的输入输出进行处理

6 线程与线程同步

6.1 线程基本概念

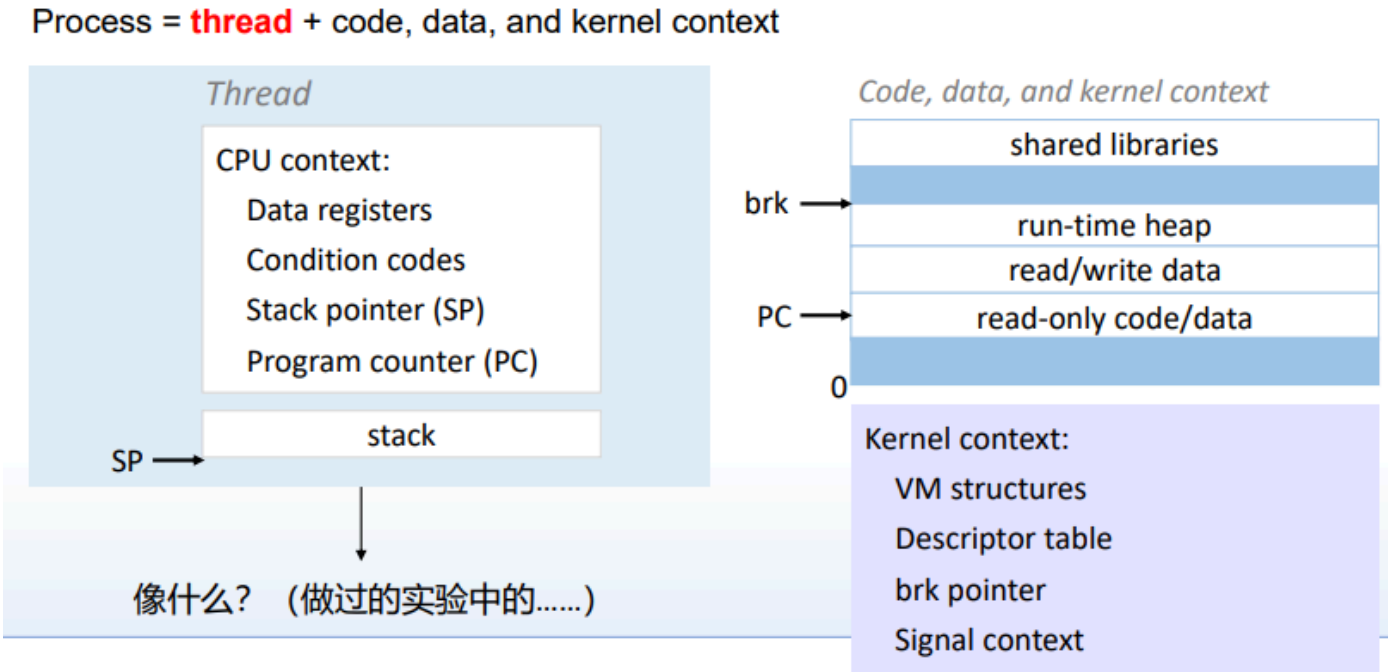
1. 进程与线程

进程：Process = process context + code, data and stack
进程上下文（含处理器状态 以及 **内核上下文**）与虚存空间

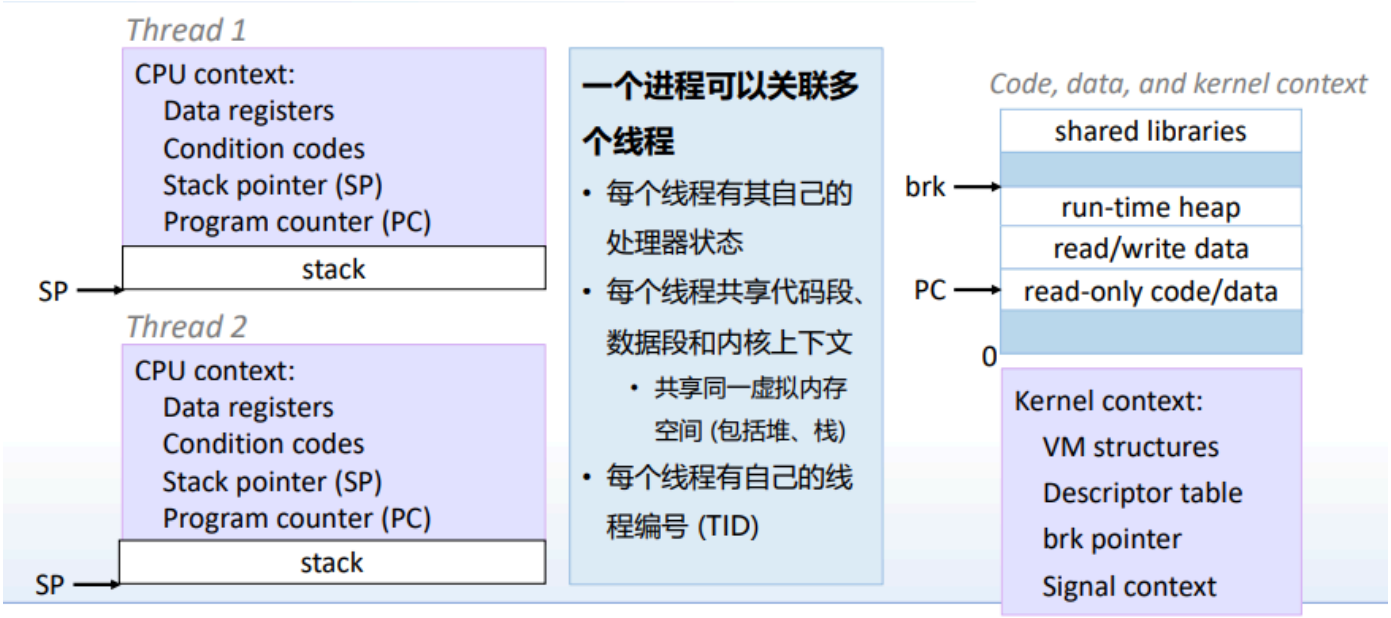


线程：Process = thead + code, data and kernel context

> 线程thread 和 进程progress 的区别：有独立的**处理器状态**，和主进程共享**内核上下文**
> 线程thread 和 协程coroutine 的区别：前者OS调用，后者为用户调用thread

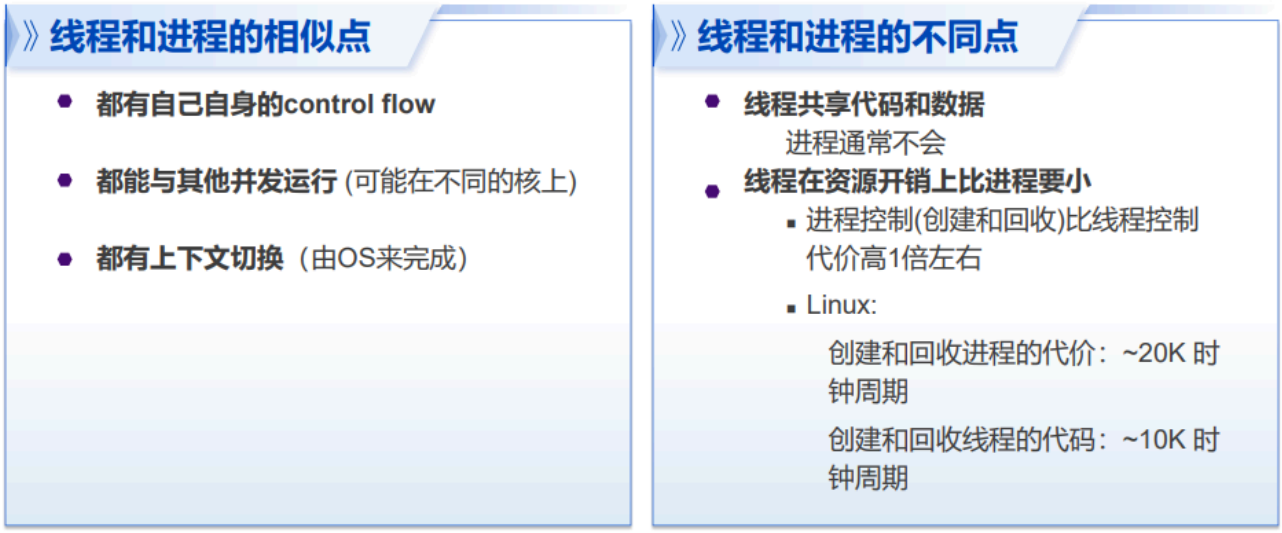


1. **Process with Two Threads**

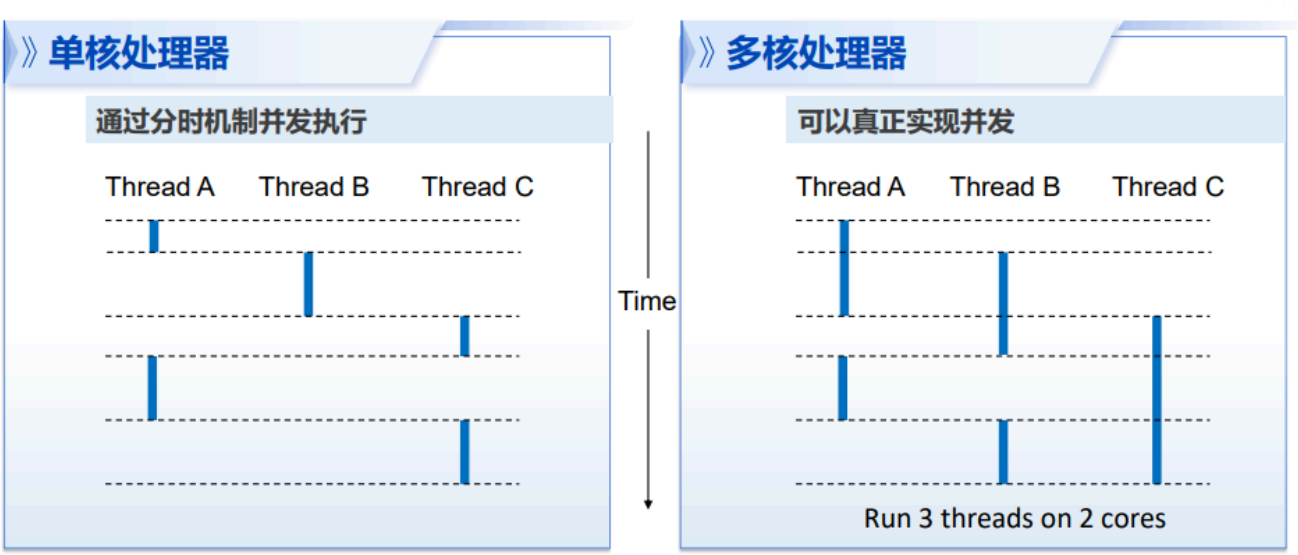


每个线程有自己的处理器状态：CPU context
每个线程有自己的线程编号：**TID**

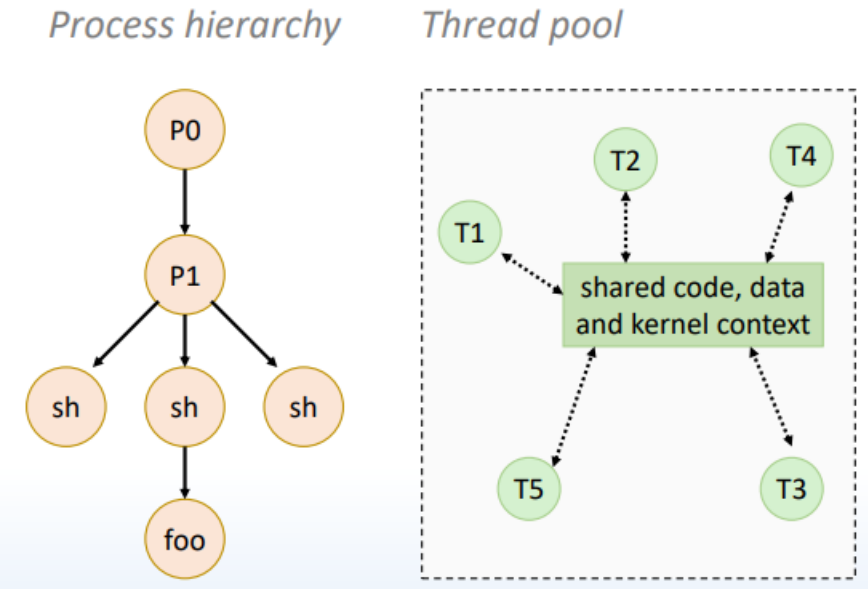
2. **Threads vs Processes**



并发(concurrency)：一个处理器同时处理多个任务（逻辑上的同时发生，依赖于轮询）
并行(parallel)：多个处理器或者多核处理器同时处理多个不同的任务（物理上的同时发生）



- 与某进程关联的线程组成一个**线程池**
  Each thread can kill any other
  Each thread can wait for any other
  Main thread: first threads to run in a process
- 与此相对，进程会形成一个**树状结构**



2. Pthreads Interface

Pthreads: Standard interface for ~60 functions that manipulate threads from C programs

- Threads run thread routines:
  `void *threadroutine(void *vargp)`
- Creating and reaping threads:
  `pthread_create(pthread_t *tid, …, func *f, void *arg)`
  `pthread_join(pthread_t tid, void **thread_return)`
- Determining your thread ID:
  `pthread_self()`
- Terminating threads:
  `pthread_cancel(pthread_t tid)`

`pthread_exit(void *thread_return)`

`return (in primary thread routine terminates the thread)`

`exit (terminates all threads)`

```
xxxxxxxxxx

#include "csapp.h"

void *thread(void *vargp);

int main() {

    pthread_t tid;

    Pthread_create(&tid, NULL, thread, NULL);

    // NULL = Thread attributes(usually NULL)

    // NULL = Thread arguments(void *p)

    Pthread_join(tid, NULL); // assigns return value(void **p)

    // 是为了防止主线程没有给其他线程执行时间就返回而设计的

    exit(0);

}

/* thread routine */

void *thread(void *vargp) {

    printf("Hello, world!\n");

    return NULL;

}
```

3. Pros and Cons of Thread-Based Designs

Pros: 易于在线程之间共享数据；线程比进程更有效率
Cons：无意的共享可能会导致微妙的、难以重现的错误

6.2 共享（sharing）

Definition：一个变量x是**共享的**，当且仅当多个线程引用x的某个实例

1. Threads Memory Model

概念模型：

- **多个线程在单个进程内运行**
- **每个线程有独立的线程上下文**
  线程号，栈，栈指针，PC，条件码，以及通用寄存器

- **所有线程共享其余的进程资源**
  进程虚拟空间的代码、数据、堆、共享库
  打开的文件以及设置好的信号处理句柄（handler）

操作层面：

- **寄存器是严格隔离和受到保护的，但虚存并不如此**
- **任何线程可以读、写任何其他线程的栈**

2. Mapping Variable Instances to Memory

全局变量： 定义在所有函数外面的变量
虚拟内存包含全局变量的唯一实例
局部变量：定义在函数内部的变量，无static属性
每个线程栈包含每个局部变量的一个实例
局部静态变量：定义在函数内部，有static属性
虚拟内存包含局部静态变量的唯一实例

```
xxxxxxxxxx

#include <pthread.h>


char **ptr; /* global */


void *thread(void *vargp) {

    int myid = (int)(__int64_t)vargp;

    static int cnt = 0;

    printf("[%d]: %s (svar=%d)\n", myid, ptr[myid], ++cnt);

}


int main() {

    int i;

    pthread_t tid;

    char *msgs[2] = {

        "Hello from foo",

        "Hello from bar"

    };

    ptr = msgs;

    for (i = 0; i < 2; i++)

        pthread_create(&tid, NULL, thread, (void *)i);
```

```
    pthread_exit(NULL);
}

/*
编译选项：链接-lpthread（通常只默认链接标准库-lstdc++）
    g++ test.cpp -lpthread -o test
运行结果：
    [0]: Hello from foo (svar=1)
    [1]: Hello from bar (svar=2)
In this case:
    ptr, cnt and msgs are shared
    i and myid are not shared
*/
```

6.3 互斥（mutual exclusion）

```
xxxxxxxxxx
//badcnt.c:失败的线程间同步的例子
void *thread(void *vargp) {
    int i, niters = *((int *)vargp);
    for (i = 0; i < niters; i++) cnt++;
    return NULL;
}

volatile int cnt = 0; /* global */
int main(int argc, char **argv) {
    int niters = atoi(argv[1]);
    pthread_t tid1, tid2;
    Pthread_create(&tid1, NULL, thread, &niters);
    Pthread_create(&tid2, NULL, thread, &niters);
    Pthread_join(tid1, NULL);
```

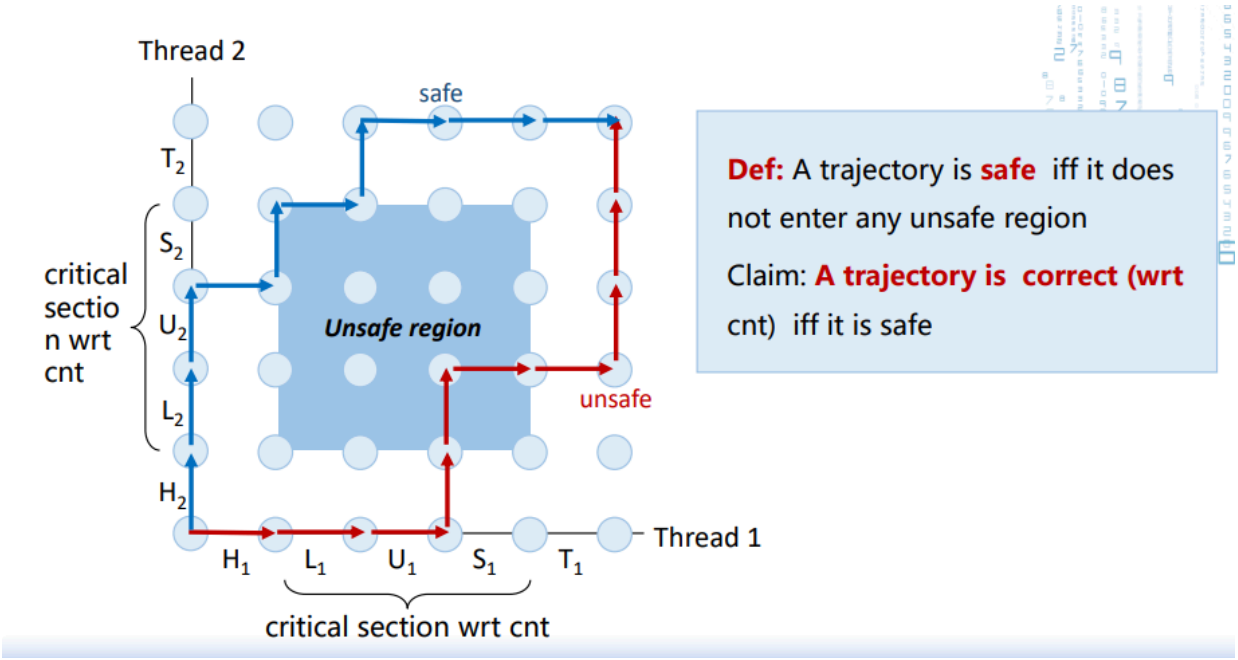
```
    Pthread_join(tid2, NULL);
    /* Check result */
    if (cnt != (2 * niters))
        printf("BOOM! cnt=%d\n", cnt);
    else
        printf("OK cnt=%d\n", cnt);
    exit(0);
}

/*
linux> ./badcnt 10000
OK cnt=20000
linux> ./badcnt 10000
BOOM! cnt=13056
*/

/*
更改汇编后重新编译：g++ test.s -lpthread -o test
    #movl   cnt(%rip), %eax // L: load cnt
    #addl   $1, %eax // U: update cnt
    #movl   %eax, cnt(%rip) // S: store cnt
    incl    cnt(%rip)
*/
```

1. 进程图 progress graph

We can analyze the behavior using a **progress graph 进度图**
A **trajectory** is a sequence of legal state transitions that describes one possible concurrent execution of threads. 轨迹描述了状态转换的顺序，它代表一种可能的线程并发执行方式。
critical section 临界区：w.r.t the shared variable *cnt*



2. 同步 synchronize

We must **synchronize 同步** the execution of the threads so that they never have an unsafe trajectory (i.e., need to guarantee **mutually exclusive access** to critical region)

经典方法：**Semaphores 信息量** (Edsger Dijkstra)

> Other Approaches:
> Mutex and condition variables (Pthreads)
> Monitors (Java)

6.4 信息量 Semaphores

Semaphore：非负整数的，用于同步的全局变量
由P、V操作组成：$P(s)$：`[ while (s == 0) wait(); s--; ]`，$V(s)$：`[ s++; ]`
OS内核确保`[]`之间的操作**不可分割**：一次只有一个P或者V操作修改s；只有P能减小s
Semaphore invariant: `(s >= 0)`

Pthreads functions:

```
xxxxxxxxxx

#include <semaphore.h>

int sem_init(sem_t* sem, 0, unsigned int val); // s = val

int sem_wait(sem_t* s); // P(s)

int sem_post(sem_t *s); // V(s)
```

1. Using Semaphores for Mutual Exclusion

Basic idea:

- 对每一个共享变量（或者相关的共享变量集合），关联一个唯一的信号量mutex，初始为1
- 将相应地临界区用`P(mutex)`和`V(mutex)`保护

Terminology:

- **Binary semaphore 二值/元信号量**
  semaphore whose value is always 0 or 1
- **Mutex 互斥锁**
  binary semaphore used for mutual exclusion
  - P operation: **locking** the mutex
  - V operation: **unlocking** or **releasing** the mutex
  - **Holding** a mutex: locked and not yet unlocked
- **Counting semaphore 计数信号量**
  used as a counter for set of available resources

2. goodcnt.c: 正确同步的例子

```
xxxxxxxxxx

#include <semaphore.h>

volatile int cnt = 0; /* Counter */

sem_t mutex; /* Semaphore that protects cnt */

Sem_init(&mutex, 0, 1); /* mutex = 1 */


// Surround critical section with P and V

void *thread(void *vargp) {

    int i, niters = *((int *)vargp);

    for (i = 0; i < niters; i++) {

        P(&mutex);

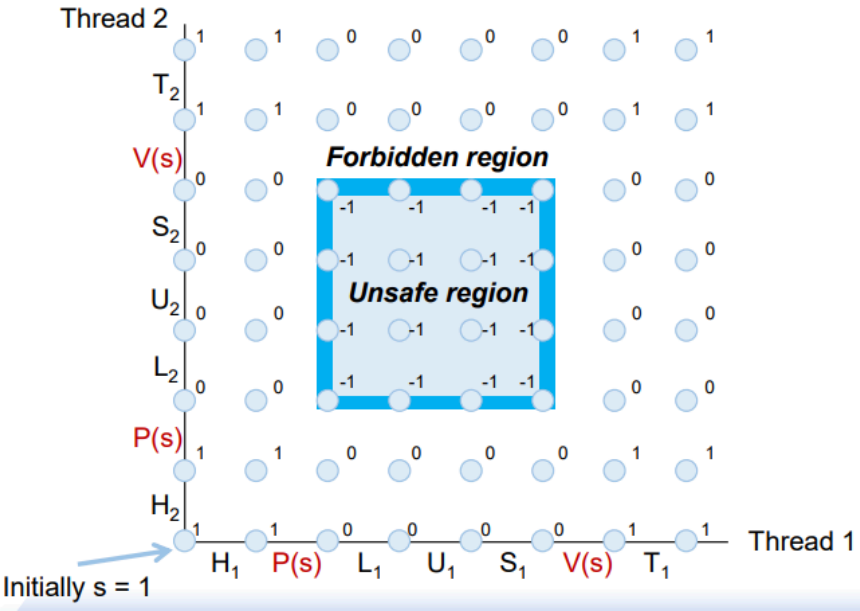
        cnt++;

        V(&mutex);

    }

}
```

> However, it's much slower than badcnt.c



通过信号量s（初始值设置为1）上的P、V操作保护关键区域，提供对共享变量的互斥访问

信号不变量创建了一个禁区，其中包含不能由任何路径进入的不安全区域。

3. 共享资源

基本思想：线程使用信号量来通知另一个线程某些条件为真
使用计数信号量来跟踪资源使用状态，并通知其他线程
使用互斥信号量保护对共享资源的访问

典型的例子：生产者 / 消费者问题 The Producer-Consumer Problem



> N个元素的生产者消费者问题：需要一个互斥信号量和两个计数信号量
> `mutex`：对缓冲区的互斥访问
> `slots`：计算缓冲区中可用的槽位
> `items`：计算缓冲区中可用的条目
> `slots` + `items` = `n`

1. 结构体实现

```
xxxxxxxxxx

//sbuf.h

#include <semaphore.h>
```

```
typedef struct {
    int *buf; /* Buffer array */
    int n; /* Maximum number of slots */
    int front; /* buf[(front+1)%n] is first item */
    int rear; /* buf[rear%n] is last item */
    sem_t mutex; /* Protects accesses to buf */
    sem_t slots; /* Counts available slots */
    sem_t items; /* Counts available items */
} sbuf_t;

void sbuf_init(sbuf_t *sp, int n) {
    sp->buf = Calloc(n, sizeof(int));
    sp->n = n; /* Buffer holds max of n items */
    sp->front = sp->rear = 0; /* Empty buffer iff front == rear */
    Sem_init(&sp->mutex, 0, 1); /* Binary semaphore for locking */
    Sem_init(&sp->slots, 0, n); /* buf has n empty slots */
    Sem_init(&sp->items, 0, 0); /* Initially, buf has 0 items */
}
void sbuf_deinit(sbuf_t *sp) {
    Free(sp->buf);
}
void sbuf_insert(sbuf_t *sp, int item) {
    // insert item onto the rear of shared buffer sp
    P(&sp->slots); /* Wait for available slot */
    P(&sp->mutex); /* Lock the buffer */
    sp->buf[(++sp->rear) % (sp->n)] = item; /* Insert the item */
    V(&sp->mutex); /* Unlock the buffer */
    V(&sp->items); /* Announce available item */
}
int sbuf_remove(sbuf_t *sp) {
    // remove and return the first item from buffer sp
```

```
    int item;

    P(&sp->items); /* Wait for available item */

    P(&sp->mutex); /* Lock the buffer */

    item = sp->buf[(++sp->front) % (sp->n)]; /* Remove the item */

    V(&sp->mutex); /* Unlock the buffer */

    V(&sp->slots); /* Announce available slot */

    return item;

}
```

- 补充：理发师理发问题

为解决同步问题，我们使用三个**信号量**（semaphores）。
mutex：控制对全局变量 customers 的访问；　customer：顾客的资源信号量；
barber：理发师的资源信号量。
（1）指出这三个信号量的初始值。
mutex =1　customer=0　barber=0
（2）仅使用以下语句将上述代码补充正确，每个语句不限使用次数。
P(&mutex);　V(&mutex);　P(&customer);　V(&customer);　P(&barber);　V(&barber);

补充后的代码是：

```
void* customer() {
   P(&mutex);
   if (customers > N) {
      V(&mutex);
      return NULL;
   }
   customers ++;
   V(&customer);
   V(&mutex);
   P(&barber);
   getHairCut();
   return NULL;
}
```

```
void* barber() {
   while(1) {
      P(&customer);
      P(&mutex);
      customers --;
      V(&barber);
      V(&mutex);
      cutHair();
   }
}
```

2. 竞争：程序的正确性依赖于某种顺序

发生race的示例：

```
xxxxxxxxxx

// race.c

#include <pthread.h>

#include <cstdlib>
```

```
#include <cstdio>


void *thread(void *vargp) {
    int myid = *((int *)vargp);
    printf("Hello from thread %d\n", myid);
    return NULL;
}


int main() {
    pthread_t tid[N];
    int i;
    for (i = 0; i < N; i++)
        Pthread_create(&tid[i], NULL, thread, &i);
    for (i = 0; i < N; i++)
        Pthread_join(tid[i], NULL);
    exit(0)
}
```

主线程对i的改变与子线程对vargp的引用之间发生**竞争**

race的消除：

```
xxxxxxxxxx
#include <cstdlib>
#include <cstdio>
#include <pthread.h>
#include <malloc.h>


void *thread(void *vargp) {
    int myid = *((int *)vargp);
    Free(vargp);
    printf("Hello from thread %d\n", myid);
    return NULL;
}
```

```
int main() {
    pthread_t tid[N];
    int i, *ptr;
    for (i = 0; i < N; i++) {
        ptr = Malloc(sizeof(int));
        *ptr = i;
        Pthread_create(&tid[i], NULL, thread, ptr);
    }
    for (i = 0; i < N; i++)
        Pthread_join(tid[i], NULL);
    exit(0);
}
```

4. 死锁

定义：等待一个永远不会发生的条件
经典场景：进程1和进程2需要两个资源A和B以继续，进程1获得了A等待B，进程2获得了B等待A

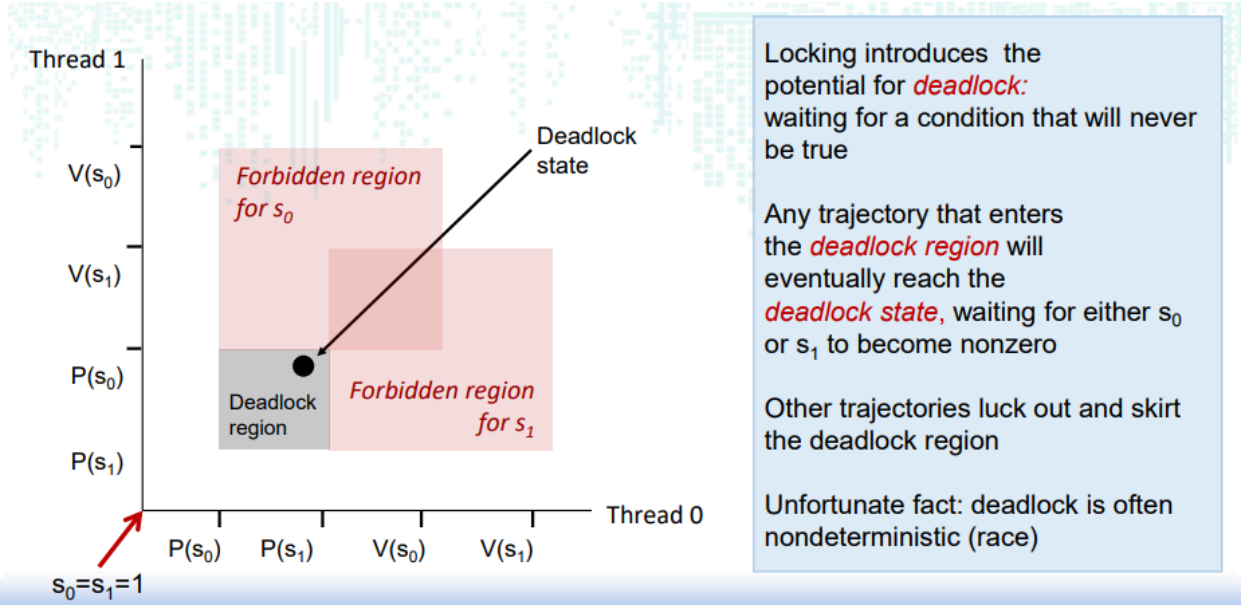
```
xxxxxxxxxx
#include <cstdlib>
#include <cstdio>
#include <pthread.h>
#include <unistd.h>
#include <semaphore.h>

int cnt = 0;
int NITERS;
sem_t mutex[2];

void *count(void *vargp) {
    int i;
    int id = (int)(__int64_t)vargp;
    for (i = 0; i < NITERS; i++) {
```

```
        sem_wait(&mutex[id]); sem_wait(&mutex[1-id]);
        cnt++;
        sem_post(&mutex[id]); sem_post(&mutex[1-id]);
    }
    return NULL;
}

int main(int argc, char** argv) {
    NITERS = atoi(argv[1]);
    pthread_t tid[2];
    sem_init(&mutex[0], 0, 1); /* mutex[0] = 1 */
    sem_init(&mutex[1], 0, 1); /* mutex[1] = 1 */
    pthread_create(&tid[0], NULL, count, (void*) 0);
    pthread_create(&tid[1], NULL, count, (void*) 1);
    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    printf("cnt=%d\n", cnt);
    exit(0);
}
```



Locking introduces the potential for *deadlock:* waiting for a condition that will never be true

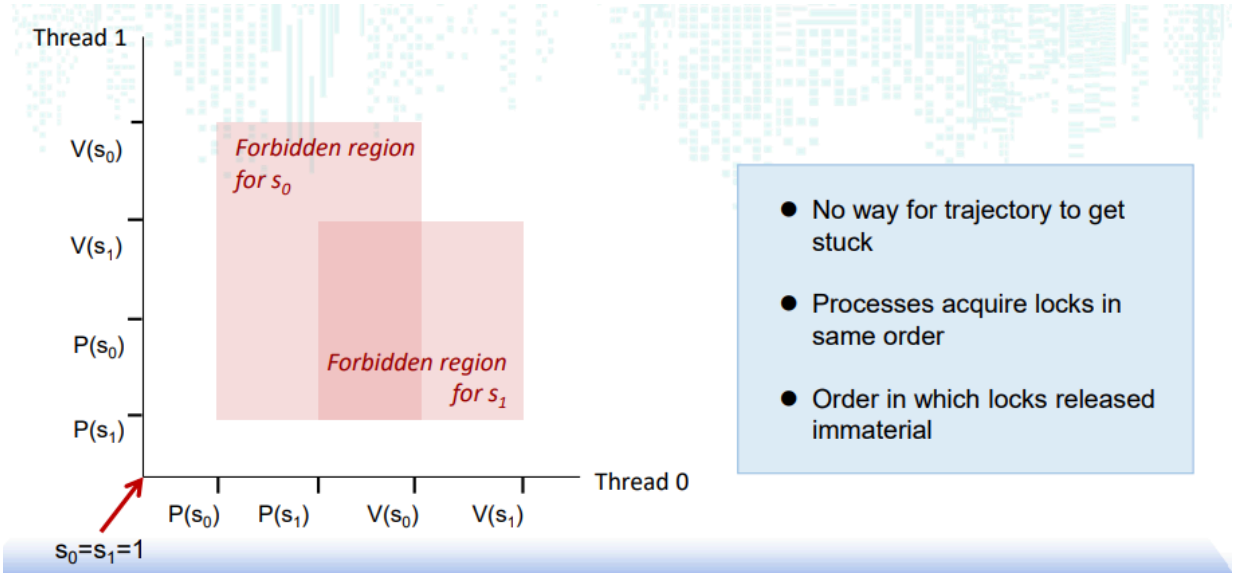
Any trajectory that enters the *deadlock region* will eventually reach the *deadlock state*, waiting for either s_0 or s_1 to become nonzero

Other trajectories luck out and skirt the deadlock region

Unfortunate fact: deadlock is often nondeterministic (race)

Tid[0]:	Tid[1]:
$P(s_0)$;	$P(s_1)$;
$P(s_1)$;	$P(s_0)$;
cnt++;	cnt++;
$V(s_0)$;	$V(s_1)$;
$V(s_1)$;	$V(s_0)$;

避免死锁：使用同样的顺序加锁



Tid[0]:	Tid[1]:
P(s0);	P(s0);
P(s1);	P(s1);
cnt++;	cnt++;
V(s0);	V(s1);
V(s1);	V(s0);

```
xxxxxxxxxx

void *count(void *vargp) {

    int i;

    int id = (int) vargp;

    for (i = 0; i < NITERS; i++) {

        P(&mutex[0]); P(&mutex[1]);
```

```
        cnt++;
        V(&mutex[id]); V(&mutex[1-id]);
    }
    return NULL;
}
```

7 X86 汇编编程

7.1 Hello World

基本概念

```
xxxxxxxxxx
#include <stdlib.h>
#include <stdio.h>
int main() {
    printf("Hello world\n");
    exit(0);
    return 0;
}
```

`gcc -S -Og helloworld.c`

```
xxxxxxxxxx
        .file   "hello.c"
        .section    .rodata
.LC0:
        .string "Hello world"
        .text
        .globl  main
        .type   main, @function
main:
        subq    $8, %rsp
        movl    $.LC0, %edi
```

```
        call    puts

        movl    $0, %edi

        call    exit
```

1. **汇编指示 Directives**
   汇编代码中以`.`开头的行都是汇编指示，如`.file`，`.def`，`.text`等，用以直到汇编器如何进行汇编。其中`.file`，`.def`，`.CFI`等均用于调试（可以忽略）
   示例：`.globl    main`表示汇编器符号`main`是全局的，`.LC0`则不是全局可见的
2. **符号 Symbol**
   以`:`结尾的字符串（如`main:`）是用以表示变量或者函数的地址的符号
3. `.text`或`.section    .text`表示代码段
   `.data`或`.section    .rodata`表示（只读）数据段
4. 按16字节对齐：`.p2align 4,,15`**指定下一行的对齐方式**第一个参数表示按2的`4`次幂字节对齐，第二个参数表示对齐时额外空间用什么数据来填充，第三个参数表示最多允许额外填充多少字节

> `int c = 1;`：初始化全局变量
>
> ```
> xxxxxxxxxx
>         .section .rodata
> LC0:    .string     "Hello World"
>         .global     c
>         .data
>         .align      4
> c:      .long       1
> ```

编译命令

`as my_file.s -o my_object_file.o`：加入`-gstabs`选项产生带调试信息的object文件
`ld my_object_file.o -o my_exe_file`：可以有多个`.o`文件；加入`-g`选项产生调试信息

"Hello World.c"

```
xxxxxxxxxx
.data # 数据段
msg: # 字符串的地址
    .ascii "Hello World\n"
    len = .-msg # “.”表示当前地址


.text # 代码段
```

```
.globl _start # 汇编程序的入口

_start:

    movq    $len, %rdx

    movq    $msg, %rsi

    movq    $1, %rax # 系统输出（write系统调用）

    movq    $1, %rdi # stdout

    syscall

    movq    $60, %rax # 程序退出

    movq    $0, %rdi # 退出值

    syscall
```

1. 系统调用

X84-64 Linux下的系统调用是通过`syscall`指令（一种异常机制）实现的
`%rax`存放系统调用的**功能号**
传给系统调用的参数必须按顺序存放到寄存器`%rdi`，`%rsi`，`%rdx`，`%r10`，`%r8`，`r9`中
当系统调用完成之后，返回值可以再寄存器`%rax`中获得（一般<0表示错误）

Call Code (rax)	System Service	Description
0	**SYS_read**	**Read data**
		rdi = file descriptor (of where to read from) rsi = address of where to store data rdx = count of bytes to read
	If unsuccessful, returns negative value If successful, returns count of characters actually read	
1	**SYS_write**	**Write data**
		rdi = file descriptor (of where to write to) rsi = address of data to write rdx = count of bytes to write
	If unsuccessful, returns negative value If successful, returns count of characters actually written	

Call Code = 2	SYS_open	Open a file.
		rdi = address of NULL terminated file name rsi = file status flags (typically 0 RD0NLY)
	If unsuccessful, returns negative value. If successful, returns file descriptor.	
3	SYS_close	Close an open file.
		rdi = file descriptor of open file to close
	If unsuccessful, returns negative value.	
8	SYS_lseek	Reposition the file read/write file offset.
		rdi = file descriptor (of where to write to) rsi = offset rdx = origin
	If unsuccessful, returns negative value	

57	SYS_fork	Fork current process
59	SYS_execve	Execute a program
		rdi = Address of NULL terminated string for name of program to execute.
60	SYS_exit	Terminate executing process.
		rdi = exit status (typically 0)
85	SYS_creat	Open/Create a file.
		rdi = address of NULL terminated file name rsi = file mode flags
	If unsuccessful, returns negative value. If successful, returns file descriptor.	
96	SYS_gettimeofday	Get date and time of day
		rdi = address of time value structure rsi = address of time zone structure
	If unsuccessful, returns negative value. If successful, returns information in the passed structures.	

2. 处理命令行参数

相当于C语言形式`int main(int argv, char *argv[])`
(*) 当一个可执行程序通过命令行启动时，命令行参数将被保存到**栈**中

```
xxxxxxxxxx

.text

.globl _start

_start:

    popq %rsi # argc

vnext:

    popq %rsi # argv[?]

    testq %rsi, %rsi # 空指针表明结束

    jz exit # 即je
```

```
    movq %rsi, %rbx
    xorq %rdx, %rdx # rdx = 0
strlen:
    movb (%rbx), %al
    incq %rdx
    incq %rbx
    testb %al, %al
    jnz strlen
    movb $10, -1(%rbx) # 10是换行键
    movq $1, %rax # 系统调用号(sys_write)
    movq $1, %rdi # 文件描述符(stdout)
    syscall
    jmp vnext
exit:
    movq $60, %rax # 程序退出
    movq $0, %rdi # 退出值
    syscall
```

> **gdb调试**:

```
(gdb) b _start
Breakpoint 1 at 0x401000
(gdb) r 12345 6789 101
Starting program: /mnt/c/Users/Catherine/Desktop/Test/test 12345 6789 101

Breakpoint 1, 0x0000000000401000 in _start ()
(gdb) info registers
rax            0x0                 0
rbx            0x0                 0
rcx            0x0                 0
rdx            0x0                 0
rsi            0x0                 0
rdi            0x0                 0
rbp            0x0                 0x0
rsp            0x7fffffffedb70      0x7fffffffedb70
r8             0x0                 0
r9             0x0                 0
r10            0x0                 0
r11            0x0                 0
r12            0x0                 0
r13            0x0                 0
r14            0x0                 0
r15            0x0                 0
rip            0x401000            0x401000 <_start>
eflags         0x202               [ IF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0
(gdb) x /6xg 0x7fffffffedb70
0x7fffffffedb70: 0x0000000000000004      0x00007fffffffeddf7
0x7fffffffedb80: 0x00007fffffffede20      0x00007fffffffede26
0x7fffffffedb90: 0x00007fffffffede2b      0x0000000000000000
(gdb) x /s 0x00007fffffffeddf7
0x7fffffffeddf7: "/mnt/c/Users/Catherine/Desktop/Test/test"
(gdb) x /s 0x00007fffffffede20
0x7fffffffede20: "12345"
(gdb) x /s 0x00007fffffffede26
0x7fffffffede26: "6789"
(gdb) x /s 0x00007fffffffede2b
0x7fffffffede2b: "101"
```

3. 调用`lib_c`库函数

汇编命令：`as hello.s -o hello.o`

`ld -lc -dynamic-linker /lib64/ld-linux-x86-64.so.2 hello.o -o hello`

```
xxxxxxxxxx

.section    .rodata

.LC0:

    .string "Hello World"

.text
```

```
.globl  _start

_start:

    movl    $.LC0, %edi

    call    puts

    movl    $0, %edi

    call    exit
```

7.2 汇编小结

1. 三个常用的段

   `.data`：数据段（声明带有初始值的数据）
   `.bss`：数据段（声明无需初始化的数据）
   `.text`：正文段（程序指令）

2. 程序入口地址：`_start`符号表示默认的起始点
   如想汇编内部的符号能够被外部模块访问，需赋予`.globl`属性，如`.globl    _start`

1. data段

```
xxxxxxxxxx

output:

    .ascii  "hello world"

pi:

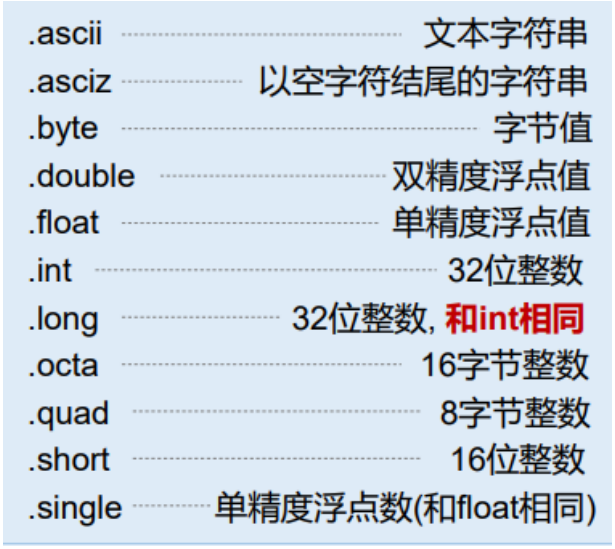
    .float  3.14

# 声明可以在一行中定义多个值，如:

age:

    .int    20, 10, 30, 40 # 数组
```

类型说明:



2. bss段

和`data`段不同, 无需声明特定的数据类型, 只需声明为所需目的保留的一定长度的内存即可

`.comm`声明为未初始化的**全局**内存区域

`.lcomm` 声明为未初始化的**局部**内存区域

```
xxxxxxxxxx

.section  .bss

    .lcomm buffer, 1000

#该语句把1000字节的内存地址赋予buffer, 外部模块不能访问他们
```

divl S R[%edx] = R[%edx]:R[%eax] % S;
R[%eax] = R[%edx]:R[%eax] / S; 无符号除法

```
        movq $strNum, %rbx
        movq $0, %rdi
popLoop:
        popq %rax
        addb  $48, %al     # 字符0的ASCII码是48
        movb %al, (%rbx,%rdi,1)
        incq  %rdi
        loop popLoop

        movb $NULL, (%rbx,%rdi,1)

        movq   %rdi, %rdx
        movq   %rbx, %rsi
        movq   $1, %rax       # 系统调用号(sys_write)
        movq   $1, %rdi       # 文件描述符(stdout)
        syscall

exit:
        ...
```

```
.equ NULL,  0

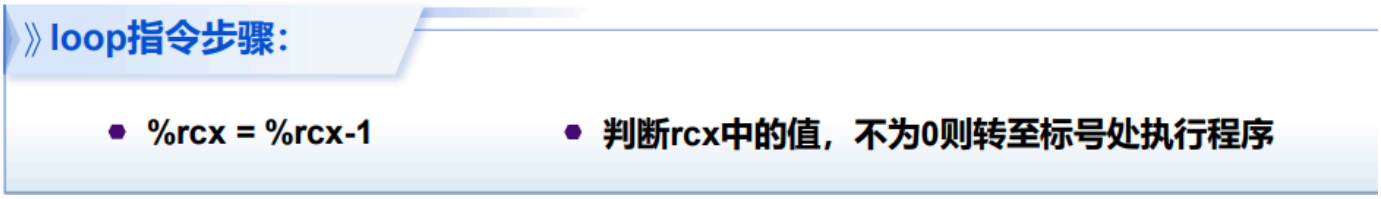
.section  .data
intNum:
        .int   1498

.section .bss
        .lcomm strNum, 10

.section .text
.globl _start
_start:
        movl   intNum, %eax
        movq  $0,   %rcx
        movl   $10, %ebx
divideLoop:
        movl   $0, %edx
        divl    %ebx
        pushq %rdx  #push remainder
        incq    %rcx
        cmpl   $0, %eax
        jne     divideLoop
```

指令	效果	描述
imull S	R[%edx]:R[%eax] = S * R[%eax]	有符号乘(结果64位)
mull S	R[%edx]:R[%eax] = S * R[%eax]	无符号乘(结果64位)
cltd	sign-extend %eax → %edx:%eax	转换为8字节 （指令也可写作CDQ） 还有类似指令cqto
idivl S	R[%edx] = R[%edx]:R[%eax] % S; R[%eax] = R[%edx]:R[%eax] / S;	有符号除法，保存余数和商
divl S	R[%edx] = R[%edx]:R[%eax] % S; R[%eax] = R[%edx]:R[%eax] / S;	无符号除法，保存余数和商





7.3 汇编示例

1. 阶乘（`factorial.s`）

```
xxxxxxxxxx

.section .text

.globl factorial    #this is unneeded unless we want to share it

.globl _start

_start:

    movl $4, %edi   #The factorial takes one argument -

                    #the number we want a factorial of.
```

```
    call factorial  #run the factorial function
    movl %eax, %edi #factorial returns the answer in %eax, but we
                    #want it in %edi to send it as our exit status
    movq $60, %rax  #exit code
    syscall


#This is the actual function definition:  factorial(n)
.type factorial, @function
factorial:
    movl $1, %eax
    cmpl $1, %edi   #If the number is 1, that is our base case, and
                    #we simply return
    je end_factorial
    pushq %rdi
    decl %edi       #otherwise, decrease the value
    call factorial  #call itself
    popq %rdi
    imull %edi, %eax #multiply that by the result of the last call to
                     #factorial; the answer is stored in %eax.
end_factorial:
    ret
```

imull S, D

这里有两个操作数，它计算S和D的乘积并截断为双字，然后存储在D中；无"带符号数"与"无符号数"的区分

2. C语言调用汇编

```
xxxxxxxxxx
#include<stdio.h>
extern void stats(int[], int, int *, int *);


int main() {
    int lst[] = {1, -2, 3, -4, 5, 7, 9, 11};
```

```
    int len = 8;
    int sum, ave;
    stats(lst, len, &sum, &ave);
    printf ("Stats:\n");
    printf ("Sum = %d \n", sum);
    printf ("Ave = %d \n", ave);
    return 0;
}
```

```
xxxxxxxxxx
# Function to find the integer sum and integer average for a passed list of signed integers.
# Call:
# stats(lst, len, &sum, &ave);
# Arguments Passed:
# 1) rdi - address of array
# 2) rsi - length of passed array
# 3) rdx - address of variable for sum
# 4) rcx - address of variable for average
# Returns:
#
.section .text
.globl stats
stats:
    pushq %r12                  # callee saved
    movq $0, %r11               # index
    movl $0, %r12d              # sum
sumLoop:
    movl (%rdi,%r11,4), %eax    #get lst[i]
    addl %eax, %r12d            #update sum
    incq %r11                   #index++
    cmpq %rsi, %r11
    jb sumLoop
```

```
    movl %r12d, (%rdx)          #return sum
    movl %r12d, %eax
    cltd                        #sign-extend %eax -> %edx:%eax
    idivl %esi
    movl %eax, (%rcx)           #return average


    #Done, return …
    popq %r12
    ret
```

3. 文件处理示例

略