Overview

考试题型

- 单选题：共20题，每题1分
- 问答题：共8题，每道10分

课程总览

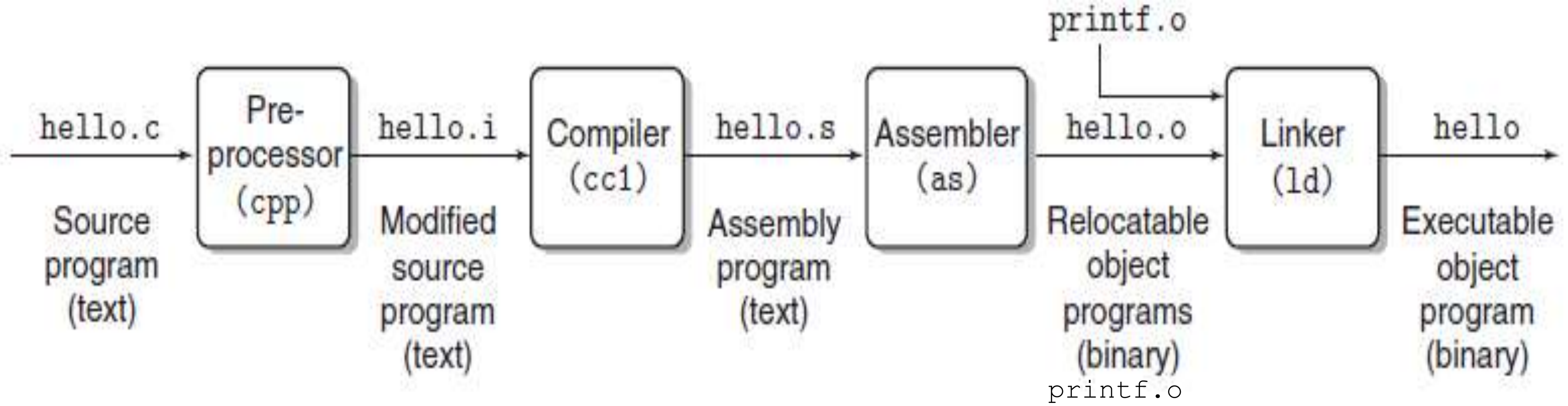
- 系统漫游
- 程序结构与执行
  - 2章-信息表示与处理
  - 3章-程序机器表示
  - 4章-处理器体系结构
  - 5章-优化程序性能
  - 6章-存储器层次结构
- 系统运行程序
  - 7章-链接
  - 8章-异常控制流
  - 9章-虚拟内存
- 程序间交互通信
  - 10章-系统级I/O
  - 11章-网络编程
  - 12章-并发编程

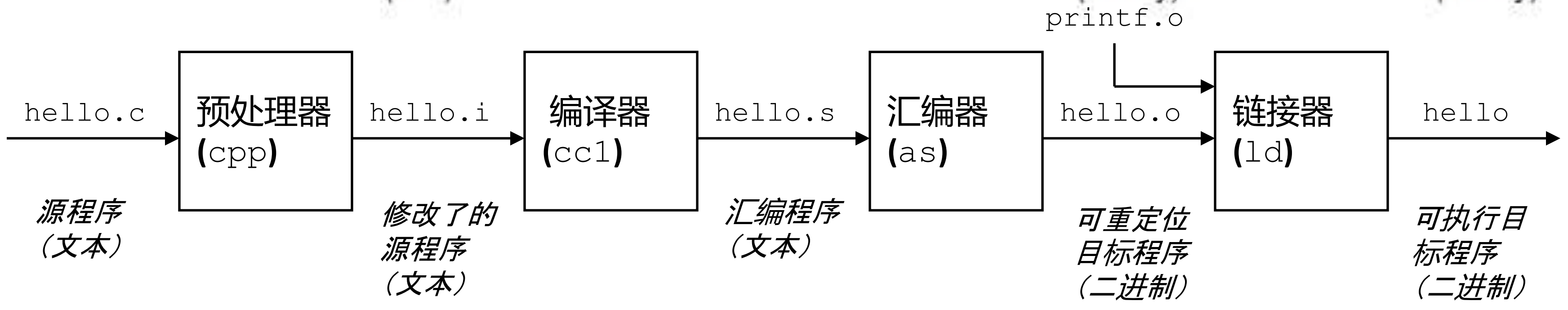
系统漫游

- 源程序
- 编译系统的阶段和用处
- 处理器硬件组成与运行过程
- 高速缓存
- 操作系统管理视图
  - 进程、线程、虚拟内存、文件
- 并发与系统层次抽象表示

编译系统 Compilation System

- Linux> gcc -o hello hello.c



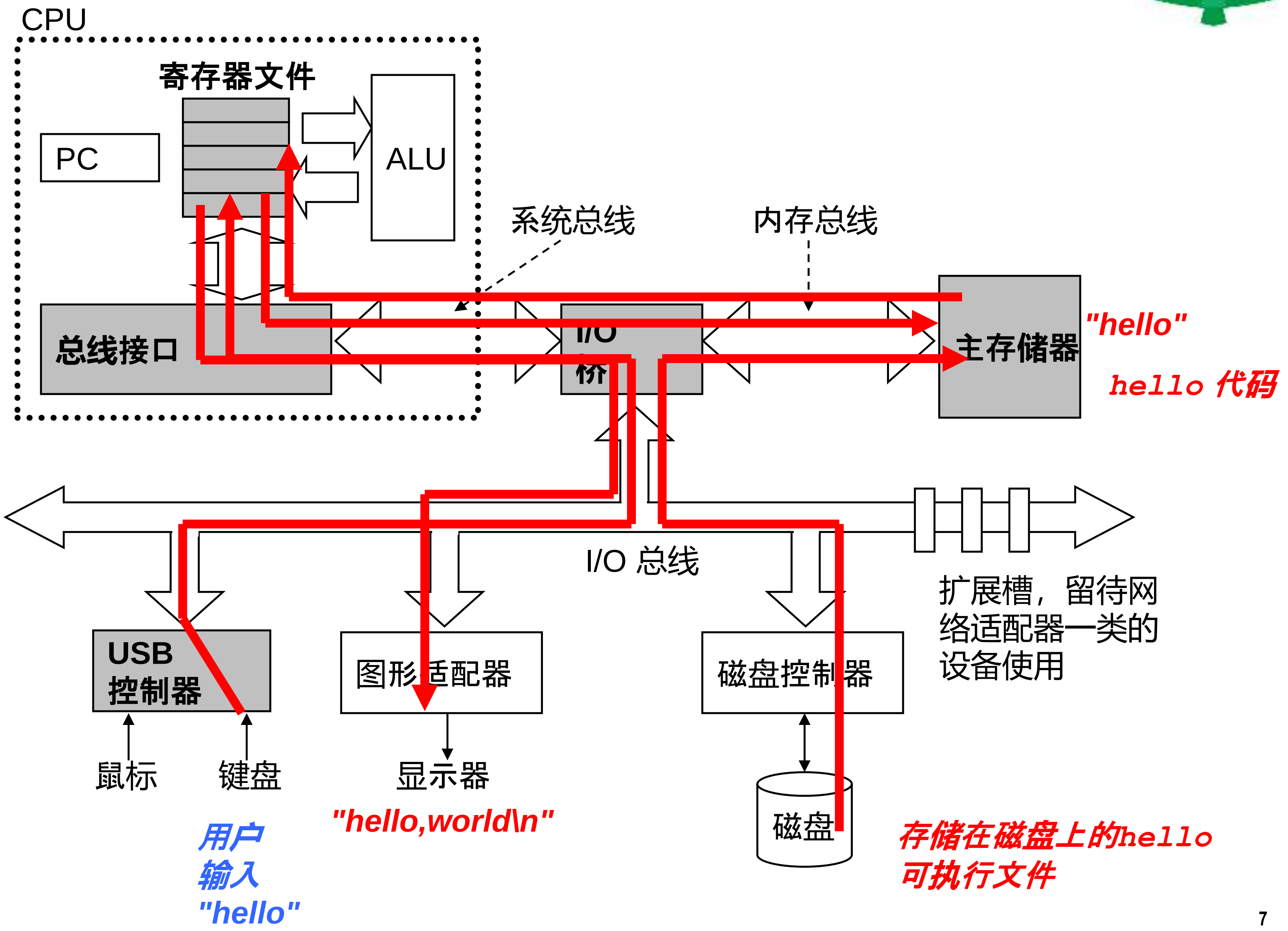


```
main:
   subq    $8, %rsp
   movl    $.LC0, %edi
   call    puts
   movl    $0, %eax
   addq    $8, %rsp
   ret
```

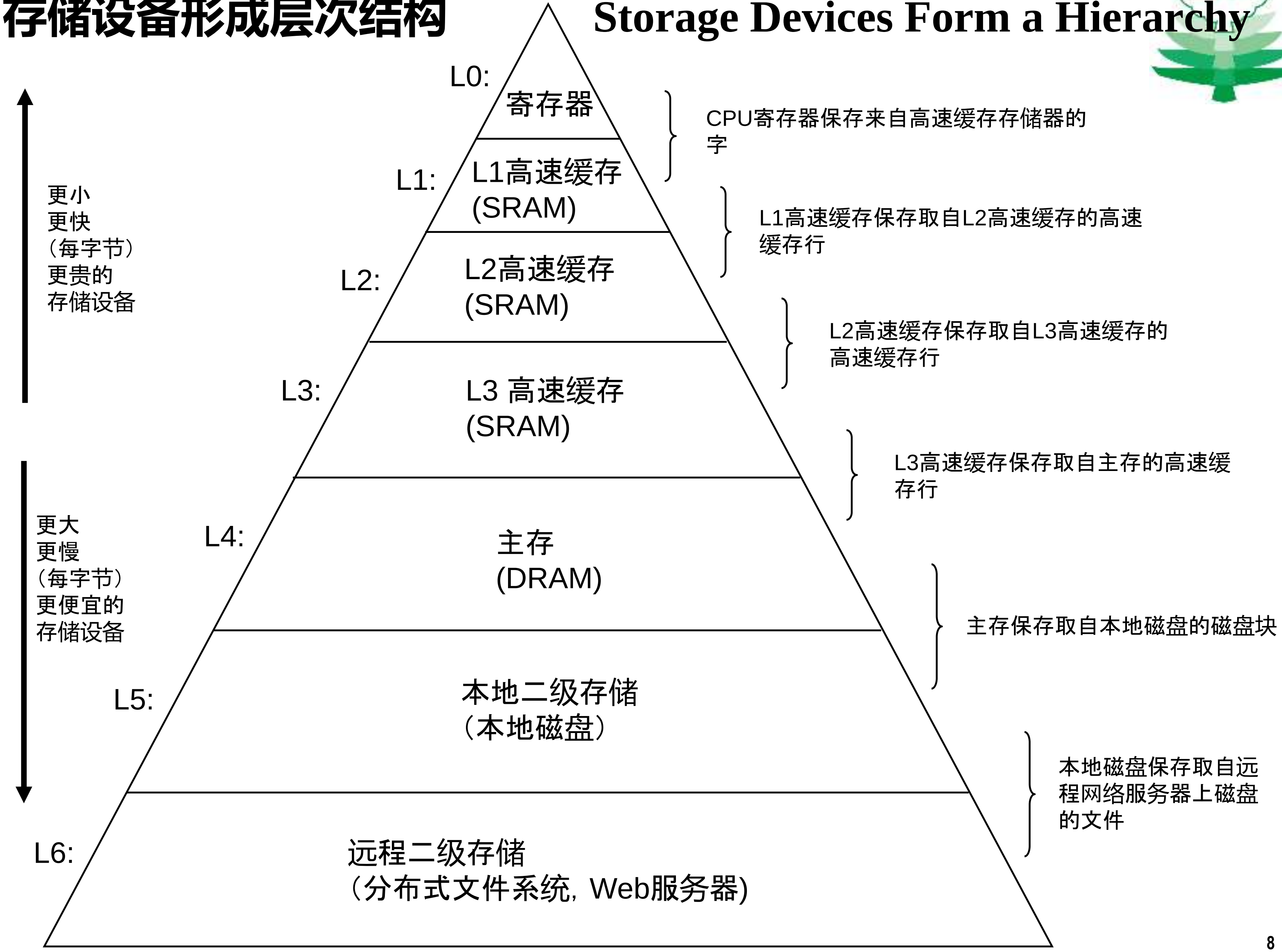
了解编译系统如何工作是大有益处的
It Pays to Understand How Compilation Systems Work

- 优化程序性能 Optimizing program performance.
  - 现代编译器通常生成很好的代码 Modern compilers usually produce good code.
  - 需要理解机器代码和编译器将不同的C语句转换成机器代码的方式 need a basic understanding of machine-level code and how the compiler translates different C statements into machine code.
- 理解链接时出现的错误 Understanding link-time errors.
  - 一些令人困惑的编程错误都与链接器操作有关，特别是构建大型软件 some of the most perplexing programming errors are related to the operation of the linker, especially trying to build large software systems.
- 避免安全漏洞 Avoiding security holes.
  - 缓冲区溢出漏洞是造成大多数网络和互联网服务器上安全漏洞的主要原因 buffer overflow vulnerabilities have accounted for many of the security holes in network and Internet servers.

运行hello程序 Running the hello Program



存储设备形成层次结构 Storage Devices Form a Hierarchy



L0:
寄存器
L1:
L1高速缓存 (SRAM)
L2:
L2高速缓存 (SRAM)
L3:
L3 高速缓存 (SRAM)
L4:
主存 (DRAM)
L5:
本地二级存储 （本地磁盘）
L6:
远程二级存储 （分布式文件系统，Web服务器)
更小 更快 （每字节） 更贵的 存储设备
更大 更慢 （每字节） 更便宜的 存储设备
CPU寄存器保存来自高速缓存存储器的字
L1高速缓存保存取自L2高速缓存的高速缓存行
L2高速缓存保存取自L3高速缓存的高速缓存行
L3高速缓存保存取自主存的高速缓存行
主存保存取自本地磁盘的磁盘块
本地磁盘保存取自远程网络服务器上磁盘的文件

操作系统管理硬件
Operating System Manages Hardware

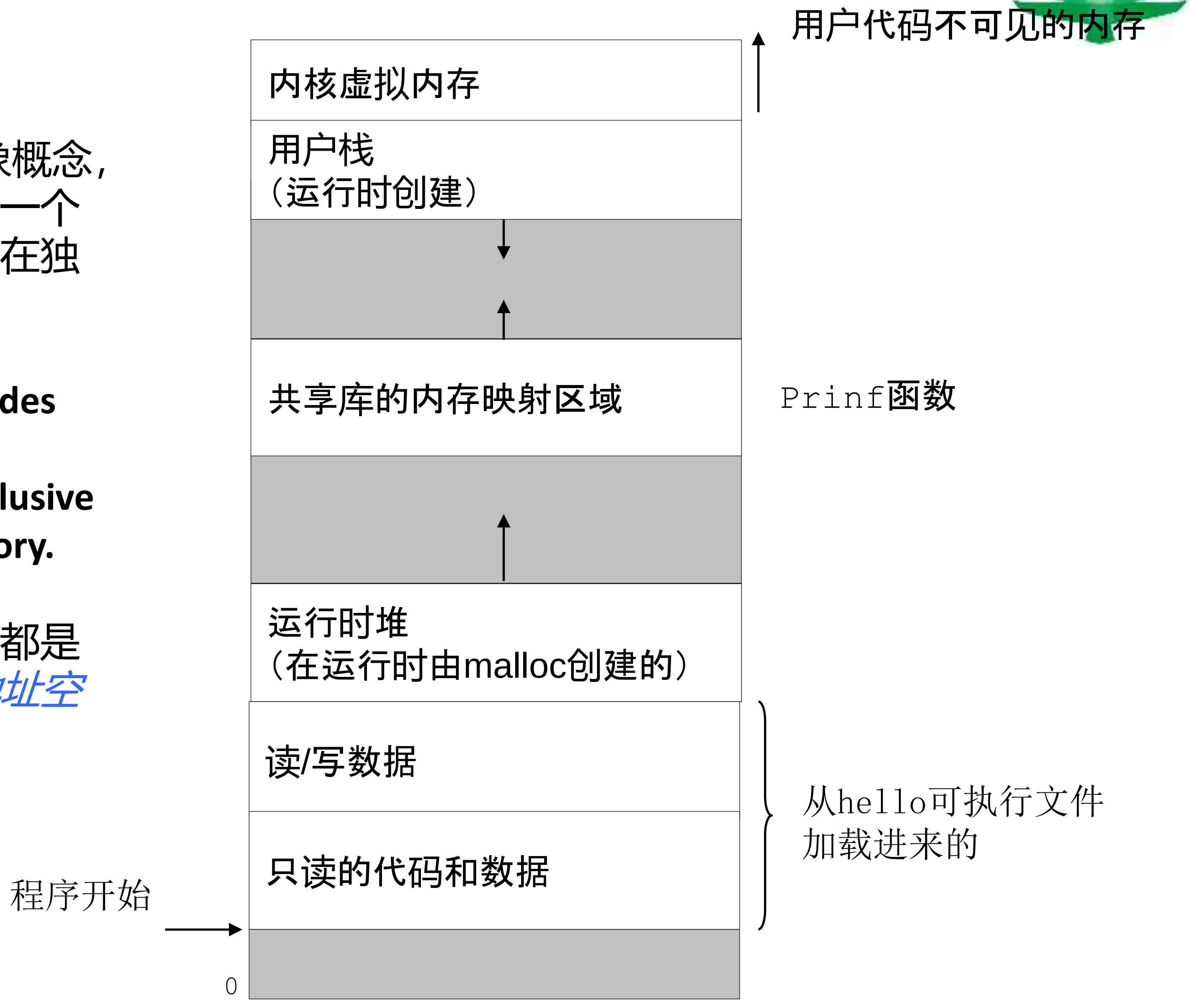
- 将操作系统看作应用程序和硬件之间插入的一个软件层 think of the operating system as a layer of software interposed between the application program and the hardware
- 两个基本目的：two primary purposes:
  - 保护硬件防止被应用误用 to protect the hardware from misuse by runaway applications.
  - 给应用提供简单统一的机制操作复杂和差异巨大的低级硬件设备 to provide applications with simple and uniform mechanisms for manipulating complicated and often wildly different low-level hardware devices.
- 基本抽象：进程、虚存和文件 fundamental abstractions: processes, virtual memory, and files.
  - 文件是I/O设备的抽象 files are abstractions for I/O devices,
  - 虚拟存储器是主存和磁盘I/O设备的抽象 virtual memory is an abstraction for both the main memory and disk I/O devices
  - 进程是处理器、主存和I/O设备的抽象 processes are abstractions for the

进程的虚拟地址空间

虚拟内存 是一个抽象概念，它为每个进程提供了一个假象，即每个进程都在独占地使用主存。

Virtual memory is an abstraction that provides each process with the illusion that it has exclusive use of the main memory.

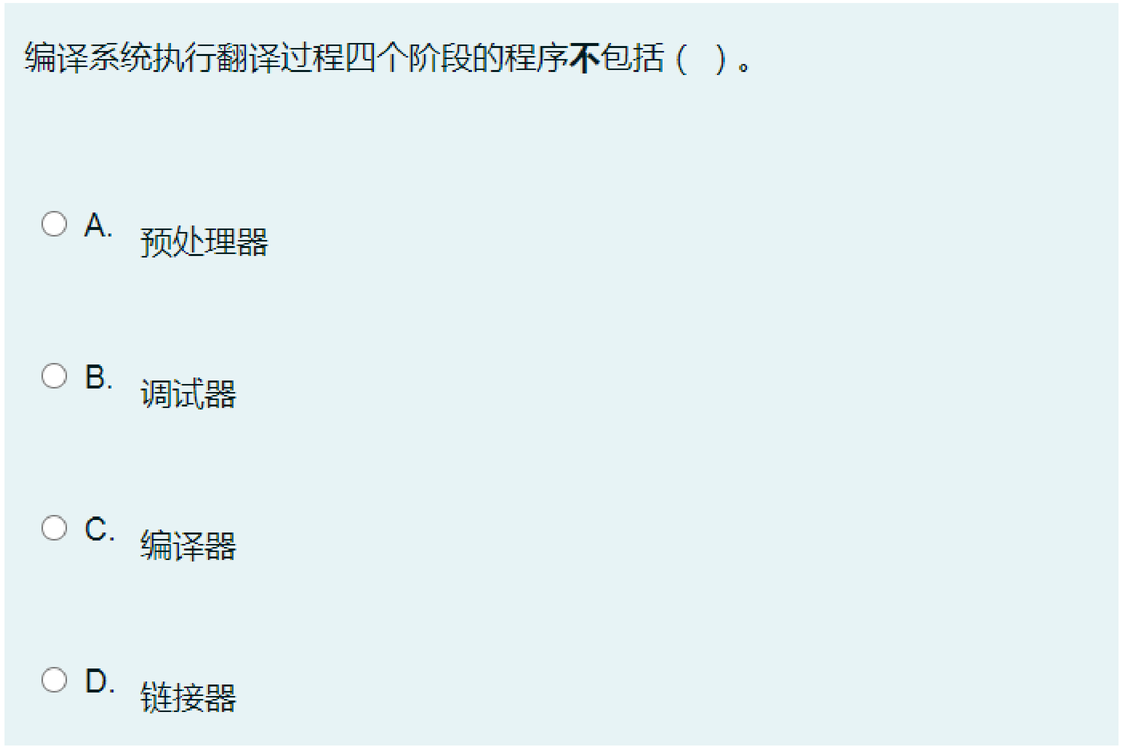
每个进程看到的内存都是一致的，称为*虚拟地址空间*。



重要主题
Important Themes

- 并发和并行 Concurrency and Parallelism
  - 并发指同时具有多个活动的系统这个通用概念 concurrency refer to the general concept of a system with multiple, simultaneous activities
  - 并行指用并发使系统运行更快 parallelism refer to the use of concurrency to make a system run faster.
  - 并行可以在计算机系统的多个抽象层次上运用 Parallelism can be exploited at multiple levels of abstraction in a computer system.
  - 三个层次，在系统层次结构中从最高到最低级 three levels, working from the highest to the lowest level in the system hierarchy
- 线程级并发 Thread-Level Concurrency
  - 在进程抽象基础上，多个程序同时执行，导致并发 building on the process abstraction, multiple programs execute at the same time, leading to concurrency.
  - 使用线程可以在单一进程中有多个控制流 With threads, have multiple control flows executing within a single process.

例题



编译系统执行翻译过程四个阶段的程序**不**包括（ ）。

○ A. 预处理器

○ B. 调试器

○ C. 编译器

○ D. 链接器

信息表示与处理

- 信息位级表示与存储
- 位级运算（C语言）
- 无符号整数编码与运算
  - 编码表示、扩展和截断
  - 加法、乘法、除以2的幂
- 补码整数编码与运算
  - 编码表示、扩展和截断
  - 加法、乘法、除以2的幂
  - 无符号与补码的转换
- 浮点数表示与运算

二进制数的性质
Binary Number Property

声明/断言 Claim

$$1 + 1 + 2 + 4 + 8 + \ldots + 2^{w\text{-}1} = 2^w$$

$$1 + \sum_{i=0}^{w-1} 2^i = 2^w$$

- **w = 0:**
  - $1 = 2^0$
- **假设对于w-1为真 Assume true for w-1:**
  - $$\underbrace{1 + 1 + 2 + 4 + 8 + \ldots + 2^{w\text{-}1}}_{=\ 2^w} + 2^w = 2^w + 2^w = 2^{w+1}$$

编码字节值 Encoding Byte Values

- **一个字节包含8比特位 Byte = 8 bits**
  - 十进制取值范围 Decimal: 0_{10} to 255_{10}
    - $255=2^8 - 1$
  - 二进制取值范围 Binary 00000000_2 to 11111111_2
  - 十六进制取值范围 Hexadecimal 00_{16} to FF_{16}
    - 基数为16的数值表示Base 16 number representation
    - 字符0-9和A-F Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
    - C语言中写作前导‘0x’，以下情况之一 Write in C with leading ‘0x’, either case
      – $0101\ 1010_2$ = 0x5a = 0x5A = 0X5a

$$\texttt{15213: } 0011\ \underbrace{1011}_{\texttt{3}}\ \underbrace{0110}_{\texttt{B}}\ \underbrace{1101}_{\texttt{6}}\ \underbrace{}_{\texttt{D}}$$

十六进制 Hex	十进制 Decimal	二进制 Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

组合字节以创建标量数据类型
Combine bytes to make *scalar data types*

	大小(字节数)Size(# of bytes)	
C Data Type	Typical 32-bit	Typical 64-bit
`char`	1	1
`short`	2	2
`int`	4	4
`long`	4	8
`float`	4	4
`double`	8	8
pointer	4	8
	“ILP32”	“LP64”

C语言中的比特级运算 Bit-Level Operations in C

- **C中可用运算 Operations &, |, ~, ^ Available in C**
  - 运用到任何整数类数据类型 Apply to any “integral” data type
    - long, int, short, char, unsigned
  - 参数视为比特位向量 View arguments as bit vectors
  - 参数运用到每个比特位 Arguments applied bit-wise
- **举例（字符数据类型） Examples (Char data type)**
  - ~0x41 → 0xBE
    - $\sim 0100\ 0001_2 \rightarrow 1011\ 1110_2$
  - ~0x00 → 0xFF
    - $\sim 0000\ 0000_2 \rightarrow 1111\ 1111_2$
  - 0x69 & 0x55 → 0x41
    - $0110\ 1001_2\ \&\ 0101\ 0101_2 \rightarrow 0100\ 0001_2$
  - 0x69 | 0x55 → 0x7D
    - $0110\ 1001_2 \mid 0101\ 0101_2 \rightarrow 0111\ 1101_2$

对比：C语言中的逻辑运算
Contrast: Logic Operations in C

- **对比比特位级的操作符 Contrast to Bit-Level Operators**
  - **逻辑运算 Logic Operations: &&, ||, !**
    - 视0为假 View 0 as “False”
    - 任何非零视为真 Anything nonzero a
    - 总是返回0或1 Always return
    - 提前终止 Early termination
- **举例（字符数据类型） Exa**
  - !0x41 → 0x00
  - !0x00 → 0x01
  - !!0x41→ 0x01

  - 0x69 && 0x55 → 0x01
  - 0x69 || 0x55 → 0x01
  - p && *p (avoids null pointer access)避免访问空指针

注意&& vs. &(以及|| vs. |)…
超级常见的C编程错误!
Watch out for && vs. & (and || vs. |)…
Super common C programming pitfall!

移位运算 Shift Operations

- **左移 Left Shift: `x << y`**
  - 位向量x左移y位 Shift bit-vector **x** left **y** positions
    - 丢弃左边多余比特 Throw away extra bits on left
  - 右边填0 Fill with 0's on right
- **右移 Right Shift: `x >> y`**
  - 位向量x右移y位 Shift bit-vector **x** right **y** positions
    - 丢弃右边多余的比特 Throw away extra bits on right
  - 逻辑移位 Logical shift
    - 左边填0 Fill with 0's on left
  - 算术移位 Arithmetic shift
    - 左边复制最高有效位 Replicate most significant bit on left
- **未定义行为 Undefined Behavior**
  - 移位量小于零或大于等于字长 Shift amount < 0 or ≥ word size

Argument **x**	01100010
<< 3	00010000
Log. >> 2	00011000
Arith. >> 2	00011000

Argument **x**	10100010
<< 3	00010000
Log. >> 2	00101000
Arith. >> 2	11101000

编码整数 Encoding Integers

无符号 Unsigned

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

补码 Two's Complement

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

符号位
Sign
Bit

```
short int x =  15213;
short int y = -15213;
```

- **C语言并不强制使用二进制补码 C does not mandate using two's complement**
  - 然而大部分机器一般会用二进制补码进行运算, 我们也如此假设
    But, most machines do, and we will assume so
- **C语言中short为2字节长 C `short` 2 bytes long**

	十进制 Decimal	十六进制 Hex	二进制 Binary
x	15213	3B 6D	00111011 01101101
y	-15213	C4 93	11000100 10010011

- **符号位 Sign Bit**
  - 对于补码最高有效位是符号位 For 2's complement, most significant bit indicates sign
    - 0表示非负 0 for nonnegative
    - 1表示负 1 for negative

无符号数表示 Unsigned Representation

- **二进制（物理上） Binary (physical)**
  - 位向量 Bit vector [x_{w-1},x_{w-2},x_{w-3},…x_0]
- **二进制到无符号数（逻辑上） Binary to Unsigned (logical)**

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

补码 Two’s Complement

- **二进制（物理上） Binary (physical)**
  - 位向量 Bit vector [$x_{w-1}, x_{w-2}, x_{w-3}, \ldots x_0$]
- **二进制到有符号数（逻辑上） Binary to Signed (logical)**

$$B2T(X) \;=\; \underbrace{-x_{w-1}}_{\text{符号位 Sign Bit}} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

- **补码 2’s complement**

从补码到二进制(x是负数)
From Two’s Complement to Binary

$$\mathrm{x} = -2^{w-1} + \sum_{i=0}^{w-2} \mathrm{x}_i 2^i = -y = -\sum_{i=0}^{w-2} y_i 2^i$$

$$\sum_{i=0}^{w-2} \mathrm{x}_i 2^i = 2^{w-1} - \sum_{i=0}^{w-2} y_i 2^i = \sum_{i=0}^{w-2} (1 - y_i) 2^i + 1$$

$$2^{w-1} = \sum_{i=0}^{w-2} 2^i + 1 \qquad x_{w-1} = 1 \qquad y_{w-1} = 0$$

补码 Two's Complement

$$\sum_{i=0}^{w-1} x_i 2^i = \sum_{i=0}^{w-1} (1 - y_i) 2^i + 1$$

- **这个公式意味着什么呢?　What does it mean?**
  - 计算x的绝对值成w位的二进制数 Computing the negation of x into binary with w-bits
  - 对结果变反（求补） Complementing the result
  - 加一 Adding 1

数值范围 Numeric Ranges

- **无符号值 Unsigned Values**
  - *UMin* = 0
    000…0
  - *UMax* = $2^w - 1$
    111…1

- **补码值 Two's Complement Values**
  - *TMin* = -2^{w-1}
    100…0
  - *TMax* = $2^{w-1} - 1$
    011…1

- **其它值 Other Values**
  - 负一 （-1） Minus 1
    111…1

Values for *W* = 16

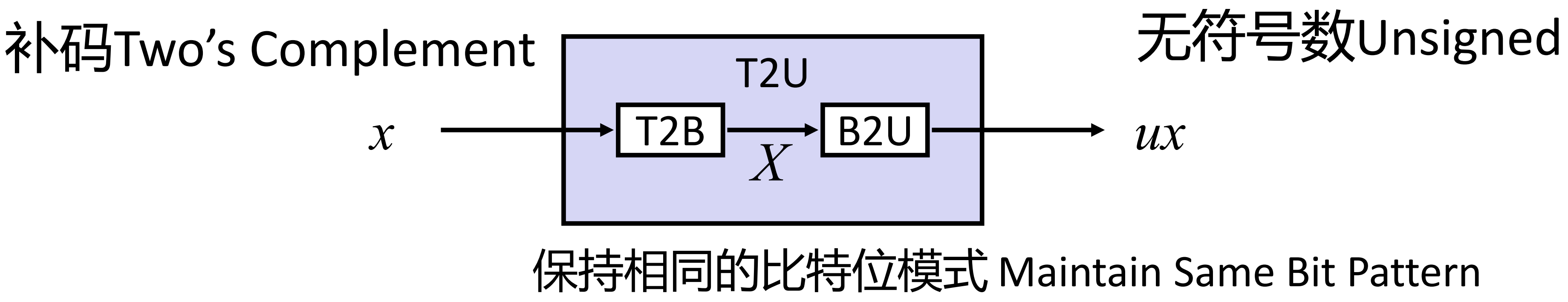
	Decimal	Hex	Binary
UMax	**65535**	**FF FF**	11111111 11111111
TMax	**32767**	**7F FF**	01111111 11111111
TMin	**-32768**	80 00	10000000 00000000
-1	**-1**	**FF FF**	11111111 11111111
0	**0**	00 00	00000000 00000000

无符号和有符号数的数值
Unsigned & Signed Numeric Values

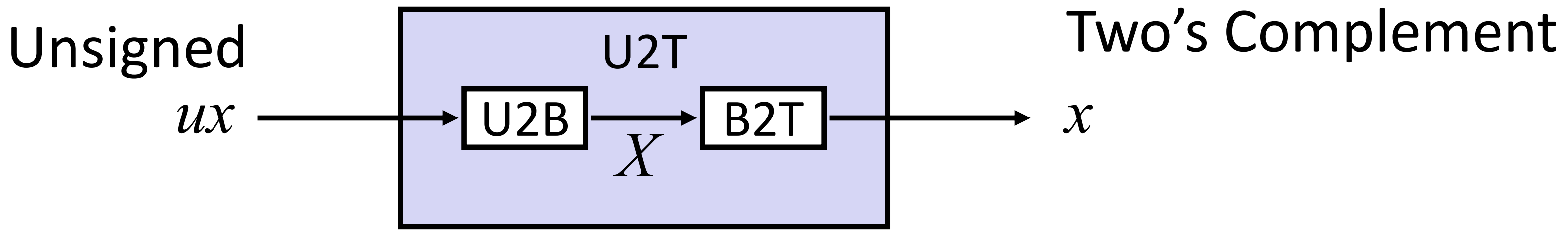
X	B2U(*X*)	B2T(*X*)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	–8
1001	9	–7
1010	10	–6
1011	11	–5
1100	12	–4
1101	13	–3
1110	14	–2
1111	15	–1

- **等同的 Equivalence**
  - 非负值同样的编码 Same encodings for nonnegative values
- **惟一性 Uniqueness**
  - 每个比特模式表示惟一的整数值 Every bit pattern represents unique integer value
  - 每个可表示的整数有惟一的比特位编码 Each representable integer has unique bit encoding
- **⇒ 能够逆映射Can Invert Mappings**
  - U2B(x) = B2U^{-1}(x)
    - 无符号整数比特模式Bit pattern for unsigned integer
  - T2B(x) = B2T^{-1}(x)
    - 补码整数比特模式 Bit pattern for

有符号数和无符号数之间进行映射 Mapping Between Signed & Unsigned



保持相同的比特位模式 Maintain Same Bit Pattern



保持相同的比特位模式 Maintain Same Bit Pattern

- **无符号数和补码之间进行映射Mappings between unsigned and two’s complement numbers:**
  保持位表示并重新解释 Keep bit representations and reinterpret

C语言整数数据类型典型取值范围-64位
Typical Ranges for C integral data types 64

C语言声明 C declaration	典型64位 Typical 64-bit	
	最小 minimum	最大 maximum
char unsigned char	-128 0	127 255
short [int] unsigned short	-32,768 0	32,767 65,535
int unsigned [int]	-2,147,483,648 0	2,147,483,647 4,294,967,295
long [int] unsigned long	-9,223,372,036,854,775,800 0	9,223,372,036,854,775,800 18,446,744,073,709,551,615
int32_t uint32_t	-2,147,483,648 0	2,147,483,647 4,294,967,295
int64_t uint64_t	-9,223,372,036,854,775,800 0	9,223,372,036,854,775,800 18,446,744,073,709,551,615

C语言中的有符号数和无符号数 Signed vs. Unsigned in C

- **常量 Constants**
  - 默认为有符号整数 By default are considered to be signed integers
  - 有U做后缀表示无符号数 Unsigned if have “U” as suffix

```
0U, 4294967259U
```

- **强制类型转换 Casting**
  - 显示强制类型转换有/无符号数等同于U2T/T2U Explicit casting between signed & unsigned same as U2T and T2U

```
int tx, ty;
unsigned ux, uy;
tx = (int) ux;
uy = (unsigned) ty;
```

  - 隐式强制类型转换通过赋值和过程调用也会发生 Implicit casting also occurs via assignments and procedure calls

```
tx = ux;
uy = ty;
```

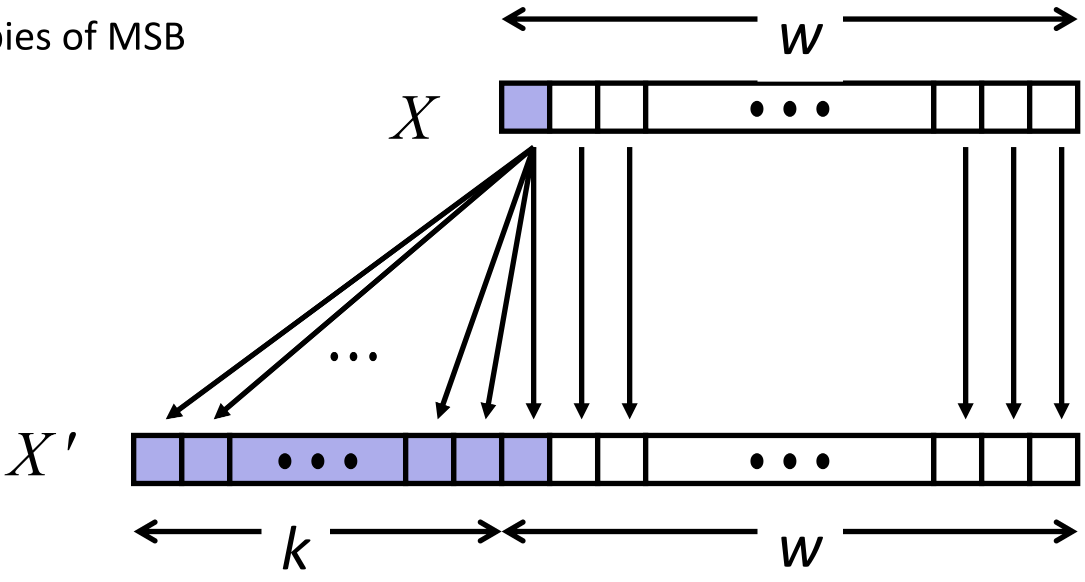
从短到长的转换 From short to long

```
short int x =  12345;
int ix = (int) x;
short int y = -12345;
int iy = (int) y;
```

- 我们需要扩展数据长度 We need to expand the data size
- 无符号类型之间强制类型转换正常 Casting among unsigned types is normal
- 有符号类型之间强制类型转换需要技巧 Casting among signed types is trick

符号扩展 Sign Extension

- **任务 Task:**
  - 给定w位的带符号整数x Given *w*-bit signed integer *x*
  - 转换成数值相同的w+k位整数 Convert it to *w*+*k*-bit integer with same value
- **规则 Rule: MSB-最高有效位**
  - 把符号位复制k位 Make *k* copies of sign bit:
  - $X' = \underbrace{x_{w-1},\ldots, x_{w-1}}_{k \text{ copies of MSB}}, x_{w-1}, x_{w-2},\ldots, x_0$



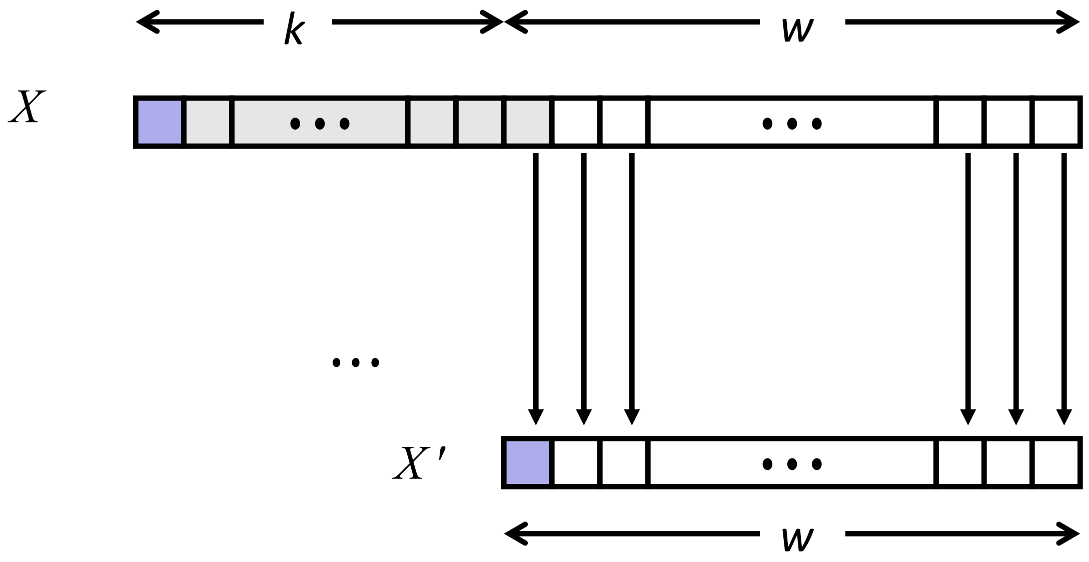
从长到短的转换 From long to short

```
int       x  = 53191;
short int sx = x;
int       y  = -12345;
Short int sy = y;
```

- 我们需要截断数据大小 We need to truncate the data size
- 强制类型转换从长到短需要技巧 Casting from long to short is trick

截断 Truncation

- **任务 Task:**
  - 给定k+w位有符号或无符号整数 X Given k+*w*-bit signed or unsigned integer *X*
  - 转换成w位整数X’，对于“足够小的”X具有同样的值 Convert it to *w*-bit integer X’ with same value for “small enough” X
- **规则 Rule:**
  - 丢弃头k位 Drop top *k* bits:
  - $X' = x_{w-1}, x_{w-2}, \ldots, x_0$



截断数值 Truncating Numbers

- Unsigned Truncating

$$B2U_w([x_w, x_{w\text{-}1}, \cdots, x_0]) mod 2^k$$
$$= B2U_k([x_k, x_{k-1}, \cdots, x_0])$$

- Signed Truncating

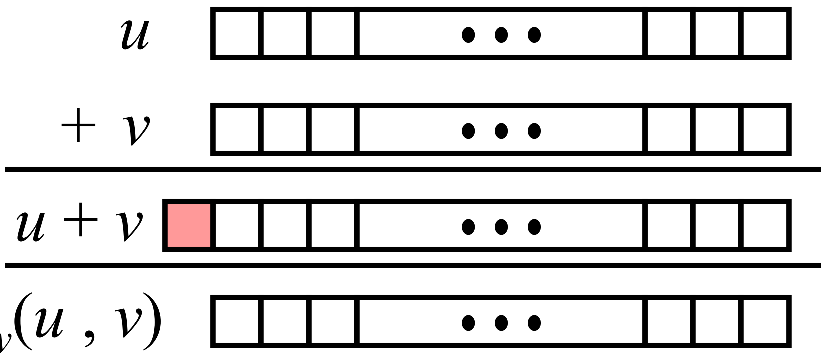
$$B2\mathrm{T}_k([x_k, x_{k-1}, \cdots, x_0])$$
$$= B2\mathrm{T}_k(B2U_w([x_w, x_{w-1}, \cdots, x_0]) mod 2^k)$$

小结 Summary:
扩展和截断：基本规则
Expanding, Truncating: Basic Rules

- **扩展（例如short扩展成int）Expanding (e.g., short int to int)**
  - 无符号数：添加零 Unsigned: zeros added
  - 有符号数：符号位扩展 Signed: sign extension
  - 都还会产生期望的结果 Both yield expected result

- **截断（例如无符号int截断成无符号short）Truncating (e.g., unsigned to unsigned short)**
  - 无/有符号数：比特位截断 Unsigned/signed: bits are truncated
  - 结果重新解释 Result reinterpreted
  - 无符号数：模取余运算 Unsigned: mod operation
  - 有符号数：类似模取余 Signed: similar to mod
  - 对于小的数值还会产生期望的行为 For small numbers yields expected behavior

无符号数加法
Unsigned Addition



操作数w位 Operands: *w* bits

真和w+1位 True Sum: *w*+1 bits

丢弃进位后和为w位
Discard Carry: *w* bits

- **标准加法功能 Standard Addition Function**
  - 忽略进位输出 Ignores carry output
- **实现取模运算 Implements Modular Arithmetic**

$s \quad = \quad \mathrm{UAdd}_w(u\ ,\ v) \quad = \quad u + v \bmod 2^w$

无符号字符
`unsigned char`

```
    1110 1001       E9       233
+   1101 0101     + D5     + 213
 ____________     ____     _____

 ____________     ____     _____
```

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

数学上性质 Mathematical Properties

- **模数加法形成阿贝尔群 Modular Addition Forms an *Abelian Group***
  - **封闭的加法 Closed** under addition

    $0 \leq \text{UAdd}_w(u, v) \leq 2^w - 1$
  - **交换性 Commutative**

    $\text{UAdd}_w(u, v) = \text{UAdd}_w(v, u)$
  - **结合性 Associative**

    $\text{UAdd}_w(t, \text{UAdd}_w(u, v)) = \text{UAdd}_w(\text{UAdd}_w(t, u), v)$
  - **0是加性恒等(单位元) 0** is additive identity

    $\text{UAdd}_w(u, 0) = u$
  - 每个元素都有加法逆元 Every element has additive **inverse**
    - Let $\text{UComp}_w(u) = 2^w - u$

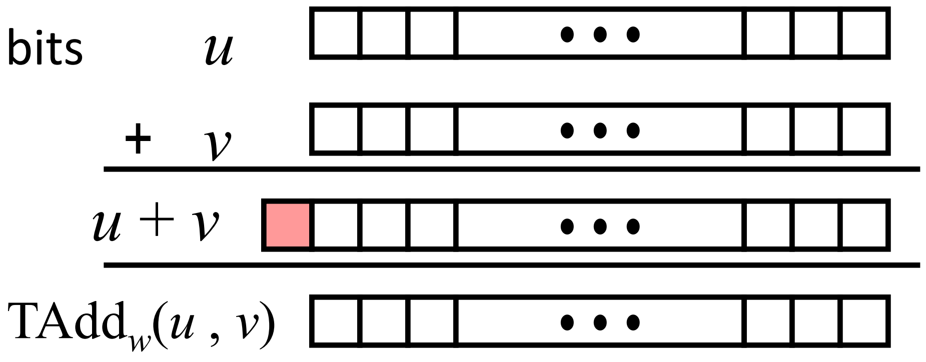
      $\text{UAdd}_w(u, \text{UComp}_w(u)) = 0$

补码加法 Two's Complement Addition

操作数w位 Operands: *w* bits

真正和w+1位
True Sum: *w*+1 bits

丢弃进位
Discard Carry: *w* bits



- **有符号和无符号数加法有同样的比特位级行为 TAdd and UAdd have Identical Bit-Level Behavior**
  - C语言中带符号和无符号数加法 Signed vs. unsigned addition in C:

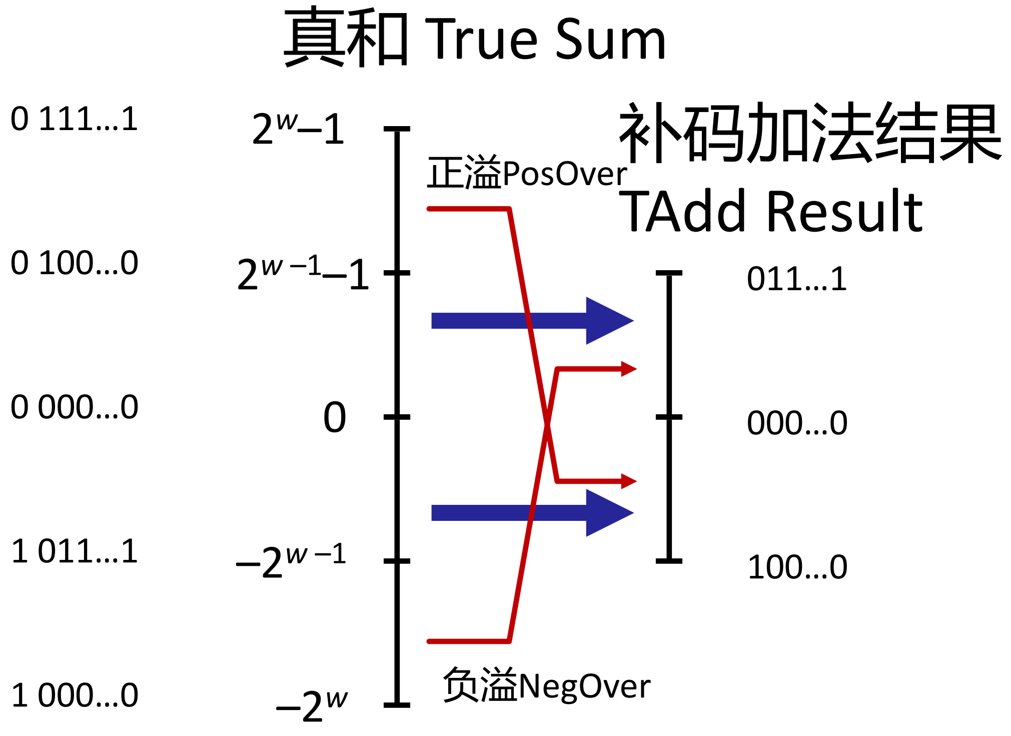
```
int s, t, u, v;
s = (int) ((unsigned) u + (unsigned) v);
t = u + v
```

  - 结果s和t相同 Will give `s == t`

```
     1110 1001       E9      -23
  +  1101 0101     + D5    + -43
   1 1011 1110      1BE      -66
     1011 1110       BE      -66
```

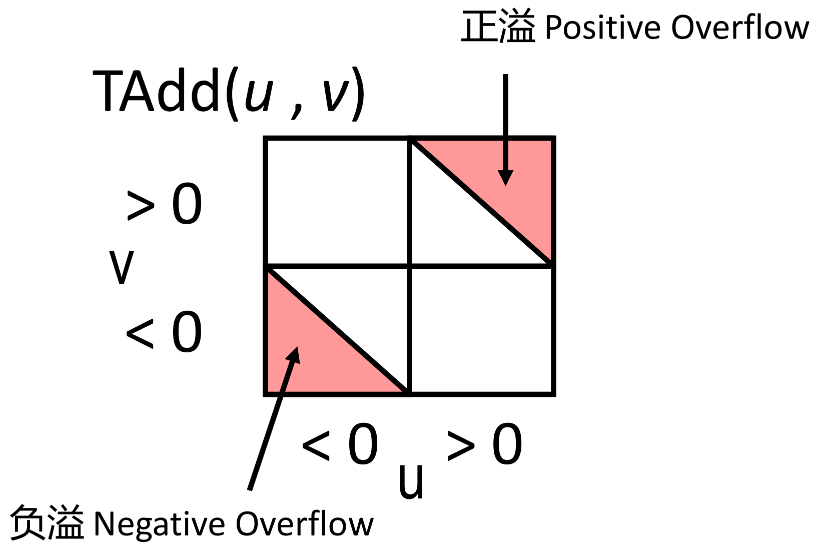
有符号数加法溢出 TAdd Overflow

- **功能 Functionality**
  - 真和需要w+1位 True sum requires *w*+1 bits
  - 丢弃最高有效位 Drop off MSB
  - 剩余位作为补码整数对待 Treat remaining bits as 2’s comp. integer



有符号数加法特征 Characterizing TAdd

- **功能 Functionality**
  - 真正的和需要w+1位 True sum requires *w*+1 bits
  - 丢弃最高有效位 Drop off MSB
  - 剩余位看成补码整数 Treat remaining bits as 2's comp. integer



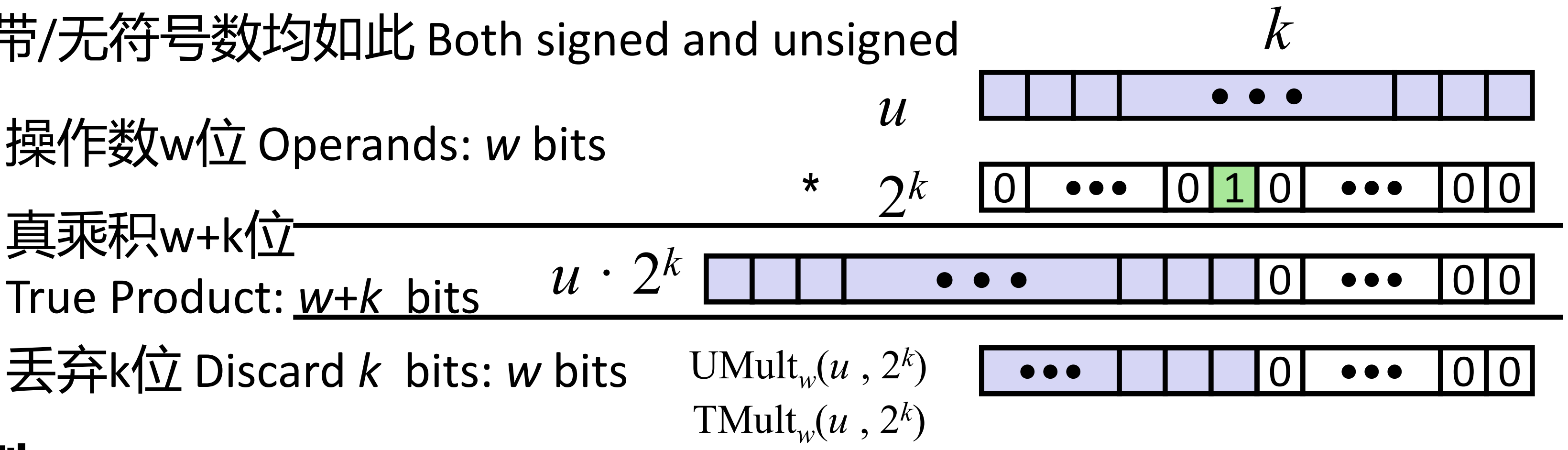
$$TAdd_w(u,v) = \begin{cases} u+v+2^w & u+v < TMin_w \quad \text{(负溢 NegOver)} \\ u+v & TMin_w \le u+v \le TMax_w \\ u+v-2^w & TMax_w < u+v \quad \text{(正溢 PosOver)} \end{cases}$$

乘法 Multiplication

- **目标：计算w位的数x和y的乘积 Goal: Computing Product of *w*-bit numbers *x*, *y***
  - 要么是有符号的，要么是无符号的 Either signed or unsigned
- **精确的结果比w位大得多 exact results can be bigger than *w* bits**
  - 无符号数：到2w位 Unsigned: up to 2*w* bits
    - 结果范围： Result range: $0 \le x * y \le (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
  - 补码最小（负数）：到2w-1位 Two's complement min (negative): Up to 2*w*-1 bits
    - 结果范围： Result range: $x * y \ge (-2^{w-1})*(2^{w-1}-1) = -2^{2w-2} + 2^{w-1}$
  - 补码最大（正数）：到2w位，但仅限于$(TMin_w)^2$ Two's complement max (positive): Up to 2*w* bits, but only for $(TMin_w)^2$
    - 结果范围： Result range: $x * y \le (-2^{w-1})^2 = 2^{2w-2}$
- **所以，保持精确的结果。。。So, maintaining exact results…**
  - 需要在计算每个乘积时不断扩大乘积结果表示的字节数 would need to keep expanding word size with each product computed
  - 如果需要由软件完成 is done in software, if needed
    - 例如，任意精度算术软件包 e.g., by "arbitrary precision" arithmetic

用移位实现2的整数次幂乘法
Power-of-2 Multiply with Shift

- **运算 Operation**
  - **左移k位等于乘以2ᵏ** `u << k` gives `u *` 2^k
  - 带/无符号数均如此 Both signed and unsigned



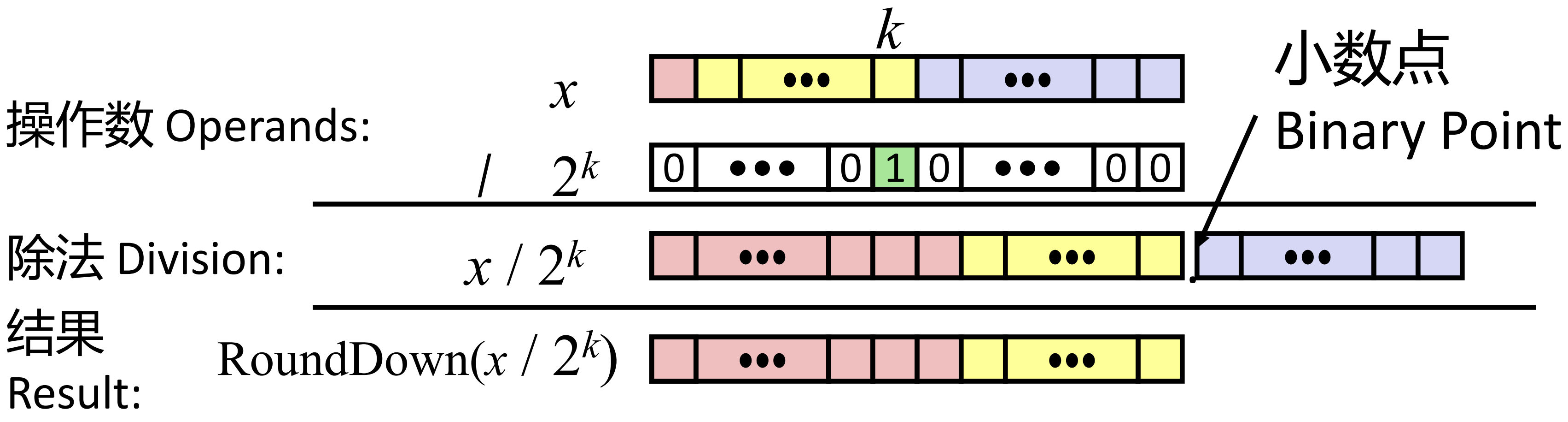
操作数w位 Operands: *w* bits
真乘积w+k位
True Product: *w+k* bits
丢弃k位 Discard *k* bits: *w* bits
k
u
$* \quad 2^k$
0 ••• 0 1 0 ••• 0 0
$u \cdot 2^k$
0 ••• 0 0
$\mathrm{UMult}_w(u\ ,2^k)$
$\mathrm{TMult}_w(u\ ,2^k)$
0 ••• 0 0

- **举例 Examples**
  - `u << 3          ==    u * 8`
  - `(u << 5) - (u << 3)==      u * 24`
  - 大多数机器移位和加法比乘法更快 M
    faster than multiply
    - 编译器自动生成这种代码 Compi
      automatically

重要教训： Important Lesson:
信任编译器 Trust Your Compiler!

用移位实现有符号数2的整数次幂除法 Signed Power-of-2 Divide with Shift

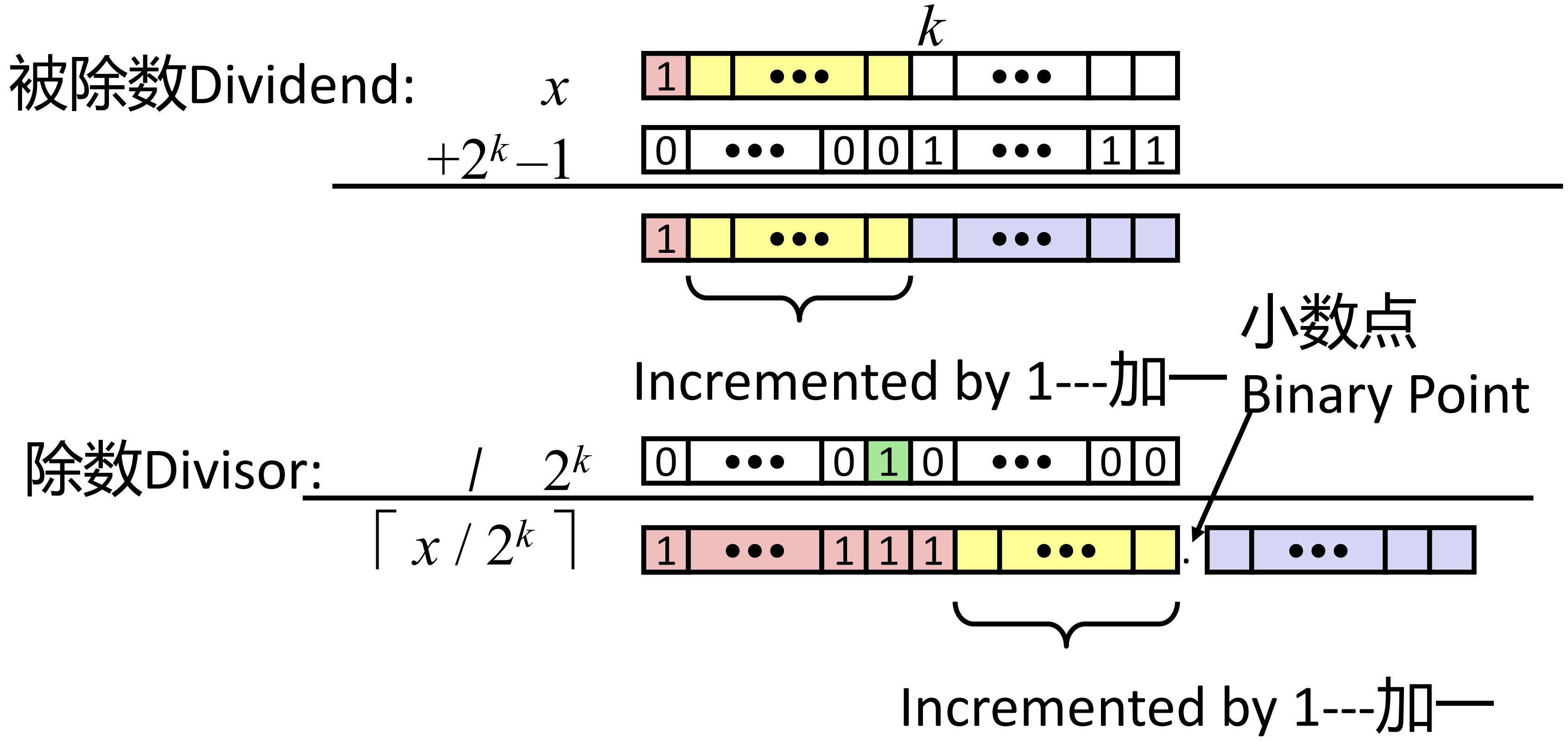
- 有符号数除以2的整数次幂的商 Quotient of Signed by Power of 2
  - x右移k位等于整除2^k 向下舍入 `x >> k` gives $\lfloor x / 2^k \rfloor$
  - 使用算术移位 Uses arithmetic shift
  - 当x<0时向错误的方向舍入 Rounds wrong direction when `x < 0`



	Division	Computed	Hex	Binary
`y`	-15213	-15213	`C4 93`	`11000100 10010011`
`y >> 1`	-7606.5	-7607	`E2 49`	`11100010 01001001`
`y >> 4`	-950.8125	-951	`FC 49`	`11111100 01001001`
`y >> 8`	-59.4257813	-60	`FF C4`	`11111111 11000100`

修正2的整数次幂除法（续）
Correct Power-of-2 Divide (Cont.)

Case 2: Rounding 向上取整



偏置给最终结果加一 Biasing adds 1 to final result

补码非：求补和递增
Negation: Complement & Increment

- **通过求补和加一得到补码非 Negate through complement and increase**

  `~x + 1 == -x`

- **示例 Example**
  - Observation: `~x + x == 1111…111 == -1`

$$\begin{array}{rr} & x & \boxed{1}\boxed{0}\boxed{0}\boxed{1}\boxed{1}\boxed{1}\boxed{0}\boxed{1} \\ + & \sim x & \boxed{0}\boxed{1}\boxed{1}\boxed{0}\boxed{0}\boxed{0}\boxed{1}\boxed{0} \\ \hline & -1 & \boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1} \end{array}$$

x = 15213

	Decimal	Hex	Binary
`x`	15213	3B 6D	00111011 01101101
`~x`	-15214	C4 92	11000100 10010010
`~x+1`	-15213	C4 93	11000100 10010011
`y`	-15213	C4 93	11000100 10010011

字节顺序 Byte Ordering

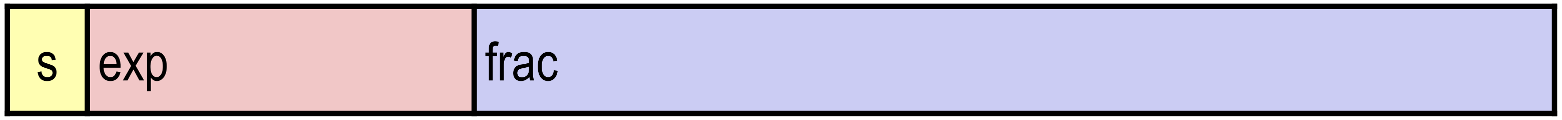
- **因此，字中的多个字节在内存里如何排序？So, how are the bytes within a multi-byte word ordered in memory?**
- **约定 Conventions**
  - 大端法：Big Endian: Sun, PPC Mac, Internet
    - 最低有效字节有最高的地址 Least significant byte has highest address
  - 小端法：Little Endian: x86, ARM processors running Android, iOS, and Windows
    - 最低有效字节有最低地址 Least significant byte has lowest address

浮点表示 Floating Point Representation

Example:
$15213_{10} = (-1)^0 \times 1.1101101101101_2 \times 2^{13}$

- **浮点数形式 Numerical Form:**
  $(-1)^s\ M\ 2^E$
  - 符号位s确定数值是负还是正 **Sign bit *s*** determines whether number is negative or positive
  - 尾数M是范围在[1.0,2.0)之间的普通小数 **Significand *M*** normally a fractional value in range [1.0,2.0).
  - 阶码E是给浮点数指定2的E次幂权重 **Exponent *E*** weights value by power of two

- **编码 Encoding**
  - 最高位是符号位s MSB s is sign bit ***s***
  - exp字段编码E（但不等于E）exp field encodes ***E*** (but is not equal to E)
  - frac字段编码M（但不等于M）frac field encodes ***M*** (but is not equal to M)



“规格化”值
“Normalized” Values

$v = (-1)^s M\, 2^E$

- 当阶码非全零和全一时 **When: exp ≠ 000…0 and exp ≠ 111…1**

- 阶码编码为一个有偏置值的有符号数：***E* = *Exp* – *Bias***
  Exponent coded as a *biased* value: *E* = *Exp* – *Bias*
  - *Exp*：*exp*字段的无符号值 *Exp*: unsigned value of exp field
  - 偏置*Bias*= $2^{k-1} - 1$其中k是阶码位的位数 *Bias* = 2^{k-1} - 1, where *k* is number of exponent bits
    - 单精度：127 Single precision: 127 (Exp: 1…254, E: -126…127)
    - 双精度：1023 Double precision: 1023 (Exp: 1…2046, E: -1022…1023)

- **尾数编码为带隐含的一个前导1 Significand coded with implied leading 1: *M* =** $1.\text{xxx…x}_2$
  - frac字段的比特位 xxx…x: bits of frac field
  - 当frac全零时值最小 Minimum when frac=000…0 (M = 1.0)
  - 当frac全一时值最大 Maximum when frac=111…1 (M = 2.0 – ε)

非规格化值 Denormalized Values

$v = (-1)^s \, M \, 2^E$

$E = 1 - Bias$

- 条件：exp为全零 **Condition:** exp = 000…0

- 阶码值：**Exponent value: *E* = 1 – Bias (instead of *E* = 0 – *Bias*)**
- 尾数编码为隐含的一个前导零 **Significand coded with implied leading 0: *M* = 0.xxx…x$_2$**
  - **`frac`**字段比特位 **`xxx…x`**: bits of **`frac`**
- 情况 **Cases**
  - **`exp`** = 000…0, **`frac`** = 000…0
    - 代表0值 Represents zero value
    - 注意区别值：+0和-0（为何？）Note distinct values: +0 and –0 (why?)
  - **`exp`** = 000…0, **`frac`** ≠ 000…0
    - 最接近0.0的数值 Numbers closest to 0.0
    - 平均分布的 Equispaced

特殊值 Special Values

- 条件：exp为全一 **Condition: `exp` = `111…1`**

- 情况 **Case: `exp` = `111…1`, `frac` = `000…0`**
  - 代表值无穷大 Represents value ∞ (infinity)
  - 溢出运算 Operation that overflows
  - 正负均如此 Both positive and negative
  - 例如 E.g., 1.0/0.0 = −1.0/−0.0 = +∞, 1.0/−0.0 = −∞

- 情况 **Case: `exp` = `111…1`, `frac` ≠ `000…0`**
  - 不是一个数 Not-a-Number (NaN)
  - 代表无法确定数值的情况 Represents case when no numeric value can be determined
  - 例如 E.g., sqrt(–1), ∞ − ∞, ∞ × 0

动态范围（仅正数）
Dynamic Range (Positive Only)

$v = (-1)^s\, M\, 2^E$

n: E = Exp – Bias

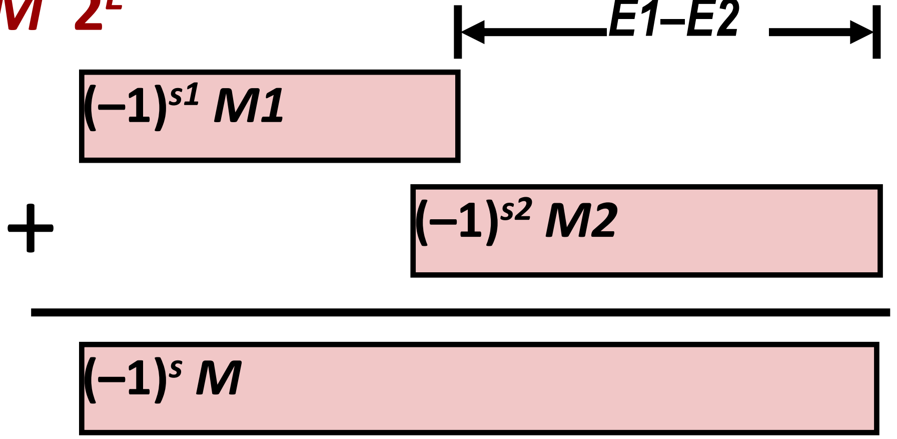
d: E = 1 – Bias

	s	exp	frac	E	Value	
	0	0000	000	-6	0	
	0	0000	001	-6	1/8*1/64 = 1/512	最接近0 closest to zero
非规格化数 Denormalized numbers	0	0000	010	-6	2/8*1/64 = 2/512	
	…					
	0	0000	110	-6	6/8*1/64 = 6/512	
	0	0000	111	-6	7/8*1/64 = 7/512	最大非规格化数 largest denorm
	0	0001	000	-6	8/8*1/64 = 8/512	最小规格化数 smallest norm
	0	0001	001	-6	9/8*1/64 = 9/512	
	…					
	0	0110	110	-1	14/8*1/2 = 14/16	
	0	0110	111	-1	15/8*1/2 = 15/16	最接近1以下 closest to 1 below
规格化数 Normalized numbers	0	0111	000	0	8/8*1 = 1	
	0	0111	001	0	9/8*1 = 9/8	最接近1以上 closest to 1 above
	0	0111	010	0	10/8*1 = 10/8	
	…					
	0	1110	110	7	14/8*128 = 224	
	0	1110	111	7	15/8*128 = 240	最大规格化数 largest norm
	0	1111	000	n/a	inf	无穷大 inf

浮点加法 Floating Point Addition

- $(-1)^{s1}\, M1\ \ 2^{E1}\ \ +\ \ (-1)^{s2}\, M2\ \ 2^{E2}$
  - 假设 Assume *E1* > *E2*

小数点对齐 **Get binary points lined up**



- 精确的结果 **Exact Result: $(-1)^s\ M\ \ 2^E$**
  - 符号位s，尾数M：
  - Sign *s*, significand *M*:
    - 有符号数对齐并相加的结果
    - Result of signed align & add
  - 阶码E是E1 Exponent *E*: *E1*

- 修正 **Fixing**
  - 如果M大于等于2，M右移，E加一 If *M* ≥ 2, shift *M* right, increment *E*
  - 如果M小于1，M左移k位，E减k if *M* < 1, shift *M* left *k* positions, decrement *E* by *k*
  - 如果E超过范围则溢出 Overflow if *E* out of range
  - 舍入M到适合frac的精度 Round *M* to fit `frac` precision

浮点乘法 FP Multiplication

- $(-1)^{s1}\ M1\ \ 2^{E1}\ \ \times\ \ (-1)^{s2}\ M2\ \ 2^{E2}$
- 精确的结果 **Exact Result:** $(-1)^{s}\ M\ \ 2^{E}$
  - 符号位s：异或 Sign *s*: *s1 ^ s2*
  - 尾数M：相乘 Significand *M*: *M1* x *M2*
  - 阶码E：相加 Exponent *E*: *E1* + *E2*
- 修正 **Fixing**
  - 如果M大于等于2，M右移，阶码E加一 If *M* ≥ 2, shift *M* right, increment *E*
  - 如果E超过范围，溢出 If *E* out of range, overflow
  - 舍入M到适合frac的精度 Round *M* to fit `frac` precision
- 实现 **Implementation**
  - 最繁琐的工作是尾数相乘 Biggest chore is multiplying significands

4 位尾数: $1.010*2^2 \times 1.110*2^3 = 10.0011*2^5$
$= 1.00011*2^6 = 1.001*2^6$

C语言中的浮点数 Floating Point in C

- C语言确保两个级别的浮点数 **C Guarantees Two Levels**

`float`	single precision	单精度
`double`	double precision	双精度

- 转换/强制转换 **Conversions/Casting**
  - 在int，float和double之间强制转换改变比特位表示 Casting between **`int`**, **`float`**, and **`double`** changes bit representation
  - **`double`/`float` → `int`**
    - 截断尾数部分 Truncates fractional part
    - 就像向零舍入 Like rounding toward zero
    - 当超过范围或NaN时没有定义：一般设置为TMin Not defined when out of range or NaN: Generally sets to TMin
  - **`int` → `double`**
    - 精确转换，只要int字长小于等于53位 Exact conversion, as long as **`int`** has ≤ 53 bit word size
  - **`int` → `float`**
    - 将按照舍入模式进行舍入 Will round according to rounding mode

浮点数难题 Floating Point Puzzles

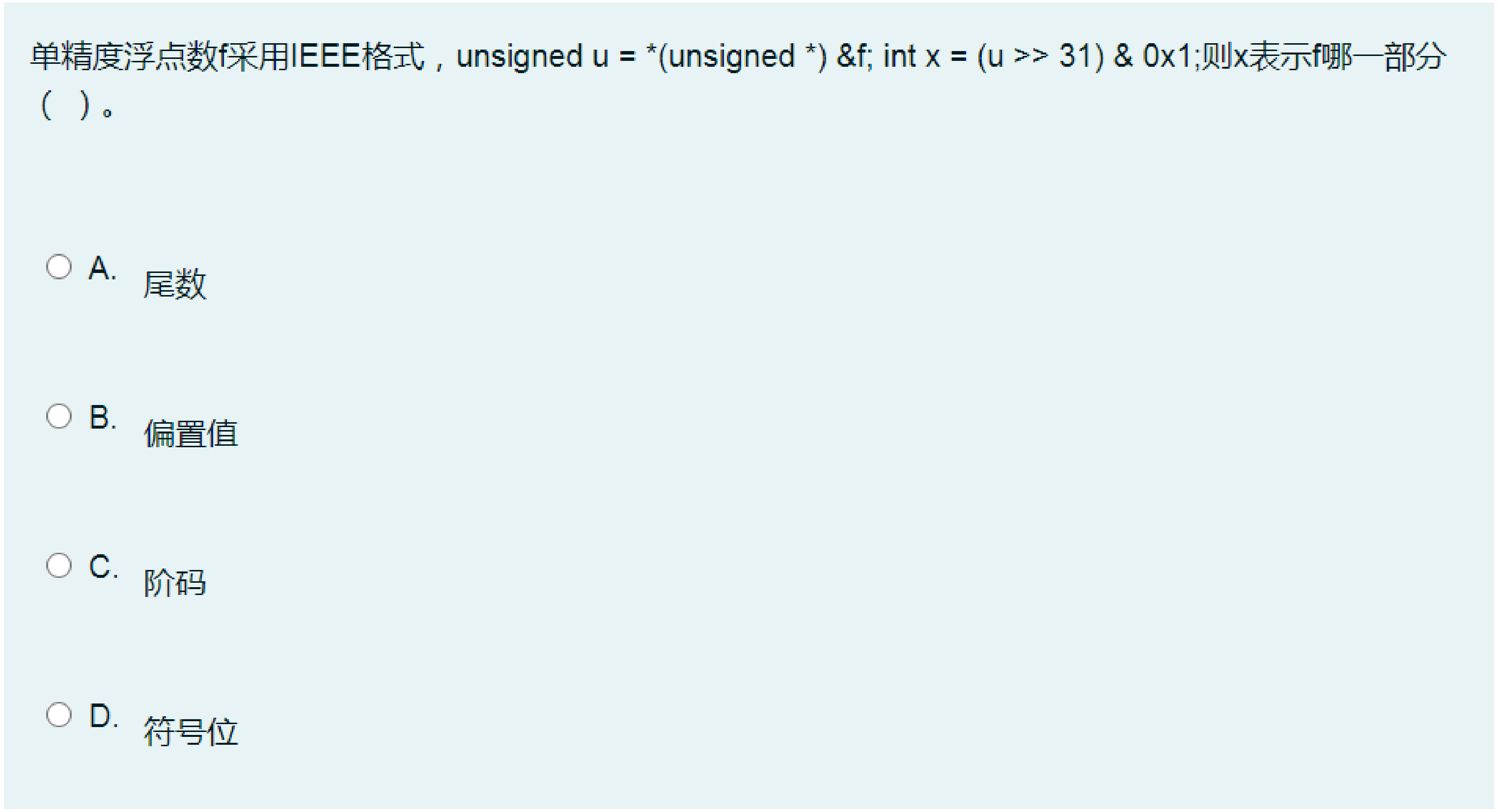
- 对于下面的每个C表达式 **For each of the following C expressions, either:** 完成其中一个工作
  - 对于所有的参数值解释其值为真 Argue that it is true for all argument values
  - 解释为何不为真 Explain why not true

```
int x = …;
float f = …;
double d = …;
```

假设d和f都不是NaN
Assume neither
`d` nor `f` is NaN

• `x == (int)(float) x`	✗
• `x == (int)(double) x`	✓
• `f == (float)(double) f`	✓
• `d == (double)(float) d`	✗
• `f == -(-f);`	✓
• `2/3 == 2/3.0`	✗
• `d < 0.0     ⇒  ((d*2) < 0.0)`	✓
• `d > f       ⇒  -f > -d`	✓
• `d * d >= 0.0`	✓
• `(d+f)-d == f`	✗

例题



单精度浮点数f采用IEEE格式，unsigned u = *(unsigned *) &f; int x = (u >> 31) & 0x1;则x表示f哪一部分（ ）。

○ A. 尾数

○ B. 偏置值

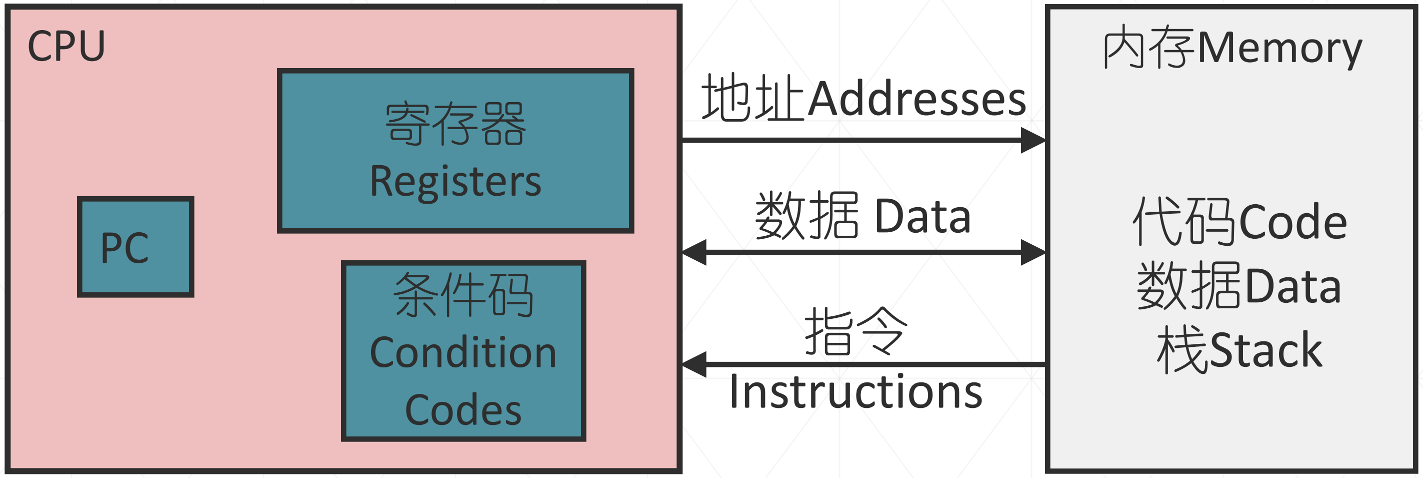
○ C. 阶码

○ D. 符号位

程序的机器表示

- 机器级代码
  - 机器指令、寄存器、操作数与寻址
- 一般指令操作
  - 传送、控制、算术、逻辑、控制流
- 过程调用
  - 递归过程、参数传送、栈帧设计、寄存器
- 构造数据类型
  - 数组、结构、联合
- 问题：内存越界、溢出及解决
- 浮点操作指令

汇编/机器代码视图Assembly/Machine Code View

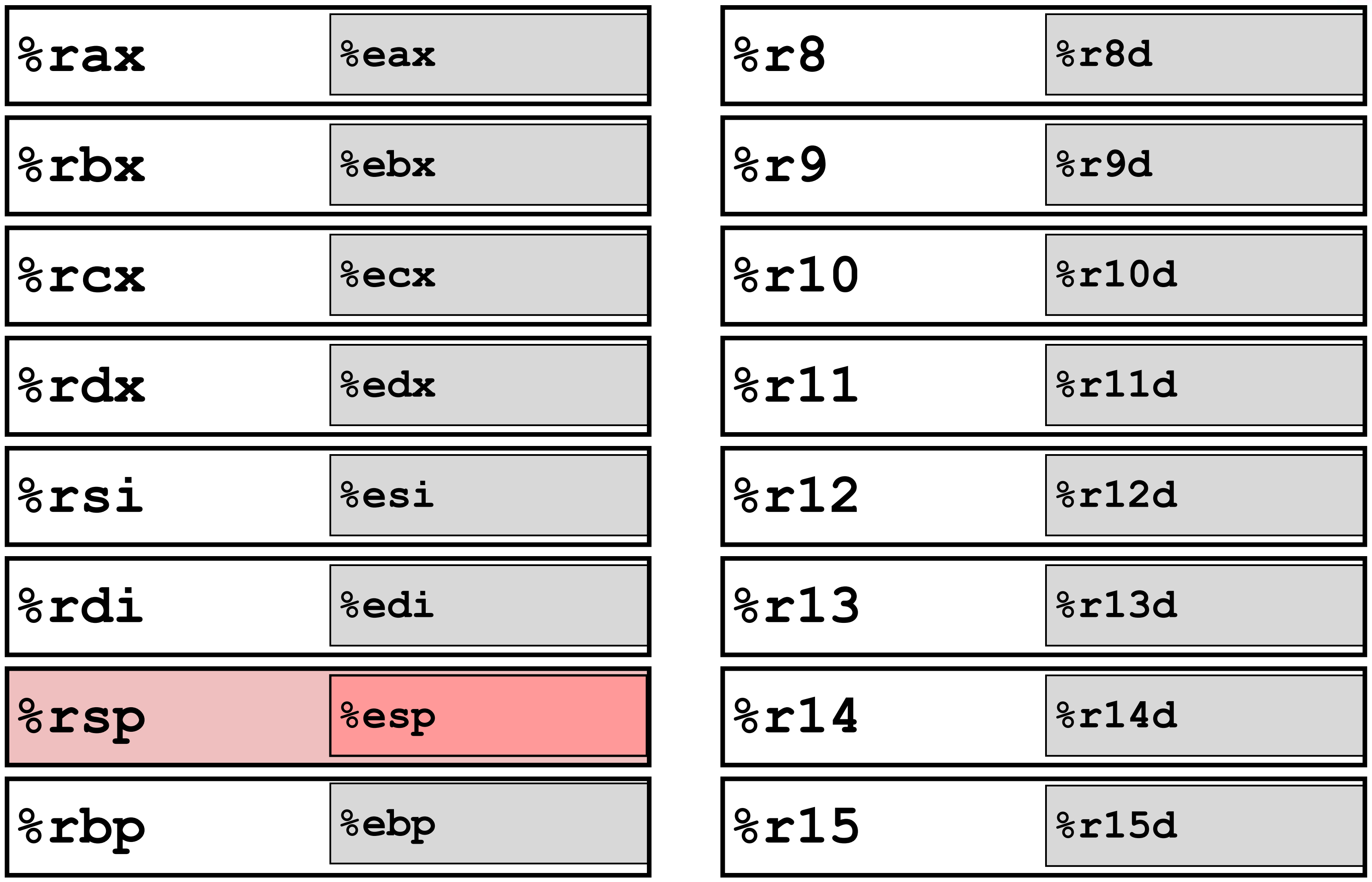


程序员可见的状态Programmer-Visible State

- **PC程序计数器 PC: Program counter**
  - 下条指令地址 Address of next instruction
  - 称为RIP Called “RIP” (x86-64)
- **寄存器堆 Register file**
  - 程序频繁使用的数据 Heavily used program data
- **条件码 Condition codes**
  - 存储有关最近的算术和逻辑运算的状态信息 Store status information about most recent arithmetic or logical operation

- **内存 Memory**
  - 字节可寻址的数组 Byte addressable array
  - 代码和用户数据Code and user data
  - 支持过程的栈 Stack to support procedures

x86-64 Integer Registers 整数寄存器



- 可以引用低4字节（也可以引用低1或2字节） Can reference low-order 4 bytes (also low-order 1 & 2 bytes)
- 不是内存（或cache）的一部分 Not part of memory (or cache)

汇编语言：操作 Assembly: Operations

- **在内存和寄存器之间传送数据 Transfer data between memory and register**
  - 从内存装载数据到寄存器 Load data from memory into register
  - 存储寄存器数据到内存 Store register data into memory

- **对寄存器或内存数据执行算术运算 Perform arithmetic function on register or memory data**

- **传递控制 Transfer control**
  - 无条件跳转到/从过程 Unconditional jumps to/from procedures
  - 条件分支 Conditional branches
  - 间接分支 Indirect branches

简单内存寻址方式 Simple Memory Addressing Modes

- **正常 Normal (R) Mem[Reg[R]]**
  - 寄存器R指定内存地址 Register R specifies memory address
  - C语言中的指针间接引用 Aha! Pointer dereferencing in C

```
movq (%rcx),%rax
```

- **变址 Displacement D(R) Mem[Reg[R]+D]**
  - 寄存器R指定内存区域的起始地址 Register R specifies start of memory region
  - 常量D指定偏移 Constant displacement D specifies offset

```
movq 8(%rbp),%rdx
```

复杂内存寻址方式 Complete Memory Addressing Modes

- **最通用形式 Most General Form**

  D(Rb,Ri,S)　　Mem[Reg[Rb]+S*Reg[Ri]+ D]

  - D:　常量“变址”1,2或4字节 Constant “displacement” 1, 2, or 4 bytes
  - Rb:　基指针：16个整数寄存器中任意一个 Base register: Any of 16 integer registers
  - Ri:　索引寄存器：任意，除了%rbp Index register: Any, except for `%rsp`
  - S:　比例因子：1,2,4或8(为何这几个数) Scale: 1, 2, 4, or 8 (*why these numbers?*)

- **特殊情况 Special Cases**

  (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]]

地址计算指令 Address Computation Instruction

- `leaq` ***Src*, *Dst***
  - *Src*是寻址方式表达式 *Src* is address mode expression
  - 设置Dst成为表达式指示的地址 Set *Dst* to address denoted by expression

- **用法 Uses**
  - 计算地址不必引用内存 Computing addresses without a memory reference
    - 例如 E.g., translation of **`p = &x[i];`**
  - 计算x + k*y形式的算术表达式 Computing arithmetic expressions of the form x + k*y
    - k = 1, 2, 4, or 8

- **Example**

```
long m12(long x)
{
  return x*12;
}
```

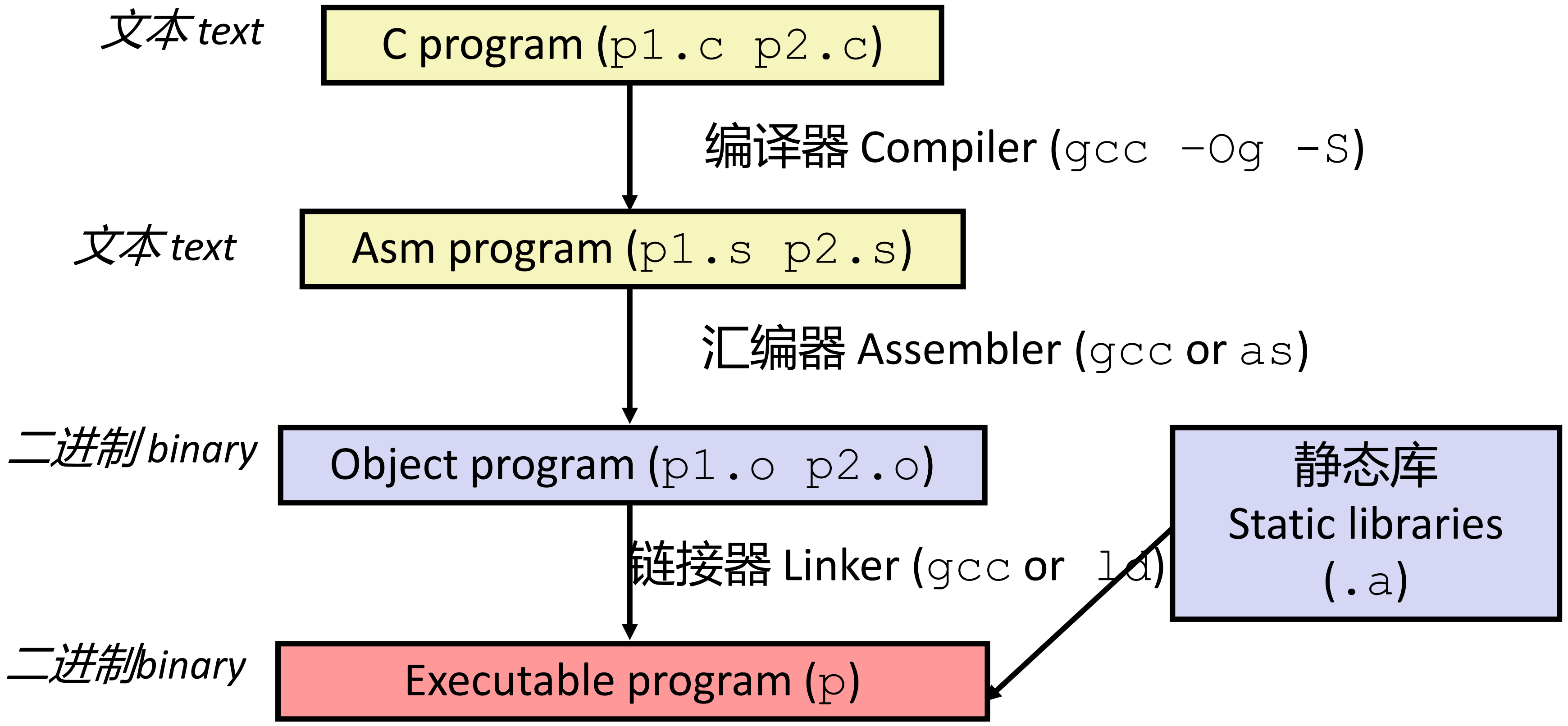
编译器转换成汇编程序

Converted to ASM by compiler:

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

转换C源程序为目标代码
Turning C into Object Code

- 代码在文件p1和p2中 Code in files `p1.c p2.c`
- 编译命令 Compile with command: `gcc -Og p1.c p2.c -o p`
  - 基本优化(-Og)Use basic optimizations (**-Og**) [New to recent versions of GCC]
  - 将二进制结果放在文件p中 Put resulting binary in file **p**



文本 text
C program (p1.c p2.c)
编译器 Compiler (gcc -Og -S)
文本 text
Asm program (p1.s p2.s)
汇编器 Assembler (gcc or as)
二进制 binary
Object program (p1.o p2.o)
静态库 Static libraries (.a)
链接器 Linker (gcc or ld)
二进制binary
Executable program (p)

汇编语言的控制流
Control flow in assembly language

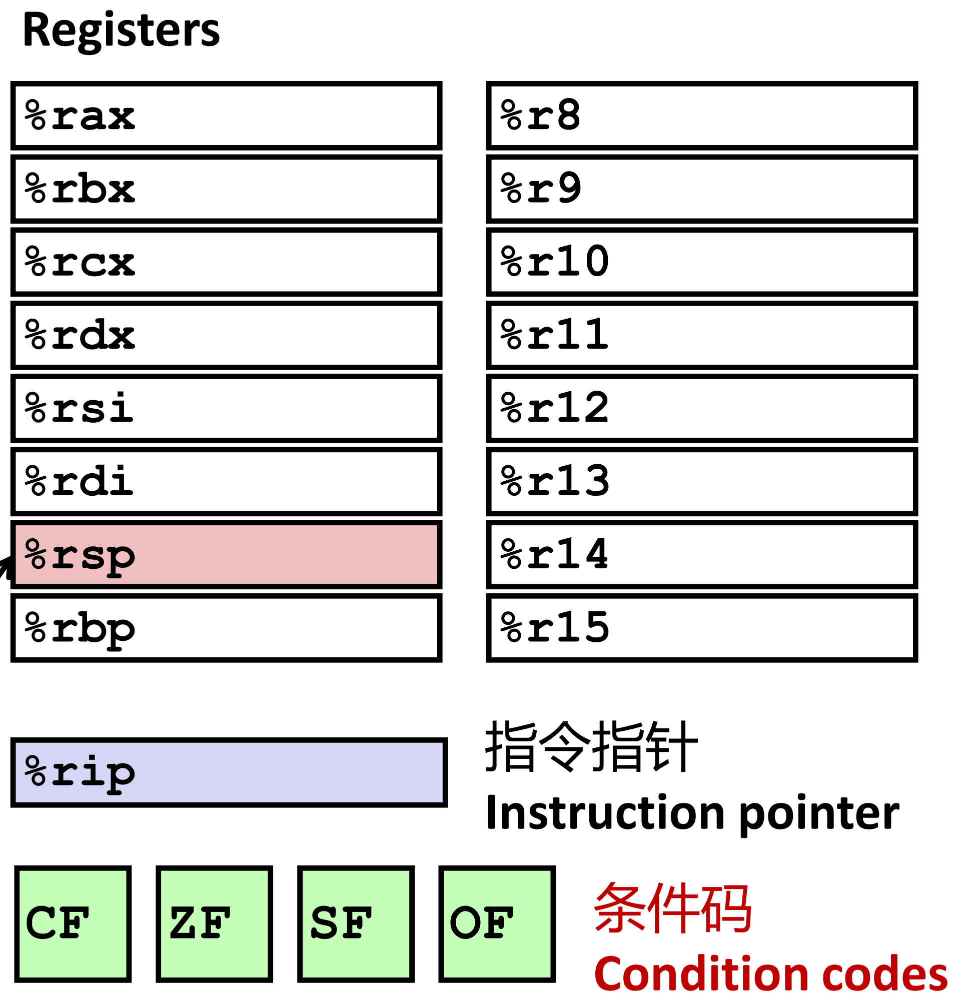
```
extern void op1(void);
extern void op2(void);

void decision(int x) {
    if (x) {
        op1();
    } else {
        op2();
    }
}
```

```
decision:
        subq    $8, %rsp
        testl   %edi, %edi
        je      .L2
        call    op1
        jmp     .L1
.L2:
        call    op2
.L1:
        addq    $8, %rsp
        ret
```

用GOTO语句来实现 It's all done with GOTO!

处理器状态（x86-64，部分）Processor State (x86-64, Partial)

- **关于当前执行程序的信息**
  Information about currently executing program
  - 临时数据 Temporary data
    (`%rax`, …)
  - 运行时栈位置 Location of runtime stack
    (`%rsp`)
  - 当前代码控制点位置Location of current code control point
    (`%rip`, …)
  - 最近测试的状态 Status of recent tests
    (**CF, ZF, SF, OF**)



在授课过程中需记住什么
What to remember during lecture

设置条件码
Set Condition Codes

- 操作：例如**addq**
  Operations: e.g. `addq`
- 比较：**Compare:** `cmp a, b`
  类似做**b-a like doing** `b-a`
- 测试：**Test:** `test a,b`
  类似做**a&b like doing** `a&b`

根据条件码跳转：je（相等跳转）jg（大于跳转）等
Jump based on condition codes: `je` **(jump if equal),** `jg` **(greater), etc.**

根据条件码设置寄存器的低字节为0/1 **Set low order byte of a register to 0/1 based on condition codes**

如果条件码置位则传送一个值 `mov` **a value if a condition code is set**

我们将深入研究，但是请像做炸弹实验一样阅读
We'll dive in, but read as you do bomb lab!

条件码（隐式设置）Condition Codes (Implicit Setting)

- **单个比特位寄存器 Single bit registers**
  - **CF** 进位标志Carry Flag (对无符号数 for unsigned) **SF** 符号标志 Sign Flag (对有符号数 for signed)
  - **ZF** 零标志 Zero Flag **OF** 溢出标志 Overflow Flag (对有符号数for signed)
- **由算术运算隐式设置（看成副作用） Implicitly set (think of it as side effect) by arithmetic operations**

  举例：Example: `addq` *Src,Dest* ↔ `t = a+b`

  CF set 如果从最高有效位进位（无符号溢出） if carry out from most significant bit (unsigned overflow)

  ZF set 如果结果为零 if `t == 0`

  SF set 如果结果小于零（有符号数） if `t < 0` (as signed)

  OF set 如果补码（有符号数）溢出 if two’s-complement (signed) overflow
  `(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)`
- **leaq指令不设置条件码 Not set by `leaq` instruction**

条件码（显式设置：比较指令）
Condition Codes (Explicit Setting: Compare)

- **由比较指令显式设置 Explicit Setting by Compare Instruction**
  - **`cmpq`** *Src2*, *Src1*
  - **`cmpq b,a`** 类似计算a-b，只是不设置目的操作数 like computing **`a-b`** without setting destination

  - **CF set** 如果从最高有效位进位（用于无符号数比较） if carry out from most significant bit (used for unsigned comparisons)
  - **ZF set** 如果相等 if **`a == b`**
  - **SF set** 如果小于（有符号数） if **`(a-b) < 0`** (as signed)
  - **OF set** 如果补码（有符号数）溢出 if two’s-complement (signed) overflow **`(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)`**

条件码（显式设置：测试指令）Condition Codes (Explicit Setting: Test)

- **由测试指令显式设置 Explicit Setting by Test instruction**
  - **`testq`** *Src2*, *Src1*
    - **`testq b,a`** 类似计算与操作，但是不设置目的操作数 like computing **`a&b`** without setting destination
  - 根据与运算的值设置条件码 Sets condition codes based on value of *Src1* & *Src2*
  - 对于用一个操作数作为掩码很有用 Useful to have one of the operands be a mask
  - **ZF set** 当与结果为0时 when **`a&b == 0`**
  - **SF set** 当与结果小于0时 when **`a&b < 0`**

非常常用 Very often:

```
testq  %rax,%rax
```

读取条件码 Reading Condition Codes

- **SetX指令 SetX Instructions**
  - 根据条件码组合设置目的操作数低字节成0或1 Set low-order byte of destination to 0 or 1 based on combinations of condition codes
  - 不要改变剩余的7个字节 Does not alter remaining 7 bytes

SetX	条件 Condition	描述 Description
`sete`	`ZF`	等于/零 Equal / Zero
`setne`	`~ZF`	不等/不为零 Not Equal / Not Zero
`sets`	`SF`	负数 Negative
`setns`	`~SF`	非负 Nonnegative
`setg`	`~(SF^OF)&~ZF`	大于（有符号）Greater (Signed)
`setge`	`~(SF^OF)`	大于或等于（有符号数）Greater or Equal (Signed)
`setl`	`(SF^OF)`	小于（有符号数）Less (Signed)
`setle`	`(SF^OF)|ZF`	小于或等于（有符号数）Less or Equal (Signed)
`seta`	`~CF&~ZF`	高于（无符号数）Above (unsigned)
`setb`	`CF`	低于（无符号数）Below (unsigned)

读取条件码 Reading Condition Codes (Cont.)

- **SetX指令 SetX Instructions:**
  - 根据条件码的组合设置单个字节 Set single byte based on combination of condition codes
- **可寻址的字节寄存器之一 One of addressable byte registers**
  - 不会修改剩余的字节 Does not alter remaining bytes
  - 典型地使用movzbl(0扩展字节到双字)来完成工作 Typically use **`movzbl`** to finish job
    - 32位指令也设置高32位为0 32-bit instructions also set upper 32 bits to 0

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

寄存器 Register	用途 Use(s)
`%rdi`	参数x Argument **`x`**
`%rsi`	参数y Argument **`y`**
`%rax`	返回值 Return value

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

跳转指令 Jumping

- 跳转指令 jX Instructions
  - 根据条件码跳转到代码的不同部分 Jump to different part of code depending on condition codes

jX	条件 Condition	描述 Description
`jmp`	`1`	无条件 **Unconditional**
`je`	`ZF`	相等/零 **Equal / Zero**
`jne`	`~ZF`	不等/非零 **Not Equal / Not Zero**
`js`	`SF`	负数 **Negative**
`jns`	`~SF`	非负数 **Nonnegative**
`jg`	`~(SF^OF)&~ZF`	大于（有符号数） **Greater (Signed)**
`jge`	`~(SF^OF)`	大于或等于（有符号数）**Greater or Equal (Signed)**
`jl`	`(SF^OF)`	小于（有符号数）**Less (Signed)**
`jle`	`(SF^OF)|ZF`	小于或等于（有符号数）**Less or Equal (Signed)**
`ja`	`~CF&~ZF`	高于**Above**（无符号数） **(unsigned)**
`jb`	`CF`	低于**Below**（无符号数） **(unsigned)**

循环翻译 “Do-While” Loop Compilation

Goto版本 Goto Version

```
long pcount_goto
  (unsigned long x) {
  long result = 0;
 loop:
  result += x & 0x1;
  x >>= 1;
  if(x) goto loop;
  return result;
}
```

寄存器 Register	用途 Use(s)
`%rdi`	参数x Argument **x**
`%rax`	结果 **result**

```
        movl    $0, %eax     #  result = 0
.L2:                         # loop:
        movq    %rdi, %rdx
        andl    $1, %edx     #  t = x & 0x1
        addq    %rdx, %rax   #  result += t
        shrq    %rdi         #  x >>= 1
        jne     .L2          #  if (x) goto loop
        rep; ret
```

通用“While”循环翻译方法#1
General “While” Translation #1

- “跳转到中间”翻译方法 **“Jump-to-middle” translation**
- 使用编译参数 **Used with –Og**

While版本 While version

```
while (Test)
    Body
```

Goto版本 Goto Version

```
    goto test;
loop:
    Body
test:
    if (Test)
        goto loop;
done:
```

通用“While”翻译方法#2
General “While” Translation #2

While版本 While version

```
while (Test)
  Body
```

- “Do-while”转换 “Do-while” conversion
- 使用编译参数 Used with –O1

Do-While版本 Do-While Version

```
  if (!Test)
    goto done;
  do
    Body
    while(Test);
done:
```

Goto版本 Goto Version

```
  if (!Test)
    goto done;
loop:
  Body
  if (Test)
    goto loop;
done:
```

"For"循环形式 "For" Loop Form

通用格式 General Form

```
for (Init; Test; Update)

    Body
```

```
#define WSIZE 8*sizeof(int)
long pcount_for
  (unsigned long x)
{
  size_t i;
  long result = 0;
  for (i = 0; i < WSIZE; i++)
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
  }
  return result;
}
```

初始 Init

```
i = 0
```

测试 Test

```
i < WSIZE
```

更新 Update

```
i++
```

循环体 Body

```
{
  unsigned bit =
      (x >> i) & 0x1;
  result += bit;
}
```

"For"循环转换成While循环

"For" Loop → While Loop

For循环版本 For Version

```
for (Init; Test; Update )

    Body
```

While循环版本 While Version

```
Init;

while (Test) {

    Body

    Update;

}
```

For-While转换 For-While Conversion

初始 Init

```
i = 0
```

测试 Test

```
i < WSIZE
```

更新 Update

```
i++
```

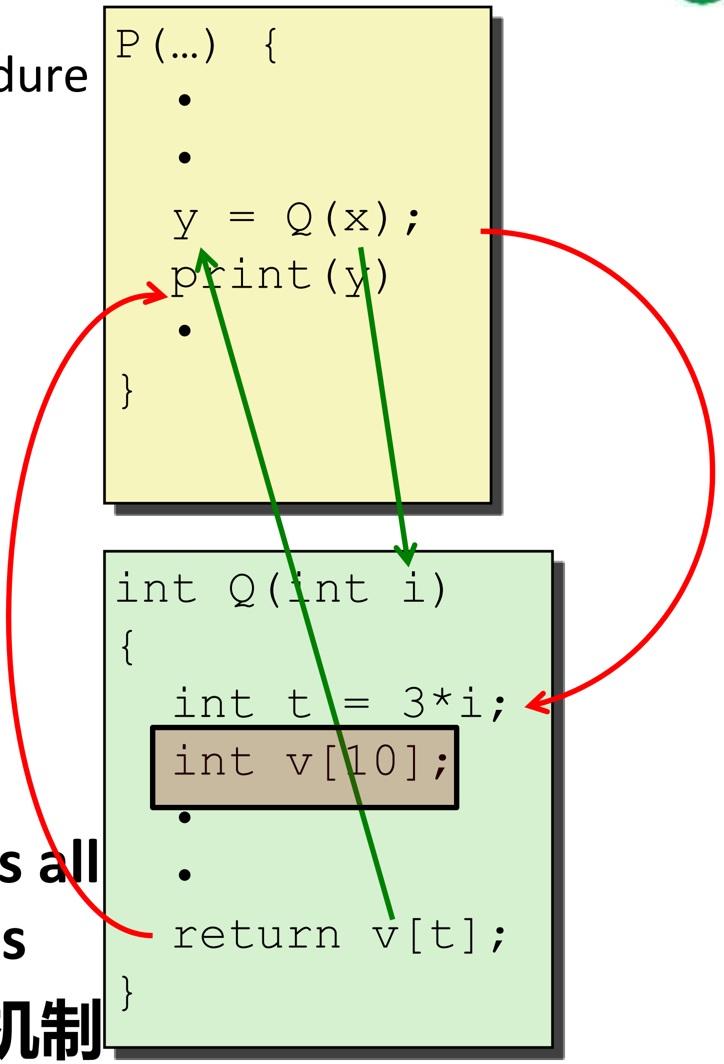
循环体 Body

```
{
  unsigned bit =
      (x >> i) & 0x1;
  result += bit;
}
```

```
long pcount_for_while
   (unsigned long x)
{
  size_t i;
  long result = 0;
  i = 0;
  while (i < WSIZE)
  {
    unsigned bit =
       (x >> i) & 0x1;
    result += bit;
    i++;
  }
  return result;
}
```

过程中的机制 Mechanisms in Procedures

- **传递控制 Passing control**
  - 进入过程代码的开始 To beginning of procedure code
  - 回到返回点 Back to return point
- **传递数据 Passing data**
  - 过程参数 Procedure arguments
  - 返回值 Return value
- **内存管理 Memory management**
  - 在过程执行期间分配内存 Allocate during procedure execution
  - 返回时释放内存 Deallocate upon return
- **所有机制由机器指令实现 Mechanisms all implemented with machine instructions**
- **x86-64的过程实现仅使用这些需要的机制 x86-64 implementation of a procedure uses only those mechanisms required**

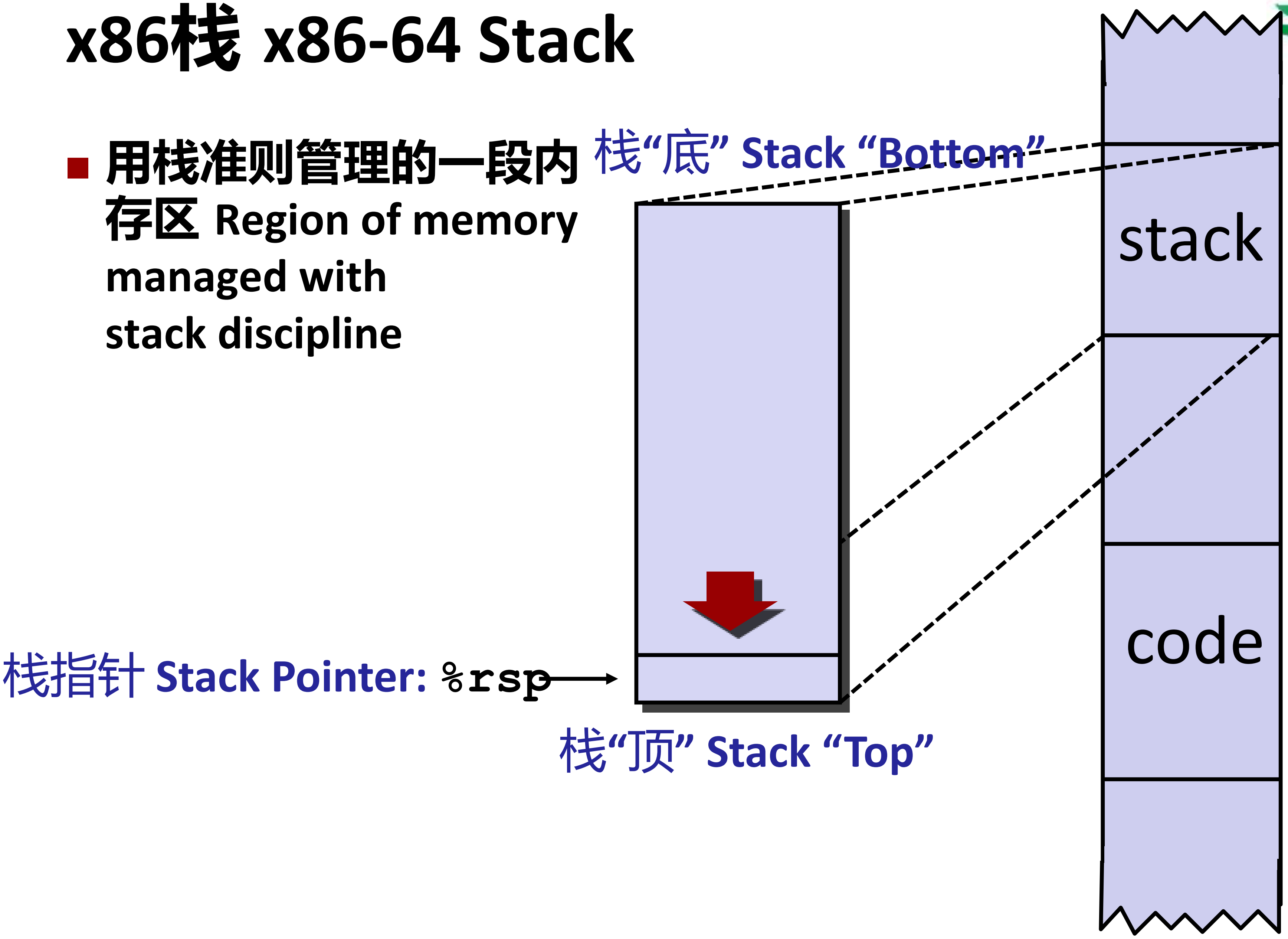


```
P(…) {
  •
  •
  y = Q(x);
  print(y)
  •
}
```

```
int Q(int i)
{
  int t = 3*i;
  int v[10];
  •
  •
  return v[t];
}
```

x86栈 x86-64 Stack

- 用栈准则管理的一段内存区 Region of memory managed with stack discipline



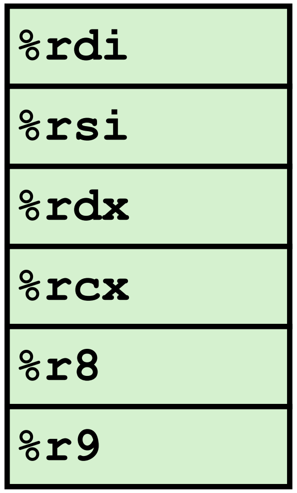
过程控制流 Procedure Control Flow

- **使用栈支持过程调用和返回 Use stack to support procedure call and return**
- **过程调用 Procedure call: `call label`**
  - 将返回地址压入栈 Push return address on stack
  - 跳转到标号处 Jump to ***label***
- **返回地址 Return address:**
  - 调用指令之后那条指令的地址 Address of the next instruction right after call
  - 反汇编的示例 Example from disassembly
- **过程返回 Procedure return: `ret`**
  - 从栈弹出地址 Pop address from stack
  - 跳转到该地址 Jump to address

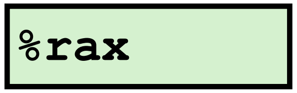
过程数据流 Procedure Data Flow

寄存器 Registers

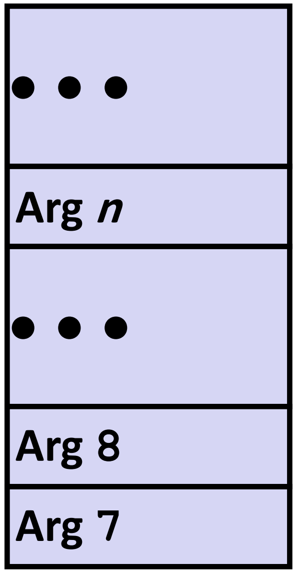
- First 6 arguments



- 返回值 Return value



栈 Stack



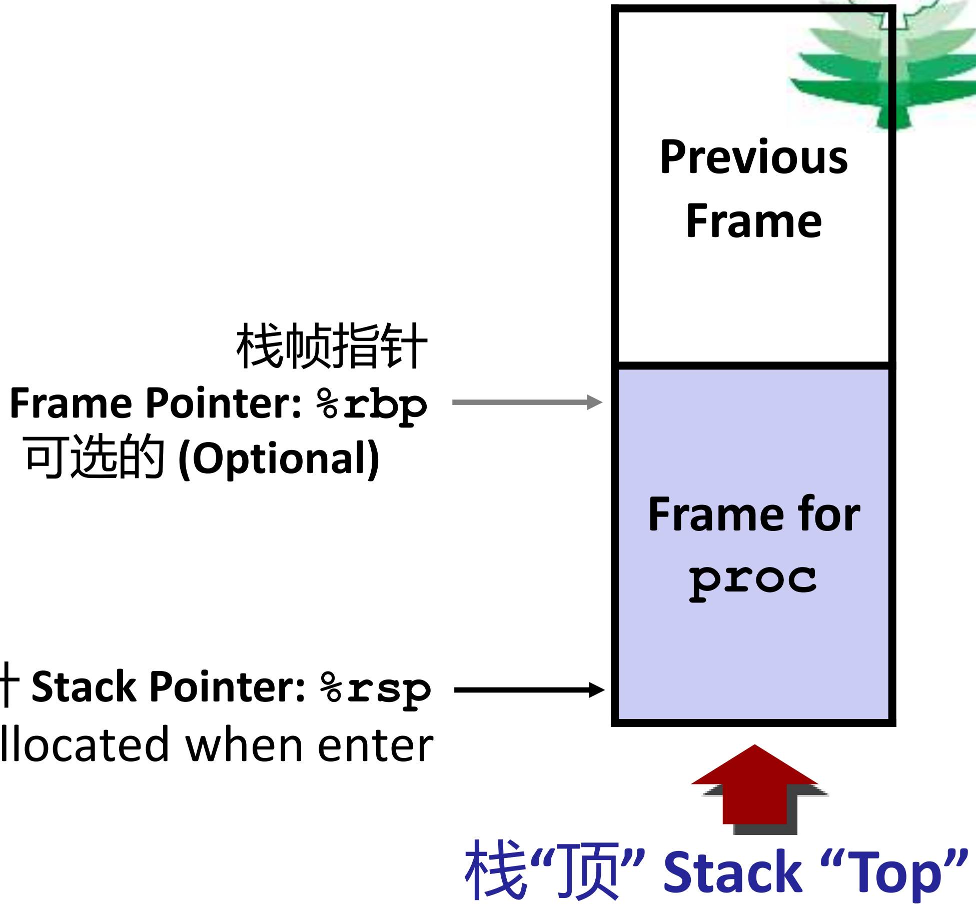
- 仅在需要时才分配栈空间
Only allocate stack space when needed

基于栈的语言 Stack-Based Languages

- **支持递归的语言 Languages that support recursion**
  - 例如 e.g., C, Pascal, Java
  - 代码必须是“可重入的” Code must be “*Reentrant*”
    - 单一过程同时有多个实例 Multiple simultaneous instantiations of single procedure
  - 需要有一些地方存储每个实例的状态 Need some place to store state of each instantiation
    - 参数 Arguments
    - 局部变量 Local variables
    - 返回指针 Return pointer
- **栈规则 Stack discipline**
  - 在限定的时间内对于给定的过程需要的状态 State for given procedure needed for limited time
    - 从过程被调用到过程返回 From when called to when return
  - 被调用者在调用者返回之前返回 Callee returns before caller does
- **栈分配以栈帧形式 Stack allocated in *Frames***
  - 单一过程实例的状态 state for single procedure instantiation

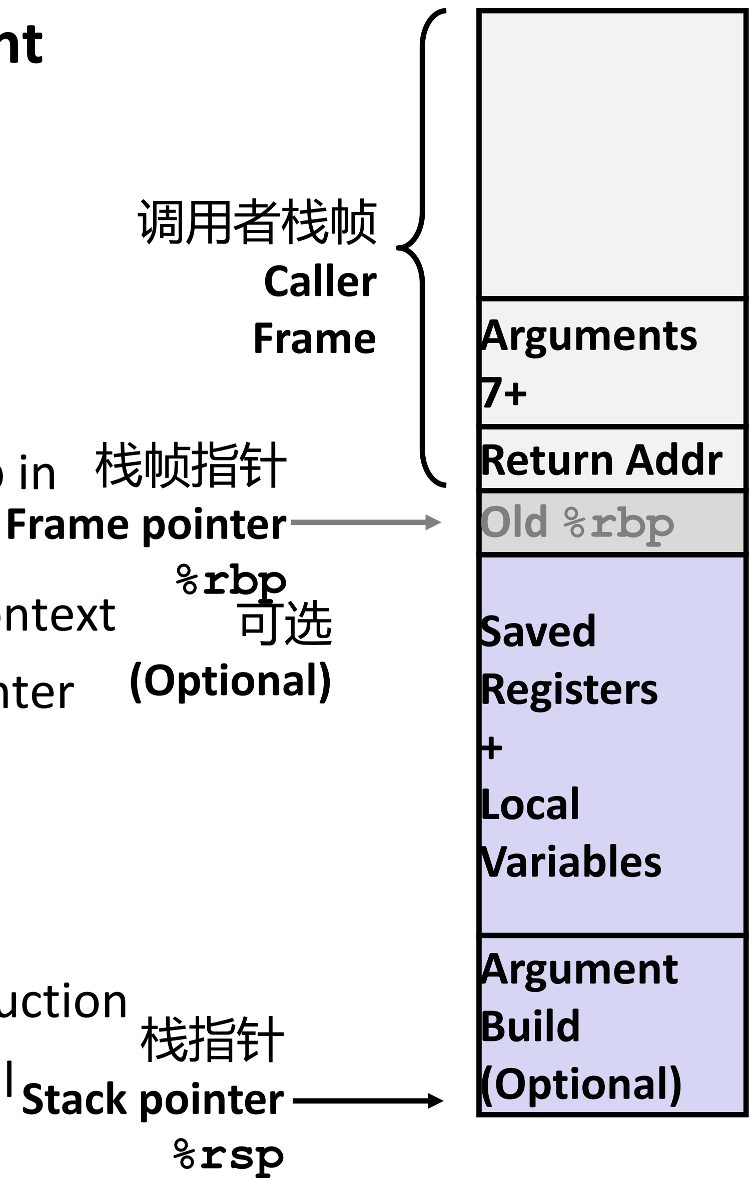
栈帧 Stack Frames

- **内容 Contents**
  - 返回信息 Return information
  - 局部存储（如果需要）
    - Local storage (if needed)
  - 临时空间（如果需要）
    - Temporary space (if needed)
- **管理 Management**
  - 当进入过程时分配空间 Space allocated when enter procedure
    - “初始”代码 “Set-up” code
    - 包括call指令的压栈 Includes push by **`call`** instruction
  - 当返回时释放空间 Deallocated when return
    - “结束”代码 “Finish” code
    - 包括ret指令的弹出栈 Includes pop by **`ret`** instruction


Previous Frame
Frame for proc
栈帧指针 Frame Pointer: %rbp 可选的 (Optional)
栈指针 Stack Pointer: %rsp
栈“顶” Stack “Top”

x86-64/Linux栈帧 x86-64/Linux Stack Frame

- **当前栈帧（自“顶”向下）Current Stack Frame (“Top” to Bottom)**
  - “参数构建：” 有关调用的函数参数 “Argument build:” Parameters for function about to call
  - 局部变量 Local variables 如果不能存储在寄存器中 If can’t keep in registers
  - 保存的寄存器上下文 Saved register context
  - 老的栈帧指针（可选） Old frame pointer (optional)
- **调用者栈帧 Caller Stack Frame**
  - 返回地址 Return address
    - Call指令压栈 Pushed by `call` instruction
  - 本次调用的参数 Arguments for this call

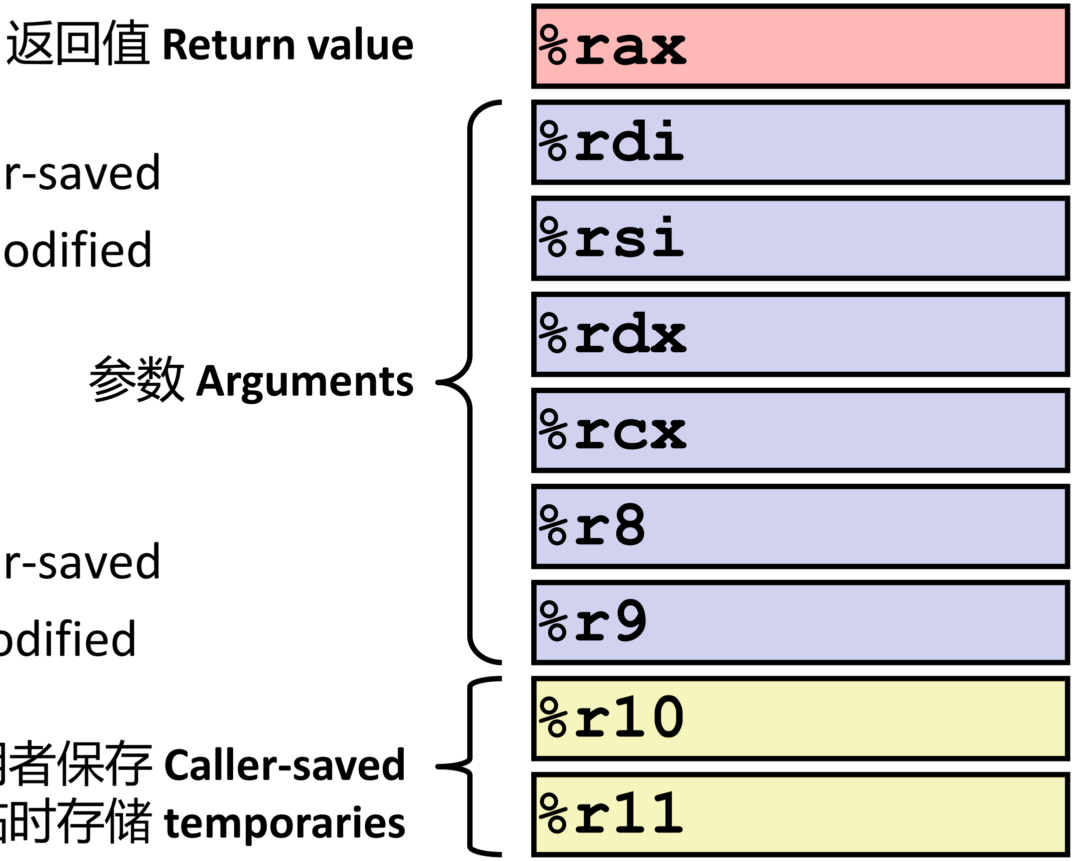


寄存器保存规则 Register Saving Conventions

- **当过程yoo调用who时 When procedure `yoo` calls `who`:**
  - `Yoo`是调用者 `yoo` is the ***caller***
  - `Who`是被调用者 `who` is the ***callee***
- **寄存器可以用于临时存储吗？ Can register be used for temporary storage?**
- **规则 Conventions**
  - *"调用者负责保存" "Caller Saved"*
    - 调用者在调用前在其栈帧中保存临时值 Caller saves temporary values in its frame before the call
  - *"被调用者负责保存" "Callee Saved"*
    - 被调用者在使用前在其栈帧中保存临时值 Callee saves temporary values in its frame before using
    - 被调用者在返回到调用者之前恢复临时值 Callee restores them before returning to caller

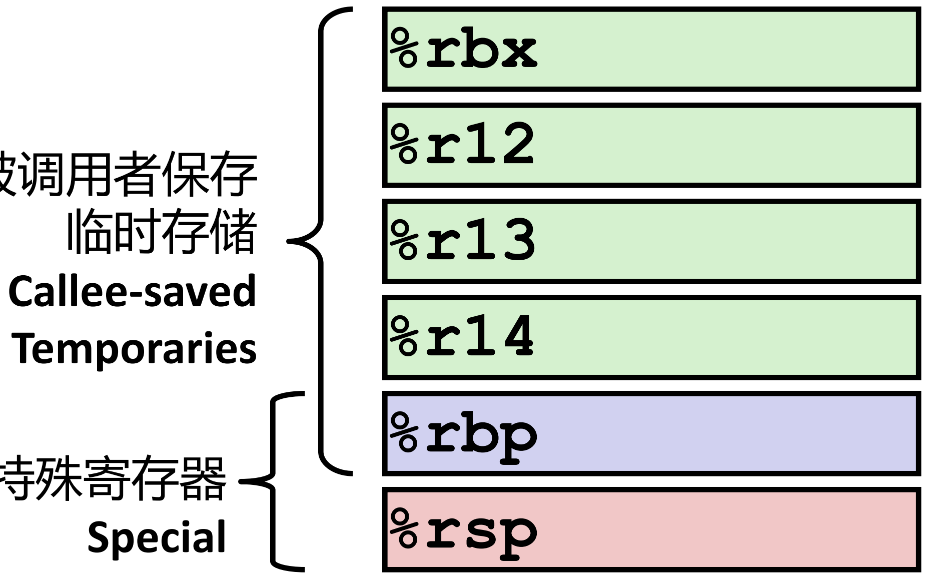
x86-64Linux寄存器用法 #1
x86-64 Linux Register Usage #1

- `%rax`
  - 返回值 Return value
  - 也是调用者保存 Also caller-saved
  - 可以被过程修改 Can be modified by procedure
- `%rdi, ..., %r9`
  - 参数 Arguments
  - 也是调用者保存 Also caller-saved
  - 可以被过程修改Can be modified by procedure
- `%r10, %r11`
  - 调用者保存 Caller-saved
  - 可以被过程修改 Can be modified by procedure



x86-64Linux寄存器用法 #2
x86-64 Linux Register Usage #2

- **`%rbx`, `%r12`, `%r13`, `%r14`**
  - 被调用者保存 Callee-saved
  - 被调用者必须保存和恢复 Callee must save & restore
- **`%rbp`**
  - 被调用者保存 Callee-saved
  - 被调用者必须保存和恢复 Callee must save & restore
  - 可能用作栈帧指针 May be used as frame pointer
  - 能够混合和匹配 Can mix & match
- **`%rsp`**
  - 被调用者保存的特殊形式 Special form of callee save
  - 从过程退出时恢复到原始值 Restored to original value upon



递归函数终止情况
Recursive Function Terminal Case

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
   movl     $0, %eax
   testq    %rdi, %rdi
   je       .L6
   pushq    %rbx
   movq     %rdi, %rbx
   andl     $1, %ebx
   shrq     %rdi
   call     pcount_r
   addq     %rbx, %rax
   popq     %rbx
.L6:
   rep; ret
```

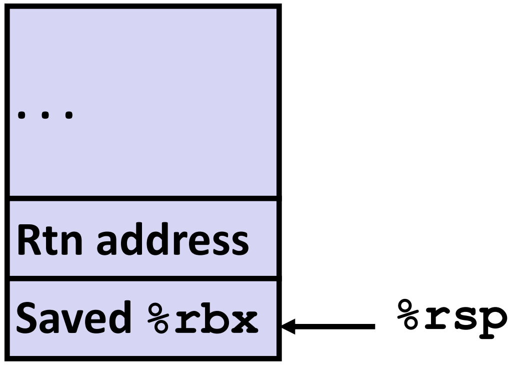
寄存器 Register	用途 Use(s)	类型 Type
%rdi	x	参数 Argument
%rax	返回值 Return value	返回值 Return value

递归函数寄存器保存
Recursive Function Register Save

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

寄存器 Register	用途 Use(s)	类型 Type
%rdi	x	参数 Argument



递归函数调用设置 Recursive Function Call Setup

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

寄存器 Register	用途 Use(s)	类型 Type
%rdi	x >> 1	递归参数 Rec. argument
%rbx	x & 1	调用者保存 Callee-saved

递归函数调用 Recursive Function Call

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

寄存器 Register	用途 Use(s)	类型 Type
`%rbx`	`x & 1`	被调用者保存 Callee-saved
`%rax`	递归调用返回值 Recursive call return value	

递归调用结果 Recursive Function Result

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

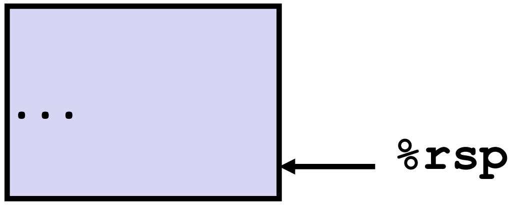
```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

寄存器 Register	用途 Use(s)	类型 Type
`%rbx`	`x & 1`	调用者保存 Callee-saved
`%rax`	返回值 Return value	

递归函数完成 Recursive Function Completion

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```



寄存器 Register	用途 Use(s)	类型 Type
%rax	返回值 Return value	返回值 Return value

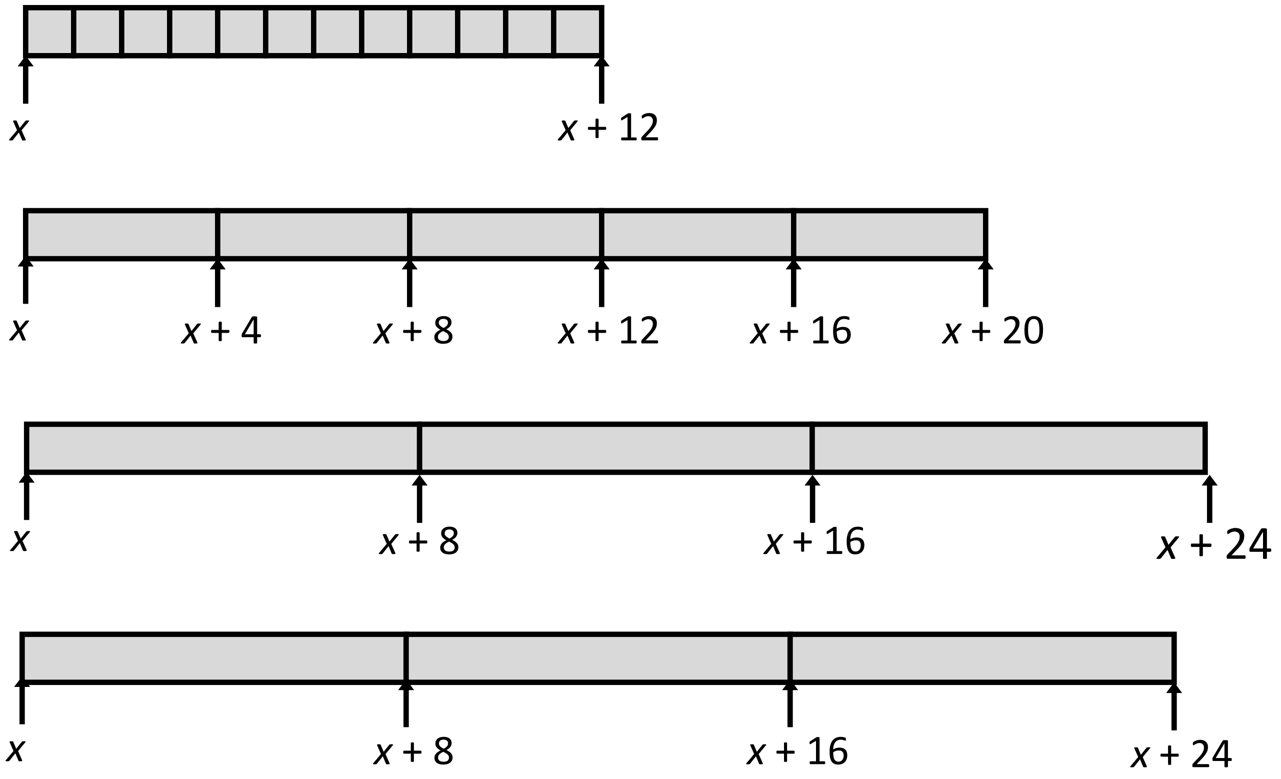
数组分配 Array Allocation

- **基本原则 Basic Principle**

  T **A[*L*];**

  - 数组的数据类型为T且长度为L Array of data type *T* and length *L*
  - 在内存中分配L*sizeof(T)字节的连续区域 Contiguously allocated region of *L* * **sizeof**（*T*）bytes in memory

```
char string[12];
int val[5];
double a[3];
char *p[3];
```



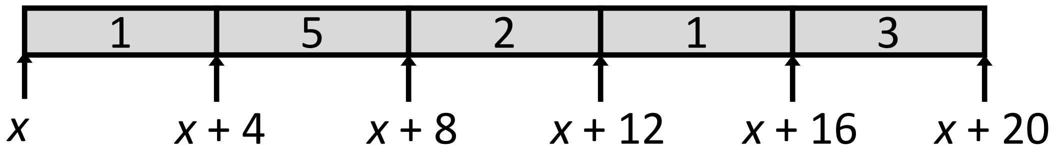
数组访问 Array Access

- **基本原则 Basic Principle**

  T **A[*L*];**

  - 数组的数据类型为T且长度为L Array of data type *T* and length *L*
  - 标识符A可以用作指向数组第0个元素的指针 Identifier **A** can be used as a pointer to array element 0: Type *T**

`int val[5];`



引用Reference	类型Type	值Value
`val[4]`	`int`	3
`val`	`int *`	*x*
`val+1`	`int *`	*x* + 4
`&val[2]`	`int *`	*x* + 8
`val[5]`	`int`	??
`*(val+1)`	`int`	5
`val + i`	`int *`	*x* + 4 *i*

理解指针和数组 #3
Understanding Pointers & Arrays #3

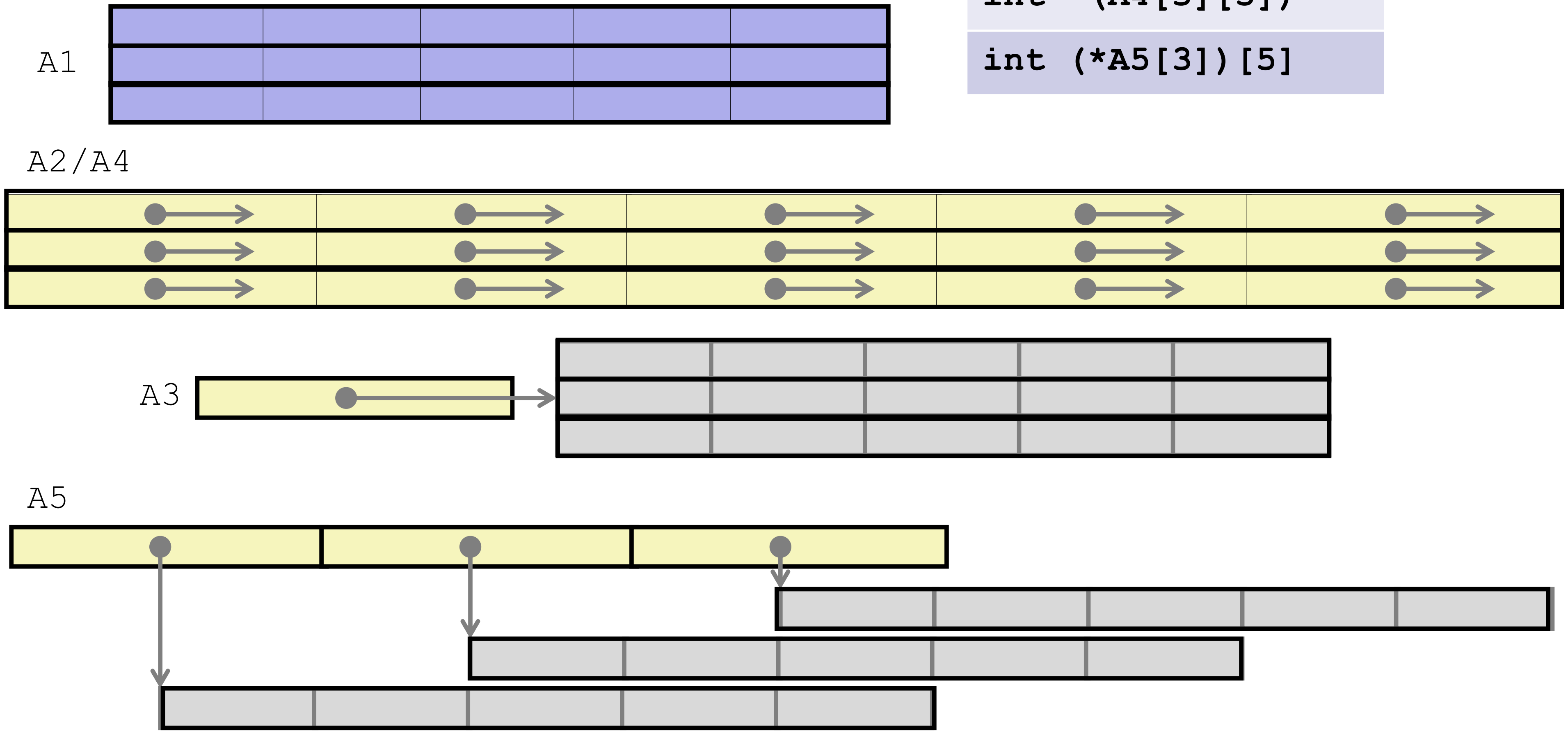
Decl	An			*An			**An		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3][5]`									
`int *A2[3][5]`									
`int (*A3)[3][5]`									
`int *(A4[3][5])`									
`int (*A5[3])[5]`									

- Cmp: Compiles (Y/N)
- Bad: Possible bad pointer reference (Y/N)
- Size: Value returned by `sizeof`

Decl	***An		
	Cmp	Bad	Size
`int A1[3][5]`			
`int *A2[3][5]`			
`int (*A3)[3][5]`			
`int *(A4[3][5])`			
`int (*A5[3])[5]`			

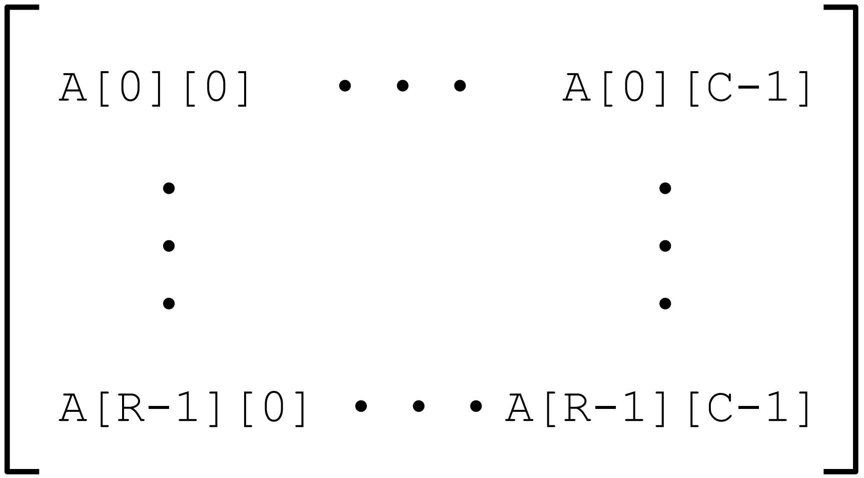
已分配指针 Allocated pointer	🟨
Allocated pointer to unallocated int	🟨 ●→
未分配指针 Unallocated pointer	⬜
已分配整数 Allocated int	🟪
未分配整数 Unallocated int	⬜

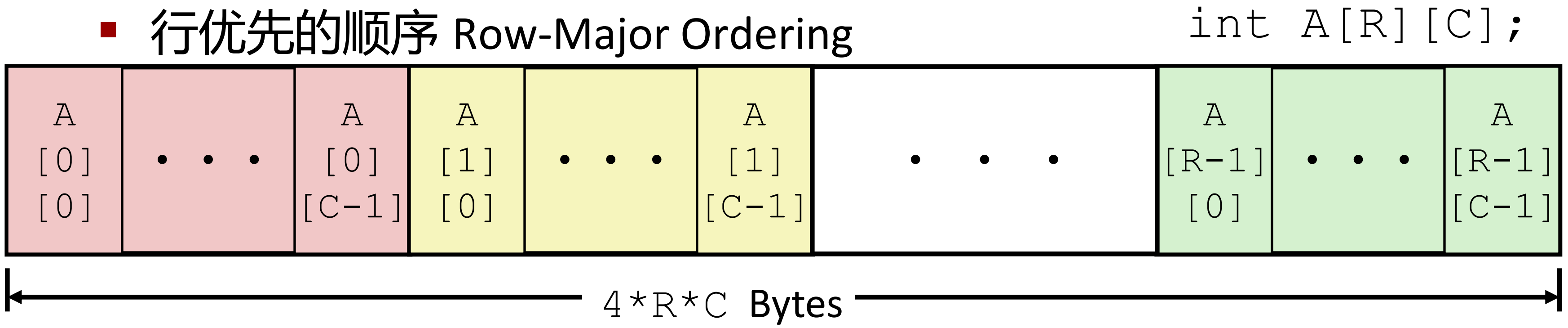
Declaration
`int A1[3][5]`
`int *A2[3][5]`
`int (*A3)[3][5]`
`int *(A4[3][5])`
`int (*A5[3])[5]`



多维（嵌套）数组 Multidimensional (Nested) Arrays

- **声明 Declaration**
  - *T* **A**[*R*][*C*];
  - 数据类型为T的二维数组 2D array of data type *T*
  - R行C列 *R* rows, *C* columns
  - 类型T元素需要K个字节 Type *T* element requires *K* bytes
- **数组大小 Array Size**
  - *R* * *C* * *K* bytes字节
- **排列 Arrangement**
  - 行优先的顺序 Row-Major Ordering

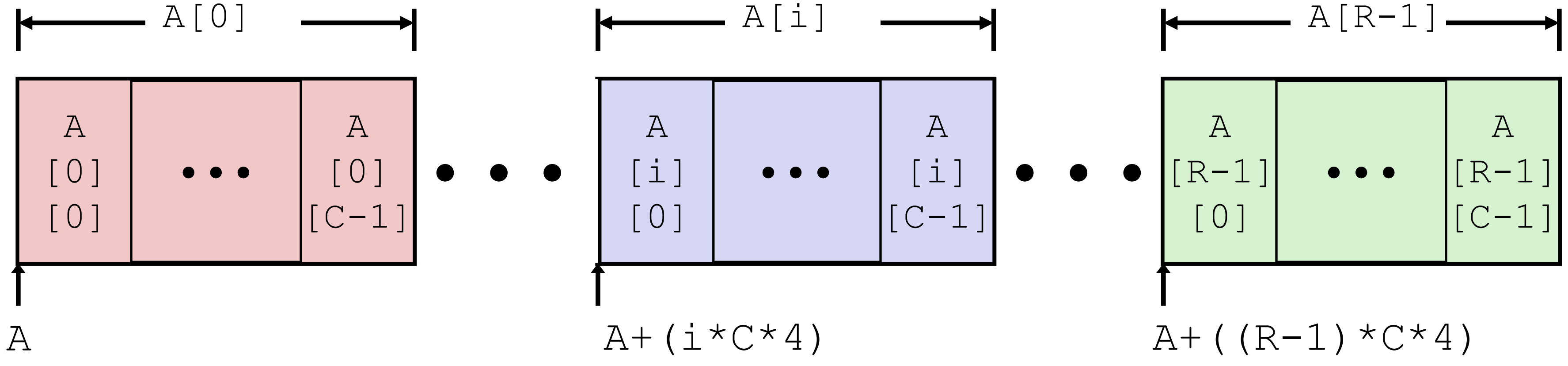




嵌套数组行访问 Nested Array Row Access

- **行向量 Row Vectors**
  - A[i]是C个元素的数组 **`A[i]`** is array of *C* elements
  - 每个类型为T的元素需要k字节 Each element of type *T* requires *K* bytes
  - 起始地址为 Starting address **`A +`** *i* * (*C* * *K*)

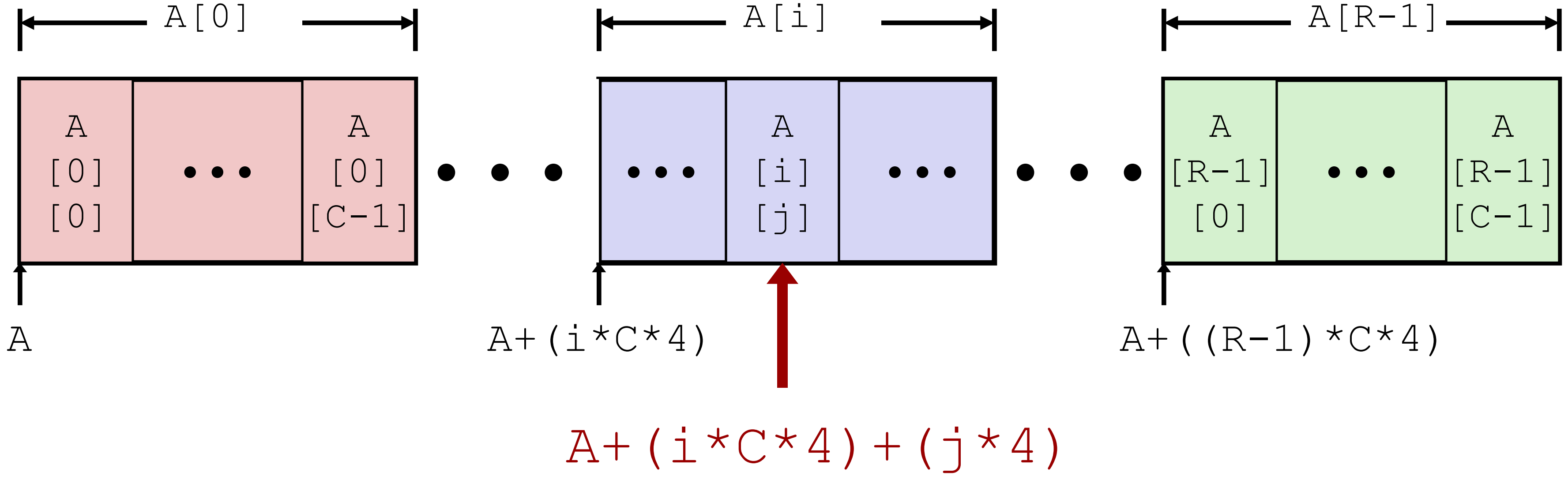
```
int A[R][C];
```



嵌套数组元素访问 Nested Array Element Access

- 数组元素 Array Elements
  - **`A[i][j]`**是类型为T的元素，需要K字节 **`A[i][j]`** is element of type *T*, which requires *K* bytes
  - 地址为 Address **`A + `** $i * (C * K) + j * K = A + (i * C + j) * K$

```
int A[R][C];
```



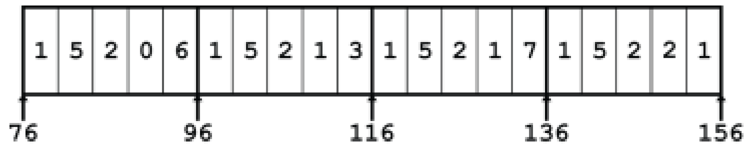
数组元素访问 Array Element Accesses

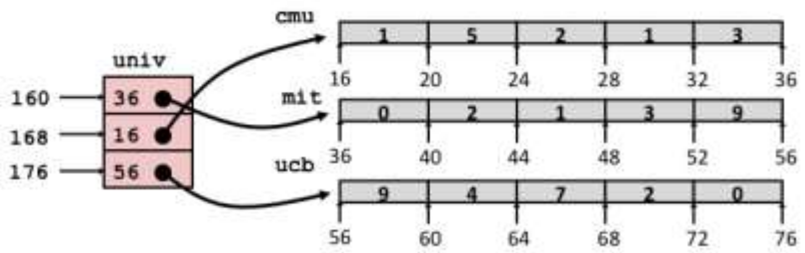
嵌套数组 Nested array

```
int get_pgh_digit
  (size_t index, size_t digit)
{
  return pgh[index][digit];
}
```

多级数组 Multi-level array

```
int get_univ_digit
  (size_t index, size_t digit)
{
  return univ[index][digit];
}
```





在C语言中访问看起来是类似的，但是地址计算方法非常不同

Accesses looks similar in C, but address computations very different:

`Mem[pgh+20*index+4*digit]`   `Mem[Mem[univ+8*index]+4*digit]`

二维矩阵代码 N X N Matrix Code

- **固定维数 Fixed dimensions**
  - 编译时知道N的值 Know value of N at compile time
- **变化维数，显式索引 Variable dimensions, explicit indexing**
  - 传统方法实现动态数组 Traditional way to implement dynamic arrays
- **变化维数，隐式索引 Variable dimensions, implicit indexing**
  - 目前gcc支持 Now supported by gcc

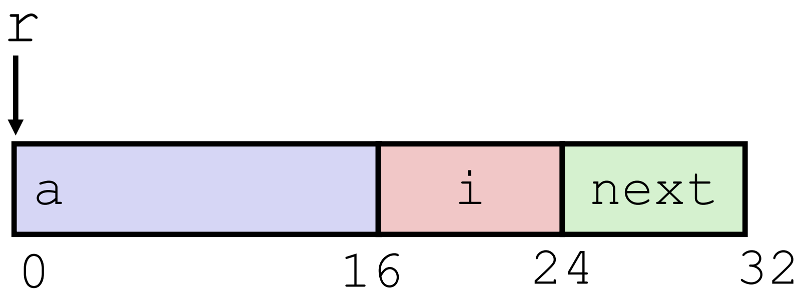
```
#define N 16
typedef int fix_matrix[N][N];
/* Get element a[i][j] */
int fix_ele(fix_matrix a,
            size_t i, size_t j)
{
  return a[i][j];
}
```

```
#define IDX(n, i, j) ((i)*(n)+(j))
/* Get element a[i][j] */
int vec_ele(size_t n, int *a,
            size_t i, size_t j)
{
  return a[IDX(n,i,j)];
}
```

```
/* Get element a[i][j] */
int var_ele(size_t n, int a[n][n],
            size_t i, size_t j) {
  return a[i][j];
}
```

结构表示 Structure Representation

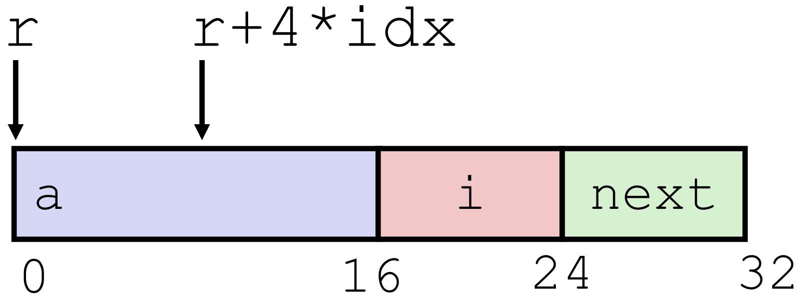
```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



- 结构表示为内存块 Structure represented as block of memory
  - 大到足够装下所有字段 Big enough to hold all of the fields
- 字段顺序按照声明顺序 Fields ordered according to declaration
  - 即使另一种顺序可能还能够更加紧凑进行表示 Even if another ordering could yield a more compact representation
- 编译器决定总体大小+字段位置 Compiler determines overall size + positions of fields
  - 机器级程序并不理解源代码中的结构 Machine-level program has no

生成结构成员的指针 Generating Pointer to Structure Member

```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



- **生成数组元素的指针 Generating Pointer to Array Element**
  - 每个结构成员的偏移在编译时确定 Offset of each structure member determined at compile time
  - 计算方法 Compute as **`r + 4*idx`**

```
int *get_ap
 (struct rec *r, size_t idx)
{
  return &r->a[idx];
}
```

```
  # r in %rdi, idx in %rsi
  leaq  (%rdi,%rsi,4), %rax
  ret
```

对齐的原则 Alignment Principles

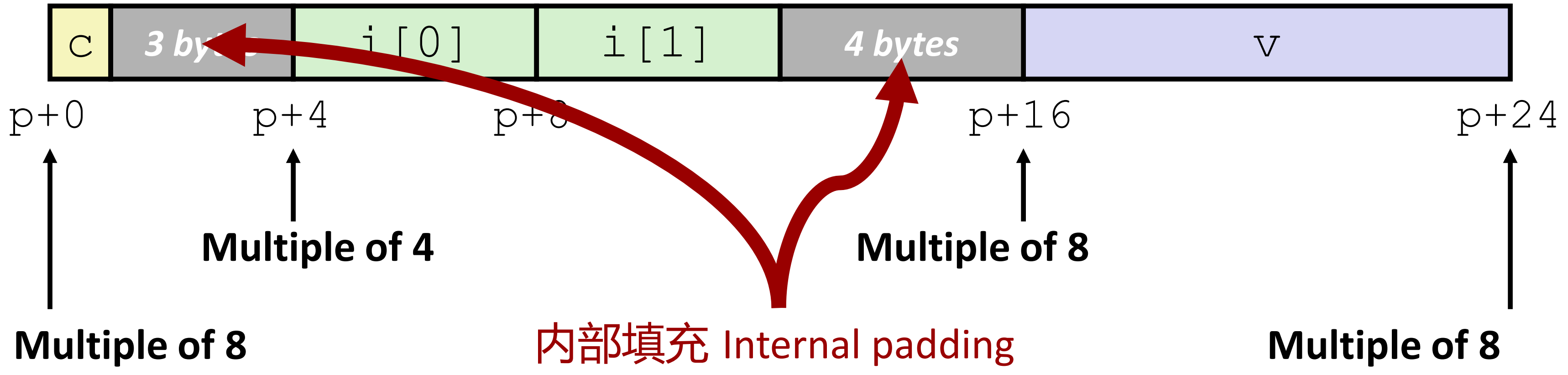
- **对齐的数据 Aligned Data**
  - 基本数据类型需要K字节 Primitive data type requires ***K*** bytes
  - 地址必须是K的整数倍 Address must be multiple of ***K***
  - 在某些机器上严格满足该要求；在x86-64上建议满足 Required on some machines; advised on x86-64
- **对齐数据的动机 Motivation for Aligning Data**
  - 内存访问以（对齐的）4或8字节数据块（依赖于不同的系统）为单位 Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
    - 装载或存储数据比较低效，因为这样会跨越四字边界 Inefficient to load or store datum that spans quad word boundaries
    - 当数据跨越2页时虚拟存储器访问更棘手 Virtual memory trickier when datum spans 2 pages
- **编译器 Compiler**
  - 在结构中插入间隔以确保字段的正确对齐 Inserts gaps in structure to ensure correct alignment of fields

结构体的对齐要求

Satisfying Alignment with Structures

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

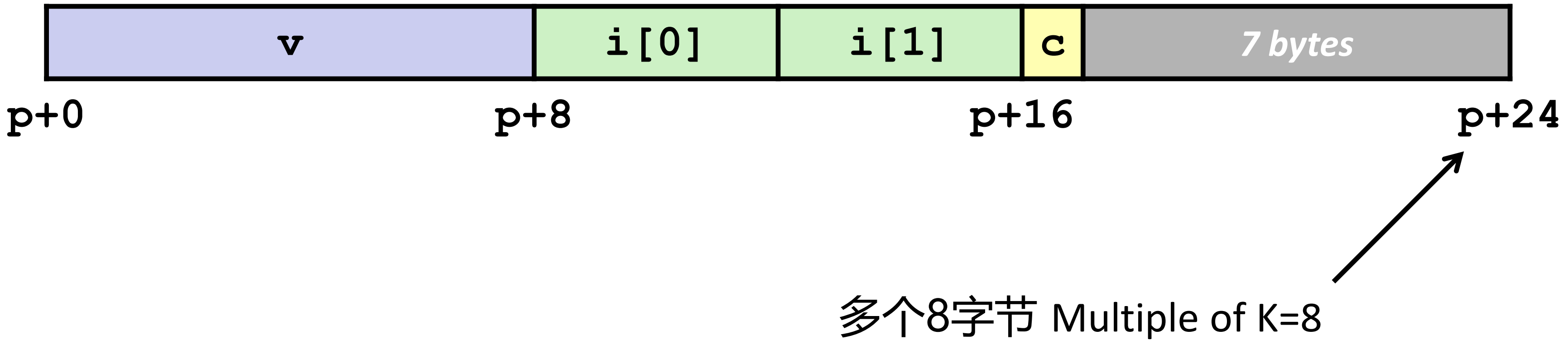
- **结构内部 Within structure:**
  - 必须满足每个元素的对齐需求 Must satisfy each element's alignment requirement
- **整个结构的排放 Overall structure placement**
  - 每个结构有对齐需求K Each structure has alignment requirement **K**
    - K为任何元素对齐要求的最大值 K = Largest alignment of any element
  - 起始地址和结构长度必须为K的整数倍 Initial address & structure length must be multiples of **K**
- **示例 Example:**
  - 由于有双精度浮点型元素，K为8字节 K = 8, due to **`double`** element



满足整体对齐需求 Meeting Overall Alignment Requirement

- 对于最大对齐需求K For largest alignment requirement K
- 整体结构必须是K的整数倍 Overall structure must be multiple of K

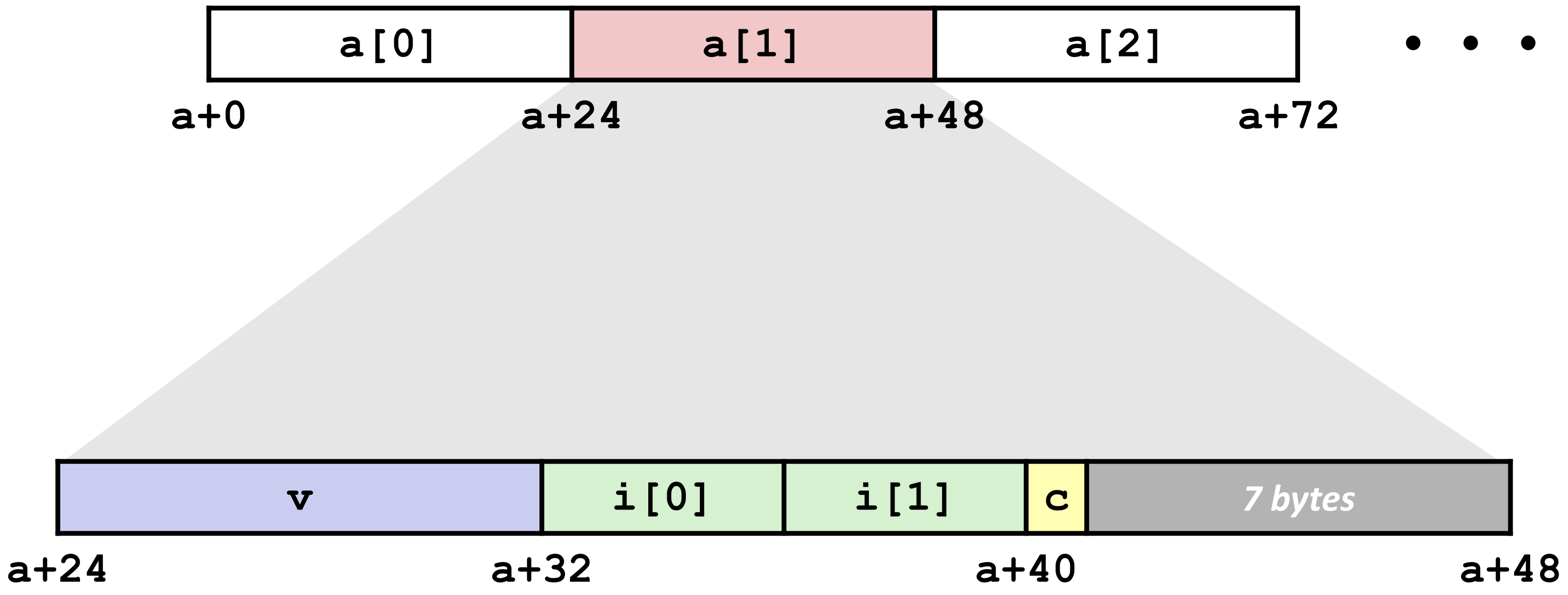
```
struct S2 {
  double v;
  int i[2];
  char c;
} *p;
```



结构数组 Arrays of Structures

- **整体结构长度是K的整数倍 Overall structure length multiple of K**
- **满足每个元素的对齐需求 Satisfy alignment requirement for every element**

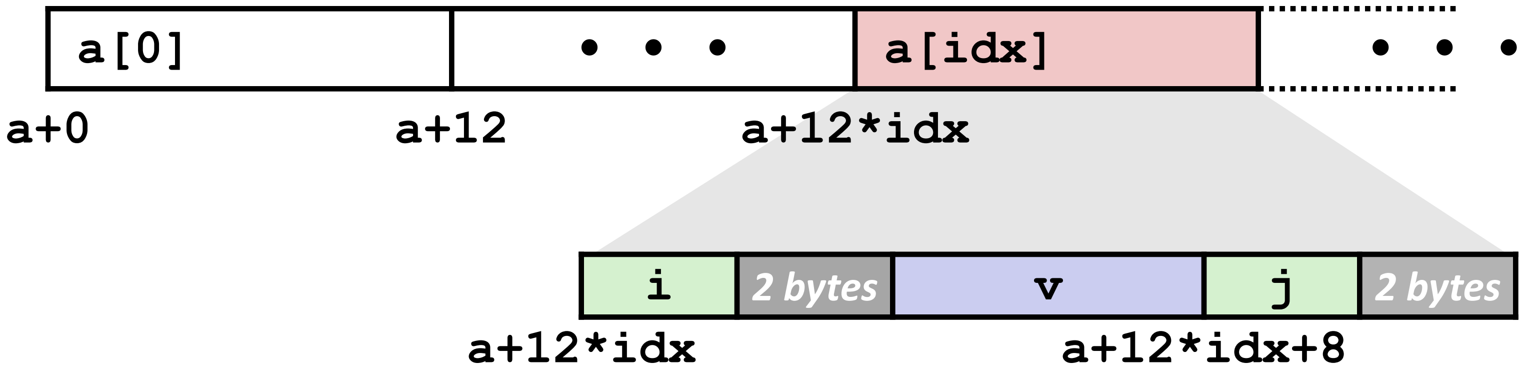
```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```



访问数组元素
Accessing Array Elements

```
struct S3 {
  short i;
  float v;
  short j;
} a[10];
```

- 计算数组的偏移 Compute array offset 12*idx
  - `sizeof(S3)`，包括对齐所需填充空白 including alignmen
- 元素j在结构内部的偏移是8 Element `j` is at offset 8 within structure
- 汇编器给出的偏移是a+8 Assembler gives offset a+8
  - 在链接时解析 Resolved during linking

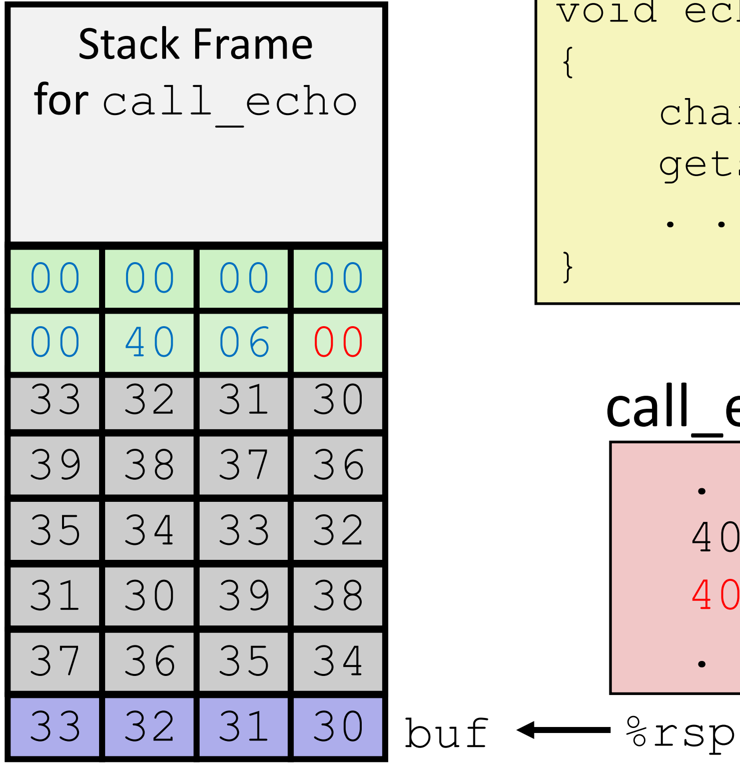


```
short get_j(int idx)
{
  return a[idx].j;
}
```

```
# %rdi = idx
leaq (%rdi,%rdi,2),%rax # 3*idx
movzwl a+8(,%rax,4),%eax
```

缓冲区溢出栈示例 #2
Buffer Overflow Stack Example #2

调用gets后 After call to gets



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

```
echo:
  subq   $0x18, %rsp
  movq   %rsp, %rdi
  call   gets
  . . .
```

call_echo:

```
 . . .
 4006be:  callq  4006cf <echo>
 4006c3:  add    $0x8,%rsp
 . . .
```

```
unix>./bufdemo-nsp
Type a string:012345678901234567890123
012345678901234567890123
Segmentation fault
```

程序"返回"到0x400600，然后崩溃 Program "returned" to 0x0400600, and then crashed.

栈击穿攻击 Stack Smashing Attacks

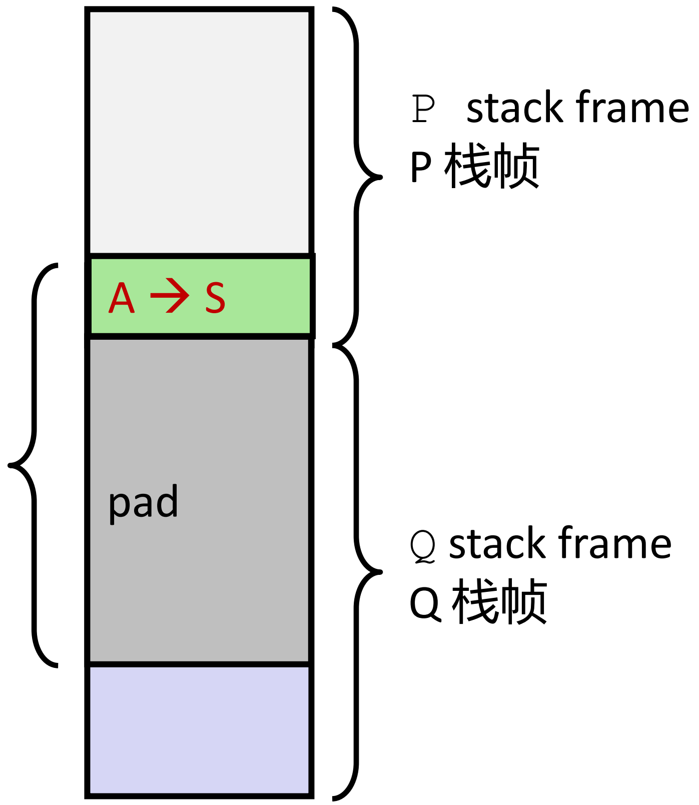
```
void P(){
  Q();
  ...
}
```

返回地址 return address A

```
int Q() {
  char buf[64];
  gets(buf);
  ...
  return ...;
}
```

```
void S(){
/* Something
   unexpected */
  ...
}
```

调用gets后的栈 Stack after call to `gets()`



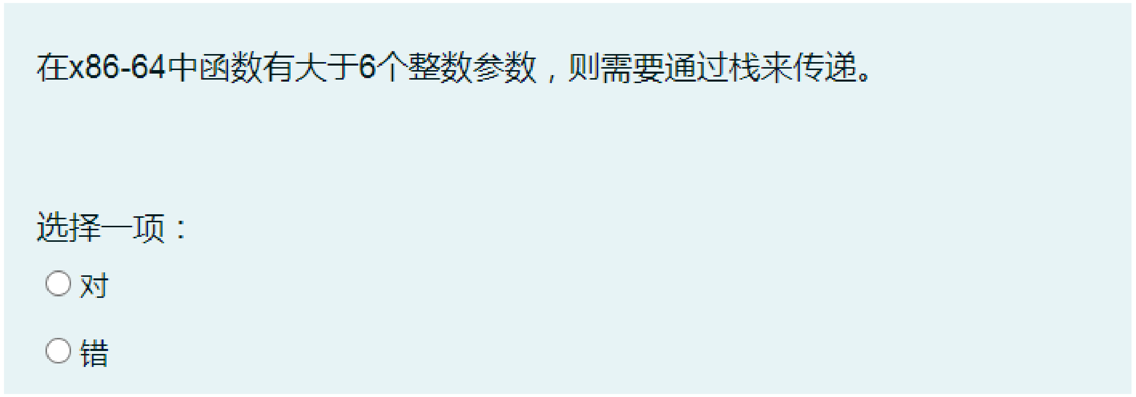
- **用某些其它代码S的地址覆盖正常的返回地址A Overwrite normal return address A with address of some other code S**
- **当Q执行ret时，会跳转到其它代码 When Q executes `ret`, will jump to other code**

如何防范缓冲区溢出攻击

OK, what to do about buffer overflow attacks

- 避免溢出漏洞 Avoid overflow vulnerabilities
- 采用系统级防护 Employ system-level protections
- 让编译器使用"栈金丝雀" Have compiler use "stack canaries"
- 让我们分别进行讨论 Lets talk about each…

例题

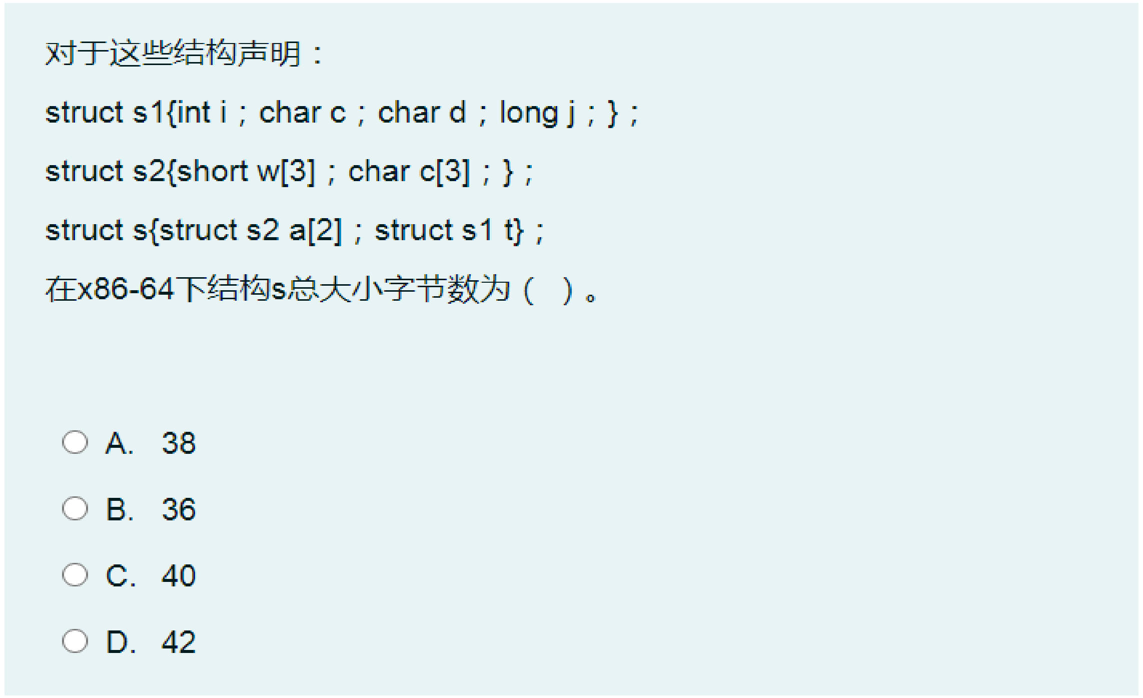


在x86-64中函数有大于6个整数参数，则需要通过栈来传递。

选择一项：

○ 对

○ 错



对于这些结构声明：

struct s1{int i；char c；char d；long j；}；

struct s2{short w[3]；char c[3]；}；

struct s{struct s2 a[2]；struct s1 t}；

在x86-64下结构s总大小字节数为（　）。

○ A.　38

○ B.　36

○ C.　40

○ D.　42

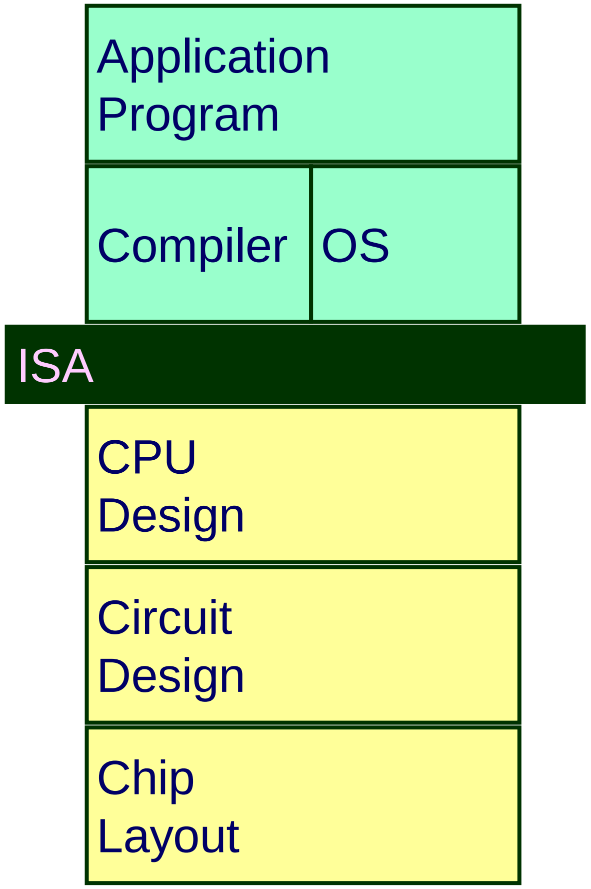
处理器体系结构

- Y86-64指令集体系结构
  - 寄存器、内存、控制码、PC、状态：
  - 指令组成与编码
  - 状态异常、常用指令详情
- 逻辑电路与硬件控制语言HCL
  - 执行单元、存储单元、时钟、逻辑表达式
- Y86-64实现
  - SEQ的硬件结构、指令时序、阶段计算；
  - 流水线实现SEQ+、PIPE-、PIPE的阶段寄存器信号
  - 流水线PIPE对数据冒险、控制冒险、异常的触发逻辑及控制机制
  - 性能分析

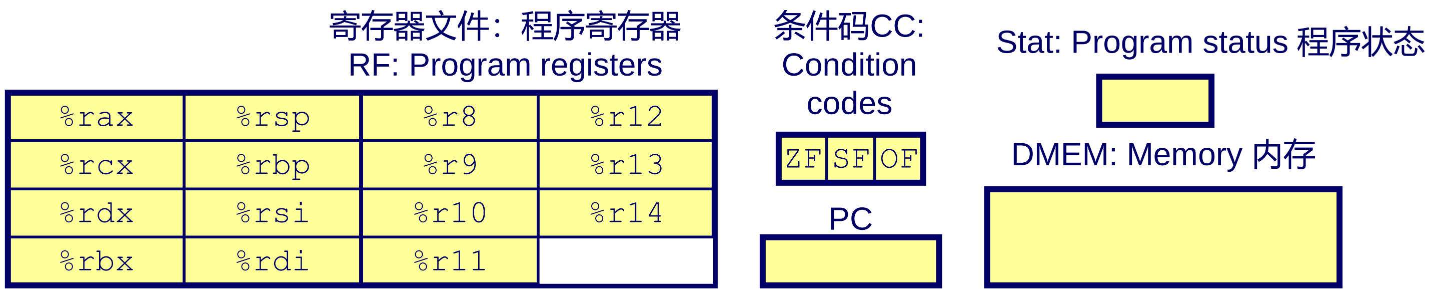
指令集体系结构 Instruction Set Architecture

抽象层次 Layer of Abstraction

- 之上：程序机器如何工作 Above: how to program machine
  - 处理器顺序执行指令 Processor executes instructions in a sequence
- 之下：需要构建什么 Below: what needs to be built
  - 使用各种技巧使其运行更快 Use variety of tricks to make it run fast
  - 例如同时执行多条指令 E.g., execute multiple instructions simultaneously



Y86-64处理器状态
Y86-64 Processor State



- **程序寄存器 Program Registers**
  - **15个寄存器（省略%r15），每个64位 15 registers (omit `%r15`). Each 64 bits**
- **条件码 Condition Codes**
  - **算术或逻辑运算指令设置单个比特位标志 Single-bit flags set by arithmetic or logical instructions**
    - » ZF: Zero零　　SF:Negative负数　　OF: Overflow溢出
- **程序计数器 Program Counter**
  - **指明下一条指令地址 Indicates address of next instruction**
- **程序状态 Program Status**
  - **指明正常运行还是一些错误情景 Indicates either normal operation or some error condition**
- **内存 Memory**
  - **字节寻址存储数组 Byte-addressable storage array**
  - **字采用小端字节顺序存储 Words stored in little-endian byte order**

Y86-64 Instructions 指令

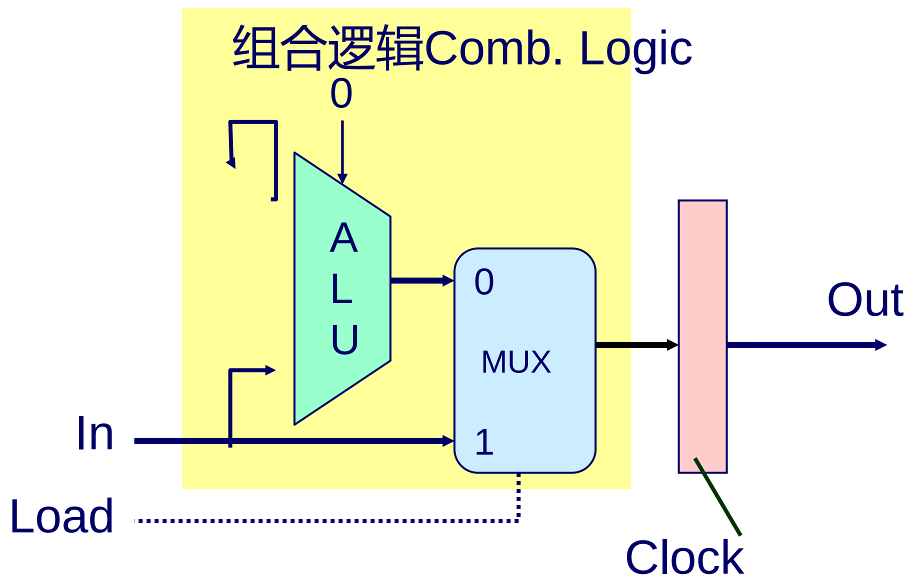
格式 Format

- 1-10字节信息从内存读出 1–10 bytes of information read from memory
  - 从第一个字节能够确定指令长度 Can determine instruction length from first byte
  - 与x86-64相比指令类型不是很多，而且编码更简单 Not as many instruction types, and simpler encoding than with x86-64
- 每次访问和修改一部分程序状态 Each accesses and modifies some part(s) of the program state

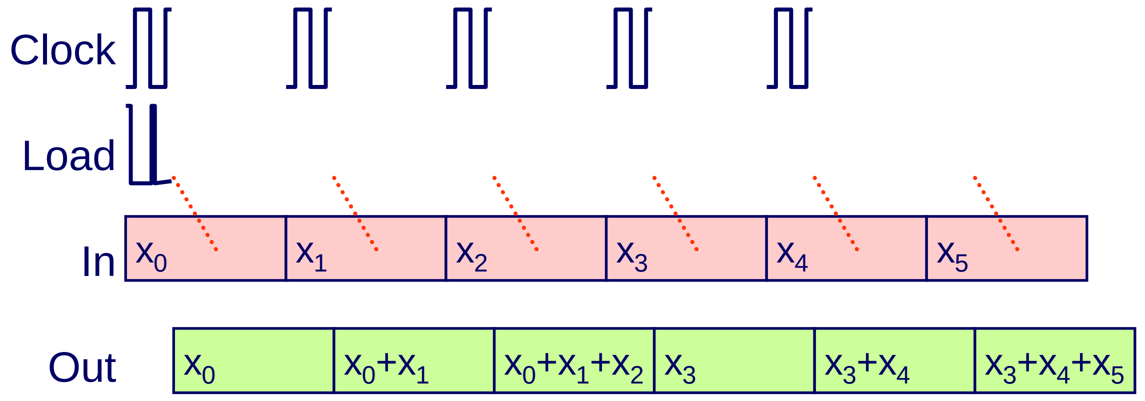
Y86-64 Instruction Set 指令集 #1

字节 Byte	0	1	2	3	4	5	6	7	8	9
`halt`	0 0									
`nop`	1 0									
`cmovXX` rA, rB	2 fn	rA rB								
`irmovq` V, rB	3 0	F rB	V							
`rmmovq` rA, D(rB)	4 0	rA rB	D							
`mrmovq` D(rB), rA	5 0	rA rB	D							
`OPq` rA, rB	6 fn	rA rB								
`jXX` Dest	7 fn	Dest								
`call` Dest	8 0	Dest								
`ret`	9 0									
`pushq` rA	A 0	rA F								
`popq` rA	B 0	rA F								

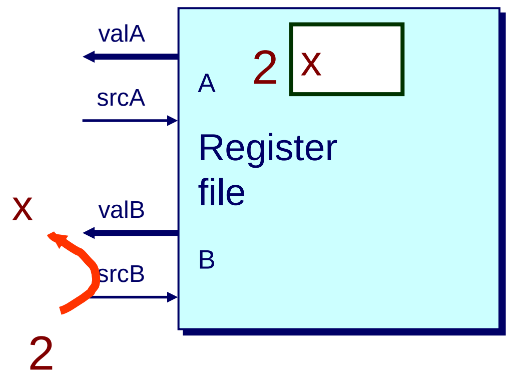
状态机示例 State Machine Example



- **累加器电路 Accumulator circuit**
- **每个周期装载或累加 Load or accumulate on each cycle**



寄存器文件时序 Register File Timing

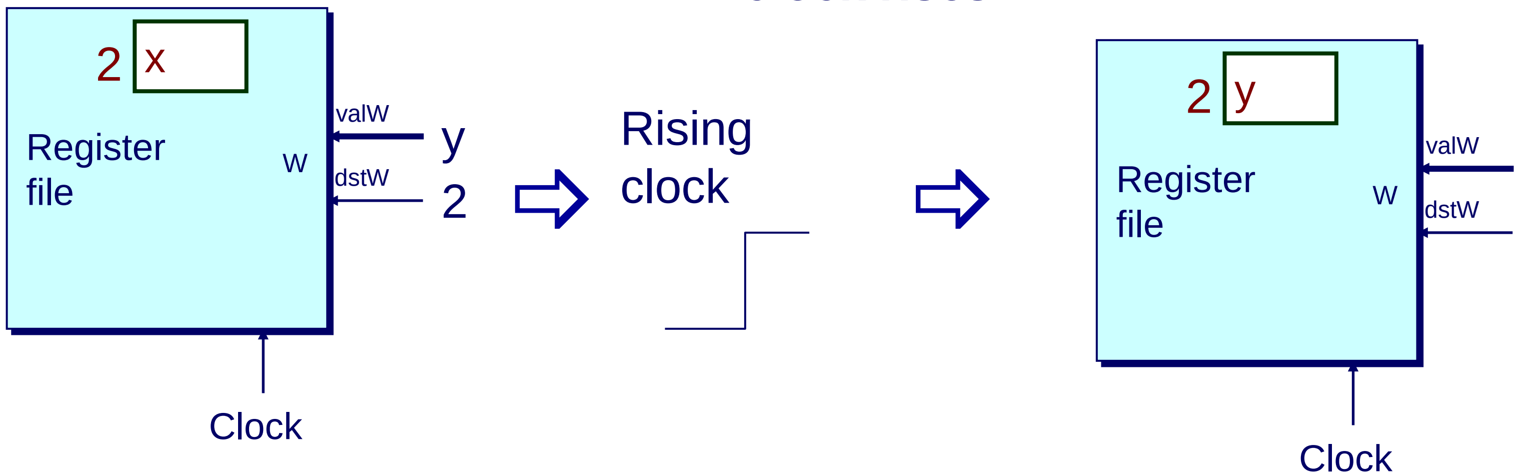


读 Reading

- 类似组合逻辑 Like combinational logic
- 根据输入地址产生输出数据 Output data generated based on input address
  - 一些时延后 After some delay

写 Writing

- 类似寄存器 Like register
- 仅在时钟上升沿更新 Update only as clock rises



硬件控制语言
Hardware Control Language

- 非常简单的硬件描述语言 Very simple hardware description language
- 仅可以表达有限的硬件操作 Can only express limited aspects of hardware operation
  - 我们想要探索和修改的部分 Parts we want to explore and modify

数据类型 Data Types

- `bool`: Boolean
  - `a`, `b`, `c`, …
- `int`: words
  - `A`, `B`, `C`, …
  - 不指定字长-字节还是64位字 Does not specify word size---bytes, 64-bit words, …

语句 Statements

- `bool a =` *`bool-expr`* `;`
- `int A =` *`int-expr`* `;`

HCL操作 HCL Operations

- 按照返回值类型来分类 Classify by type of value returned

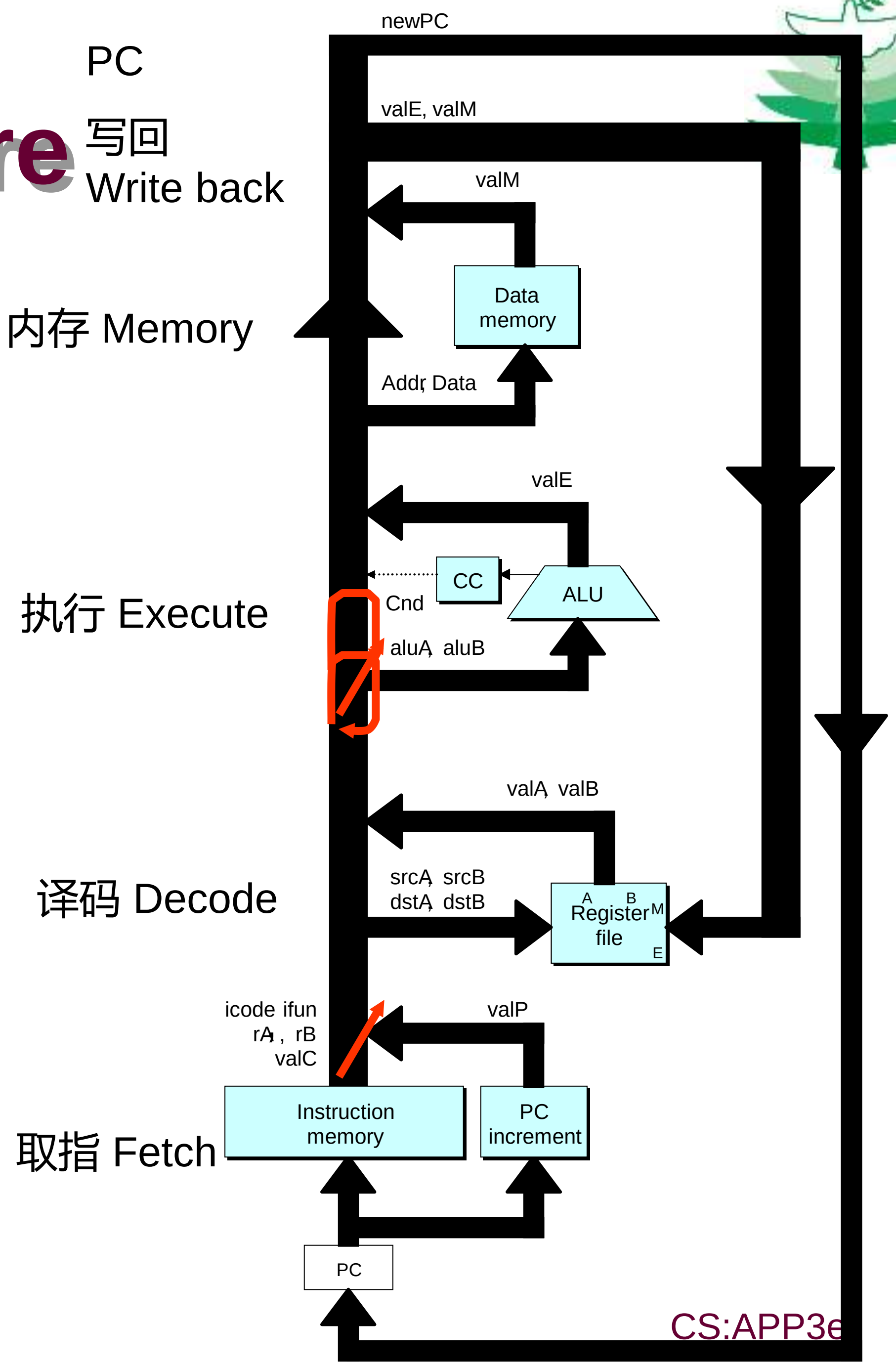
布尔表达式 Boolean Expressions

- 逻辑操作 Logic Operations
  - `a && b`, `a || b`, `!a`
- 字比较 Word Comparisons
  - `A == B`, `A != B`, `A < B`, `A <= B`, `A >= B`, `A > B`
- 集合成员关系 Set Membership
  - `A in { B, C, D }`
    - » 等同于 Same as `A == B || A == C || A == D`

字表达式 Word Expressions

- Case表达式 Case expressions
  - `[ a : A; b : B; c : C ]`
  - 按顺序评估测试表达式 Evaluate test expressions `a`, `b`, `c`, … in sequence
  - 返回第一个成功测试的字表达式 Return word expression `A`, `B`, `C`, … for first successful test

顺序硬件结构 SEQ Hardware Structure



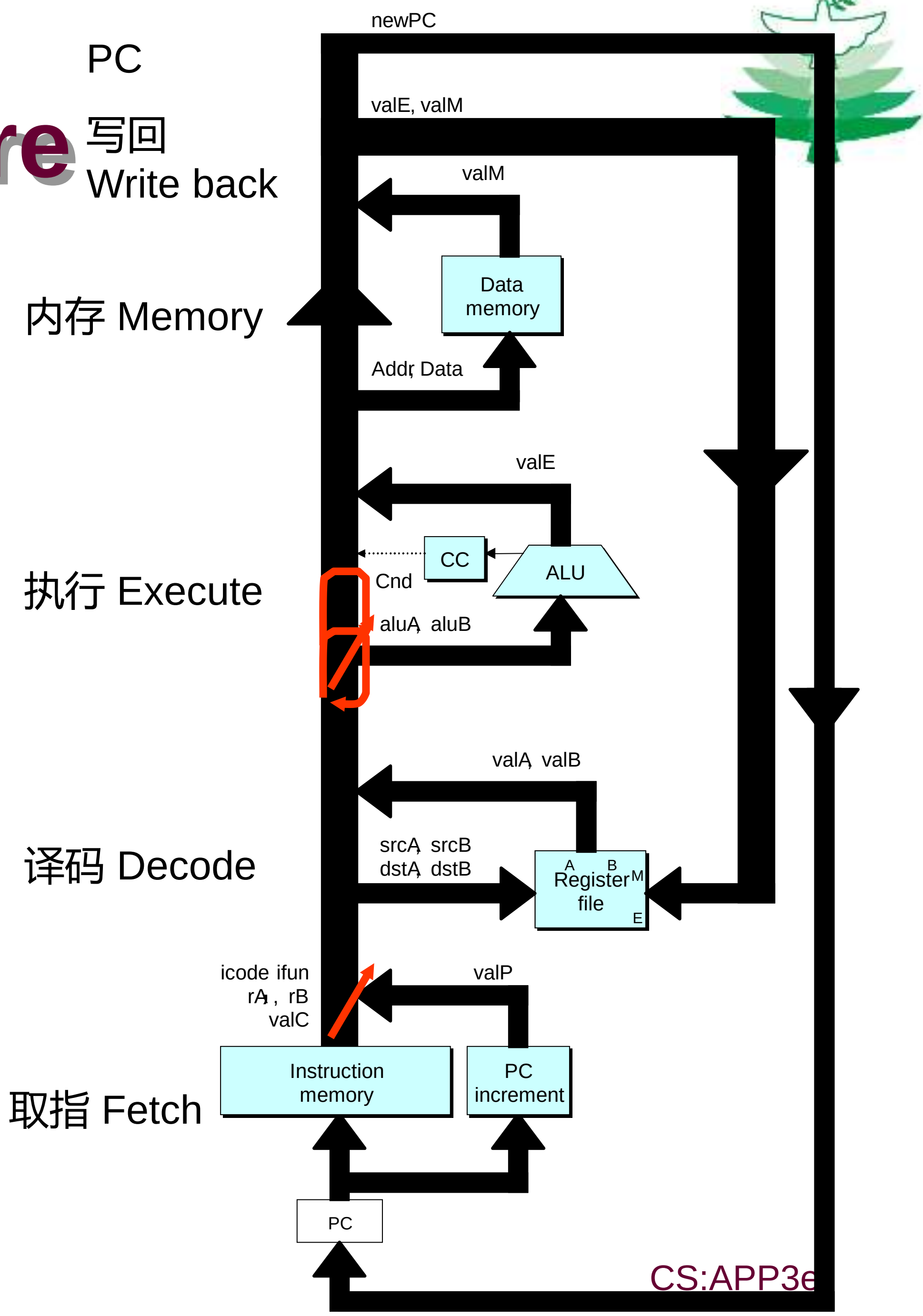
状态 State

- 程序计数器寄存器 Program counter register (PC)
- 条件码寄存器 Condition code register (CC)
- 寄存器文件（堆） Register File
- 内存 Memories
  - 访问同样的内存空间 Access same memory space
  - 数据：读/写程序数据 Data: for reading/writing program data
  - 指令：读指令 Instruction: for reading instructions

顺序硬件结构 SEQ Hardware Structure

指令流 Instruction Flow

- 读PC指定地址处的指令 Read instruction at address specified by PC
- 分成阶段处理 Process through stages
- 更新程序计数器 Update program counter



CS:APP3e

顺序阶段 SEQ Stages

取指 Fetch

- 从指令内存读指令 Read instruction from instruction memory

译码 Decode

- 读程序寄存器 Read program registers

执行 Execute

- 计算值或地址 Compute value or address

内存 Memory

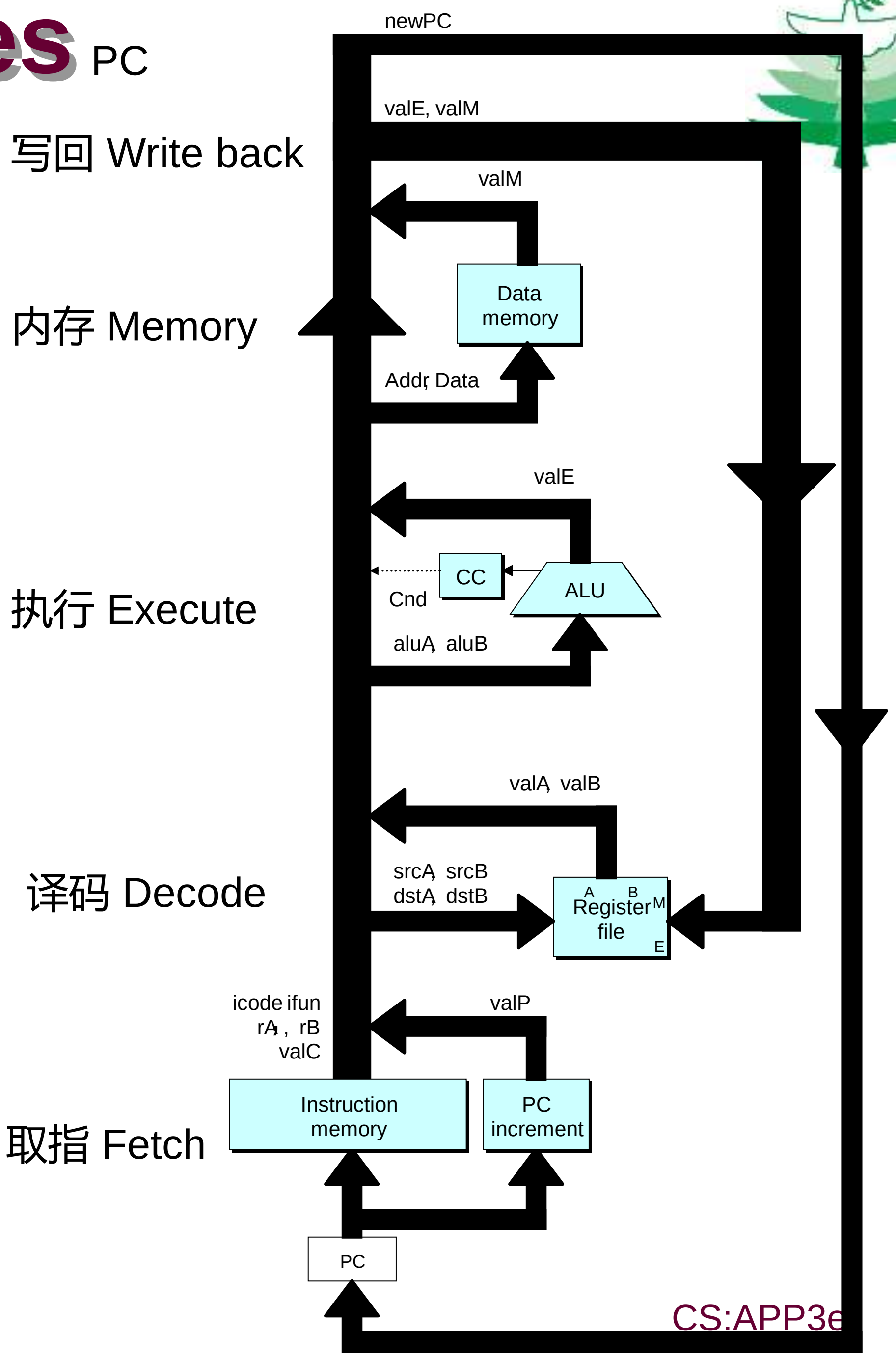
- 读或写数据 Read or write data

写回 Write Back

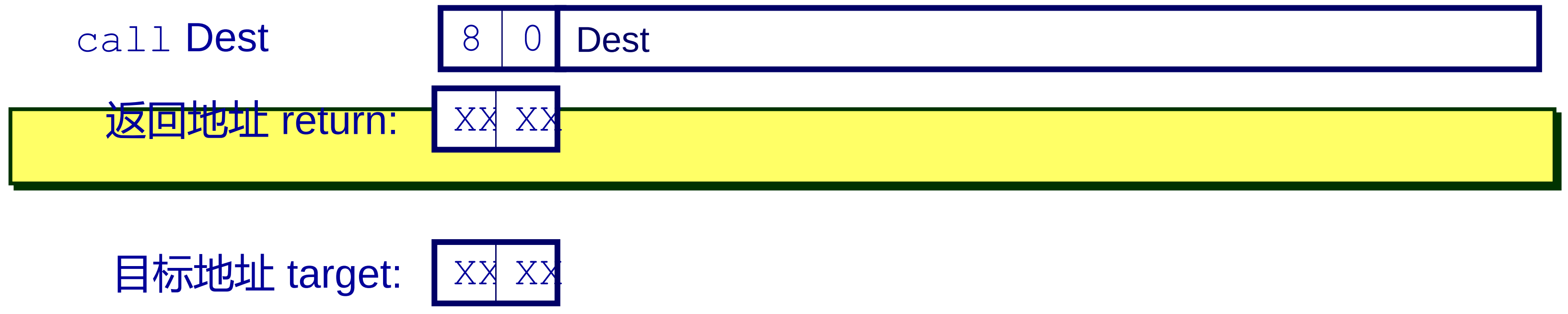
- 写程序寄存器 Write program registers

PC

- 更新程序计数器 Update program counter



执行过程调用 Executing `call`



取指 Fetch

- 读9个字节 Read 9 bytes
- PC增加9 Increment PC by 9

译码 Decode

- 读栈指针 Read stack pointer

执行 Execute

- 栈指针减8 Decrement stack pointer by 8

内存 Memory

- 将增加后PC写到栈指针新值 Write incremented PC to new value of stack pointer

写回 Write back

- 更新栈指针 Update stack pointer

PC更新 PC Update

- 设置PC为目标地址 Set PC to Dest

每个阶段的计算：过程调用 Stage Computation: `call`

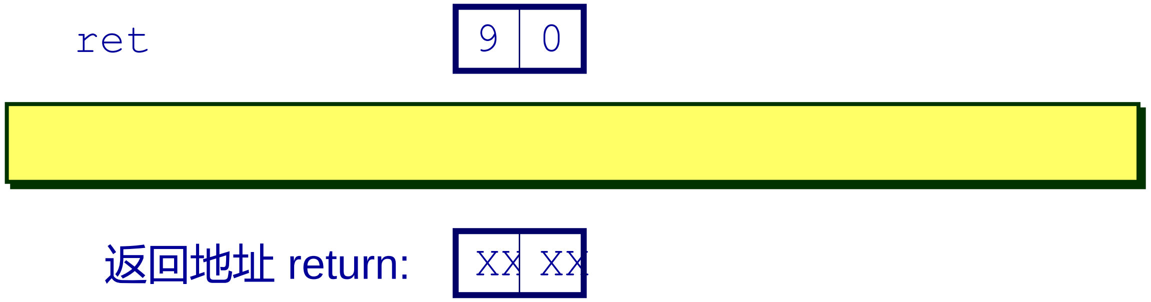
	`call` Dest	

Fetch	icode:ifun ← M_1[PC]
	
	valC ← M_8[PC+1]
	valP ← PC+9
		Read instruction byte
		
		Read destination address
		Compute return point

Decode	 valB ← R[`%rsp`]	 Read stack pointer
Execute	valE ← valB + –8	Decrement stack pointer
Memory	M_8[valE] ← valP	Write return value on stack
Write back	R[`%rsp`] ← valE	Update stack pointer
PC update	PC ← valC	Set PC to destination

- **使用ALU减栈指针 Use ALU to decrement stack pointer**
- **存储增加后的PC Store incremented PC**

执行过程返回 Executing `ret`



取指 Fetch

- 读1个字节 Read 1 byte

译码 Decode

- 读栈指针 Read stack pointer

执行 Execute

- 栈指针增加8 Increment stack pointer by 8

内存 Memory

- 从老的栈指针读返回地址 Read return address from old stack pointer

写回 Write back

- 更新栈指针 Update stack pointer

PC更新 PC Update

- 设置PC为返回地址 Set PC to return address

每个阶段的计算：过程返回 Stage Computation: `ret`

	`ret`	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte

Decode	valA ← R[`%rsp`]
	valB ← R[`%rsp`]
		Read operand stack pointer
		Read operand stack pointer

Execute	valE ← valB + 8	Increment stack pointer
Memory	valM ← M_8[valA]	Read return address
Write back	R[`%rsp`] ← valE	Update stack pointer
PC update	PC ← valM	Set PC to return address

- **使用ALU增加栈指针 Use ALU to increment stack pointer**
- **从内存读返回地址 Read return address from memory**

计算步骤 Computation Steps

		OPq rA, rB	
Fetch	icode,ifun	icode:ifun ← M_1[PC]	Read instruction byte
	rA,rB	rA:rB ← M_1[PC+1]	Read register byte
	valC		[Read constant word]
	valP	valP ← PC+2	Compute next PC
Decode	valA, srcA	valA ← R[rA]	Read operand A
	valB, srcB	valB ← R[rB]	Read operand B
Execute	valE	valE ← valB OP valA	Perform ALU operation
	Cond code	Set CC	Set/use cond. code reg
Memory	valM		[Memory read/write]
Write back	dstE	R[rB] ← valE	Write back ALU result
	dstM		[Write back memory result]
PC update	PC	PC ← valP	Update PC

- **所有指令遵循同样的通用模式 All instructions follow same general pattern**
- **差别在于每步计算什么 Differ in what gets computed on each step**

计算步骤 Computation Steps

		`call Dest`	
Fetch	icode,ifun	icode:ifun ← M_1[PC]	Read instruction byte
	rA,rB		[Read register byte]
	valC	valC ← M_8[PC+1]	Read constant word
	valP	valP ← PC+9	Compute next PC
Decode	valA, srcA		[Read operand A]
	valB, srcB	valB ← R[`%rsp`]	Read operand B
Execute	valE	valE ← valB + –8	Perform ALU operation
	Cond code		[Set /use cond. code reg]
Memory	valM	M_8[valE] ← valP	Memory read/write
Write back	dstE	R[`%rsp`] ← valE	Write back ALU result
	dstM		[Write back memory result]
PC update	PC	PC ← valC	Update PC

- **所有指令遵循同样的通用模式 All instructions follow same general pattern**
- **差别在于每步计算什么 Differ in what gets computed on each step**

计算的值 Computed Values

取指 Fetch

icode	Instruction code
ifun	Instruction function
rA	Instr. Register A
rB	Instr. Register B
valC	Instruction constant
valP	Incremented PC

译码 Decode

srcA	Register ID A
srcB	Register ID B
dstE	Destination Register E
dstM	Destination Register M
valA	Register value A
valB	Register value B

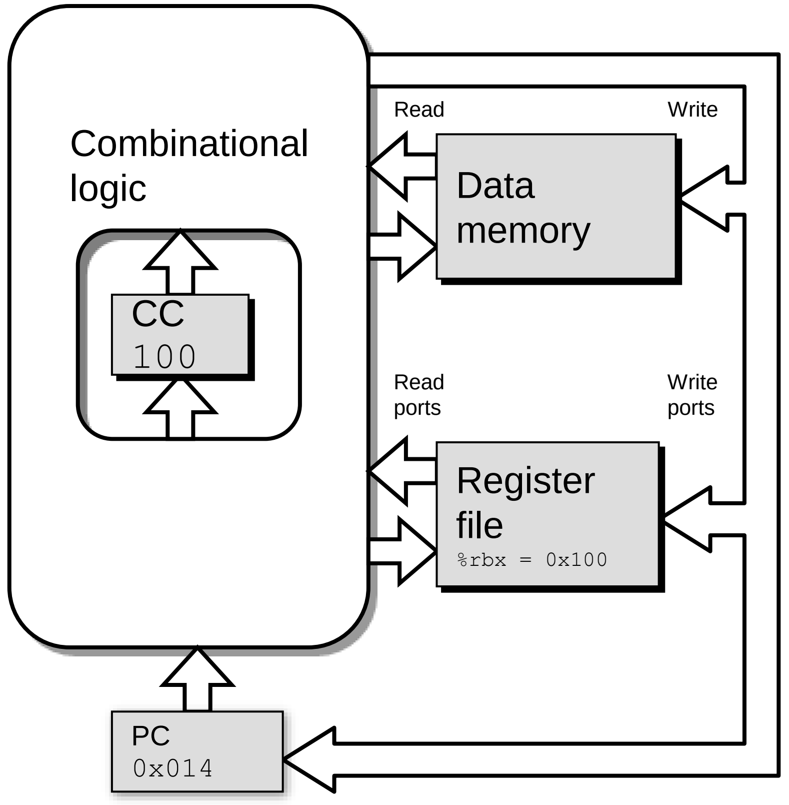
执行 Execute

- valE ALU result
- Cnd Branch/move flag

内存 Memory

- valM Value from memory

顺序处理器操作 SEQ Operation

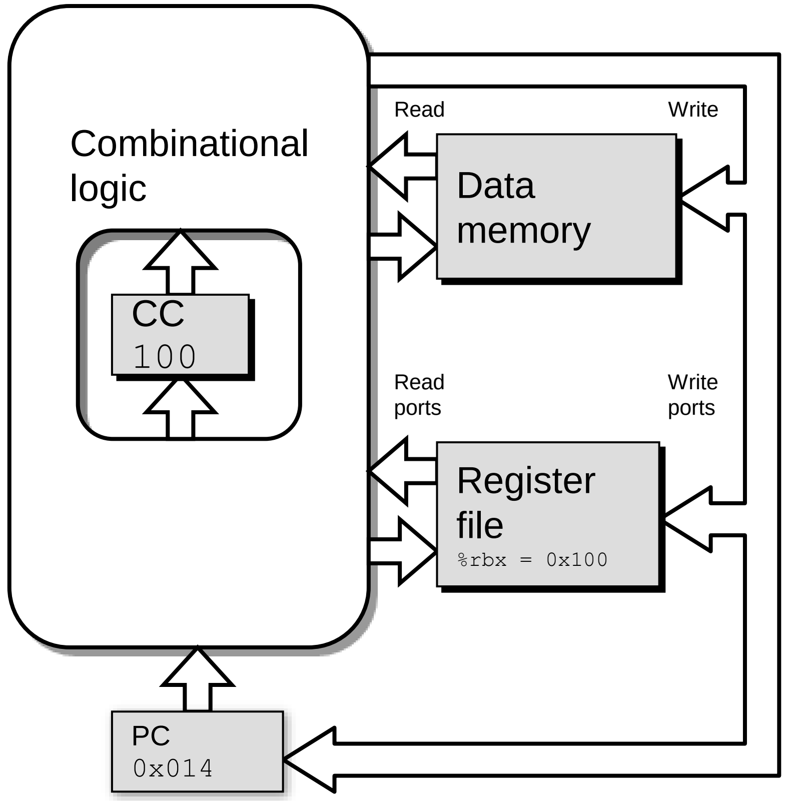


状态 State

- PC寄存器 PC register
- 条件码寄存器 Cond. Code register
- 数据内存 Data memory
- 寄存器文件（堆） Register file

所有更新在时钟上升沿 All updated as clock rises

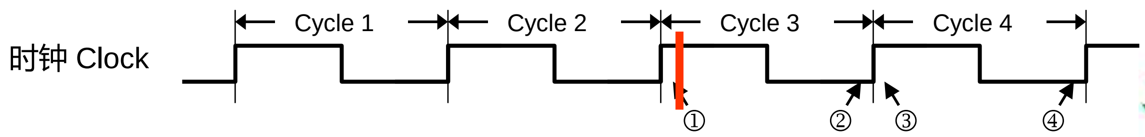
顺序处理器操作 SEQ Operation



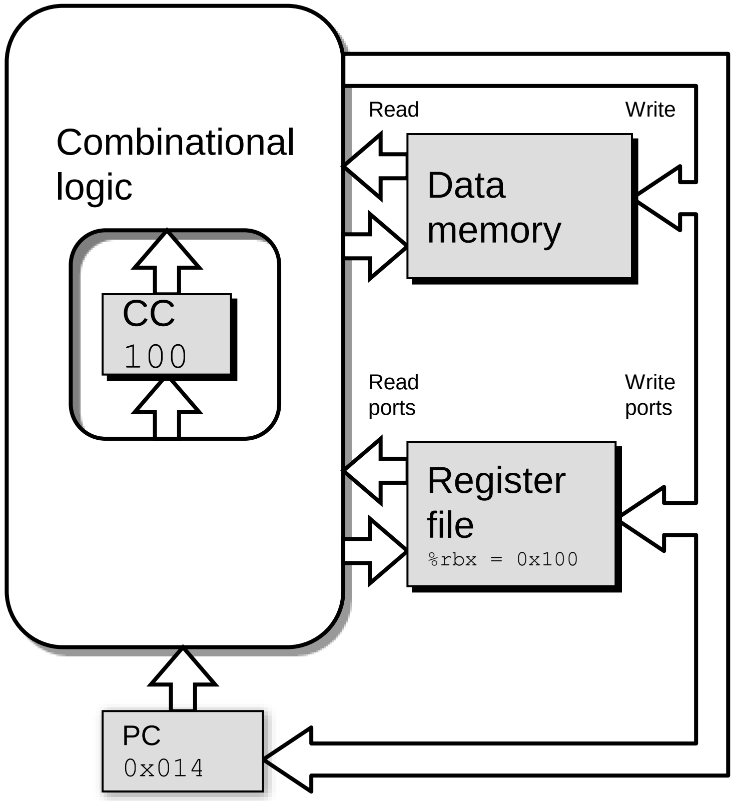
组合逻辑 Combinational Logic

- 算术/逻辑单元 ALU
- 控制逻辑 Control logic
- 内存读 Memory reads
  - 指令内存 Instruction memory
  - 寄存器文件（堆） Register file
  - 数据内存 Data memory

SEQ操作 SEQ Operation #2



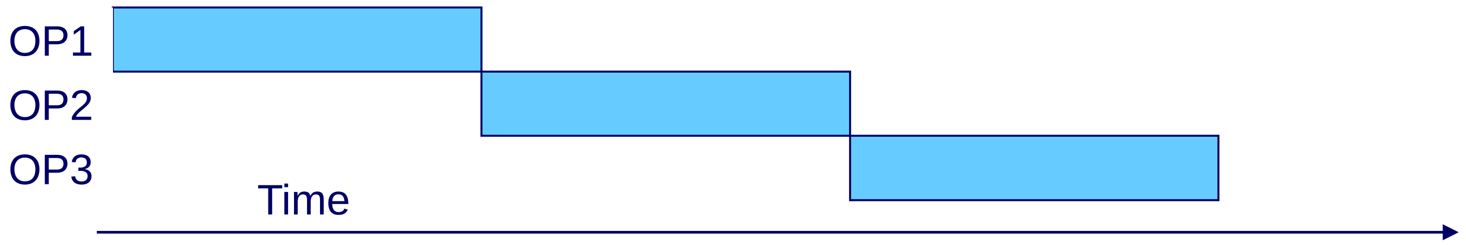
周期1 Cycle 1:	`0x000:    irmovq $0x100,%rbx   # %rbx <-- 0x100`
周期2 Cycle 2:	`0x00a:    irmovq $0x200,%rdx   # %rdx <-- 0x200`
周期3 Cycle 3:	`0x014:    addq %rdx,%rbx       # %rbx <-- 0x300 CC <-- 000`
周期4 Cycle 4:	`0x016:    je dest              # Not taken`
周期5 Cycle 5:	`0x01f:    rmmovq %rbx,0(%rdx)  # M[0x200] <-- 0x300`



- 状态设置按照第二条irmovq指令 state set according to second `irmovq` instruction
- 组合逻辑开始反应状态改变 combinational logic starting to react to state changes

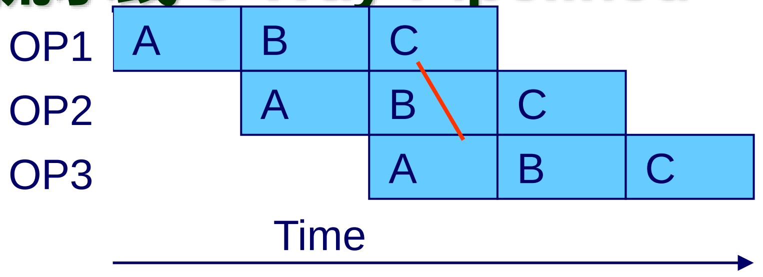
流水线图 Pipeline Diagrams

非流水线 Unpipelined



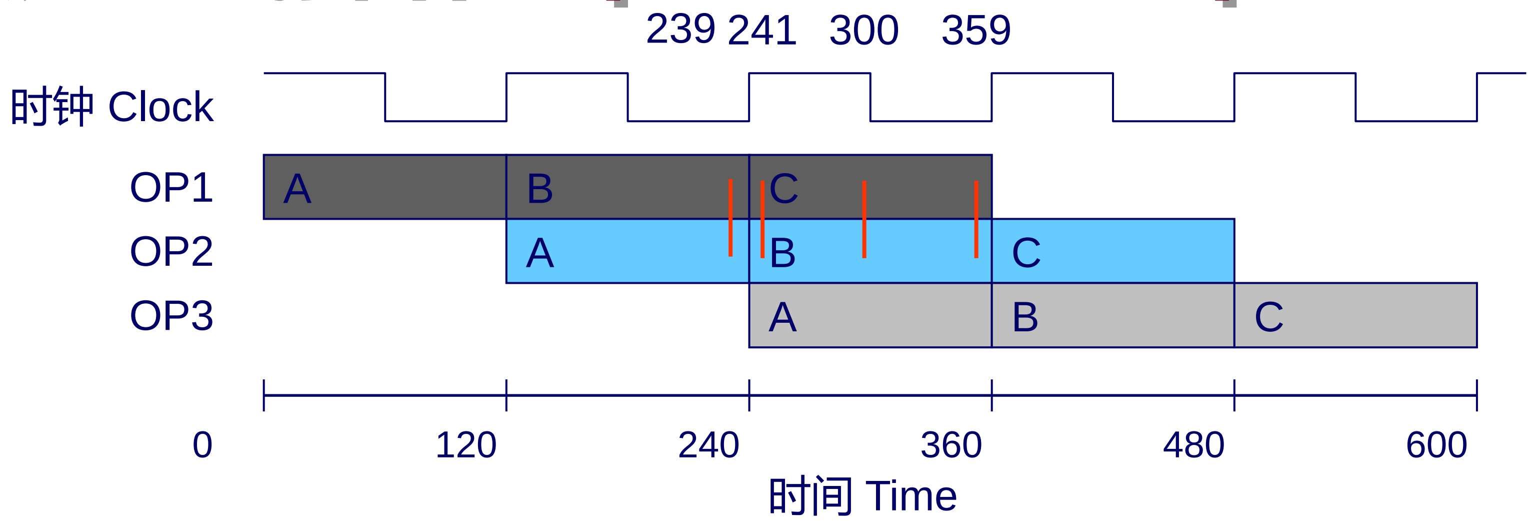
- 在上一个操作完成前不能开始新的操作 Cannot start new operation until previous one completes

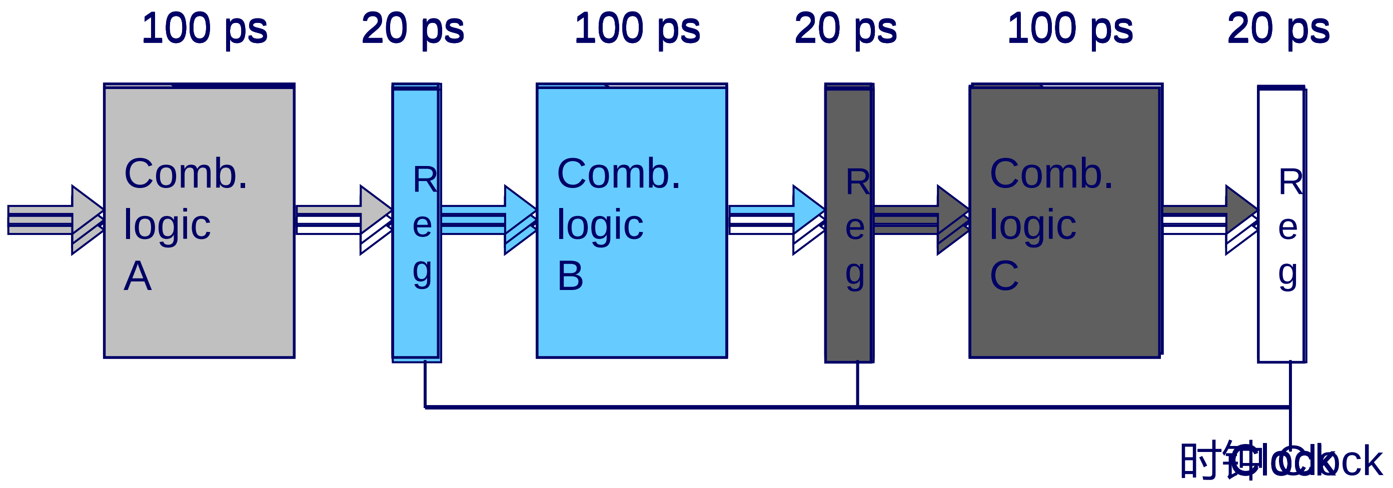
3级流水线 3-Way Pipelined



- 最多3个操作在同时处理 Up to 3 operations in process simultaneously

流水线操作 Operating a Pipeline





增加流水线寄存器 Adding Pipeline Registers

PC

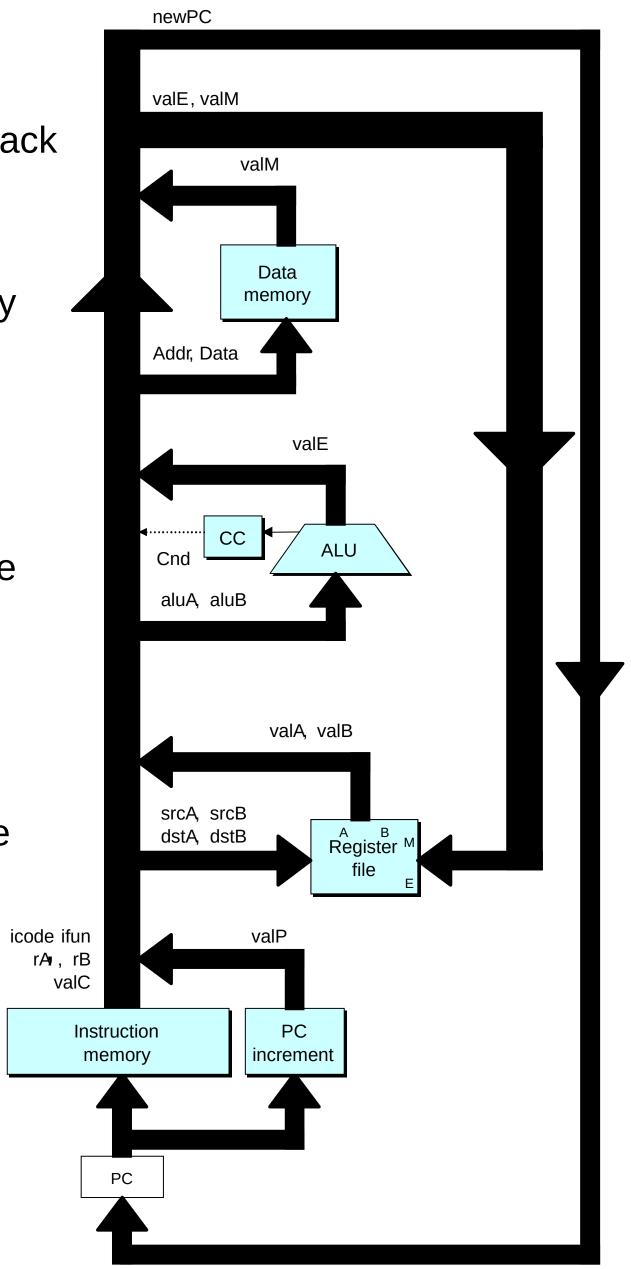
写回 Write back

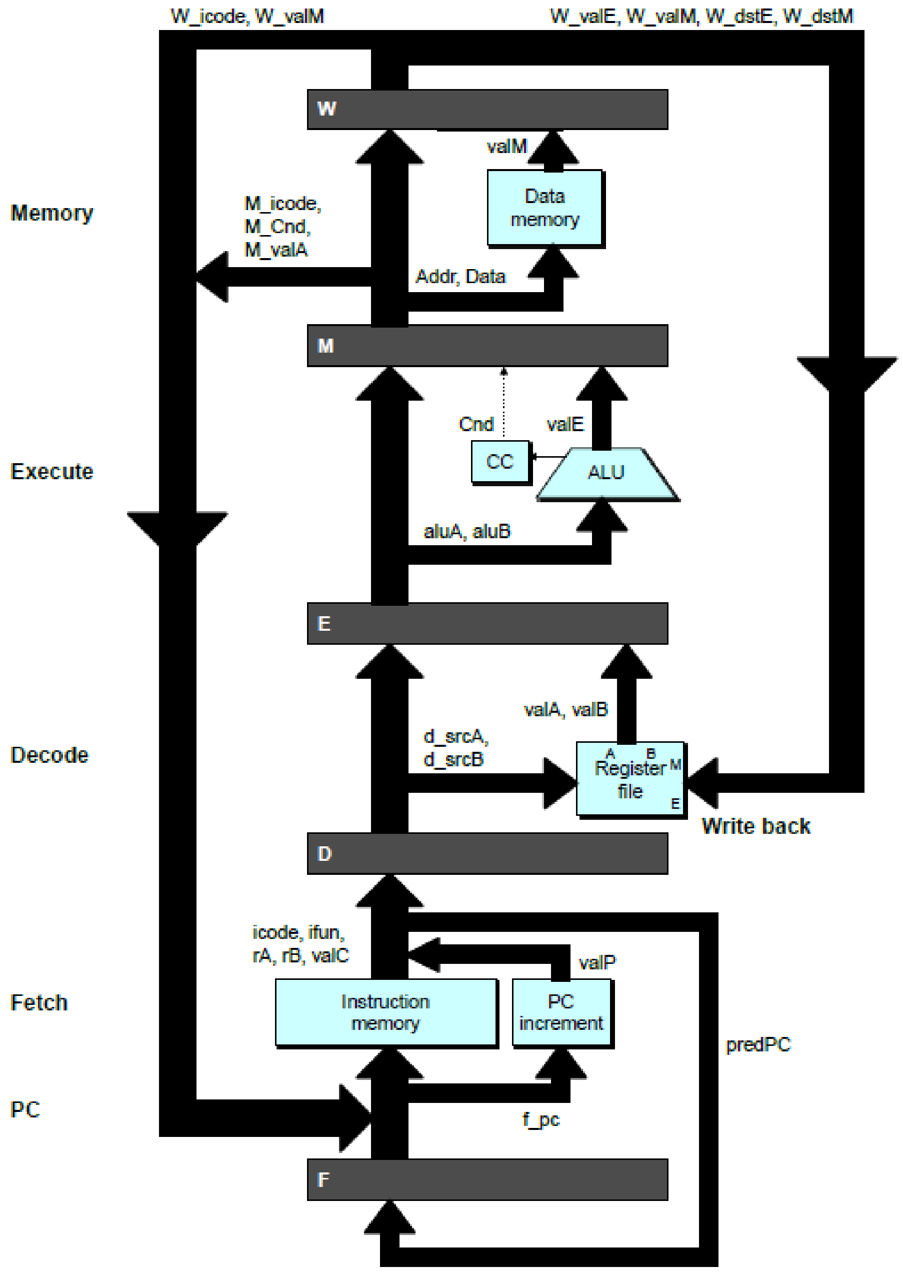
内存 Memory

执行 Execute

译码 Decode

取指 Fetch





流水线阶段 Pipeline Stages

取指 Fetch

- 选择当前PC Select current PC
- 读指令 Read instruction
- 计算PC增加值 Compute incremen PC

译码 Decode

- 读程序寄存器 Read program regis

执行 Execute

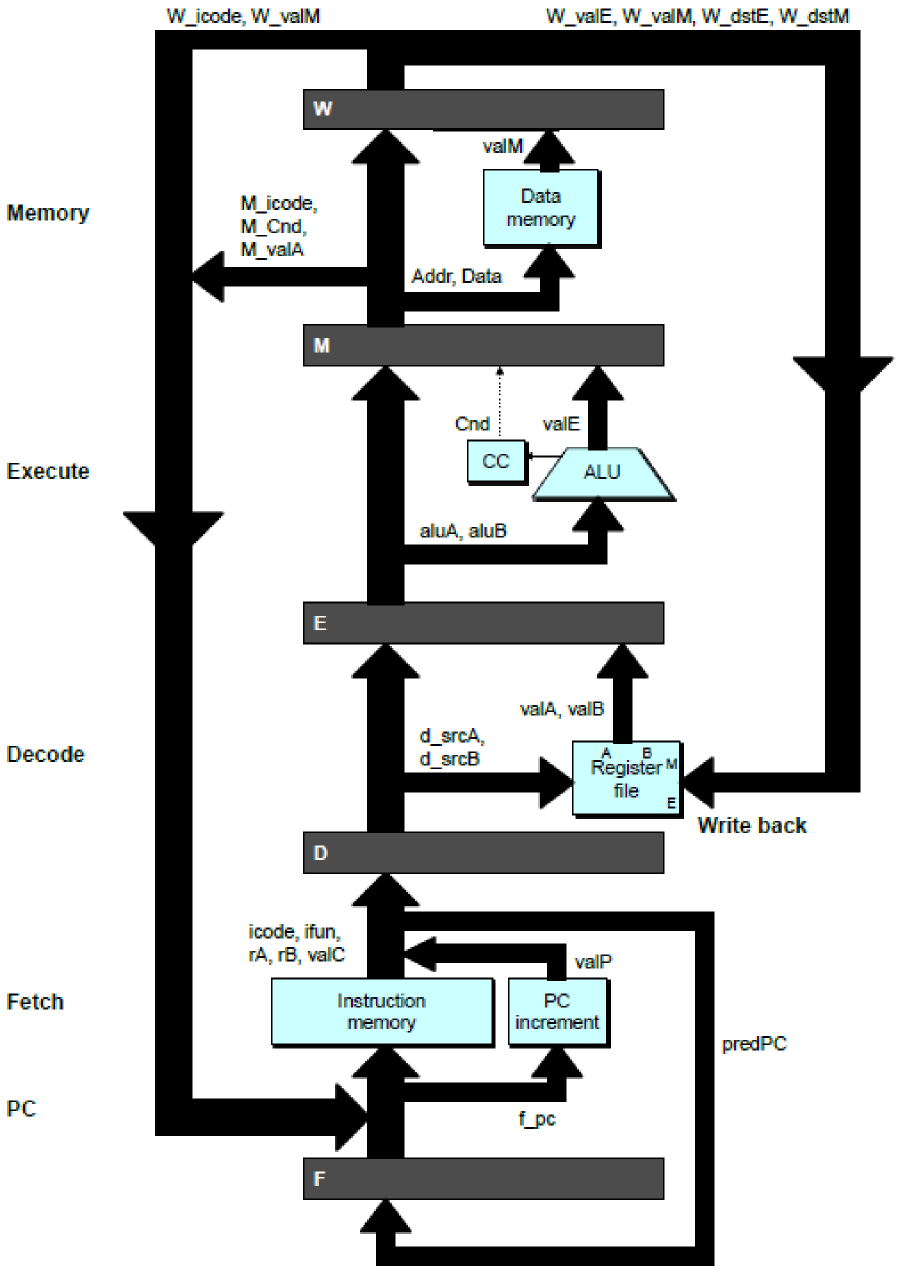
- 操作ALU Operate ALU

内存 Memory

- 读或写数据内存 Read or write data memory

写回 Write Back

- 更新寄存器文件（堆） Update register

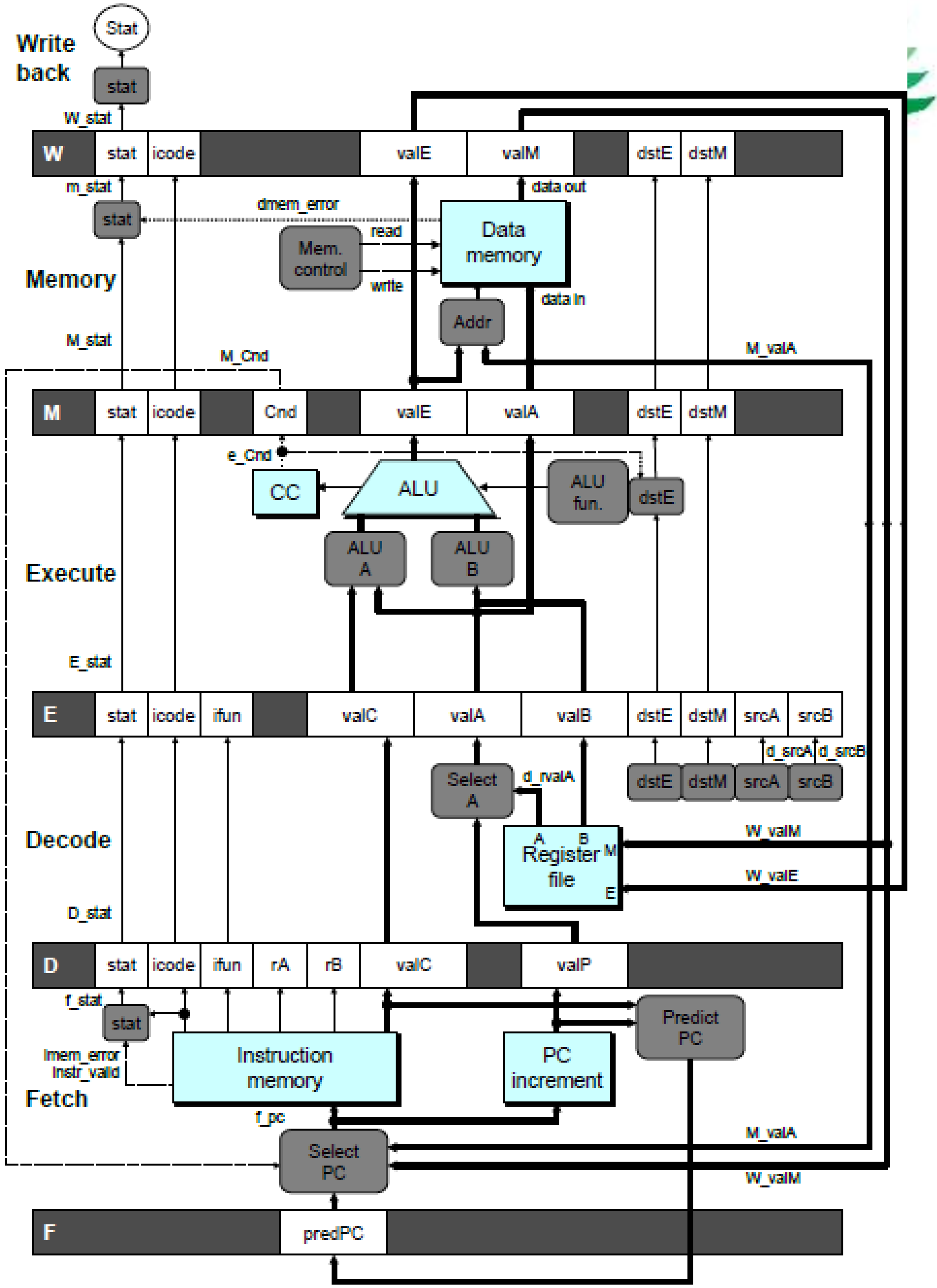


PIPE-硬件 PIPE- Hardware

- 流水线寄存器存储指令执行中的中间值 Pipeline registers hold intermediate values from instruction execution

转发（向前）路径 Forward (Upward) Paths

- 从一个阶段到下一个阶段传递的值 Values passed from one stage to next
- 不能回跳到过去的阶段 Cannot jump past stages
  - 例如valC直传通过译码阶段 e.g., valC passes through decode



概述 Overview

使流水线处理器工作 Make the pipelined processor work!

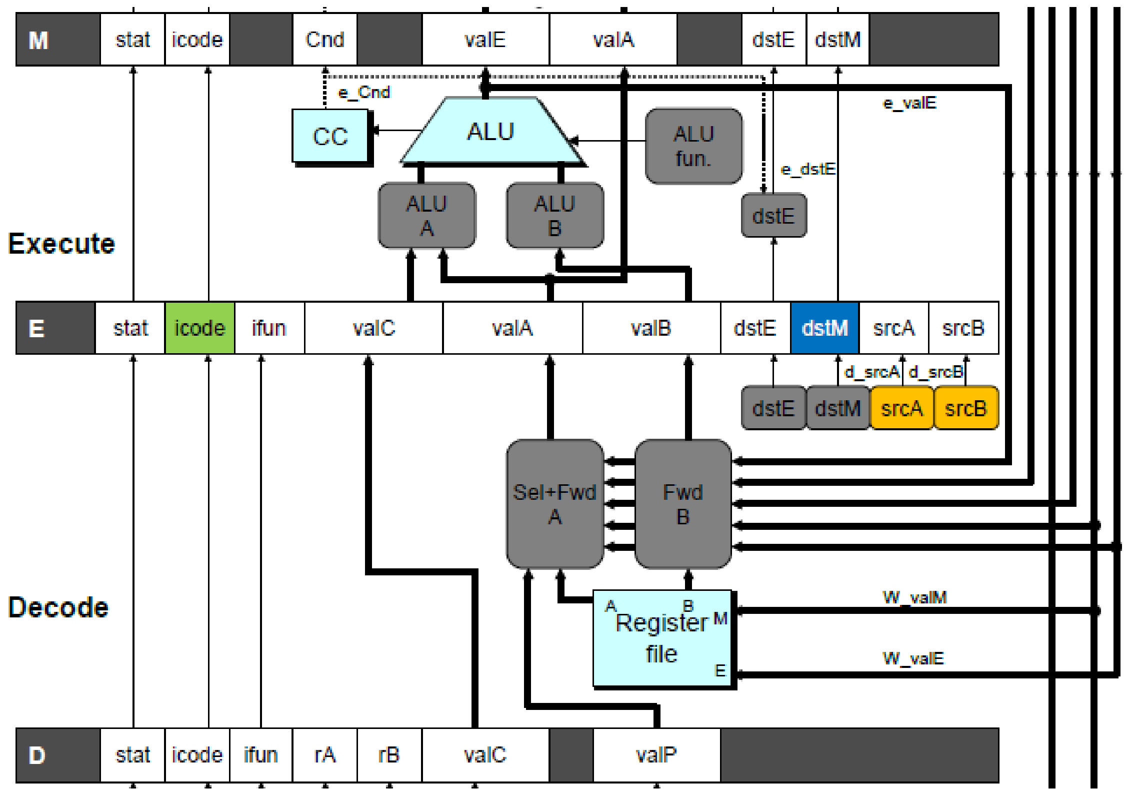
数据冒险 Data Hazards

- 以寄存器 R 作为源的指令紧跟在以寄存器 R 作为目的的指令之后 Instruction having register R as source follows shortly after instruction having register R as destination
- 常见情况，不想减慢流水线 Common condition, don’t want to slow down pipeline

控制冒险 Control Hazards

- 错误预测条件分支 Mispredict conditional branch
  - 我们的设计预测所有分支为选择分支 Our design predicts all branches as being taken
  - 朴素流水线执行两条额外指令 Naïve pipeline executes two extra instructions
- 获得ret指令的返回地址 Getting return address for `ret` instruction
  - 朴素流水线执行三条额外指令 Naïve pipeline executes three extra instructions

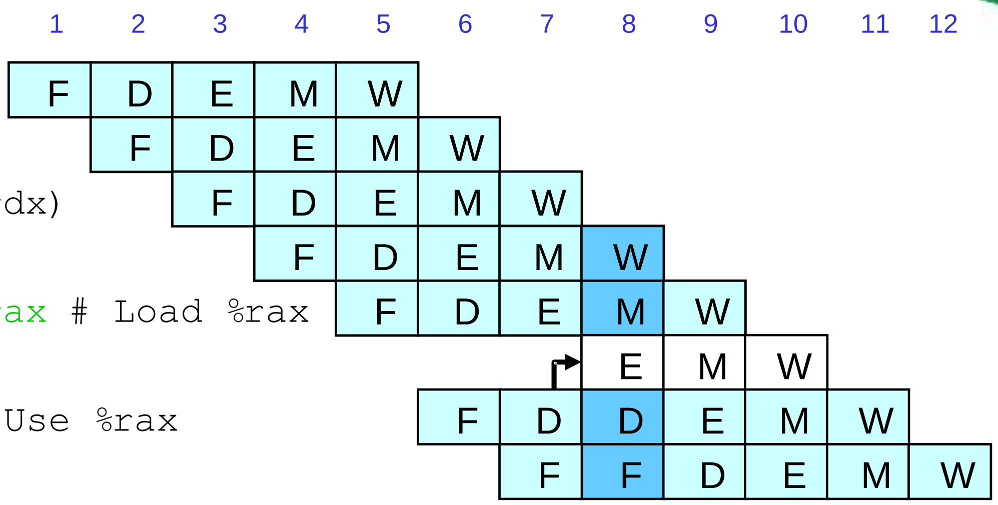
检测装载/使用冒险 Detecting Load/Use Hazard



状况 Condition	触发 Trigger
装载/使用冒险 Load/Use Hazard	`E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB }`

装载/使用冒险控制 Control for Load/Use Hazard

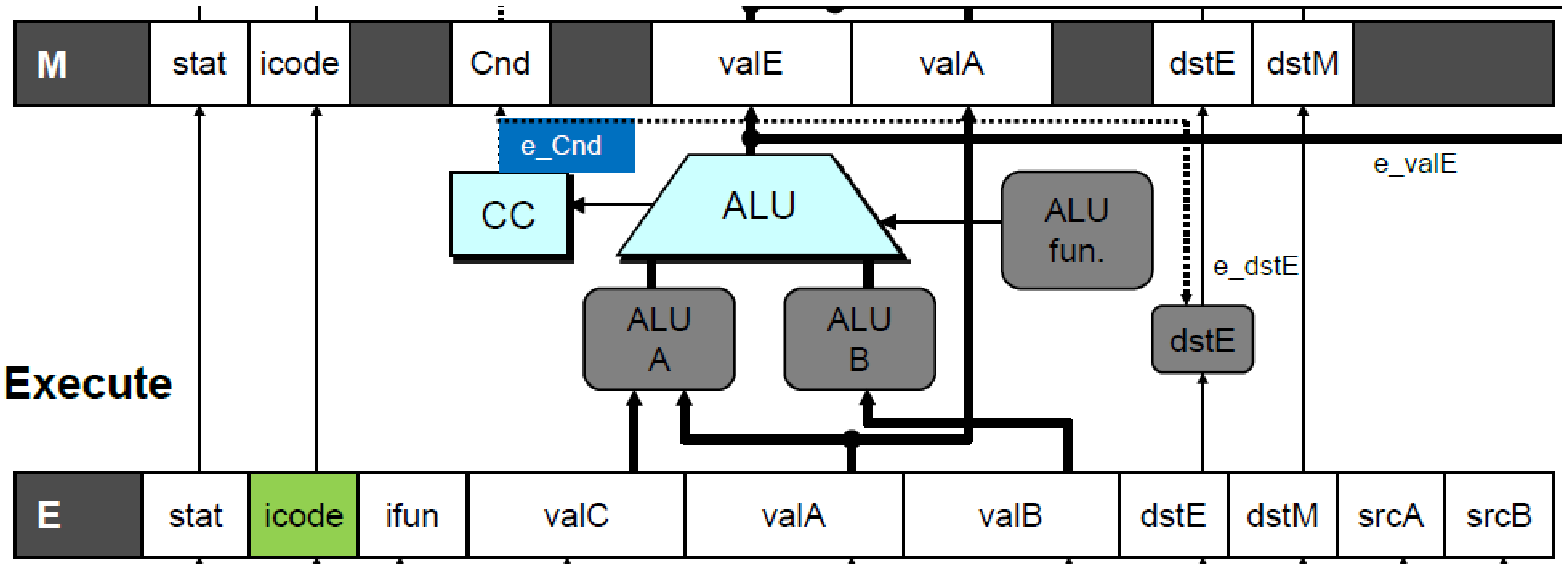
```
# demo-luh.ys

0x000:  irmovq $128,%rdx
 0x00a: irmovq  $3,%rcx
 0x014: rmmovq %rcx,  0(%rdx)
 0x01e: irmovq $10,%ebx
 0x028: mrmovq 0(%rdx),%rax # Load %rax
              bubble
0x032: addq %ebx,%rax # Use %rax
0x034: halt
```



- **暂停指令在取指和译码阶段 Stall instructions in fetch and decode stages**
- **注入气泡到执行阶段 Inject bubble into execute stage**

状况 Condition	F	D	E	M	W
装载/使用冒险 Load/Use Hazard	暂停 stall	暂停 stall	气泡 bubble	正常 normal	正常 normal

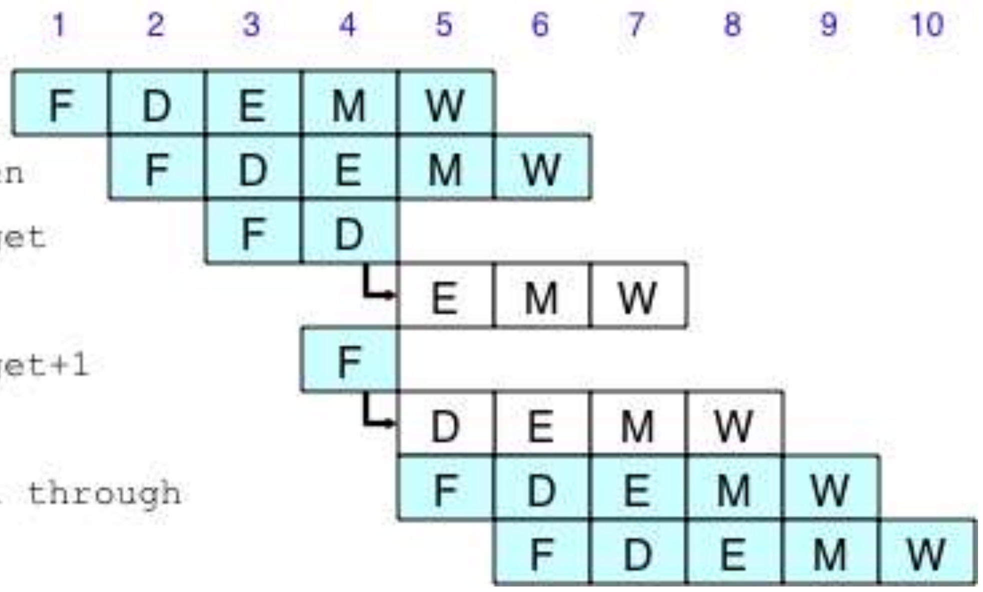
检测分支预测错误 Detecting Mispredicted Branch



状况 Condition	触发 Trigger
分支预测错误 Mispredicted Branch	`E_icode = IJXX & !e_Cnd`

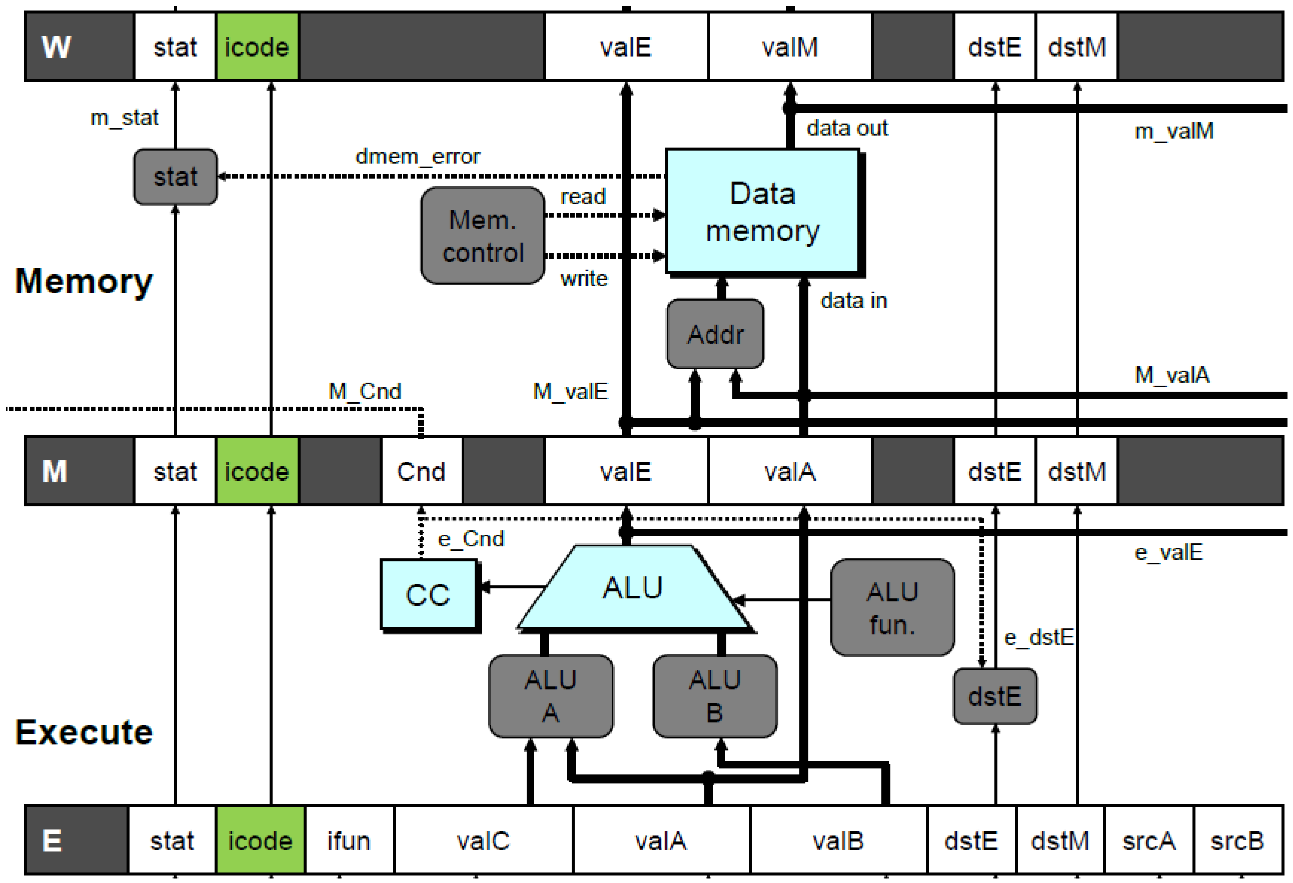
预测错误控制 Control for Misprediction

```
# demo-j.ys
0x000: xorq %rax,%rax
0x002: jne target # Not taken
0x016: irmovq $2,%rdx # Target
       bubble
0x020: irmovq $3,%rbx # Target+1
       bubble
0x00b: irmovq $1,%rax # Fall through
0x015: halt
```



状况 Condition	F	D	E	M	W
分支预测错误 Mispredicted Branch	正常 normal	气泡 bubble	气泡 bubble	正常 normal	正常 normal

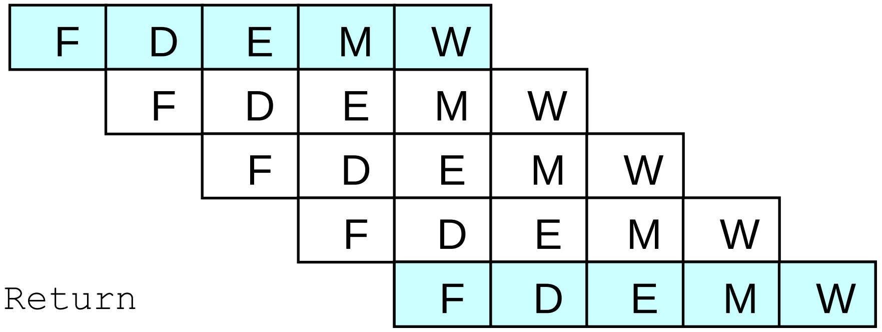
检测返回 Detecting Return



状况 Condition	触发 Trigger
处理ret Processing `ret`	`IRET in { D_icode, E_icode, M_icode }`

返回控制 Control for Return

```
# demo-retb
0x026:     ret
           bubble
           bubble
           bubble
  0x014:    irmovq $5,%rsi # Return
```



状况 Condition	F	D	E	M	W
处理ret Processing `ret`	暂停 stall	气泡 bubble	正常 normal	正常 normal	正常 normal

特殊控制情况 Special Control Cases

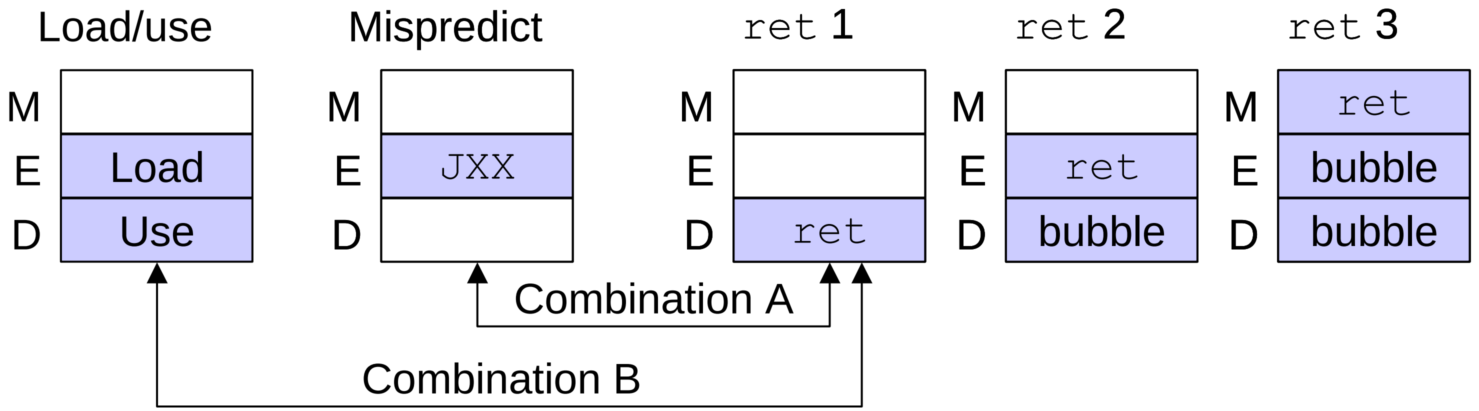
检测 Detection

状况 Condition	触发 Trigger
处理ret Processing `ret`	IRET in { D_icode, E_icode, M_icode }
装载/使用冒险 Load/Use Hazard	E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB }
分支预测错误 Mispredicted Branch	E_icode = IJXX & !e_Cnd

动作（下一个周期） Action (on next cycle)

Condition	F	D	E	M	W
处理ret Processing `ret`	暂停 stall	气泡 bubble	正常 normal	正常 normal	正常 normal
装载/使用冒险 Load/Use Hazard	暂停 stall	暂停 stall	气泡 bubble	正常 normal	正常 normal
分支预测错误 Mispredicted Branch	正常 normal	气泡 bubble	气泡 bubble	正常 normal	正常 normal

控制组合 Control Combinations



- 在同一个时钟周期可能引起的特殊情况 Special cases that can arise on same clock cycle

组合A Combination A

- 分支不选择 Not-taken branch
- Ret指令在分支目标处 `ret` instruction at branch target

组合B Combination B

- 指令从内存读到%rsp Instruction that reads from memory to `%rsp`
- 后跟ret指令 Followed by `ret` instruction

状态改变的控制逻辑 Control Logic for State Changes

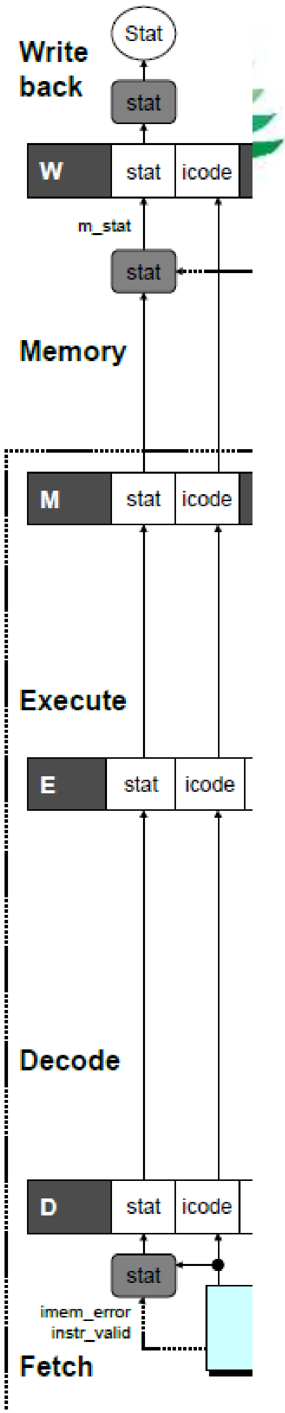
设置条件码 Setting Condition Codes

```
# Should the condition codes be updated?
bool set_cc = E_icode == IOPQ &&
        # State changes only during normal operation
        !m_stat in { SADR, SINS, SHLT }
        && !W_stat in { SADR, SINS, SHLT };
```

阶段控制 Stage Control

- 也控制内存更新 Also controls updating of memory

```
# Start injecting bubbles as soon as exception passes
through memory stage
bool M_bubble = m_stat in { SADR, SINS, SHLT }
        || W_stat in { SADR, SINS, SHLT };

# Stall pipeline register W when exception encountered
bool W_stall = W_stat in { SADR, SINS, SHLT };
```



流水线处理器的CPI CPI for PIPE

CPI ≈ 1.0

- 每个时钟周期取一条指令Fetch instruction each clock cycle
- 几乎每个周期有效处理一条新指令 Effectively process new instruction almost every cycle
  - 尽管每条单独指令都有5个周期的时延 Although each individual instruction has latency of 5 cycles

CPI > 1.0

- 有时必须暂停或取消分支 Sometimes must stall or cancel branches

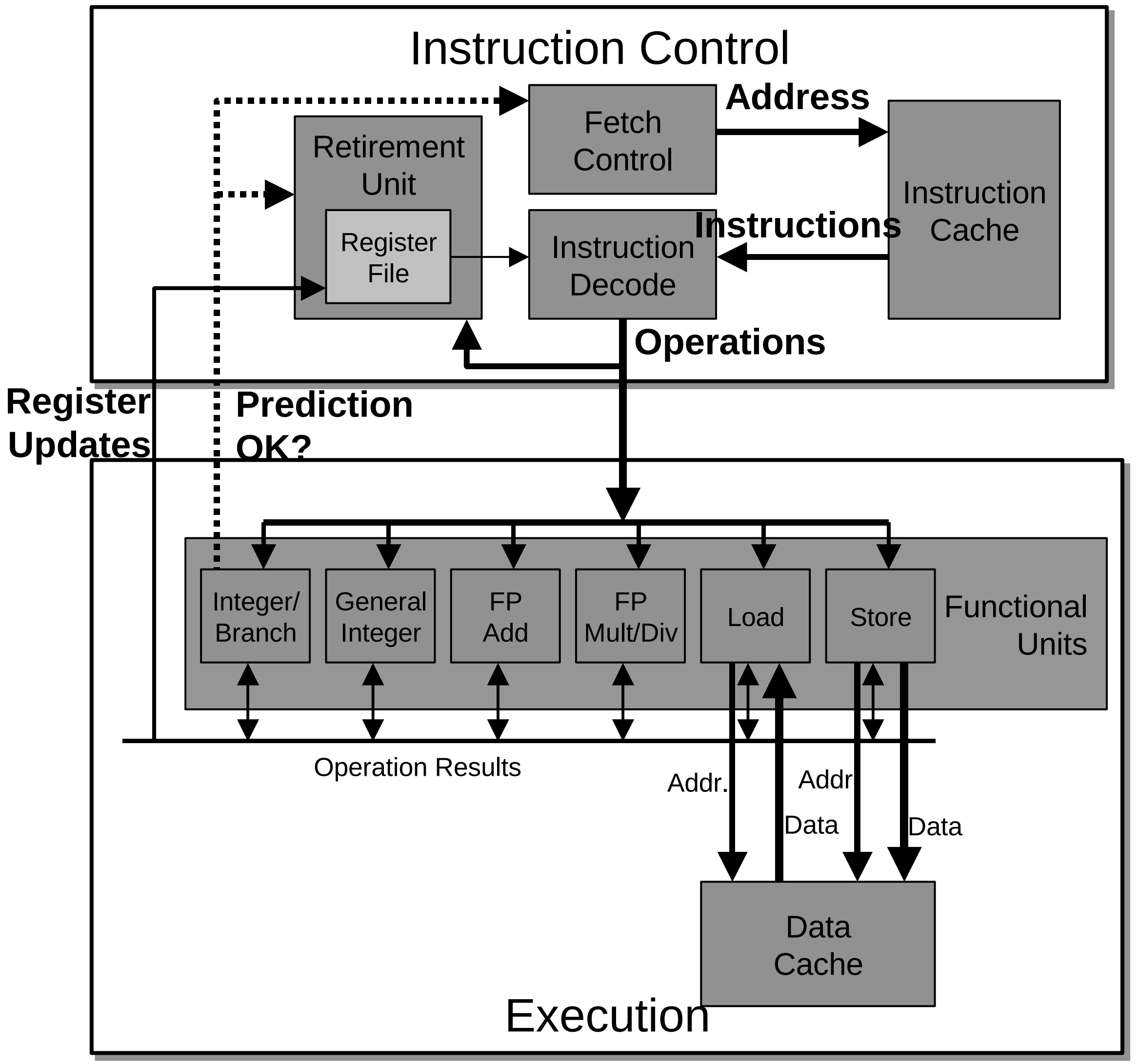
计算CPI Computing CPI

- C是时钟周期数 C clock cycles
- I是执行完成的指令数 I instructions executed to completion
- B是注入的气泡气泡数 B bubbles injected (C = I + B)

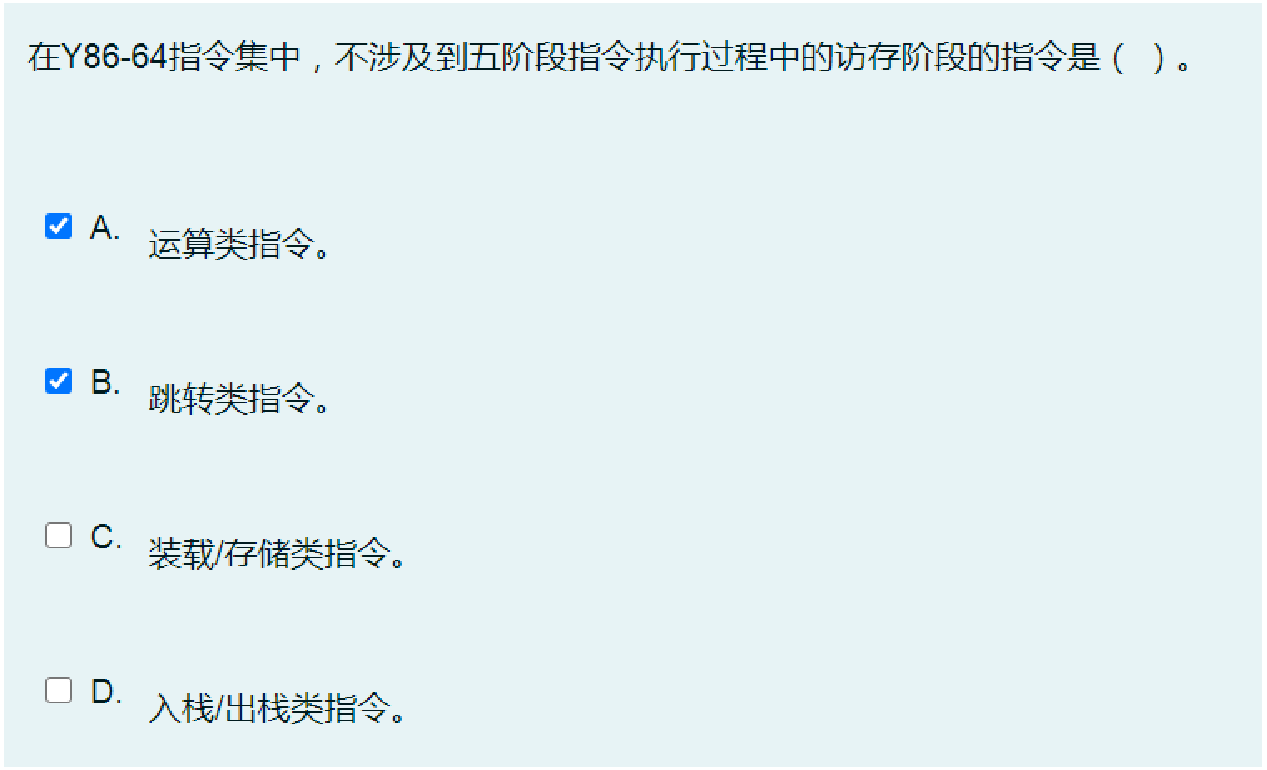
$$CPI = C/I = (I+B)/I = 1.0 + B/I$$

- 因子B/I表示气泡的平均惩罚 Factor B/I represents average penalty due to bubbles

现代CPU设计 Modern CPU Design



例题



在Y86-64指令集中，不涉及到五阶段指令执行过程中的访存阶段的指令是（ ）。

[✓ A. 运算类指令。]

[✓ B. 跳转类指令。]

☐ C. 装载/存储类指令。

☐ D. 入栈/出栈类指令。

优化程序性能

- 编译器的优化与局限性
  - 代码外提/频度削弱、强度削弱、公共子表达式删除、减少不必要的函数调用
  - 调用与内存别名
- 指令级并行挖掘
  - 程序性能量化CPE
  - 编译优化-基本优化
  - 超标量处理器的循环展开与重新结合
- 现代处理器的分支预测

优化障碍:过程调用/Optimization Blocker: Procedure Calls

- ***编译器为什么不会自动将strlen外提? /Why couldn't compiler move*** `strlen` ***out of inner loop?***
  - 过程可能会有副作用/Procedure may have side effects
    - 每次调用时改变全局状态/Alters global state each time called
  - 对于同样的参数每次返回结果可能不同/Function may not return same value for given arguments
    - 依赖于全局状态/Depends on other parts of global state
    - 过程lower可能与strlen互相作用/Procedure `lower` could interact with `strlen`
- **Warning:**
  - 编译器将过程调用看做黑盒/Compiler treats procedure call as a black box
  - 很少做优化/Weak optimizations near them
- **Remedies:**
  - GCC –O1单个文件内优化/Use of inline functions
    - GCC does this with –O1
      - Within single file
  - 需要手动移动代码/Do your own code motion

```
size_t lencnt = 0;
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
        s++; length++;
    }
    lencnt += length;
    return length;
}
```

优化障碍:内存别名/Optimization Blocker: Memory Aliasing

- **别名/Aliasing**
  - 两个不同的指针指向/引用同一个位置/Two different memory references specify single location
  - C语言里面很常见/Easy to have happen in C
    - 允许地址计算/Since allowed to do address arithmetic
    - 直接访问存储结构/Direct access to storage structures
  - 尽可能使用局部变量/Get in habit of introducing local variables
    - 随着循环累积/Accumulating within loops
    - 以这种方式告知编译器不需要做别名/Your way of telling compiler not to check for aliasing

基本优化/Basic Optimizations

```
void combine4(vec_ptr v, data_t *dest)
{
  long i;
  long length = vec_length(v);
  data_t *d = get_vec_start(v);
  data_t t = IDENT;
  for (i = 0; i < length; i++)
    t = t OP d[i];
  *dest = t;
}
```

- **将vec_length移出循环/Move vec_length out of loop**
- **每个迭代避免边界检查/Avoid bounds check on each cycle**
- **使用临时变量进行累积/Accumulate in temporary**

循环展开/Loop Unrolling (2x1)

```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = (x OP d[i]) OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

- **每个迭代做2倍更有用的工作/Perform 2x more useful work per iteration**

循环展开与单独累加器/Loop Unrolling with Separate Accumulators (2x2)

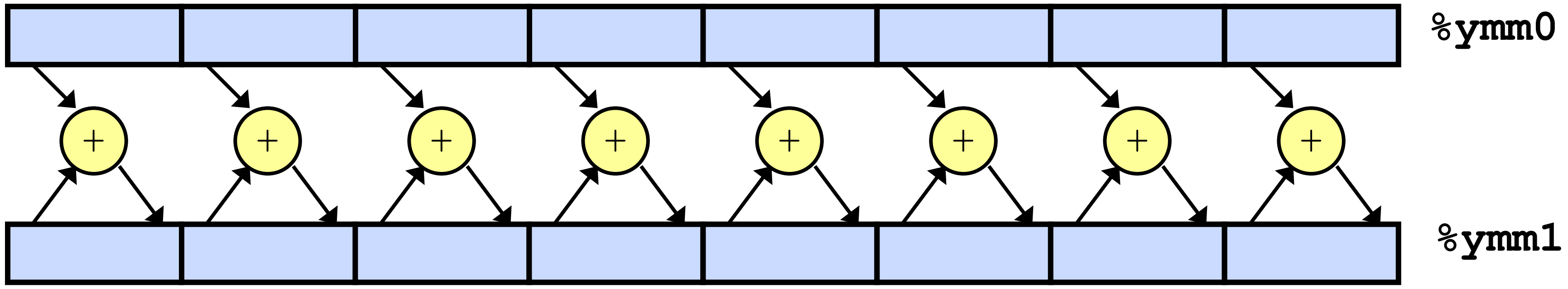
```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x0 = IDENT;
    data_t x1 = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
       x0 = x0 OP d[i];
       x1 = x1 OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x0 = x0 OP d[i];
    }
    *dest = x0 OP x1;
}
```

- 与association不同/Different form of reassociation

SIMD操作/SIMD Operations

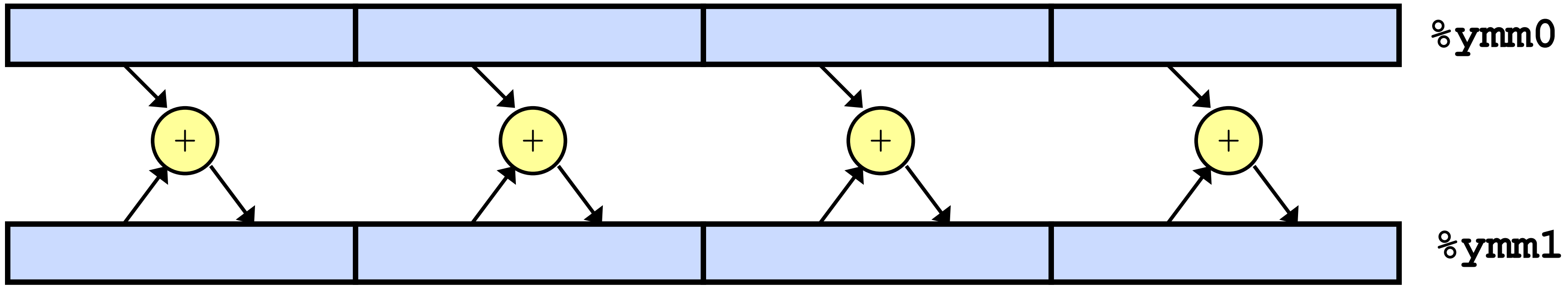
- SIMD操作：单精度/SIMD Operations: Single Precision

```
vaddsd %ymm0, %ymm1, %ymm1
```



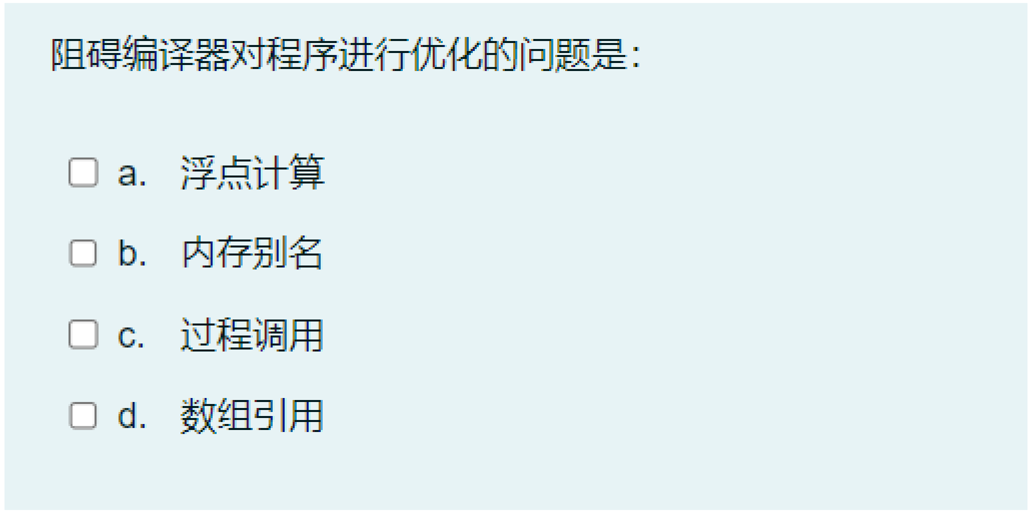
- SIMD操作：双精度/SIMD Operations: Double Precision

```
vaddpd %ymm0, %ymm1, %ymm1
```



例题1

- 多选题：



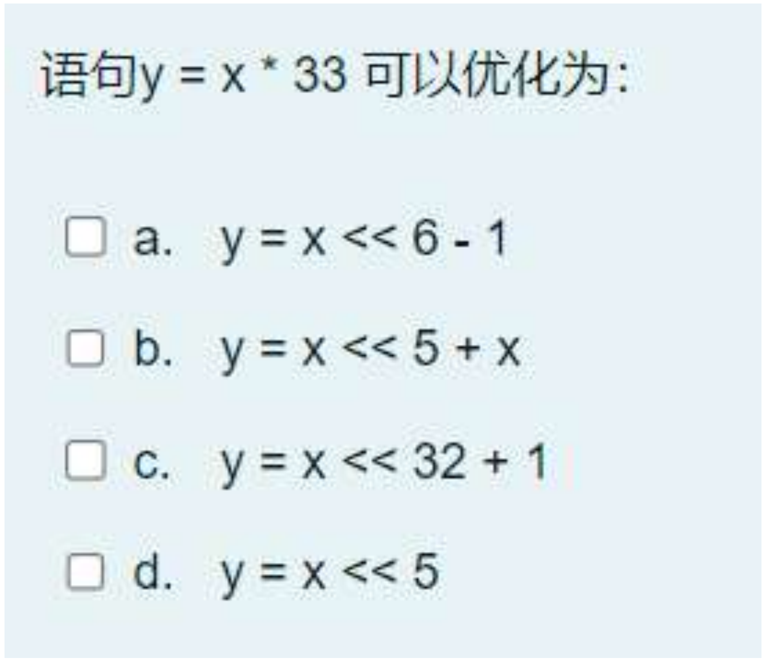
阻碍编译器对程序进行优化的问题是：

- ☐ a. 浮点计算
- ☐ b. 内存别名
- ☐ c. 过程调用
- ☐ d. 数组引用

- 答案：B、C

例题2

- 单选题



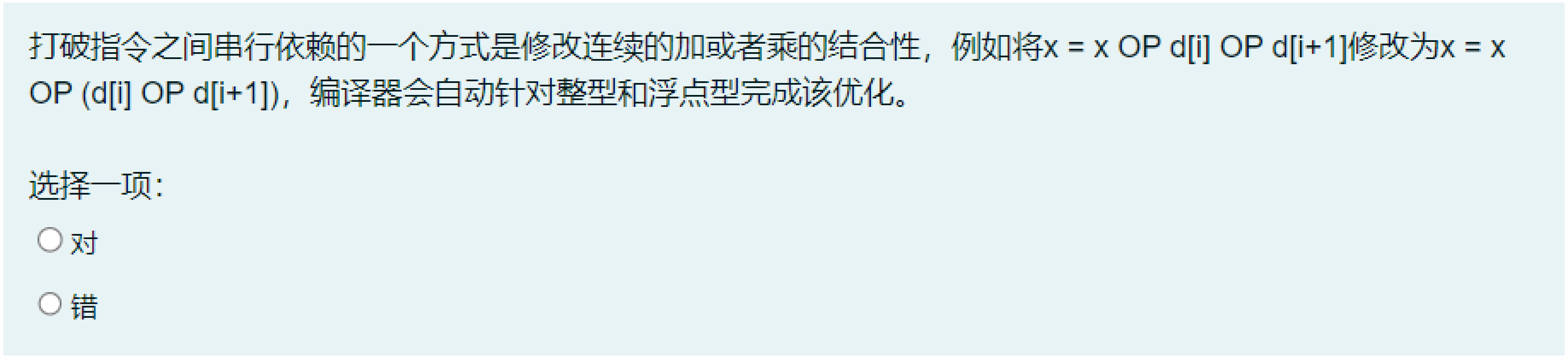
语句y = x * 33 可以优化为:

- ☐ a. y = x << 6 - 1
- ☐ b. y = x << 5 + x
- ☐ c. y = x << 32 + 1
- ☐ d. y = x << 5

- 答案：B

例题3

- **判断题**


打破指令之间串行依赖的一个方式是修改连续的加或者乘的结合性，例如将x = x OP d[i] OP d[i+1]修改为x = x OP (d[i] OP d[i+1])，编译器会自动针对整型和浮点型完成该优化。

选择一项:

○ 对

○ 错

- **答案：错，编译器不会对浮点型数完成该优化**

存储器的层级结构

- 存储器的类型与特点
- 数据局部性与指令局部性
- 层次结构
  - 缓存结构与地址编码
  - 缓存命中、不命中机制
- 高速缓存的性能影响

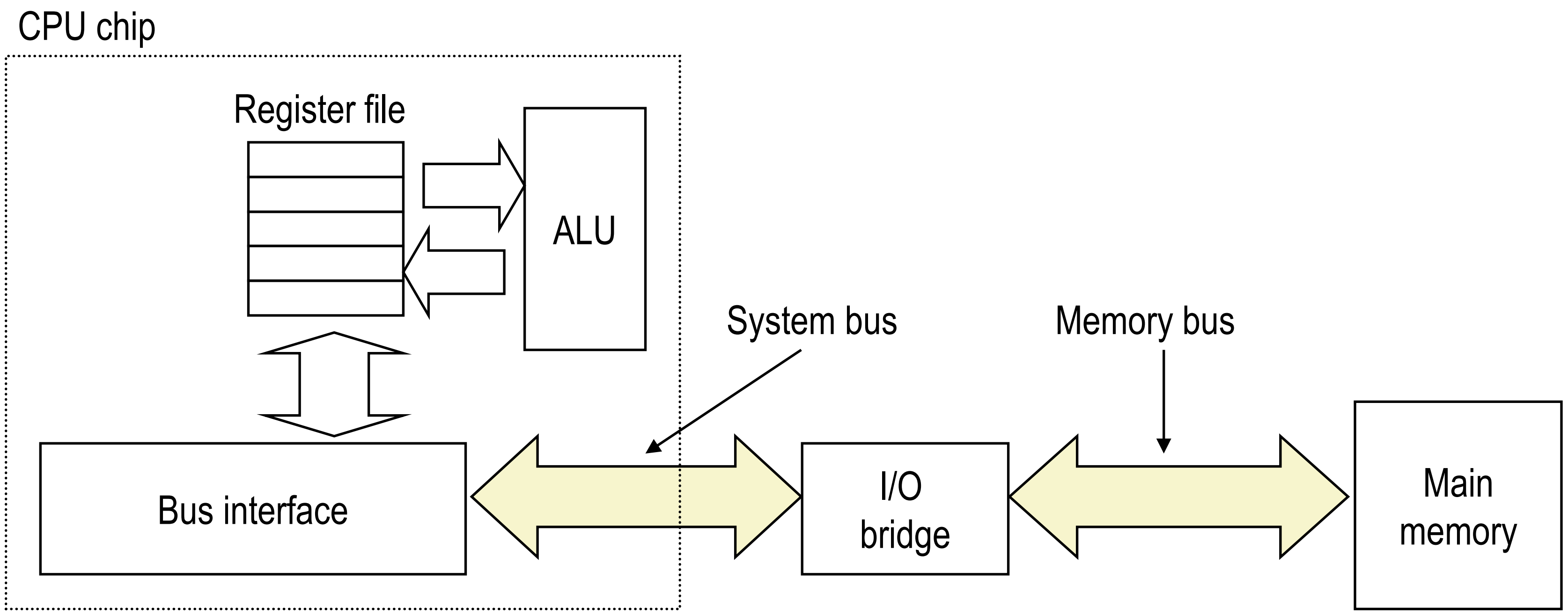
随机访问内存/Random-Access Memory (RAM)

- **主要特点/Key features**
  - RAM通常封装为一个芯片/RAM is traditionally packaged as a chip.
  - 基本存储单元是一个cell（每个cell是一个bit）Basic storage unit is normally a cell (one bit per cell).
  - 多个RAM芯片构成一个内存/Multiple RAM chips form a memory.

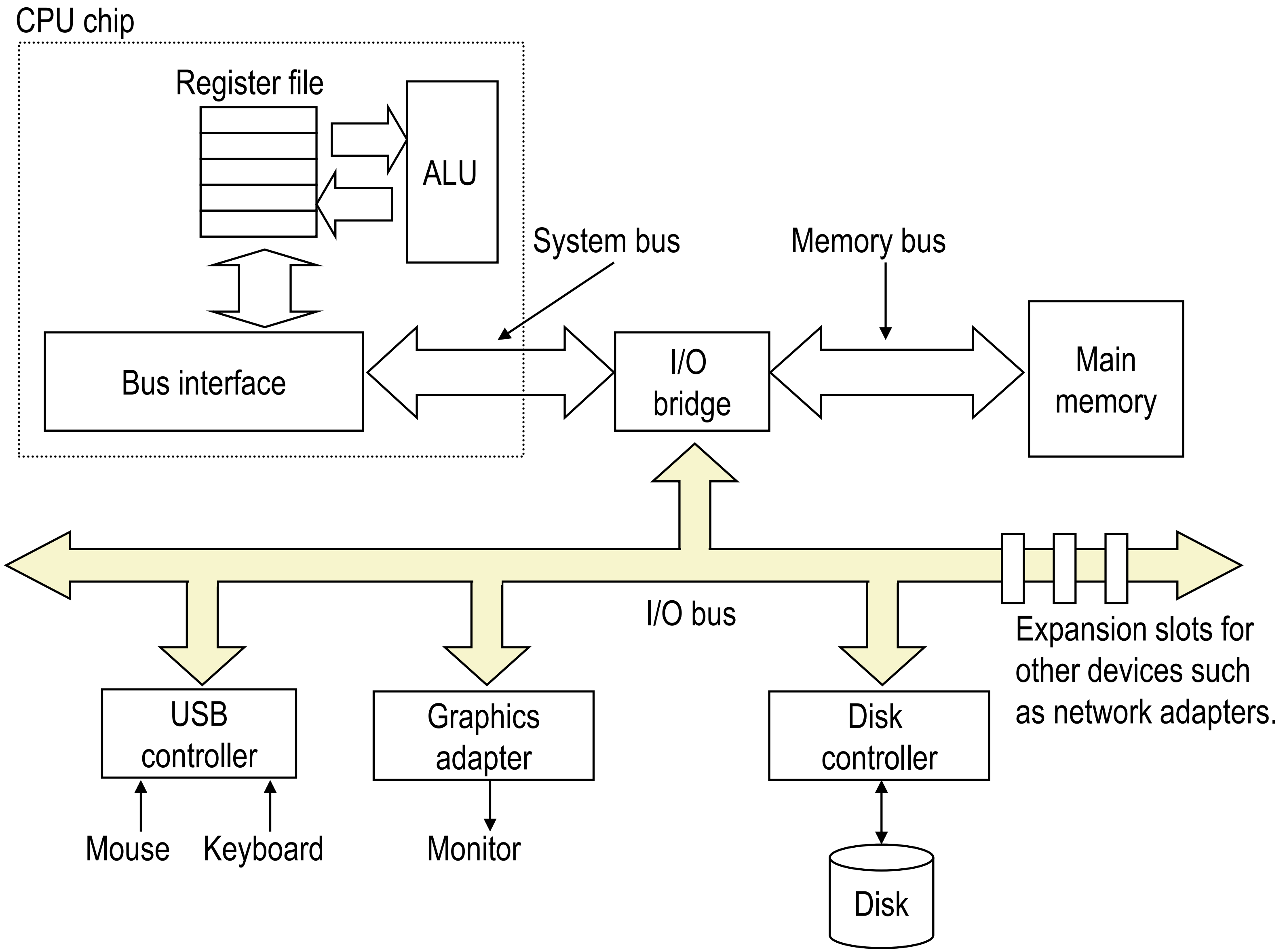
- **RAM有两种类型/RAM comes in two varieties:**
  - SRAM 静态RAM/SRAM (Static RAM)
  - DRAM 动态RAM/DRAM (Dynamic RAM)

传统CPU和内存之间的互连结构/Traditional Bus Structure Connecting CPU and Memory

- 总线是一组并行的用于传输地址、数据和控制信号的导线/A **bus** is a collection of parallel wires that carry address, data, and control signals.
- 总线通常是多个设备共享的/Buses are typically shared by multiple devices.

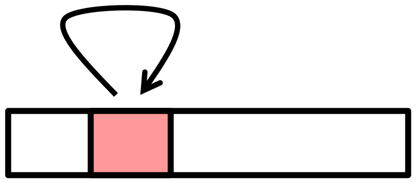


I/O总线/ I/O Bus

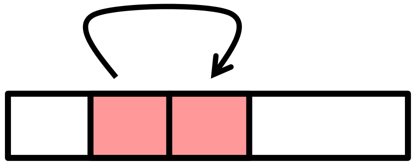

CPU chip
Register file
ALU
System bus
Memory bus
Bus interface
I/O bridge
Main memory
I/O bus
Expansion slots for other devices such as network adapters.
USB controller
Graphics adapter
Disk controller
Mouse
Keyboard
Monitor
Disk

局域性/Locality

- **局域性原理/Principle of Locality: 程序更倾向于使用临近或者最近使用过的数据和指令/Programs tend to use data and instructions with addresses near or equal to those they have used recently**

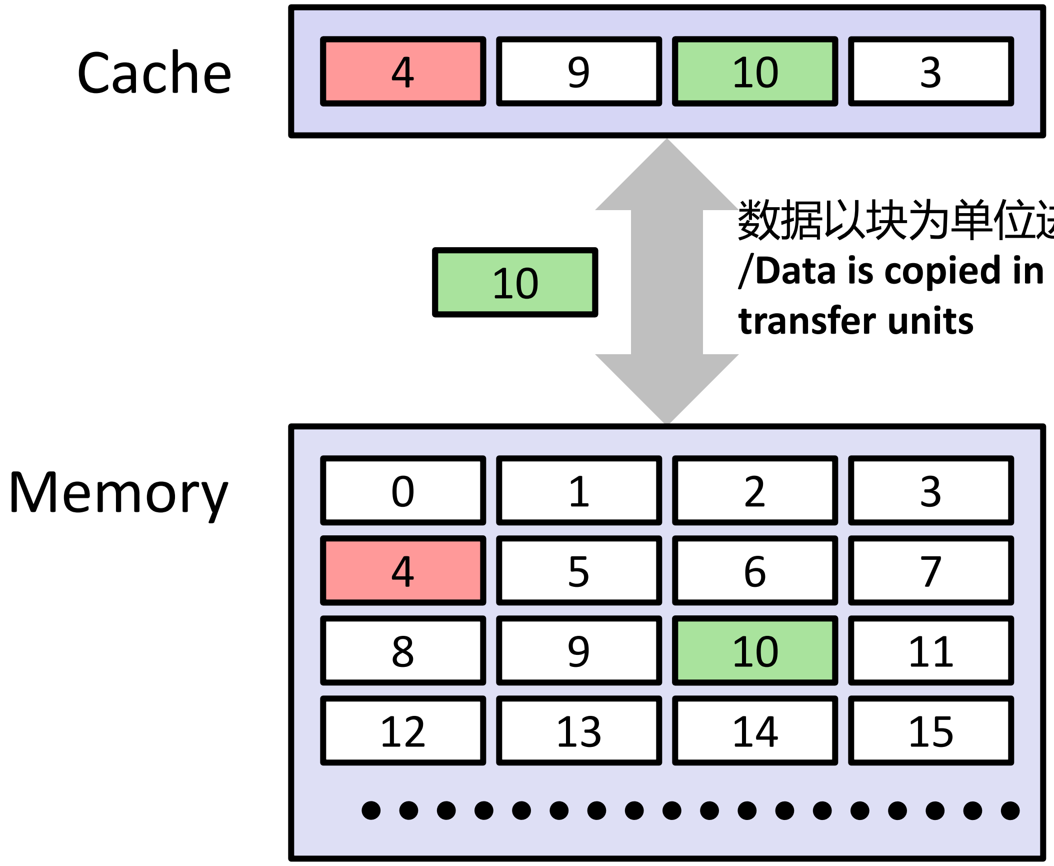


- **时间局域性/Temporal locality:**
  - 最近使用的数据在不远的将来会继续使用/Recently referenced items are likely to be referenced again in the near future



- **空间局域性/Spatial locality:**
  - 相邻的数据会在一段时间内一起访问/Items with nearby addresses tend to be referenced close together in time

Cache基本概念/General Cache Concepts

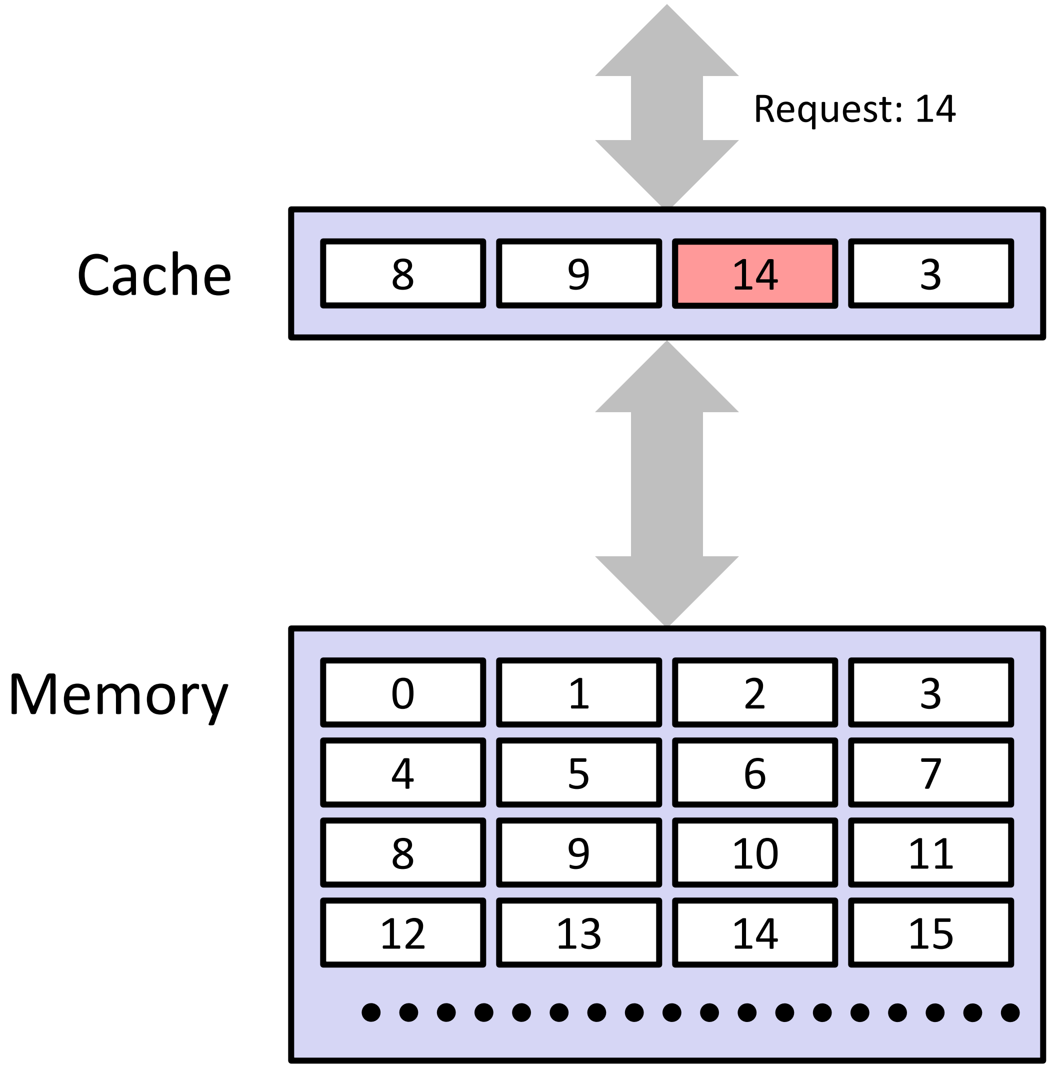


更小、更快、更贵，缓存了一部分块/
Smaller, faster, more expensive memory caches a subset of the blocks

数据以块为单位进行拷贝传输
/Data is copied in block-sized transfer units

更大、更慢、更便宜的主存被划分为多个块/
Larger, slower, cheaper memory viewed as partitioned into “blocks”

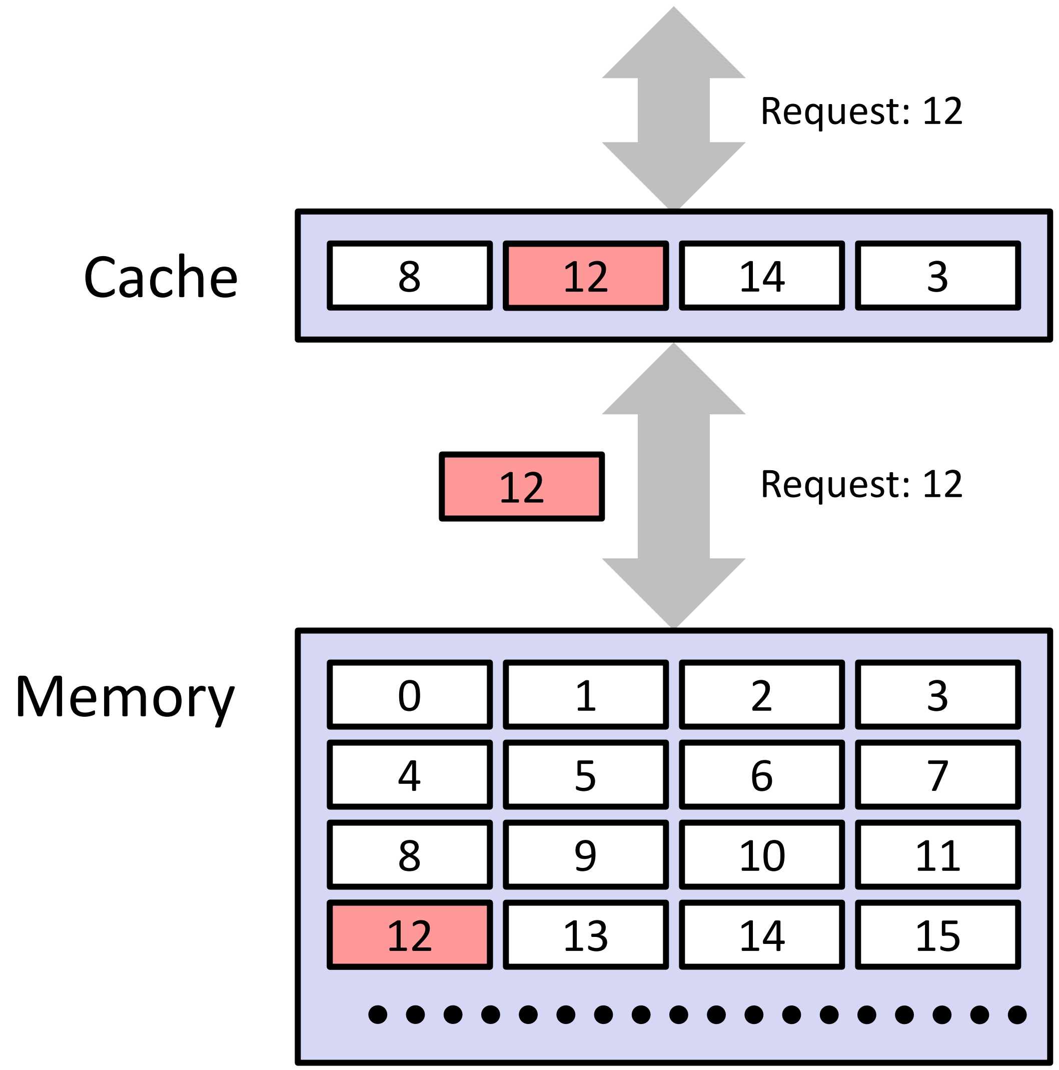
Cache命中/General Cache Concepts: Hit



需要访问块b中的数据/
Data in block b is needed

块b在cache中：命中/
Block b is in cache:
Hit!

Cache丢失/General Cache Concepts: Miss



需要访问块b中的数据/
Data in block b is needed

块b不在cache中：丢失！/
Block b is not in cache:
Miss!

从内存中加载块b/
Block b is fetched from
memory

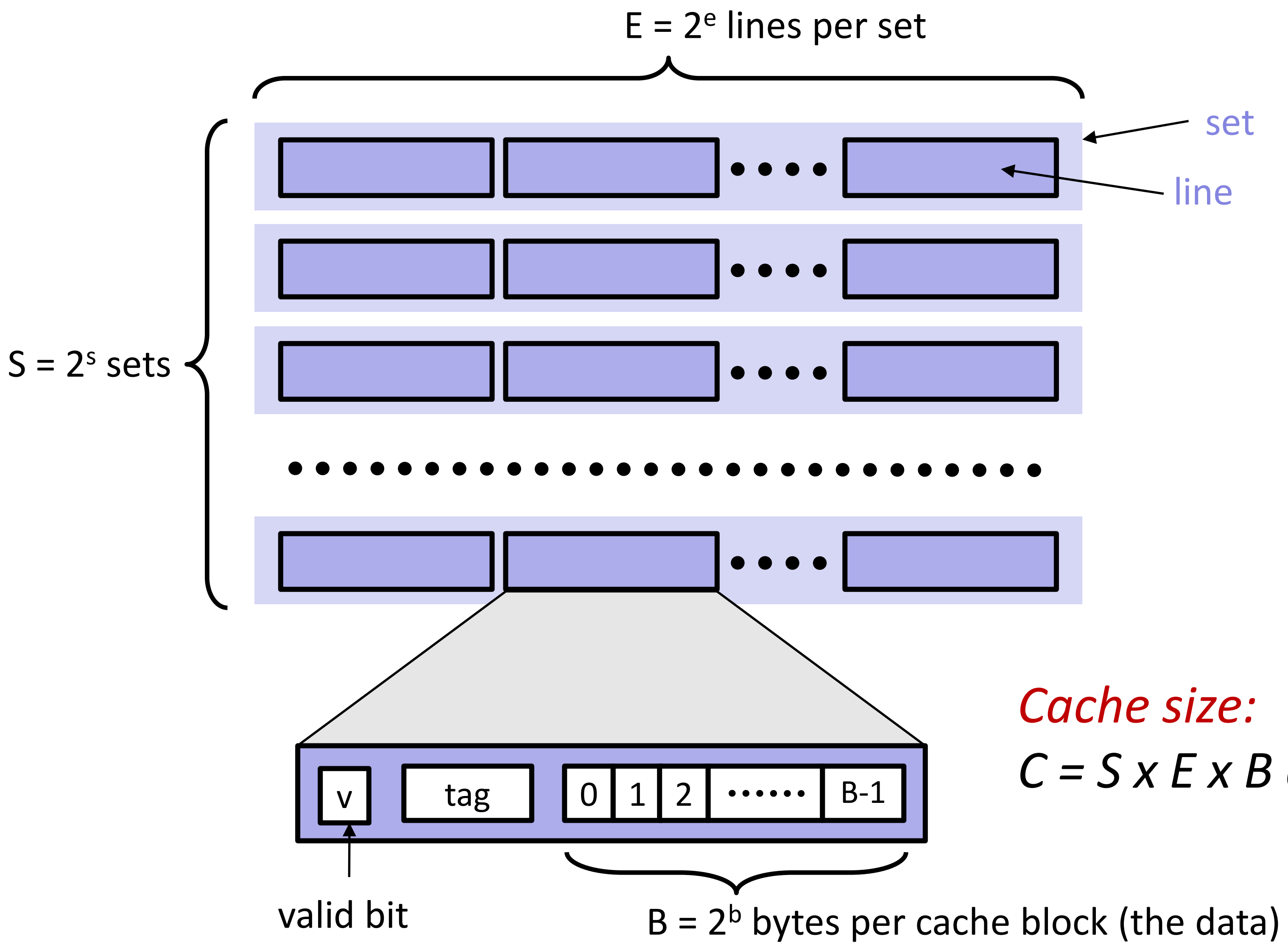
将块b存储在cache中/
Block b is stored in cache

- 放置策略Placement policy: 决定b放在哪里/determines where b goes
- 替换策略Replacement policy: 决定把那个块换出/determines which block gets evicted (victim)

Cache丢失类型/General Caching Concepts: Types of Cache Misses

- **冷启动丢失/Cold (compulsory) miss**
  - 冷启动丢失是因为刚开始Cache是空的/Cold misses occur because the cache is empty.
- **冲突丢失/Conflict miss**
  - 大多数Cache将k+1层的数据块映射到k层更小的一个或者多个块位置/Most caches limit blocks at level k+1 to a small subset (sometimes a singleton) of the block positions at level k.
    - 例如k+1层的块被映射到k层（i mod 4）的位置
    - E.g. Block i at level k+1 must be placed in block (i mod 4) at level k.
  - 当多个块被映射到k层的同一个位置时就会出现冲突/Conflict misses occur when the level k cache is large enough, but multiple data objects all map to the same level k block.
    - 例如访问0,8,0,8,0,8就会每次导致Cache丢失/E.g. Referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time.
- **容量丢失/Capacity miss**
  - 当活跃使用的Cache块大小大于Cache容量时就会导致丢失/Occurs when the set of active cache blocks (working set) is larger than the cache.

Cache组织结构 General Cache Organization (S, E, B)

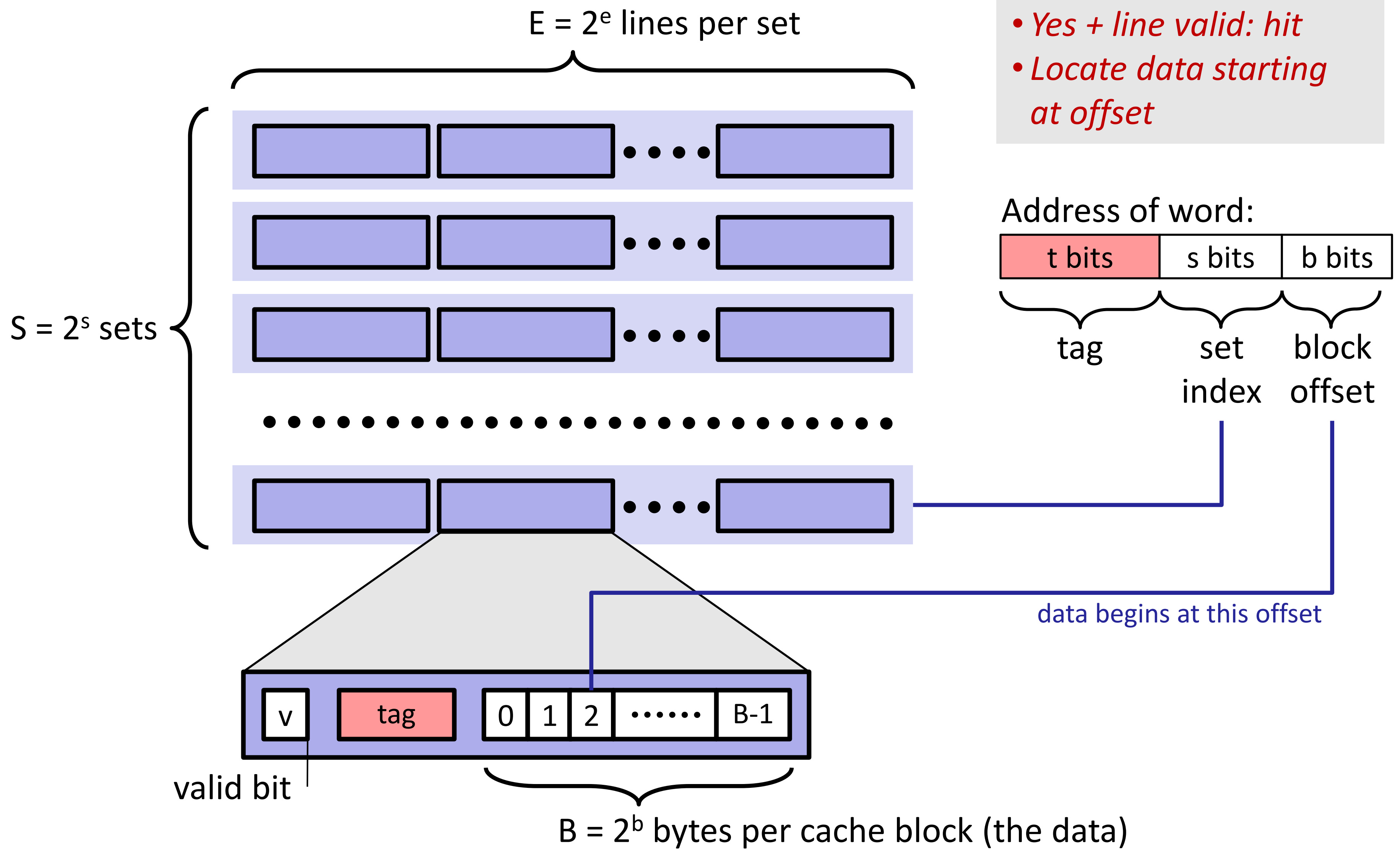


Cache size:

C = S x E x B data bytes

Cache读操作 Cache Read

- *Locate set*
- *Check if any line in set has matching tag*
- *Yes + line valid: hit*
- *Locate data starting at offset*



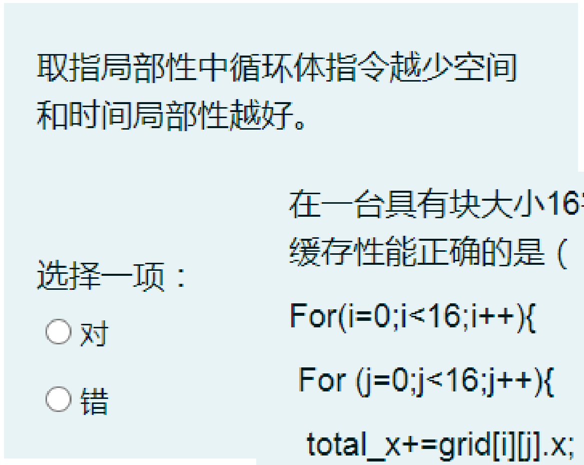
Cache写操作 What about writes?

- **多数据副本 Multiple copies of data exist:**
  - L1, L2, L3, Main Memory, Disk
- **写命中时如何处理？ What to do on a write-hit?**
  - 写透（直接写入内存）Write-through (write immediately to memory)
  - 写回（替换式写回）Write-back (defer write to memory until replacement of line)
    - 需要脏比特位标识 Need a dirty bit (line different from memory or not)
- **写丢失时如何处理？ What to do on a write-miss?**
  - 写分配（装载进Cache后进行更新）Write-allocate (load into cache, update line in cache)
    - 如果后续还有写操作时比较好 Good if more writes to the location follow
  - 非写分配（直接写入内存，不装载）No-write-allocate (writes straight to memory, does not load into cache)
- **通常策略 Typical**
  - 写透 + 非写分配 Write-through + No-write-allocate
  - **写回 + 写分配 Write-back + Write-allocate**

Cache性能评价 Cache Performance Metrics

- **丢失率 Miss Rate**
  - 内存引用没有在Cache中找到的比率 Fraction of memory references not found in cache (misses / accesses)
    = 1 – hit rate
  - 通常的Cache丢失率 Typical numbers (in percentages):
    - 3-10% for L1
    - L2也可能很小，依赖Cache大小 can be quite small (e.g., < 1%) for L2, depending on size, etc.
- **命中时间 Hit Time**
  - 从Cache行到处理器的时间 Time to deliver a line in the cache to the processor
    - 包括判断Cache是否命中的时间 includes time to determine whether the line is in the cache
  - 通常的时间 Typical numbers:
    - L1 Cache 4个时钟周期 4 clock cycle for L1
    - L2 Cache 10个时钟周期 10 clock cycles for L2
- **丢失开销 Miss Penalty**
  - 丢失需要额外的时间 Additional time required because of a miss
    - 主存的访问周期50~200 typically 50-200 cycles for main memory (Trend: increasing!)

例题

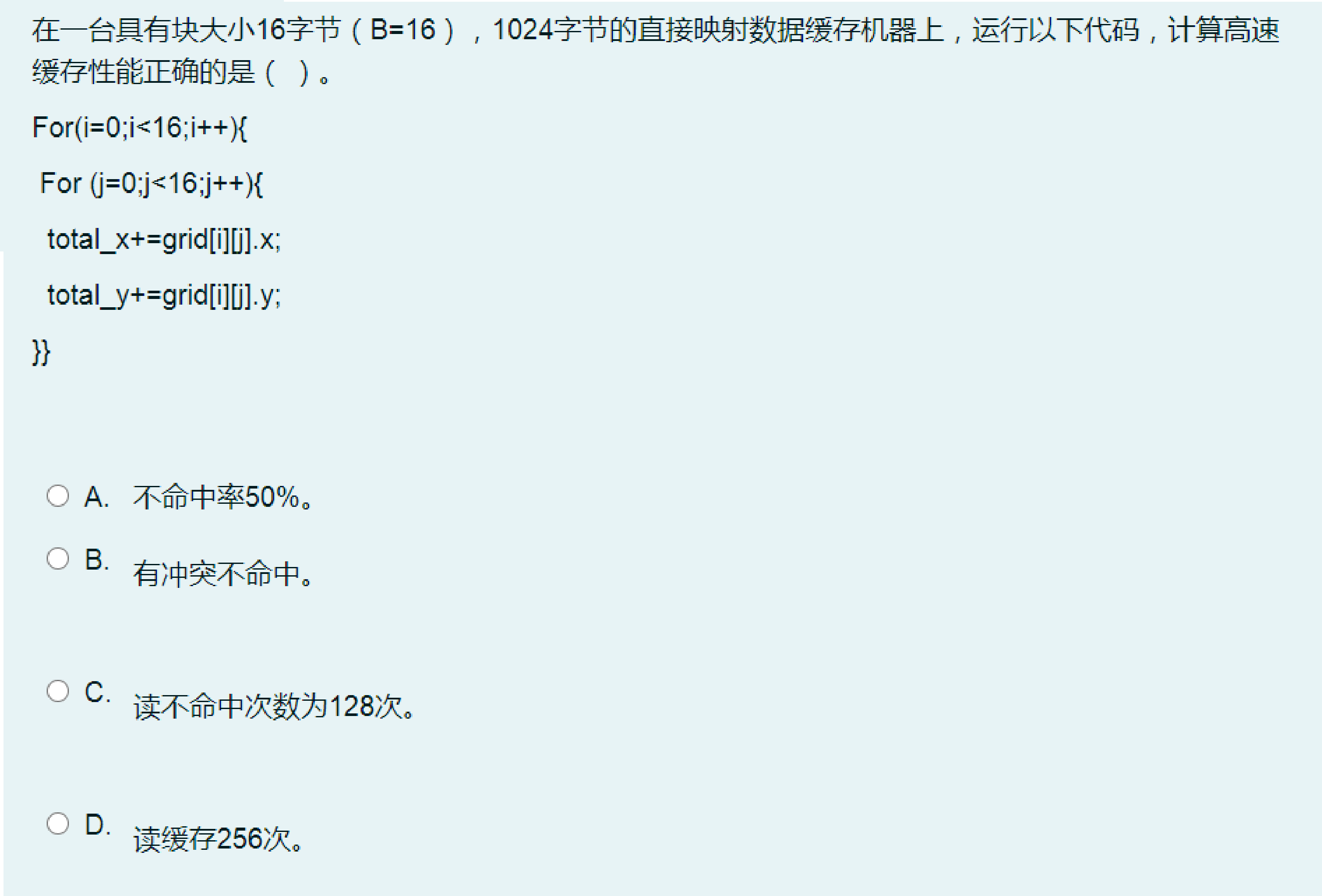


取指局部性中循环体指令越少空间和时间局部性越好。

选择一项：

○ 对

○ 错



在一台具有块大小16字节（B=16），1024字节的直接映射数据缓存机器上，运行以下代码，计算高速缓存性能正确的是（ ）。

```
For(i=0;i<16;i++){
 For (j=0;j<16;j++){
  total_x+=grid[i][j].x;
  total_y+=grid[i][j].y;
}}
```

○ A. 不命中率50%。

○ B. 有冲突不命中。

○ C. 读不命中次数为128次。

○ D. 读缓存256次。

链接

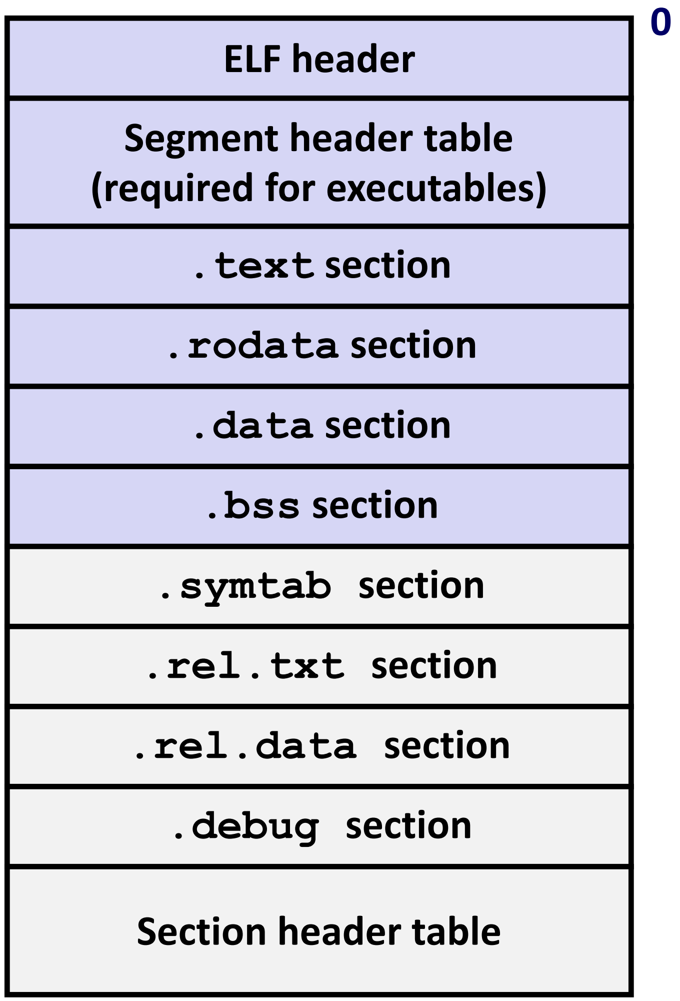
- 编译器与可执行文件格式（ELF)
- 静态链接的符号解析与重定向
- 动态链接及共享库
  - 加载时链接
  - 运行时链接
- 库打桩
- 位置无关代码

三种不同的目标文件（模块）/Three Kinds of Object Files (Modules)

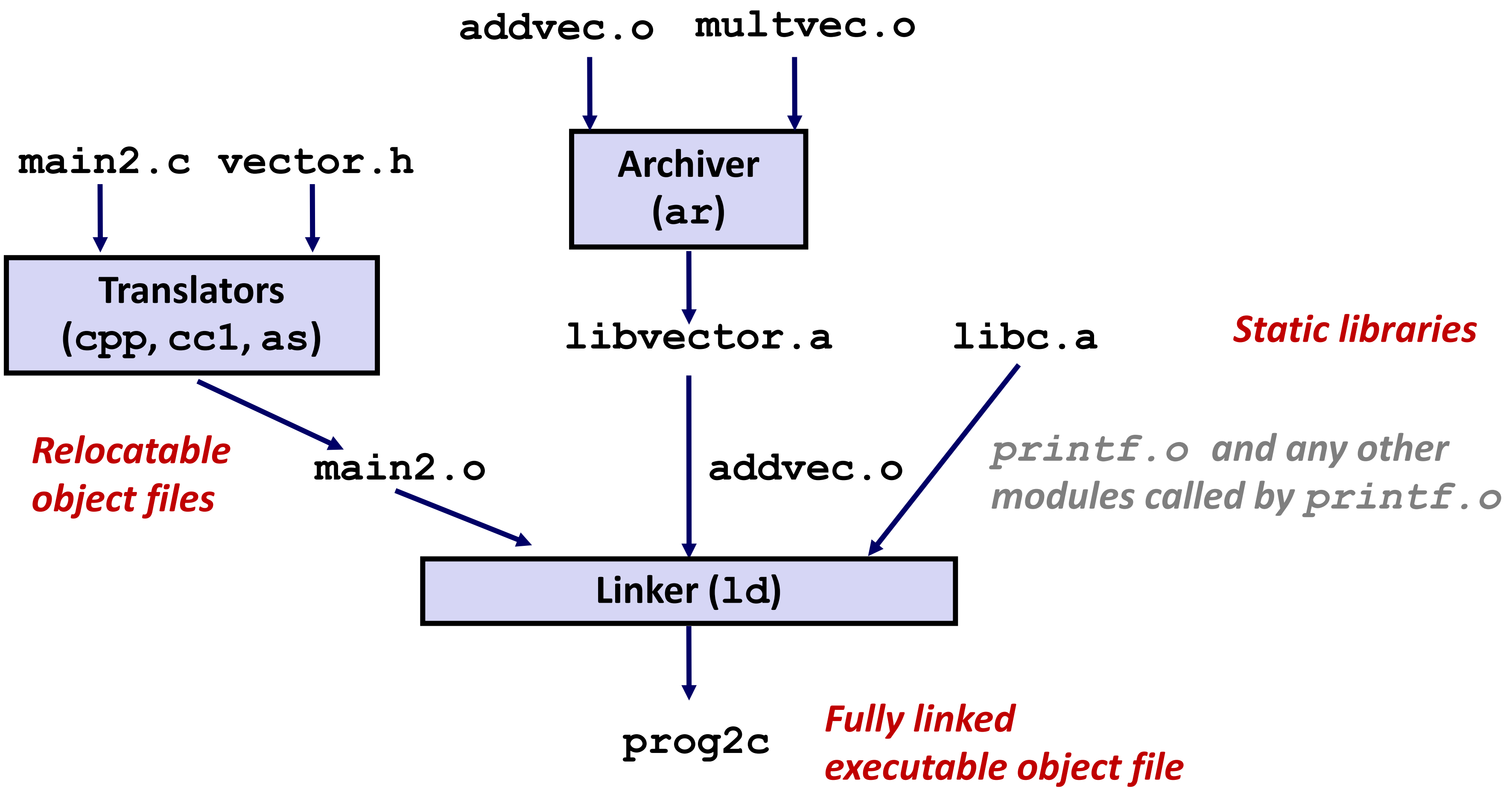
- **可重定位目标文件/Relocatable object file (`.o` file)**
  - 其中的代码和数据可以和其他可重定位目标文件一起形成可执行目标文件/Contains code and data in a form that can be combined with other relocatable object files to form executable object file.
    - 每个.o文件从一个源码文件.c获得/Each `.o` file is produced from exactly one source (`.c`) file
- **可执行目标文件/Executable object file (`a.out` file)**
  - 其中的代码和数据可以直接拷贝到内存中执行/Contains code and data in a form that can be copied directly into memory and then executed.
- **共享目标文件/Shared object file (`.so` file)**
  - 特定的可重定位目标文件，能以在加载时或者运行时装进内存并进行动态链接/Special type of relocatable object file that can be loaded into memory and linked dynamically, at either load time or run-time.
  - Windows上通常称为动态链接库/Called *Dynamic Link Libraries* (DLLs) by Windows

ELF目标文件格式/ELF Object File Format

- ELF头/Elf header
  - 字大小、字节顺序、文件类型、机器类型等/Word size, byte ordering, file type (.o, exec, .so), machine type, etc.
- 段头表/Segment header table
  - 页大小、虚拟地址内存段(节)、段大小/Page size, virtual addresses memory segments (sections), segment sizes.
- `.text`节/`.text` section
  - 代码/Code
- `.rodata`节/`.rodata` section
  - 只读数据：跳转表,.../Read only data: jump tables, ...
- `.data`节/`.data` section
  - 初始化的全局数据/Initialized global variables
- `.bss`节/`.bss` section
  - 未初始化的全局变量/Uninitialized global variables
  - “Block Started by Symbol”
  - “Better Save Space”
  - 不占内存空间/Has section header but occupies no space



链接静态链接库/Linking with Static Libraries

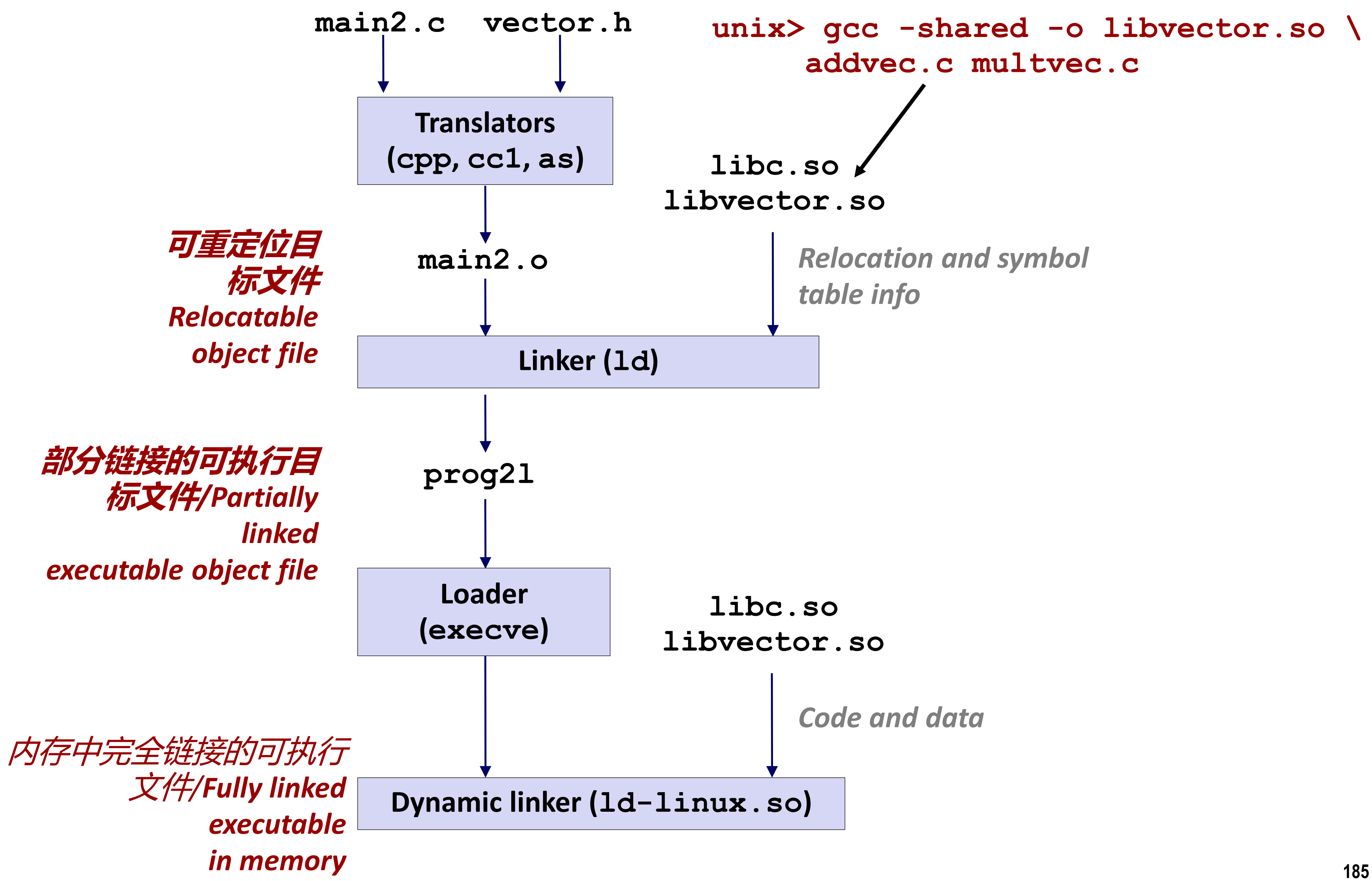


"c" for "compile-time"

链接器符号消解规则/Linker’s Symbol Rules

- **规则1：不允许有多个强符号/Rule 1: Multiple strong symbols are not allowed**
  - 每个符号只允许被定义一次/Each item can be defined only once
  - 否则会出现链接错误/Otherwise: Linker error
- **规则2：当有一个强符号和多个弱符号时，选择强符号/Rule 2: Given a strong symbol and multiple weak symbols, choose the strong symbol**
  - 对弱符号的引用改为对强符号的引用/References to the weak symbol resolve to the strong symbol
- **规则3：当有多个弱符号时，选择其中任意一个/Rule 3: If there are multiple weak symbols, pick an arbitrary one**
  - 可以通过`gcc -fno-common`指定/Can override this with `gcc -fno-common`

加载时进行动态链接/Dynamic Linking at Load-time

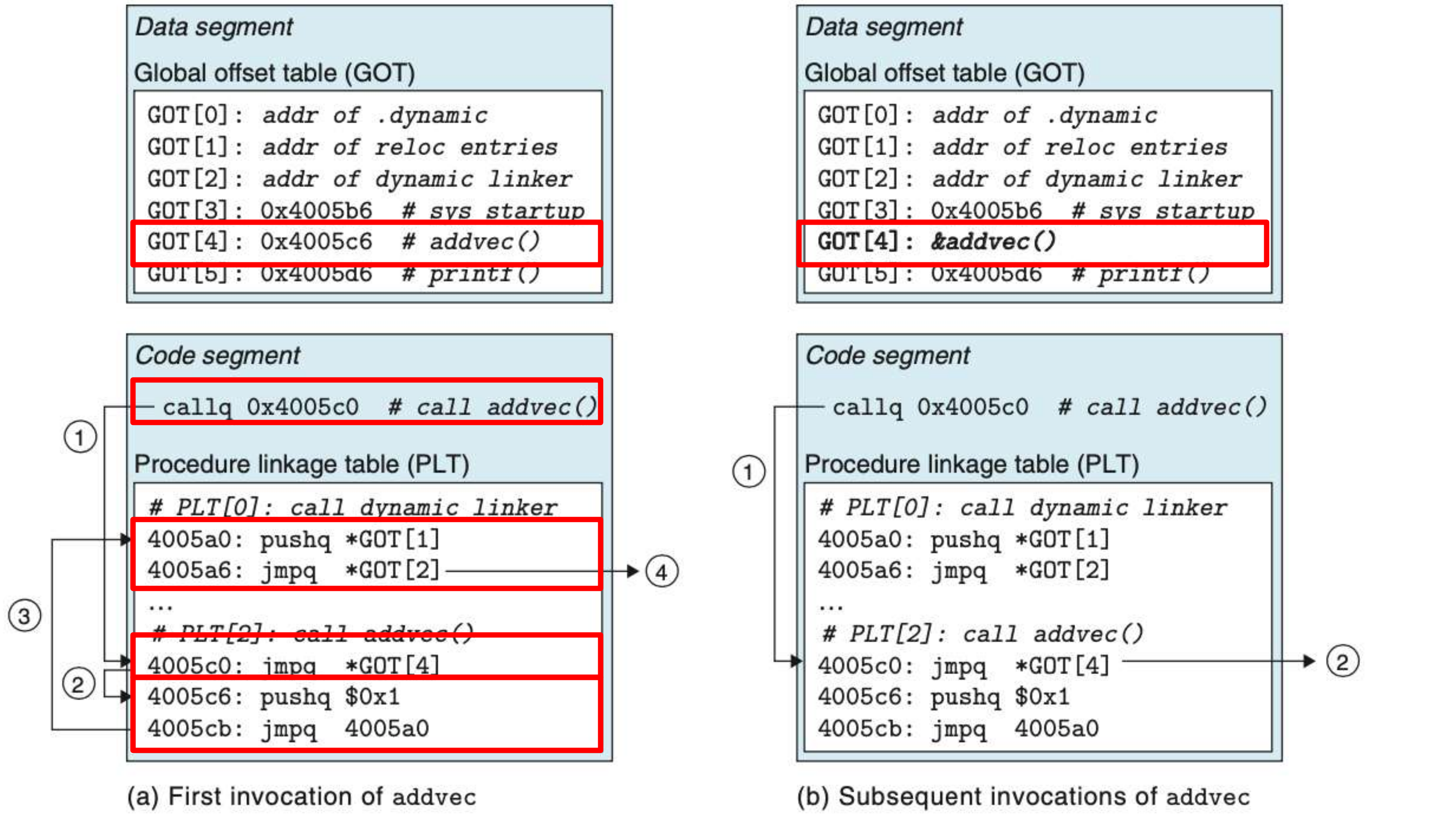

main2.c
vector.h
Translators (cpp, cc1, as)
main2.o
libc.so
libvector.so
Relocation and symbol table info
Linker (ld)
prog21
Loader (execve)
libc.so
libvector.so
Code and data
Dynamic linker (ld-linux.so)
可重定位目标文件 Relocatable object file
部分链接的可执行目标文件/Partially linked executable object file
内存中完全链接的可执行文件/Fully linked executable in memory

```
unix> gcc -shared -o libvector.so \
        addvec.c multvec.c
```

打桩回顾/Interpositioning Recap

- **编译时/Compile Time**
  - Apparent calls to malloc/free get macro-expanded into calls to mymalloc/myfree
- **链接时/Link Time**
  - Use linker trick to have special name resolutions
    - malloc → __wrap_malloc
    - __real_malloc → malloc
- **加载/运行时/Load/Run Time**
  - Implement custom version of malloc/free that use dynamic linking to load library malloc/free under different names

PIC函数调用


Data segment
Global offset table (GOT)
Code segment
Procedure linkage table (PLT)
1
2
3
4
(a) First invocation of addvec
(b) Subsequent invocations of addvec

(a) First invocation of addvec

```
Data segment
Global offset table (GOT)
GOT[0]: addr of .dynamic
GOT[1]: addr of reloc entries
GOT[2]: addr of dynamic linker
GOT[3]: 0x4005b6  # sys startup
GOT[4]: 0x4005c6  # addvec()
GOT[5]: 0x4005d6  # printf()

Code segment
 callq 0x4005c0  # call addvec()

Procedure linkage table (PLT)
# PLT[0]: call dynamic linker
4005a0: pushq *GOT[1]
4005a6: jmpq  *GOT[2]
...
# PLT[2]: call addvec()
4005c0: jmpq  *GOT[4]
4005c6: pushq $0x1
4005cb: jmpq  4005a0
```

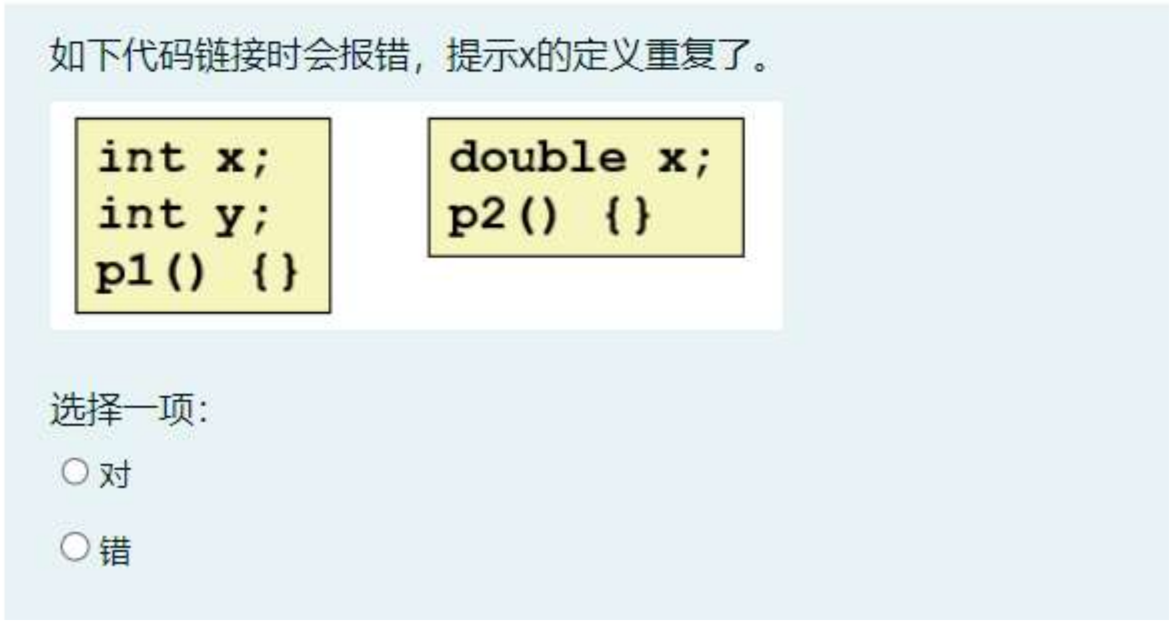
(b) Subsequent invocations of addvec

```
Data segment
Global offset table (GOT)
GOT[0]: addr of .dynamic
GOT[1]: addr of reloc entries
GOT[2]: addr of dynamic linker
GOT[3]: 0x4005b6  # sys startup
GOT[4]: &addvec()
GOT[5]: 0x4005d6  # printf()

Code segment
 callq 0x4005c0  # call addvec()

Procedure linkage table (PLT)
# PLT[0]: call dynamic linker
4005a0: pushq *GOT[1]
4005a6: jmpq  *GOT[2]
...
# PLT[2]: call addvec()
4005c0: jmpq  *GOT[4]
4005c6: pushq $0x1
4005cb: jmpq  4005a0
```

Figure 7.19 Using the PLT and GOT to call external functions. The dynamic linker resolves the address of addvec the first time it is called.

例题1

- 判断题：



如下代码链接时会报错，提示x的定义重复了。

```
int x;
int y;
p1() {}
```

```
double x;
p2() {}
```

选择一项：

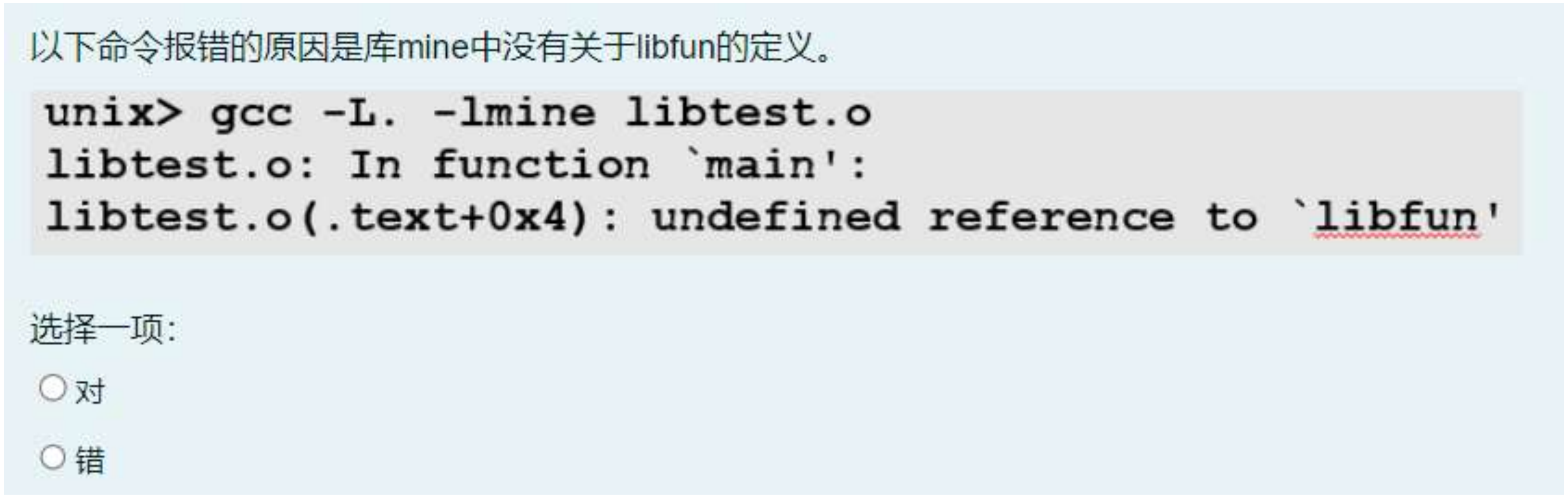
○ 对

○ 错

- 答案：对

例题2

- **判断题：**



以下命令报错的原因是库mine中没有关于libfun的定义。

```
unix> gcc -L. -lmine libtest.o
libtest.o: In function `main':
libtest.o(.text+0x4): undefined reference to `libfun'
```

选择一项:

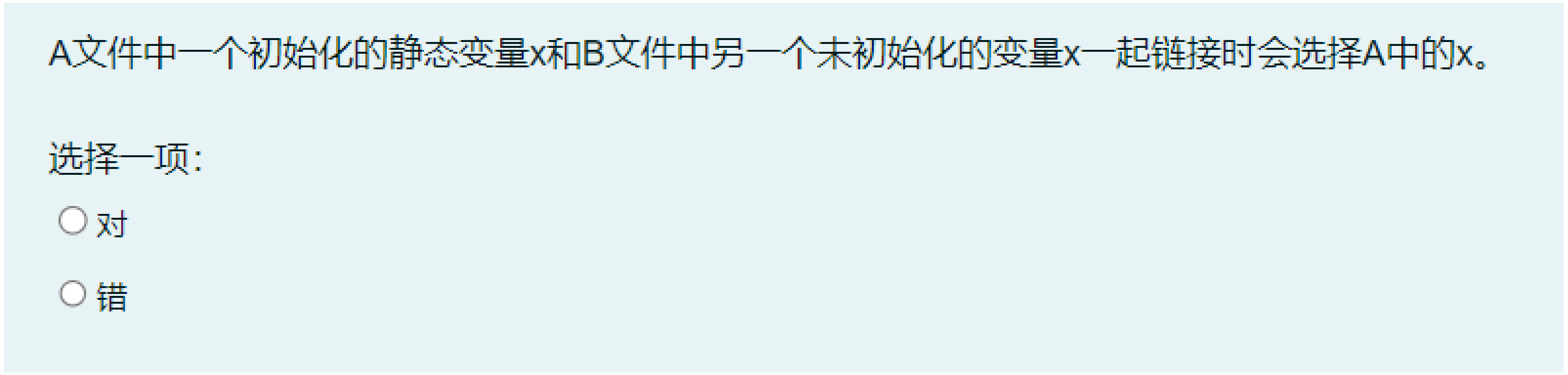
○ 对

○ 错

- **答案：错，libtest.o应该放在前面**

例题3

- **判断题：**


A文件中一个初始化的静态变量x和B文件中另一个未初始化的变量x一起链接时会选择A中的x。

选择一项：

○ 对

○ 错

- **答案：错，静态变量的作用域是文件**

异常控制流

- 异常与异常处理
- 进程控制
  - 基本概念：并发流、模式控制、上下文切换
  - 进程控制函数使用和系统调用
- 信号方式
  - shell与进程组
  - 发送信号、接收信号、挂起/阻塞、处理程序及安全信号处理、
  - 可移植信号处理、同步控制流、显式等待信号
- 非本地跳转

异常控制流 / Exceptional Control Flow

- **存在系统的每个层次 / Exists at all levels of a computer system**
- **低层次机制 / Low level mechanisms**
  - 1.异常 / **Exceptions**
    - 为响应系统事件改变控制流（例如，系统状态改变）/ Change in control flow in response to a system event (i.e., change in system state)
    - 硬件和OS软件组合实现 / Implemented using combination of hardware and OS software
- **高层次机制/Higher level mechanisms**
  - 2. 进程上下文切换 / **Process context switch**
    - 硬件时钟和OS软件实现 / Implemented by OS software and hardware timer
  - 3. 信号 **Signals**
    - OS软件实现 / Implemented by OS software
  - 4. 非局部跳转 / **Nonlocal jumps**: `setjmp()` and `longjmp()`
    - C运行时库实现/Implemented by C runtime library

异步异常（中断/Asynchronous Exceptions (Interrupts)

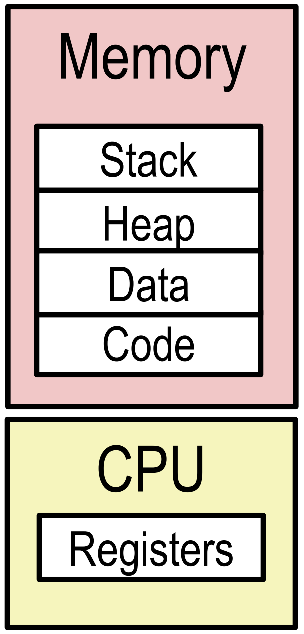
- **由处理器外部事件引起/Caused by events external to the processor**
  - 通过处理器的中断管脚给出/Indicated by setting the processor’s *interrupt pin*
  - 中断处理程序返回后执行下一条指令/Handler returns to “next” instruction
- **举例/Examples:**
  - 时钟中断/Timer interrupt
    - 大约几毫秒，外部时钟芯片触发/Every few ms, an external timer chip triggers an interrupt
    - 将控制权从用户切换到内核/Used by the kernel to take back control from user programs
  - 外部设备的I/O中断/ I/O interrupt from external device
    - 键盘输入Ctrl-C/Hitting Ctrl-C at the keyboard
    - 网络数据包到达/Arrival of a packet from a network
    - 磁盘数据到达/Arrival of data from a disk

同步异常/Synchronous Exceptions

- **指令执行导致的异常事件/Caused by events that occur as a result of executing an instruction:**
  - ***陷入/陷阱 Traps***
    - 人为的/Intentional
    - 例如：系统调用、断点、特殊指令等/Examples: ***system calls***, breakpoint traps, special instructions
    - 控制流返回下一条指令/Returns control to “next” instruction
  - ***故障 Faults***
    - 不是有意的但是大概率可恢复/Unintentional but possibly recoverable
    - 例如：缺页异常、保护异常、浮点异常/Examples: page faults (recoverable), protection faults (unrecoverable), floating point exceptions
    - 重新执行或者终止执行/Either re-executes faulting (“current”) instruction or aborts
  - ***终止 Aborts***
    - 非故意且不可恢复/Unintentional and unrecoverable
    - 例如：非法指令、校验错误、机器检查/Examples: illegal instruction, parity error, machine check
    - 终止当前程序执行/Aborts current program

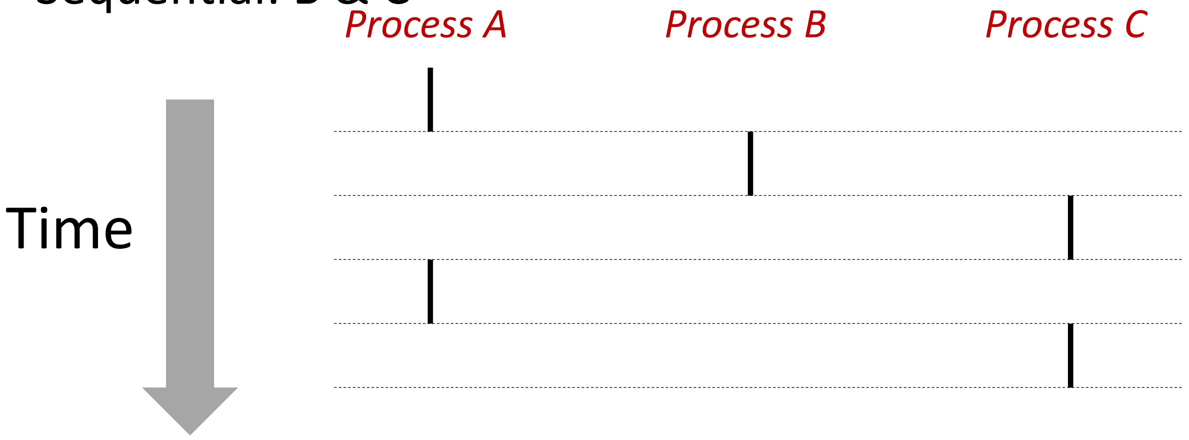
进程/Processes

- **定义：进程是程序的运行的实例/Definition: A *process* is an instance of a running program.**
  - 计算机科学最重要的概念之一/One of the most profound ideas in computer science
  - 与程序和处理器不同/Not the same as “program” or “processor”
- **进程为每个程序提供了两个关键抽象/Process provides each program with two key abstractions:**
  - ***逻辑控制流/Logical control flow***
    - 每个程序看起来独占CPU/Each program seems to have exclusive use of the CPU
    - 内核支持的上下文切换/Provided by kernel mechanism called *context switching*
  - ***私有地址空间/Private address space***
    - 每个程序看起来独占主存空间/Each program seems to have exclusive use of main memory.
    - 系统支持的虚拟内存/Provided by kernel mechanism called *virtual memory*



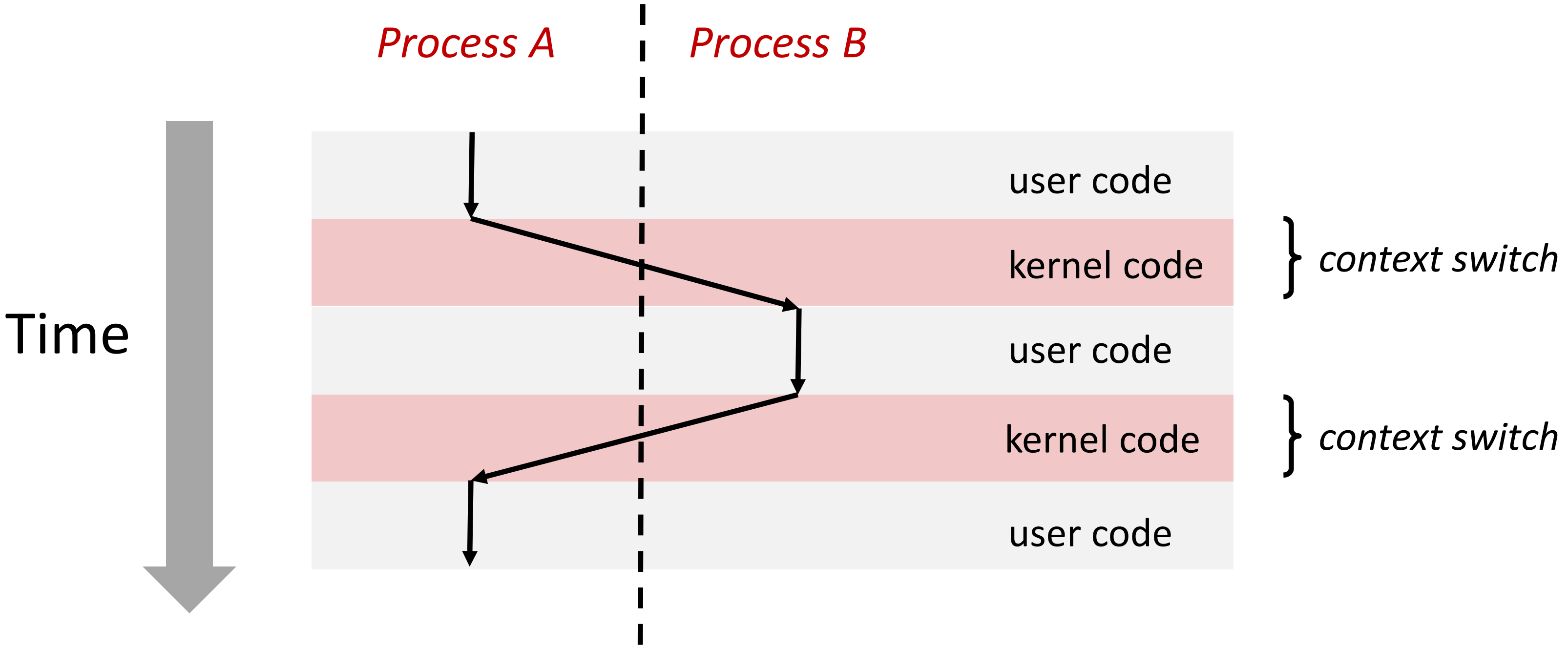
并发进程/Concurrent Processes

- 每个进程是一个逻辑控制流/Each process is a logical control flow.
- 两个进程并发运行如果在时间上重叠/Two processes *run concurrently* (*are concurrent)* if their flows overlap in time
- 否则是顺序执行/Otherwise, they are ***sequential***
- 例如 (在单个核上运行）/Examples (running on single core):
  - Concurrent: A & B, A & C
  - Sequential: B & C



上下文切换/Context Switching

- **进程是由操作系统内核管理的/Processes are managed by a shared chunk of memory-resident OS code called the *kernel***
  - 重点：内核不是一个独立的进程，而是作为某些进程的一部分运行 Important: the kernel is not a separate process, but rather runs as part of some existing process.
- **上下文切换使得控制流从一个进程切换到另一个进程 Control flow passes from one process to another via a *context switch***



fork举例/fork Example

```c
int main()
{
   pid_t pid;
   int x = 1;

   pid = Fork();
   if (pid == 0) {  /* Child */
      printf("child : x=%d\n", ++x);
            exit(0);
   }

   /* Parent */
   printf("parent: x=%d\n", --x);
   exit(0);
}
```

fork.c

```
linux> ./fork
parent: x=0
child : x=2
```

- **一次调用，两次返回/Call once, return twice**
- **并发执行/Concurrent execution**
  - 不能预测父进程和子进程之间的执行顺序/Can’t predict execution order of parent and child
- **重复的但是独立的地址空间/Duplicate but separate address space**
  - `fork`时`x`的值是`1`/`x` has a value of 1 when fork returns in parent and child
  - 后续对x的修改都是独立的/Subsequent changes to `x` are independent
- **共享打开的文件/Shared open files**
  - `stdio`在子进程和父进程中都是一样的/`stdout` is the same in both parent and child

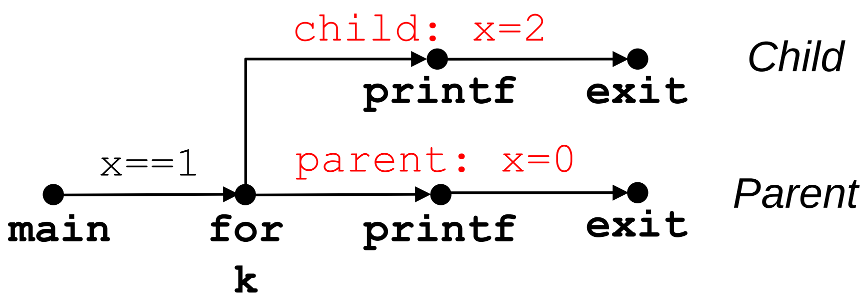
使用进程图描述fork /Modeling `fork` with Process Graphs

- **进程图是描述并发程序中语句偏序关系的有用工具/A *process graph* is a useful tool for capturing the partial ordering of statements in a concurrent program:**
  - 每个顶点是一个语句/Each vertex is the execution of a statement
  - a -> b 表示a在b之前发生/a -> b means `a` happens before b
  - 每个边可以用当前值或者变量进行标注/Edges can be labeled with current value of variables
  - `printf`节点可以用标注为`output/printf` vertices can be labeled with output
  - 每个图的开始节点没有入边/Each graph begins with a vertex with no inedges
- **图的任何拓扑排序都对应一个可行的全局序/Any *topological sort* of the graph corresponds to a feasible total ordering.**
  - 全局序节点之间的边从左指向右/Total ordering of vertices where all edges point from left to right

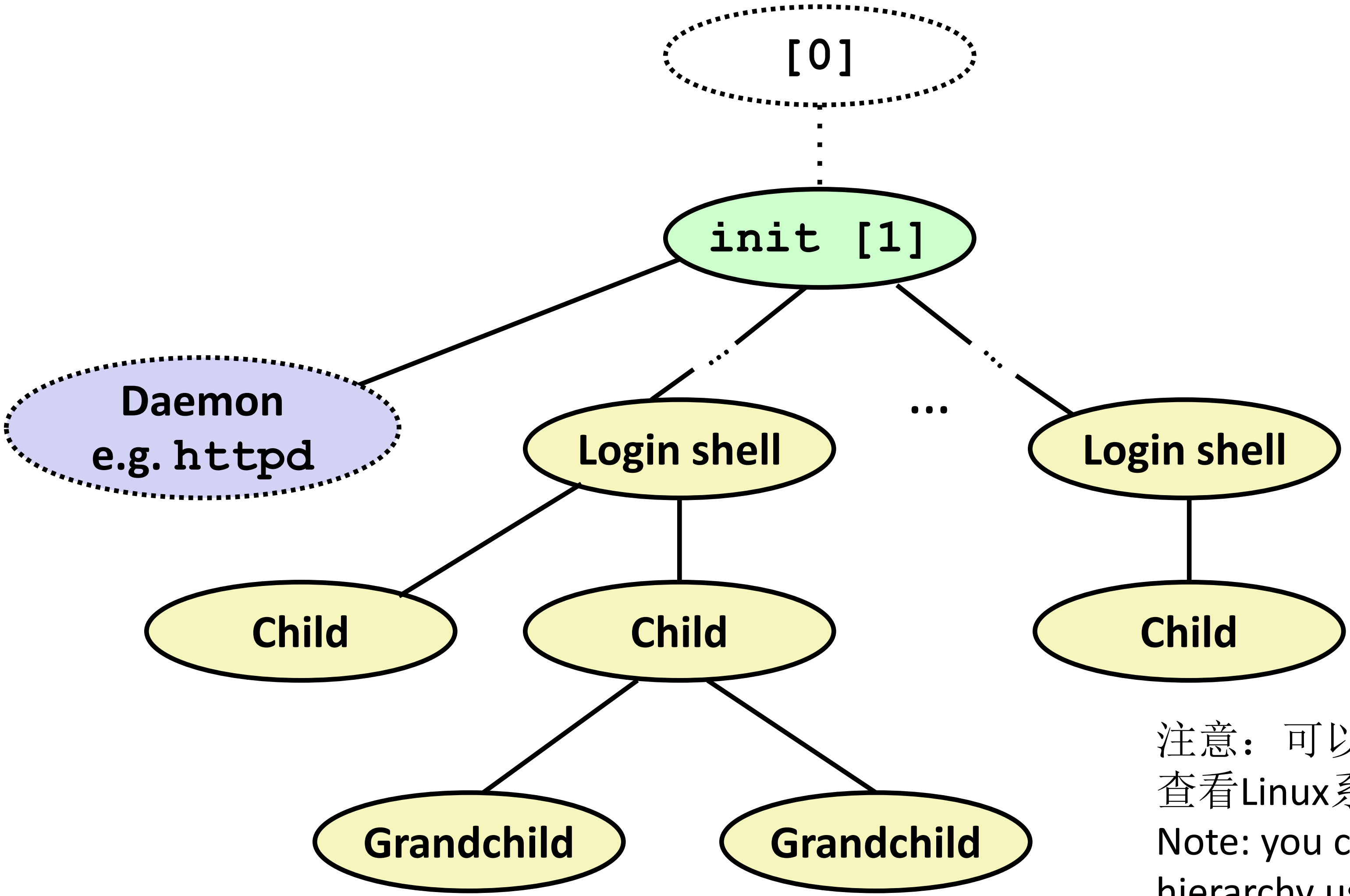
进程图举例/Process Graph Example

```
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {  /* Child */
        printf("child : x=%d\n", ++x);
                exit(0);
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    exit(0);
}
```

fork.c



Linux进程树/Linux Process Hierarchy



注意：可以用pstree命令查看Linux系统的进程树/
Note: you can view the hierarchy using the Linux `pstree` command

可以利用ECF解决/ECF to the Rescue!

- **解决方案：异常控制流/Solution: Exceptional control flow**
  - 系统在后台进程处理完成后打断正常处理流程并提醒我们/The kernel will interrupt regular processing to alert us when a background process completes
  - Unix系统中这种提醒的机制是信号/In Unix, the alert mechanism is called a ***signal***

信号/Signals

- 信号是用来通知一个进程某种类型的事件在系统中发生了/**A *signal* is a small message that notifies a process that an event of some type has occurred in the system**
  - 类似于异常和中断/Akin to exceptions and interrupts
  - 由内核发送给一个进程（有时是根据另一个进程的请求）Sent from the kernel (sometimes at the request of another process) to a process
  - 信号的类型是用1-30的小整型标识/Signal type is identified by small integer ID’s (1-30)
  - 信号的唯一信息就是这个ID以及信号达到的事实/Only information in a signal is its ID and the fact that it arrived

ID	*Name*	*Default Action*	*Corresponding Event*
2	SIGINT	Terminate	用户输入ctrl-c/User typed ctrl-c
9	SIGKILL	Terminate	杀死程序（不能覆盖或被忽略）Kill program (cannot override or ignore)
11	SIGSEGV	Terminate	段错误/Segmentation violation
14	SIGALRM	Terminate	时钟信号/Timer signal
17	SIGCHLD	Ignore	子进程停止或者终止/Child stopped or terminated

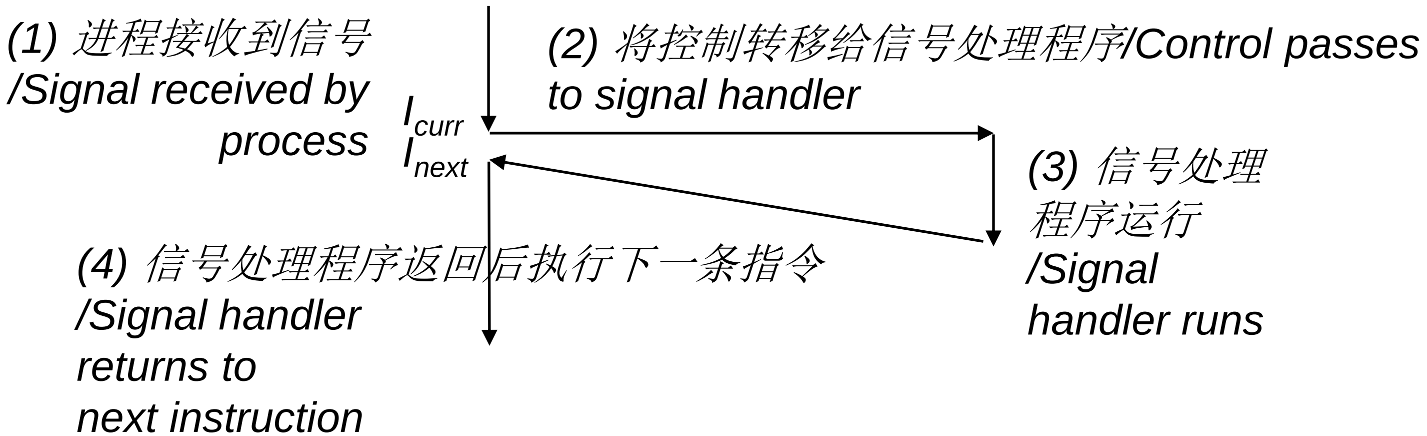
信号概念：发送一个信号/Signal Concepts: Sending a Signal

- **内核通过更新目标进程的某些状态来发送一个信号给目标进程/Kernel *sends* (delivers) a signal to a *destination process* by updating some state in the context of the destination process**

- **内核发送信号是由于以下原因中的一个/Kernel sends a signal for one of the following reasons:**
  - 内核侦测到除零错误或者子进程终止等系统事件/Kernel has detected a system event such as divide-by-zero (SIGFPE) or the termination of a child process (SIGCHLD)
  - 另外一个进程调用了kill系统调用显式请求内核发送一个信号给目标进程/Another process has invoked the `kill` system call to explicitly request the kernel to send a signal to the destination process

信号概念：接收一个信号/Signal Concepts: Receiving a Signal

- 目标进程接收信号是由于系统内核强制其对某个信号的发送做出响应 /A destination process ***receives*** a signal when it is forced by the kernel to react in some way to the delivery of the signal
- 可能的响应方式/Some possible ways to react:
  - ***忽略***信号（什么也不做）***/Ignore*** the signal (do nothing)
  - ***终止进程***（可以选择对信息转储）***/Terminate*** the process (with optional core dump)
  - ***调用***用户级信号处理函数对信号进行处理***/Catch*** the signal by executing a user-level function called ***signal handler***
    - 类似于硬件异常处理函数对异步中断的响应/Akin to a hardware exception handler being called in response to an asynchronous interrupt:



(1) 进程接收到信号 /Signal received by process
I_{curr}
I_{next}
(2) 将控制转移给信号处理程序/Control passes to signal handler
(3) 信号处理程序运行 /Signal handler runs
(4) 信号处理程序返回后执行下一条指令 /Signal handler returns to next instruction

信号概念：挂起或者阻塞的信号/Signal Concepts: Pending and Blocked Signals

- **已经发送但是没有被接收的信号处于挂起状态/A signal is *pending* if sent but not yet received**
  - 任何特定类型的信号最多有一个挂起的/There can be at most one pending signal of any particular type
  - 重要：信号不排队/Important: Signals are not queued
    - 如有某个进程有一个类型为k的信号挂起，泽后续发给该进程的其他信号被直接抛弃/If a process has a pending signal of type k, then subsequent signals of type k that are sent to that process are discarded

- **一个进程会阻塞某种特定类型信号的接收/A process can *block* the receipt of certain signals**
  - Blocked signals can be delivered, but will not be received until the signal is unblocked

- **挂起的信号最多被接收一次/A pending signal is received at most once**

信号概念：挂起/阻塞位/Signal Concepts: Pending/Blocked Bits

- **内核在每个进程的上下文维护一个挂起和阻塞的比特向量/Kernel maintains `pending` and `blocked` bit vectors in the context of each process**
  - **挂起：表示挂起的信号集合/`pending`**: represents the set of pending signals
    - 当发送了一个k类型的信号时系统设置第k个比特位/Kernel sets bit k in **`pending`** when a signal of type k is delivered
    - 当类型k的信号被接收后系统会将第k个比特位清零/Kernel clears bit k in **`pending`** when a signal of type k is received
  - **阻塞：表示阻塞的信号集合/`blocked`**: represents the set of blocked signals
    - 可以使用**`sigprocmask`** 设置或者清除/Can be set and cleared by using the **`sigprocmask`** function
    - 也称为信号掩码/Also referred to as the *signal mask*.

安装信号处理程序/Installing Signal Handlers

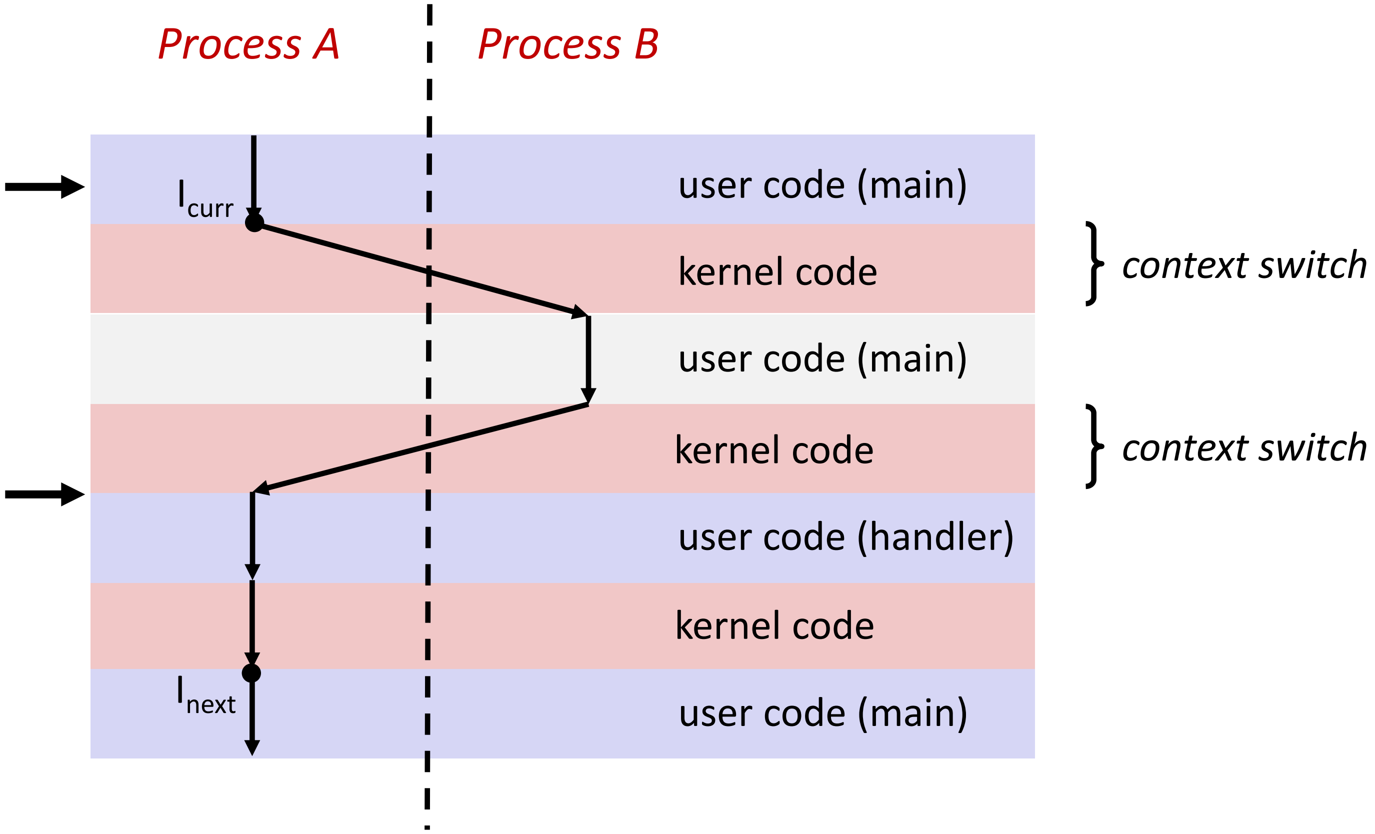
- **函数Signal修改信号signum对应的默认行为/The `signal` function modifies the default action associated with the receipt of signal `signum`:**
  - `handler_t *signal(int signum, handler_t *handler)`

- **可能的选项/Different values for `handler`:**
  - SIG_IGN: ignore signals of type **`signum`**/忽略signum类型的信号
  - SIG_DFL: revert to the default action on receipt of signals of type **`signum`**/接收到signum类型的信号时按照默认动作处理
  - 否则handler是用于级信号处理程序的地址/Otherwise, **`handler`** is the address of a user-level ***signal handler***
    - 当进程接收到类型为signum的信号时调用/Called when process receives signal of type **`signum`**
    - 称为安装信号处理程序/Referred to as ***"installing"*** the handler
    - 执行信号处理程序称为/Executing handler is called ***"catching"*** or ***"handling"*** the signal
    - 当信号处理程序返回时，控制权交给进程接收到信号时被打断的指令/When the handler executes its return statement, control passes back to instruction in the control flow of the process that was interrupted by receipt of the signal

信号处理程序作为并发控制流的另一个视图/Another View of Signal Handlers as Concurrent Flows

信号发送给进程A/
Signal delivered
to process A

信号被进程A接收/
Signal received
by process A



编写安全处理程序的提示/Guidelines for Writing Safe Handlers

- **G0: 中断处理程序越简单越好/Keep your handlers as simple as possible**
  - 例如，设置全局标记后返回/e.g., Set a global flag and return
- **G1: 只调用异步安全的信号处理函数/Call only async-signal-safe functions in your handlers**
  - `printf`, `sprintf`, `malloc`, and `exit` are not safe! /这些都不安全
- **G2: 进入和退出时保存errno/Save and restore `errno` on entry and exit**
  - So that other handlers don't overwrite your value of `errno`
- **G3:临时阻塞所有的信号后再访问全局数据结构/ Protect accesses to shared data structures by temporarily blocking all signals.**
  - 避免可能的破坏/To prevent possible corruption
- **G4: 将全局变量声明位volatile/Declare global variables as `volatile`**
  - 避免编译器将其存储在寄存器中/To prevent compiler from storing them in a register
- **G5: 将全局标记声明为volatile `sig_atomic_t` / Declare global flags as `volatile sig_atomic_t`**
  - *Flag*只读或只写的变量/:*flag*: variable that is only read or written (e.g. flag = 1, not flag++)
  - 按照这种方式声明的变量不需要像其他全局变量那样保护/Flag declared this way does not need to be protected like other globals

正确的信号处理 /Correct Signal Handling

```
int ccount = 0;
void child_handler(int sig) {
   int olderrno = errno;
   pid_t pid;
   if ((pid = wait(NULL)) < 0)
      Sio_error("wait error");
   ccount--;
   Sio_puts("Handler reaped child ");
   Sio_putl((long)pid);
   Sio_puts(" \n");
   sleep(1);
   errno = olderrno;
}

void fork14() {
   pid_t pid[N];
   int i;
   ccount = N;
   Signal(SIGCHLD, child_handler);

   for (i = 0; i < N; i++) {
      if ((pid[i] = Fork()) == 0) {
         Sleep(1);
         exit(0);  /* Child exits */
      }
   }
   while (ccount > 0) /* Parent spins */
      ;
}
```

forks.c

```
whaleshark> ./forks 14
Handler reaped child 23240
Handler reaped child 23241
```

- **挂起的信号是不排队的/Pending signals are not queued**
  - 对每个信号类型，只用一个比特位来标识是否有信号被挂起/For each signal type, one bit indicates whether or not signal is pending…
  - 因此每种最多有一个挂起的信号/…thus at most one pending signal of any particular type.
- **不可以使用信号对事件计数，例如子进程终止等/You can't use signals to count events, such as children terminating.**

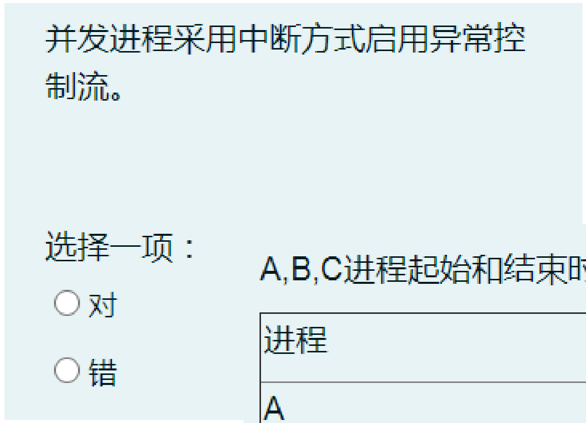
非局部跳转/Nonlocal Jumps: `setjmp/longjmp`

- **将控制转移到任意位置的强大（但比较危险）用户级机制/Powerful (but dangerous) user-level mechanism for transferring control to an arbitrary location**
  - 受控的打破call/return规则的方式/Controlled to way to break the procedure call / return discipline
  - 通常用户错误处理和信号处理/Useful for error recovery and signal handling

- **`int setjmp(jmp_buf j)`**
  - 必须在longjmp之前调用/Must be called before longjmp
  - 给出后续longjmp对应的返回位置/Identifies a return site for a subsequent longjmp
  - 一次调用，返回一次或者多次/Called **once**, returns **one or more** times

- **实现/Implementation:**
  - 通过将寄存器上下文、堆栈指针和PC值等存储在jmp_buf中记住当前位置 /Remember where you are by storing the current ***register context***, ***stack pointer***, and ***PC value*** in **`jmp_buf`**
  - 返回0/Return 0

例题

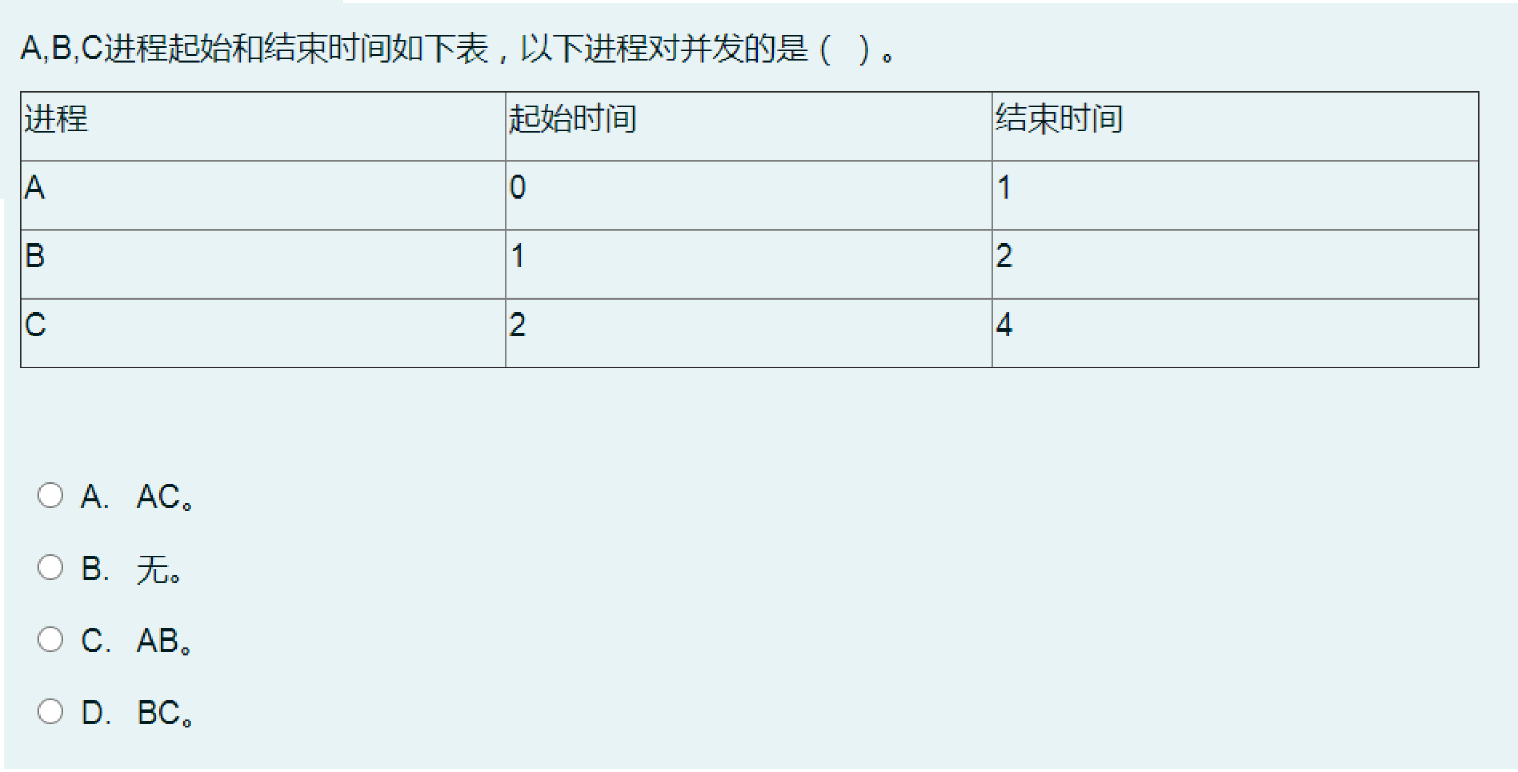


并发进程采用中断方式启用异常控制流。

选择一项：

○ 对

○ 错



A,B,C进程起始和结束时间如下表，以下进程对并发的是（　）。

进程	起始时间	结束时间
A	0	1
B	1	2
C	2	4

○ A. AC。

○ B. 无。

○ C. AB。

○ D. BC。

虚拟内存

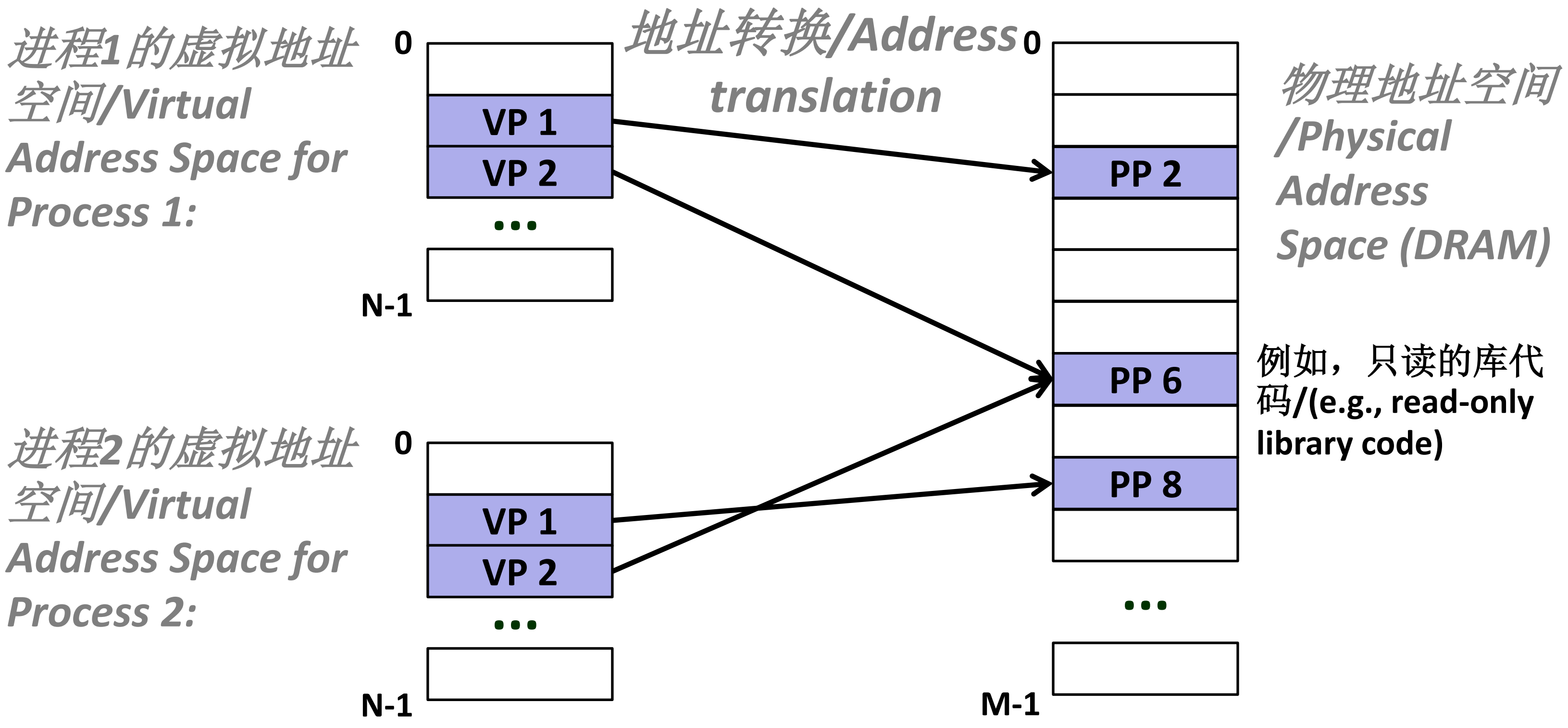
- 地址空间与虚拟内存
  - 物理地址、虚拟地址
- 虚拟内存与物理内存缓存机制
  - 页、页表、页命中、缺页处理、分配页、局域性
- 虚拟内存与内存管理与保护
  - 链接、加载、页表记录
- 地址翻译
  - 页命中、缺页处理地址翻译
  - 缓存和虚拟内存结合的地址翻译
  - 快表、多级页表

虚拟内存

- 缓存与虚拟内存的系统示例
- i7与Linux虚拟内存
  - 地址翻译、多级页表记录、进程虚拟空间
- 内存映射
  - 共享对象、fork和execve、用户内存映射
- 动态内存分配
  - 分配器、性能、碎片
  - 隐式链表、显式链表LIFO、分离空闲表
  - 垃圾回收
- 内存相关问题

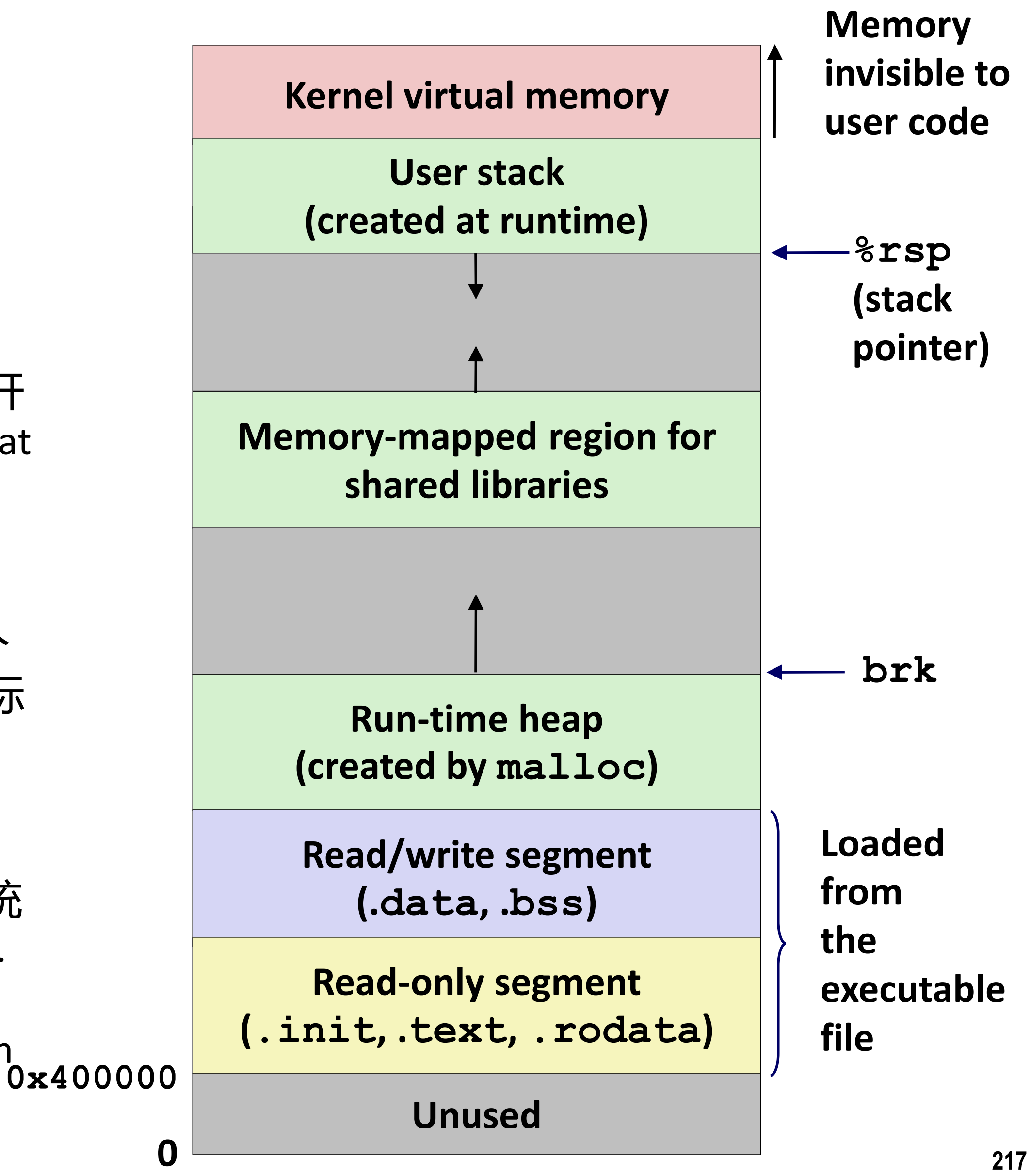
基于虚拟内存的内存管理机制/VM as a Tool for Memory Management

- **关键点：每个进程有自己的虚拟地址空间/Key idea: each process has its own virtual address space**
  - 将内存看做简单的线性数组/It can view memory as a simple linear array
  - 映射函数将地址分散到物理内存中/Mapping function scatters addresses through physical memory
    - 好的映射函数会提高局域性/Well-chosen mappings can improve locality


进程1的虚拟地址空间/Virtual Address Space for Process 1:
0
VP 1
VP 2
...
N-1
地址转换/Address translation
0
PP 2
PP 6
PP 8
...
M-1
物理地址空间/Physical Address Space (DRAM)
例如，只读的库代码/(e.g., read-only library code)
进程2的虚拟地址空间/Virtual Address Space for Process 2:
0
VP 1
VP 2
...
N-1

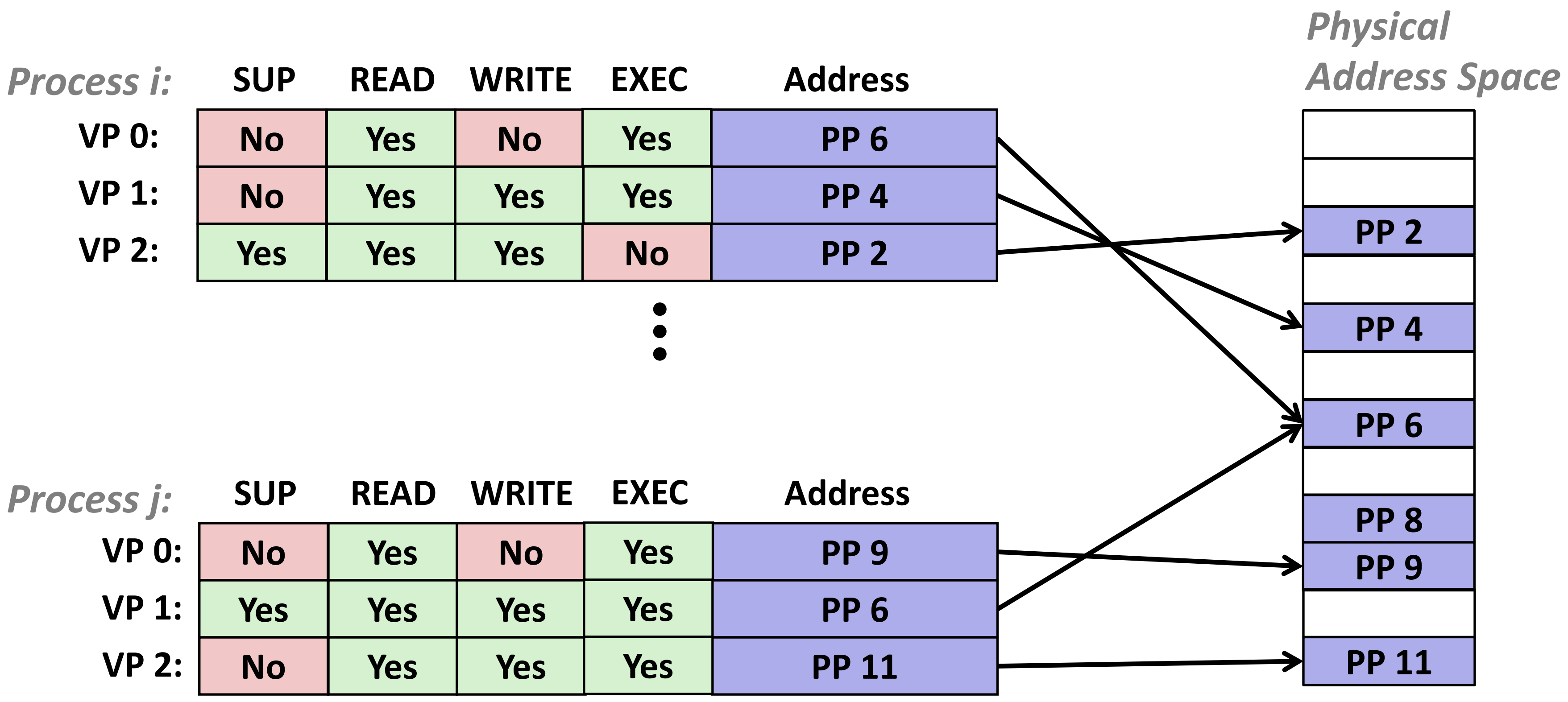
简化链接和加载/Simplifying Linking and Loading

- 链接/Linking
  - 每个程序都有类似的虚拟地址空间/Each program has similar virtual address space
  - 代码、数据和堆总是从相同的地址开始/Code, data, and heap always start at the same addresses.
- 加载/Loading
  - **execve**负责为`.text`和`.data`段分配虚拟页并创建页表记录，并将其标记位非法/**execve** allocates virtual pages for .text and .data sections & creates PTEs marked as invalid
  - .text和.data中的页是由虚拟内存系统按需拷贝的/The **.text** and **.data** sections are copied, page by page, on demand by the virtual memory system


Kernel virtual memory
Memory invisible to user code
User stack (created at runtime)
%rsp (stack pointer)
Memory-mapped region for shared libraries
brk
Run-time heap (created by malloc)
Read/write segment (.data, .bss)
Read-only segment (.init, .text, .rodata)
Loaded from the executable file
0x400000
Unused
0

基于虚拟内存的内存保护机制/VM as a Tool for Memory Protection

- 对页表记录进行扩展增加权限位/Extend PTEs with permission bits
- MMU在每个内存访问时检查/MMU checks these bits on each access



Process i:

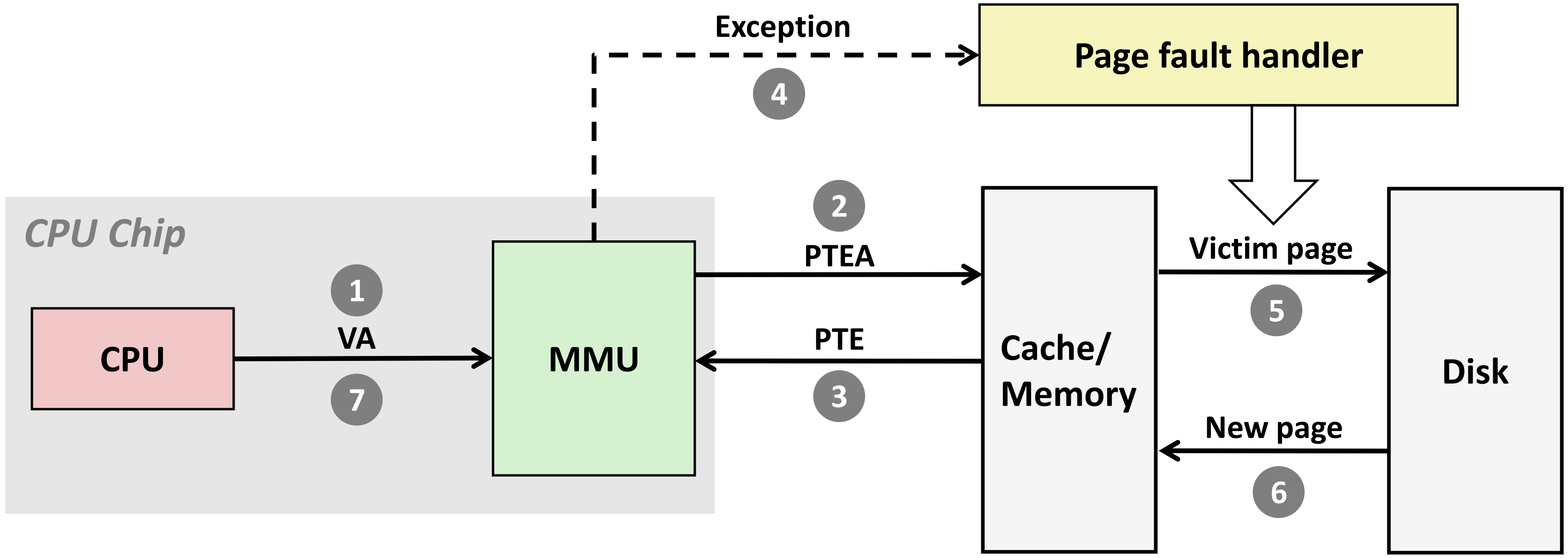
	SUP	READ	WRITE	EXEC	Address
VP 0:	No	Yes	No	Yes	PP 6
VP 1:	No	Yes	Yes	Yes	PP 4
VP 2:	Yes	Yes	Yes	No	PP 2

⋮

Process j:

	SUP	READ	WRITE	EXEC	Address
VP 0:	No	Yes	No	Yes	PP 9
VP 1:	Yes	Yes	Yes	Yes	PP 6
VP 2:	No	Yes	Yes	Yes	PP 11

地址翻译：缺页中断/Address Translation: Page Fault



1) 处理器将虚拟地址发给MMU/Processor sends virtual address to MMU

2-3)MMU从内存中的页表取出页表记录/ MMU fetches PTE from page table in memory

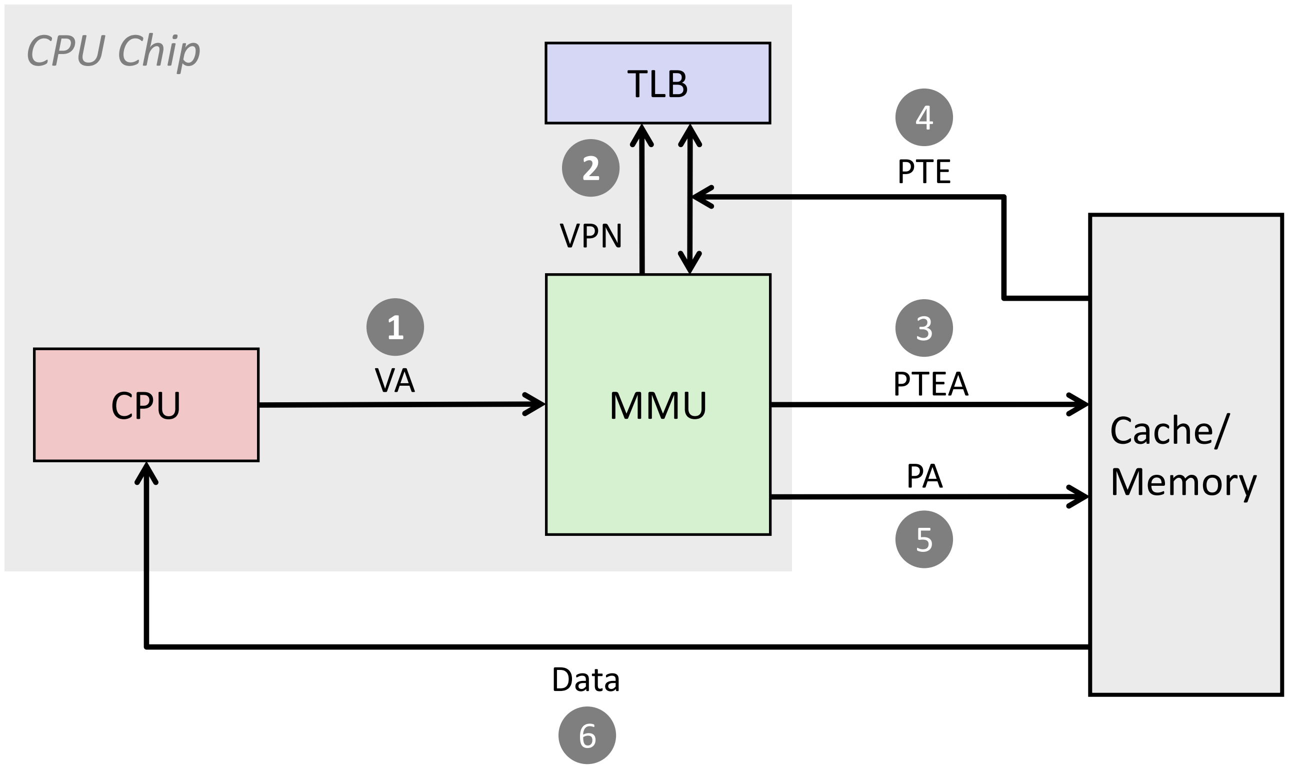
4) 当合法位为0时MMU触发缺页中断异常/Valid bit is zero, so MMU triggers page fault exception

5) 异常处理程序找到一个换出页（如果是脏页则要写回磁盘）/Handler identifies victim (and, if dirty, pages it out to disk)

6) 异常处理程序拷贝页并更新页表记录/Handler pages in new page and updates PTE in memory

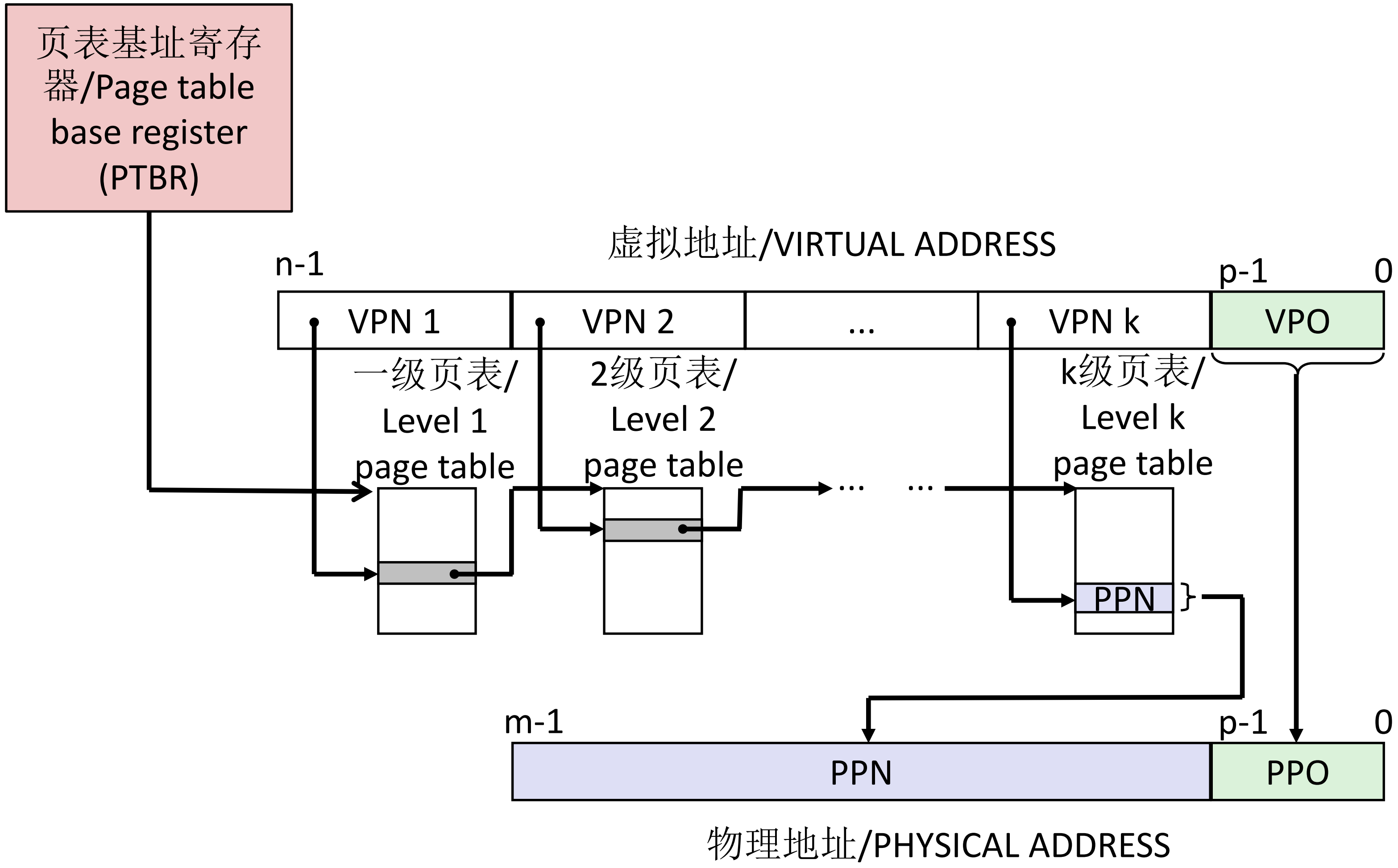
7)异常处理程序返回原进程中断的指令重新执行/Handler returns to original process, restarting faulting instruction

TLB丢失/TLB Miss



TLB丢失会导致一个额外的内存访问，幸运的是，TLB丢失很少发生
/A TLB miss incurs an additional memory access (the PTE)
Fortunately, TLB misses are rare. Why?

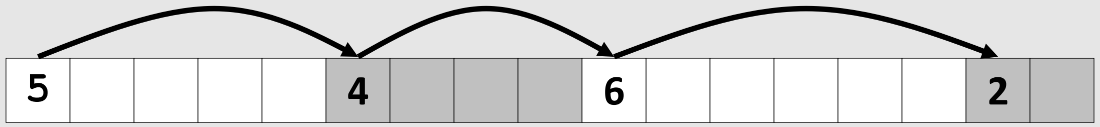
k级页表的地址翻译/Translating with a k-level Page Table



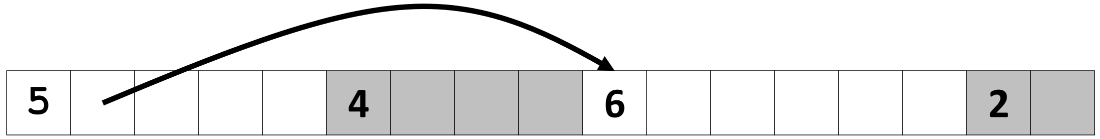
页表基址寄存器/Page table base register (PTBR)
虚拟地址/VIRTUAL ADDRESS
n-1
p-1
0
VPN 1
VPN 2
...
VPN k
VPO
一级页表/ Level 1 page table
2级页表/ Level 2 page table
k级页表/ Level k page table
... ...
PPN
m-1
p-1
0
PPN
PPO
物理地址/PHYSICAL ADDRESS

跟踪空闲块/Keeping Track of Free Blocks

- **方法1：隐式列表-使用长度链接所有块/Method 1: *Implicit list* using length—links all blocks**



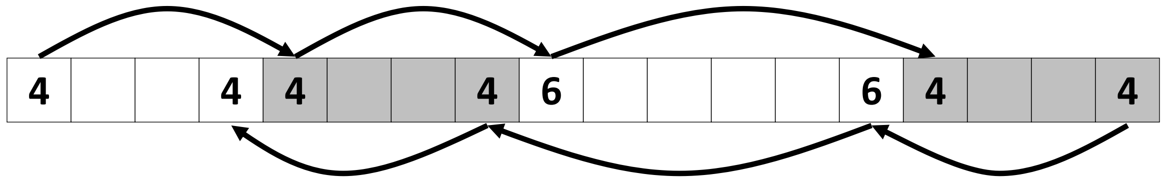
- **方法2：空闲块之间使用指针的显式列表/Method 2: *Explicit list* among the free blocks using pointers**

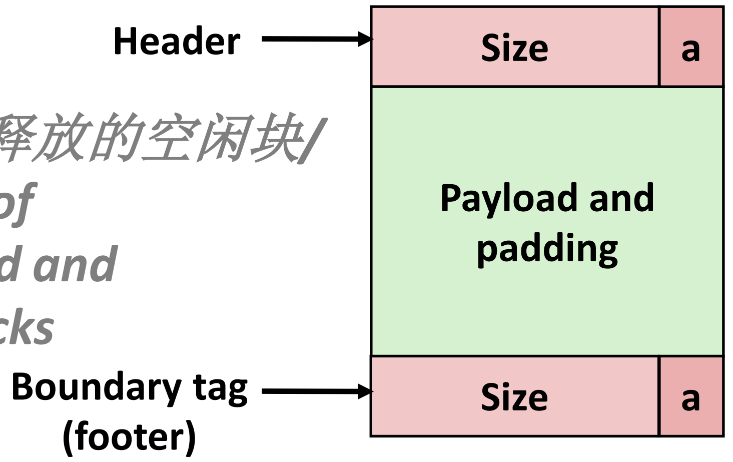


- **方法3：分离的空闲列表/Method 3: *Segregated free list***
  - 不同大小块使用不同的空闲列表/Different free lists for different size classes
- **方法4：根据大小对块排序/Method 4: *Blocks sorted by size***
  - 可以使用一个平衡树（红黑树）Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

隐式列表：双向合并/Implicit List: Bidirectional Coalescing

- *边界标记/Boundary tags* [Knuth73]
  - 在空闲块结束的位置重复标记大小/已分配字/Replicate size/allocated word at “bottom” (end) of free blocks
  - 以额外的空间换取反向遍历列表功能/Allows us to traverse the “list” backwards, but requires extra space
  - 重要和常用的技术/Important and general technique!





分配和释放的空闲块/ Format of allocated and free blocks

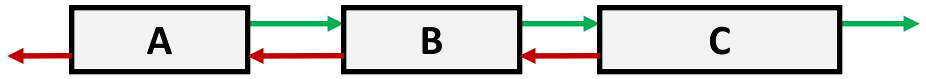
a = 1: Allocated block /分配的块
a = 0: Free block/空闲块

Size: Total block size/总体块大小

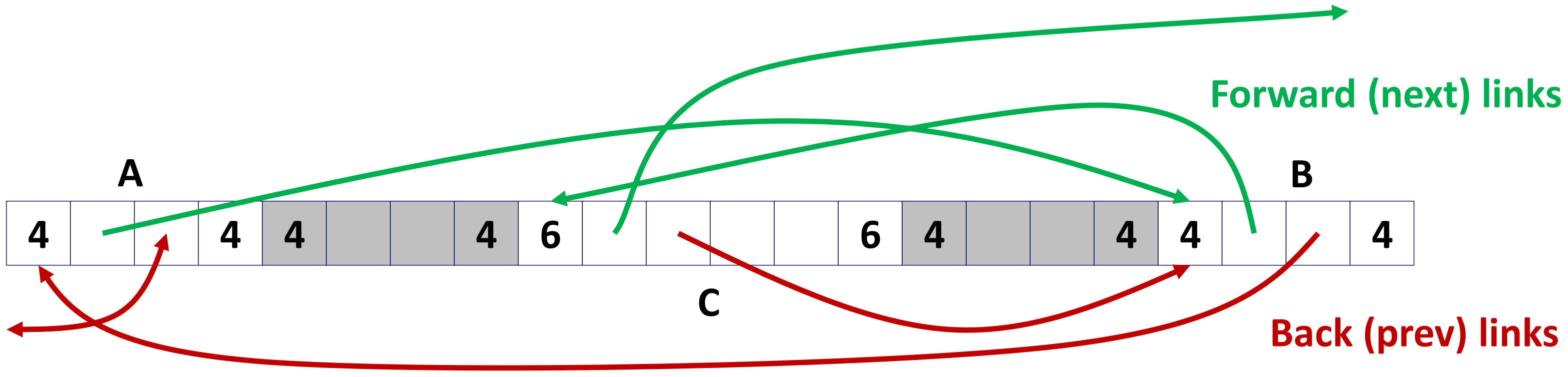
Payload: Application data (allocated blocks only)/应用程序数据

显式空闲列表/Explicit Free Lists

- 逻辑上/Logically:

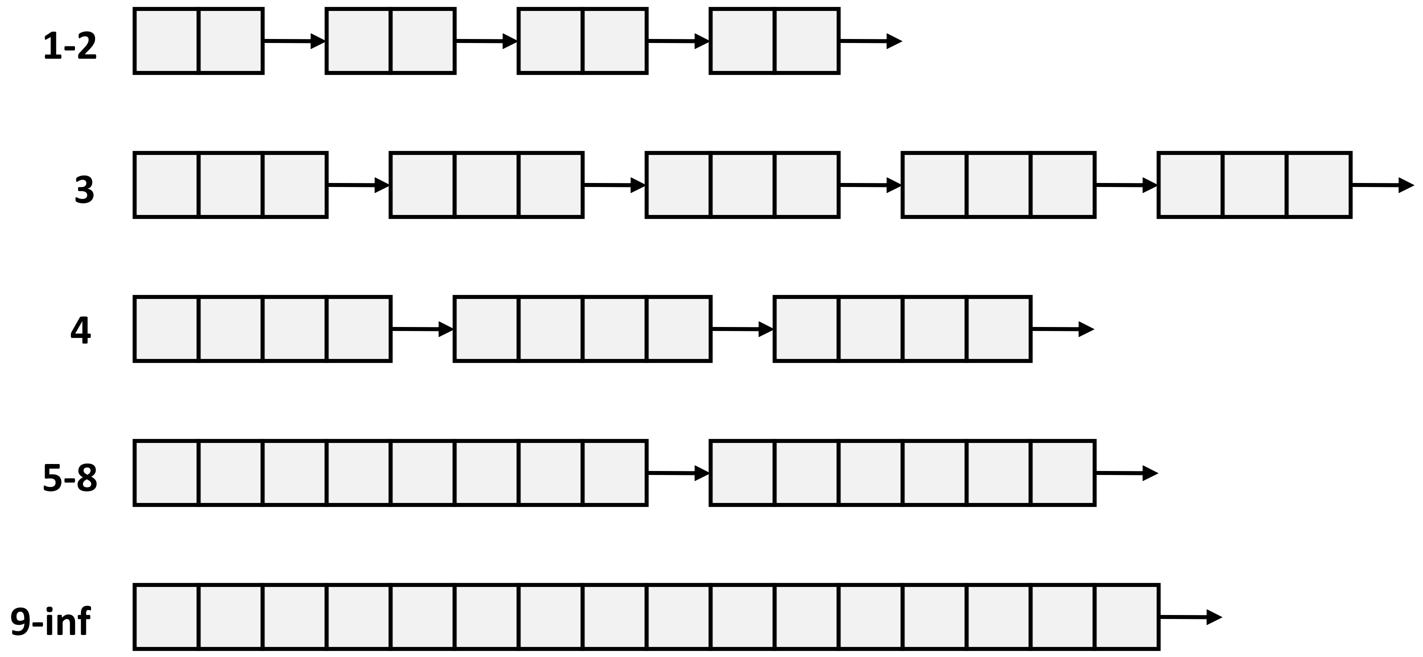


- 物理上：块可能是任意顺序/Physically: blocks can be in any order



分离空闲列表分配器/Segregated List (Seglist) Allocators

- 每个不同大小类块有自己的空闲列表/Each *size class* of blocks has its own free list



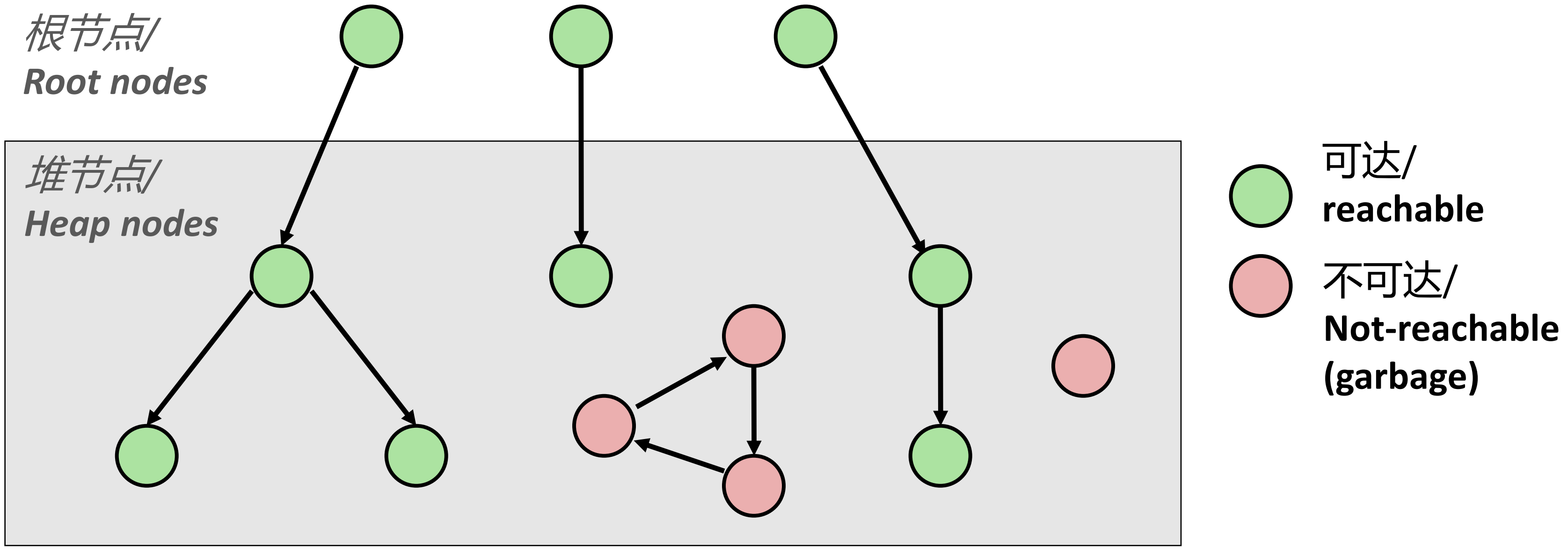
- 通常比较小的块有自己单独的类/Often have separate classes for each small size
- 对于比较大的块，每个2的指数区间有一个类/For larger sizes: One class for each two-power size

经典垃圾收集算法/Classical GC Algorithms

- **标记清除算法/Mark-and-sweep collection (McCarthy, 1960)**
  - 不需要移动内存块（除非需要平移压紧占用部分）/Does not move blocks (unless you also “compact”)
- **引用计数算法/Reference counting (Collins, 1960)**
  - 不需要移动内存块/Does not move blocks (not discussed)
- **拷贝收集算法/Copying collection (Minsky, 1963)**
  - 需要移动内存块/Moves blocks (not discussed)
- **按代垃圾收集算法/Generational Collectors (Lieberman and Hewitt, 1983)**
  - 基于生命周期的收集/Collection based on lifetimes
    - 大部分内存块很快变为垃圾/Most allocations become garbage very soon
    - 主要聚焦在最近分配的区域内开展回收工作/So focus reclamation work on zones of memory recently allocated
- **For more information:**
  Jones and Lin, “*Garbage Collection: Algorithms for Automatic Dynamic Memory*”, John Wiley & Sons, 1996.

将内存当做一个图/Memory as a Graph

- 我们将内存看做一个有向图/**We view memory as a directed graph**
  - 每个块是图中的一个节点/Each block is a node in the graph
  - 每个指针是图中的一条边/Each pointer is an edge in the graph
  - 不在堆中但是持有指向堆中指针的位置称为根节点（例如，寄存器，栈中元素，以及全局变量）/Locations not in the heap that contain pointers into the heap are called ***root*** nodes (e.g. registers, locations on the stack, global variables)

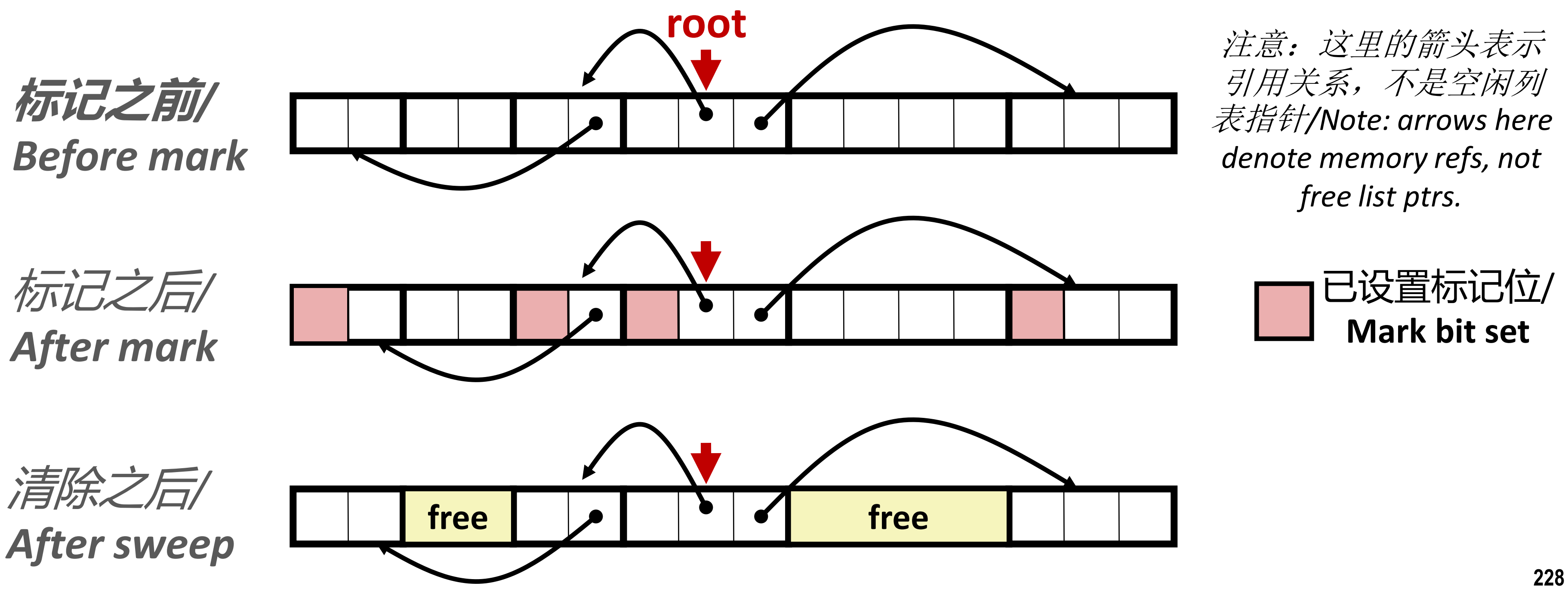


如果有从根节点到某个节点的路径则这个节点是可达的/A node (block) is *reachable* if there is a path from any root to that node.

不可达的都是垃圾（应用程序不再需要）/Non-reachable nodes are *garbage* (cannot be needed by the application)

标记清除收集算法/Mark and Sweep Collecting

- **可以基于malloc/free操作构建/Can build on top of malloc/free package**
  - 一直使用malloc直到空间不够用/Allocate using `malloc` until you “run out of space”
- **什么时候内存不够用/When out of space:**
  - 在每个块的头部使用额外的标记位/Use extra ***mark bit*** in the head of each block
  - ***Mark:*** 从根节点开始并对所有可达节点设置标记位/Start at roots and set mark bit on each reachable block
  - ***Sweep:*** 扫描所有的块并清除未标记的块/Scan all blocks and free blocks that are not marked



注意：这里的箭头表示引用关系，不是空闲列表指针/Note: arrows here denote memory refs, not free list ptrs.

内存相关的风险和陷阱/Memory-Related Perils and Pitfalls

- 解引问题指针/Dereferencing bad pointers
- 使用未初始化内存/Reading uninitialized memory
- 覆盖内存/Overwriting memory
- 引用不存在的变量/Referencing nonexistent variables
- 重复释放内存块/Freeing blocks multiple times
- 引用释放的内存/Referencing freed blocks
- 释放内存失败/Failing to free blocks

例题1

- 以下程序存在的问题是：

```
char s[8];
int i;

gets(s);  /* reads "123456789" from stdin */
```

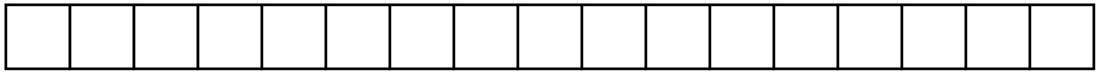
  - A：没有检查最大字符串长度
  - B：使用未初始化内存
  - C：引用不存在的变量
  - D：引用释放的内存
- 答案解析：**A**

例题2

- 多选题。垃圾收集算法中，根对象可能来自于：
  - A：寄存器
  - B：内存
  - C：stack
  - D：磁盘
- 答案解析：**A、C**

例题3

- 单选题。对于如下的内存空间和内存分配与释放序列，则以下说法正确的是：
  - A：p3分配会失败
  - B：p4分配会失败
  - C：p5分配会失败
  - D：没有分配会失败



```
p1 = malloc(4)
p2 = malloc(5)
p3 = malloc(6)
free(p2)
p4 = malloc(2)
p5 = malloc(4)
```

- 答案：**C**

例题4

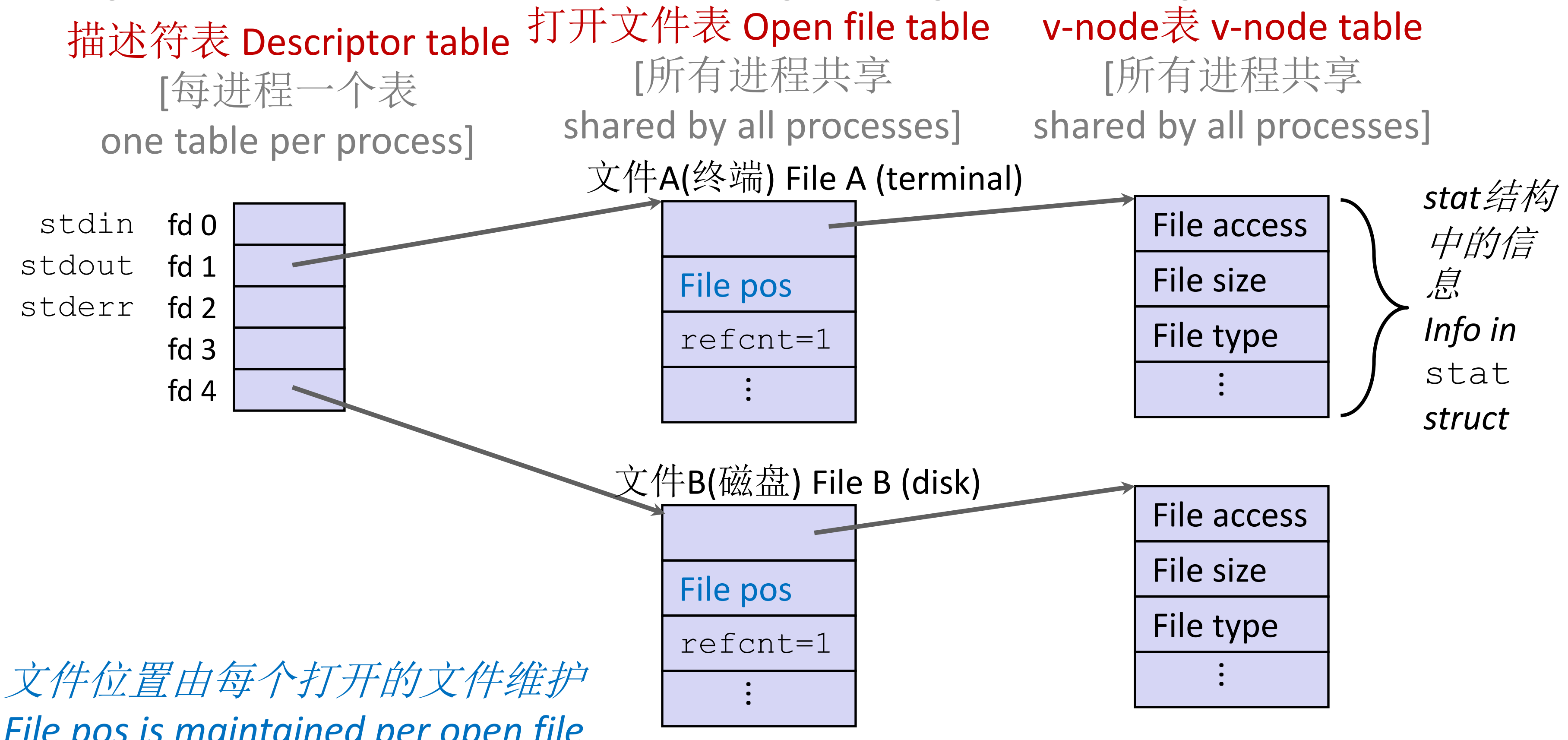
- 判断题。多个不同的进程可能会访问同一个物理内存页
- 答案：对。内核空间，动态链接库等

系统级I/O

- 文件与I/O
- RIO
- 读文件与目录
- 共享文件
- I/O重定向
- I/O标准库与函数

Unix内核如何表示打开文件
How the Unix Kernel Represents Open Files

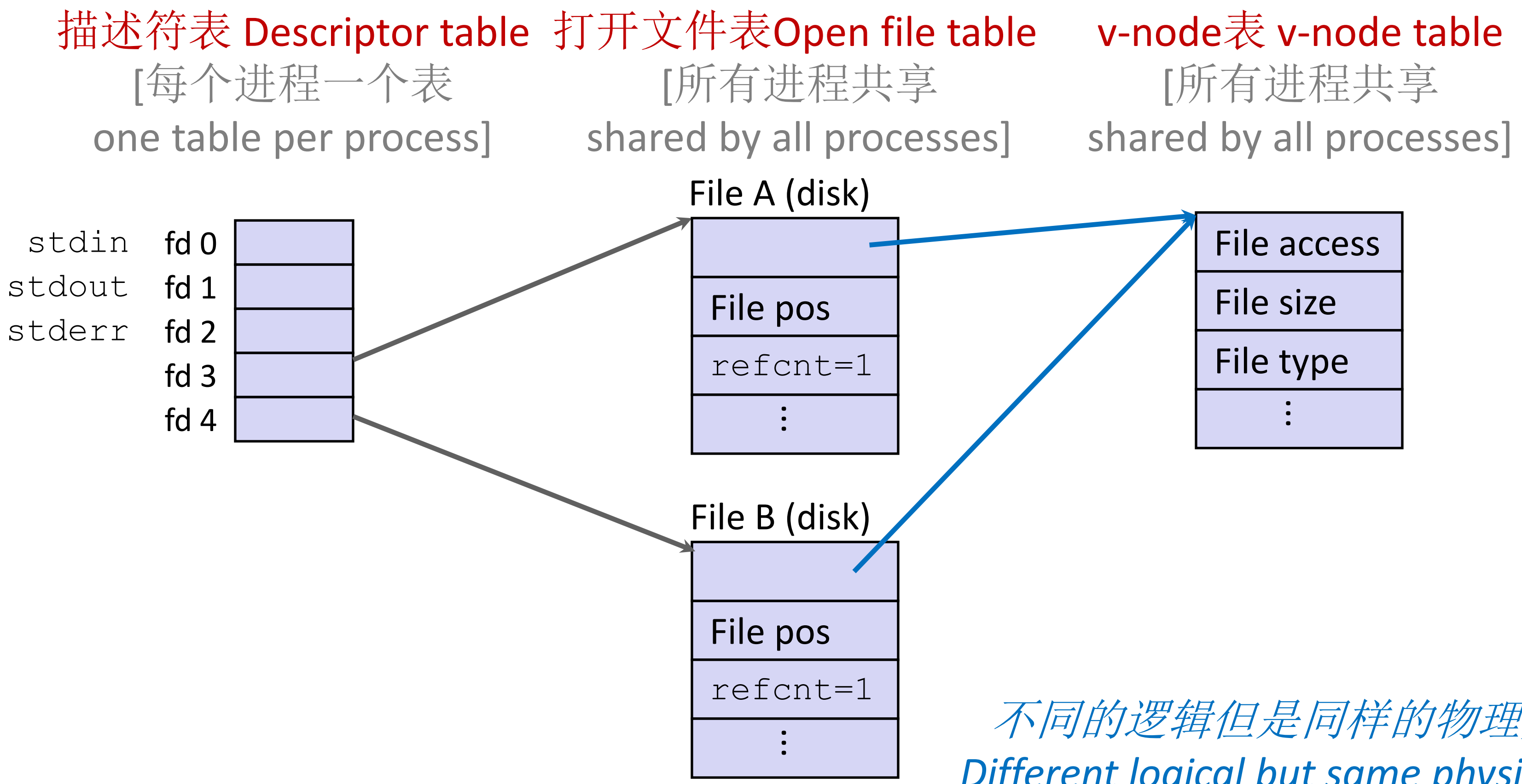
- 两个描述符引用两个不同的打开文件。描述符1（标准输出）指向终端，描述符4指向打开的磁盘文件 **Two descriptors referencing two distinct open files. Descriptor 1 (`stdout`) points to terminal, and descriptor 4 points to open disk file**



文件位置由每个打开的文件维护
File pos is maintained per open file

文件共享 File Sharing

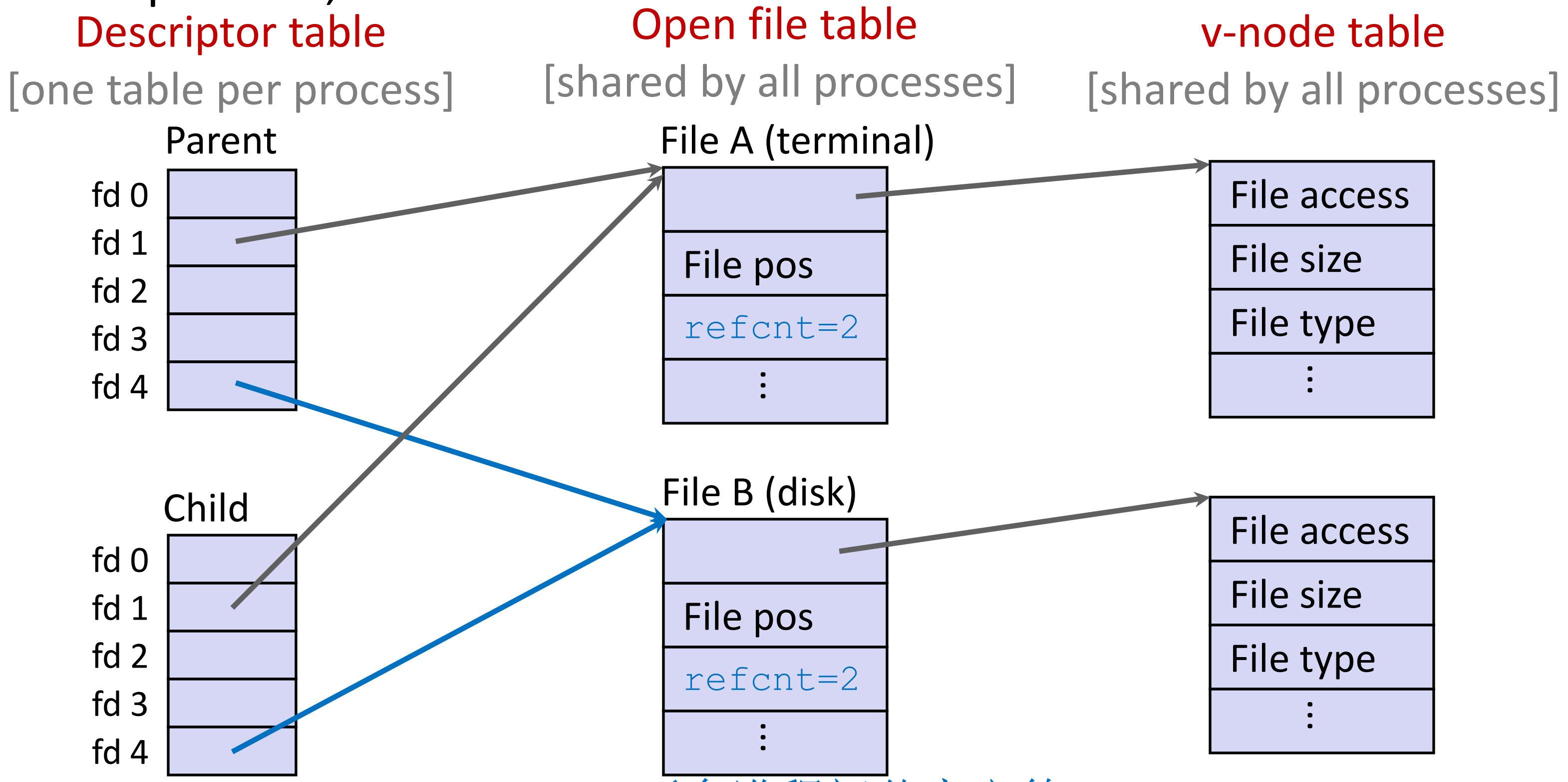
- 两个不同的描述符通过两个不同的打开文件表共享磁盘文件 **Two distinct descriptors sharing the same disk file through two distinct open file table entries**
  - 例如调用**open**两次，两次的**文件名**参数相同 e.g., calling **`open`** twice with the same **`filename`** argument



不同的逻辑但是同样的物理文件
Different logical but same physical file

进程如何共享文件：fork
How Processes Share Files: `fork`

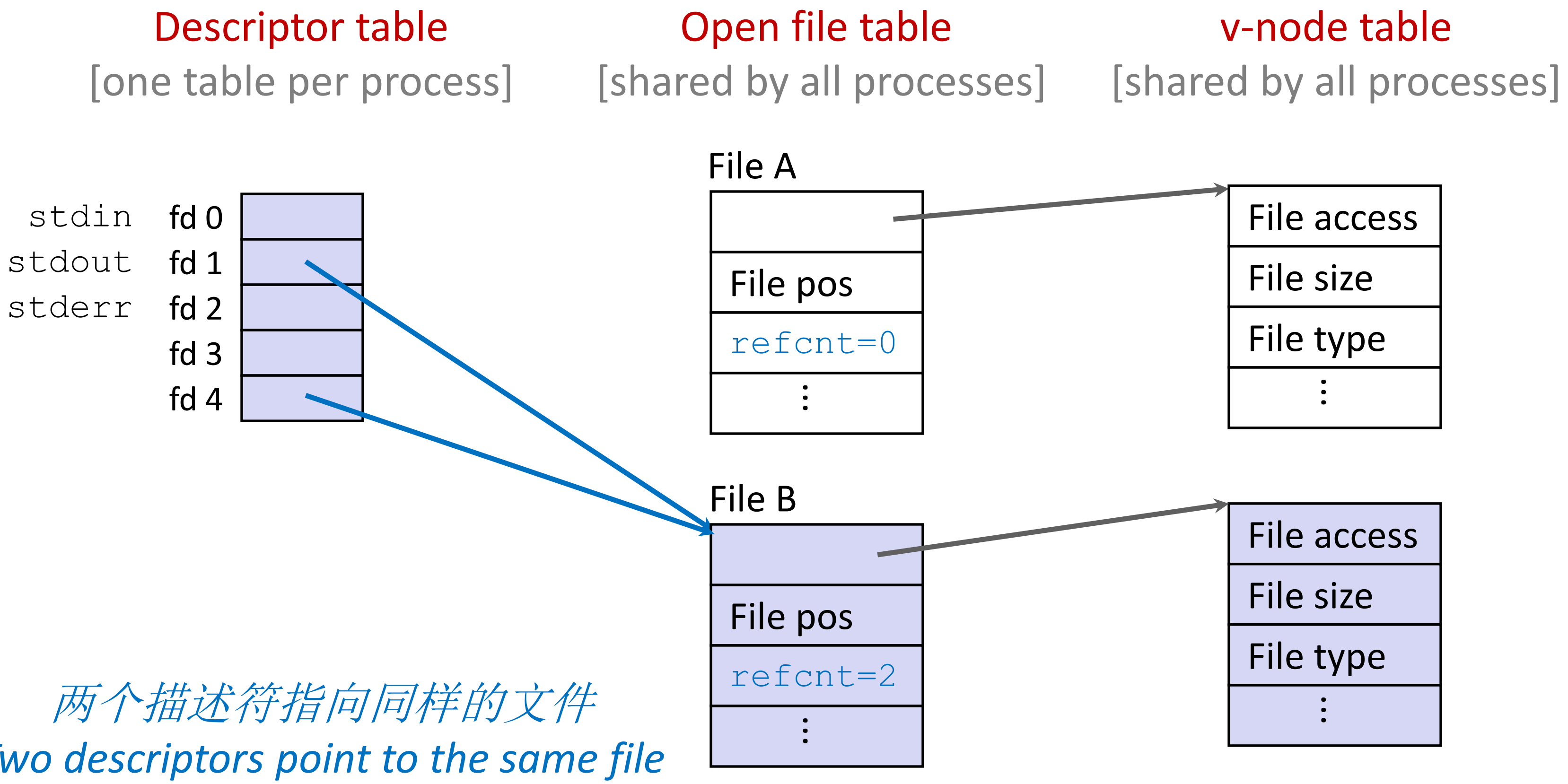
- 子进程继承其父进程的打开文件 **A child process inherits its parent's open files**
- **调用fork***之后* ***After*** **`fork`:**
  - 子进程与父进程的表一样，而且每个**refcnt**加一 Child's table same as parent's, and +1 to each **`refcnt`**



两个进程间共享文件 File is shared between processes

输入/输出重定向示例
I/O Redirection Example (cont.)

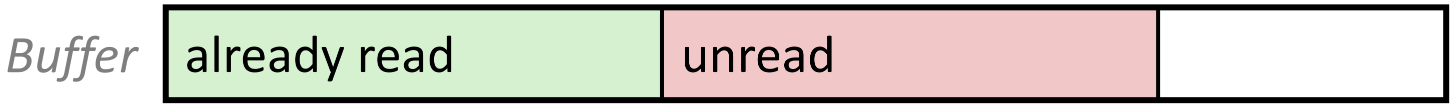
- 步骤#2：调用dup2(4,1) Step #2: call `dup2(4,1)`
  - 导致fd=1（标准输出）指向fd=4所指向的磁盘文件 Cause fd=1 (**`stdout`**) to refer to disk file pointed at by fd=4



两个描述符指向同样的文件
Two descriptors point to the same file

带缓冲的输入/输出：动机
Buffered I/O: Motivation

- **应用程序通常一次读/写一个字符 Applications often read/write one character at a time**
  - **`getc, putc, ungetc`**
  - **`gets, fgets`**
    - 读文本行，一次读一个字符，直到读到换行符为止 Read line of text one character at a time, stopping at newline
- **实现为Unix I/O调用比较费时 Implementing as Unix I/O calls expensive**
  - **读**和**写**需要`Unix`内核调用 **`read`** and **`write`** require Unix kernel calls
    - 大于1万个时钟周期 > 10,000 clock cycles



缓冲RIO输入函数 Buffered RIO Input Functions

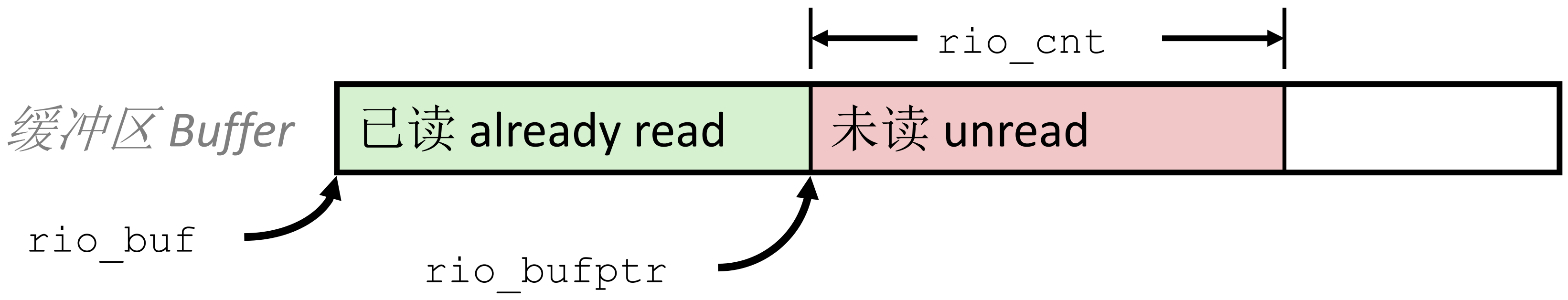
- 从部分缓存在内部内存缓冲区中的文件中有效读取文本行和二进制数据 **Efficiently read text lines and binary data from a file partially cached in an internal memory buffer**

```
#include "csapp.h"

void rio_readinitb(rio_t *rp, int fd);

ssize_t rio_readlineb(rio_t *rp, void *usrbuf, size_t maxlen);
ssize_t rio_readnb(rio_t *rp, void *usrbuf, size_t n);
                                返回：如果OK为读取字节数，遇到EOF为0，出错为-1
                                Return: num. bytes read if OK, 0 on EOF, -1 on error
```

- **`rio_readlineb`**从文件**`fd`**中读取最多**`maxlen`**字节的*文本行*，并将该行存储在**`usrbuf`**中 **`rio_readlineb`** reads a ***text line*** of up to **`maxlen`** bytes from file **`fd`** and stores the line in **`usrbuf`**
  - 特别适用于从网络套接字读取文本行 Especially useful for reading text lines from network sockets
- 停止条件 Stopping conditions
  - 读取到最大字节数 **`maxlen`** bytes read
  - 遇到EOF EOF encountered
  - 遇到换行符（“**`\n`**”） Newline (‘**`\n`**’) encountered

缓冲I/O：声明 Buffered I/O: Declaration

- 结构包含所有信息 **All information contained in `struct`**

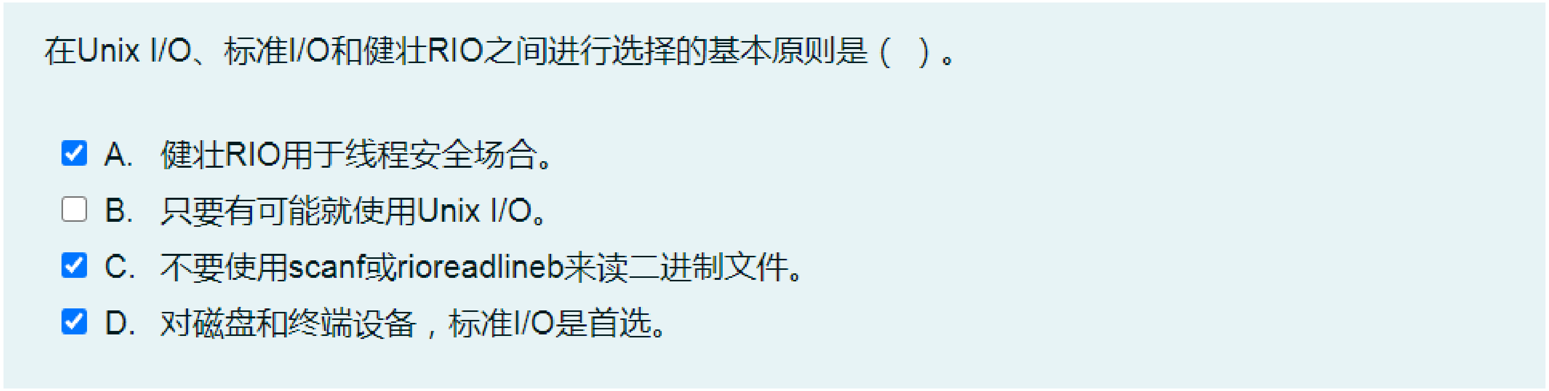


```
typedef struct {
    int rio_fd;                /* descriptor for this internal buf */
    int rio_cnt;               /* unread bytes in internal buf */
    char *rio_bufptr;          /* next unread byte in internal buf */
    char rio_buf[RIO_BUFSIZE]; /* internal buffer */
} rio_t;
```

选择输入/输出函数 Choosing I/O Functions

- **一般规则：使用您可以使用的最高级别I/O函数 General rule: use the highest-level I/O functions you can**
  - 许多C程序员能够使用标准I/O函数完成所有工作 Many C programmers are able to do all of their work using the standard I/O functions
  - 但是，一定要理解使用的函数！ But, be sure to understand the functions you use!
- **何时使用标准I/O When to use standard I/O**
  - 使用磁盘或终端文件时 When working with disk or terminal files
- **何时使用原始Unix I/O When to use raw Unix I/O**
  - *信号处理程序内部，因为Unix I/O是异步信号安全的Inside signal handlers, because Unix I/O is async-signal-safe*
  - 在极少数情况下，当需要绝对最高的性能时 In rare cases when you need absolute highest performance
- **何时使用RIO When to use RIO**
  - *当读写网络套接字时When you are reading and writing network sockets*
  - 避免对套接字使用标准I/O Avoid using standard I/O on sockets

例题



在Unix I/O、标准I/O和健壮RIO之间进行选择的基本原则是（ ）。

[✓ A.] 健壮RIO用于线程安全场合。

B. 只要有可能就使用Unix I/O。

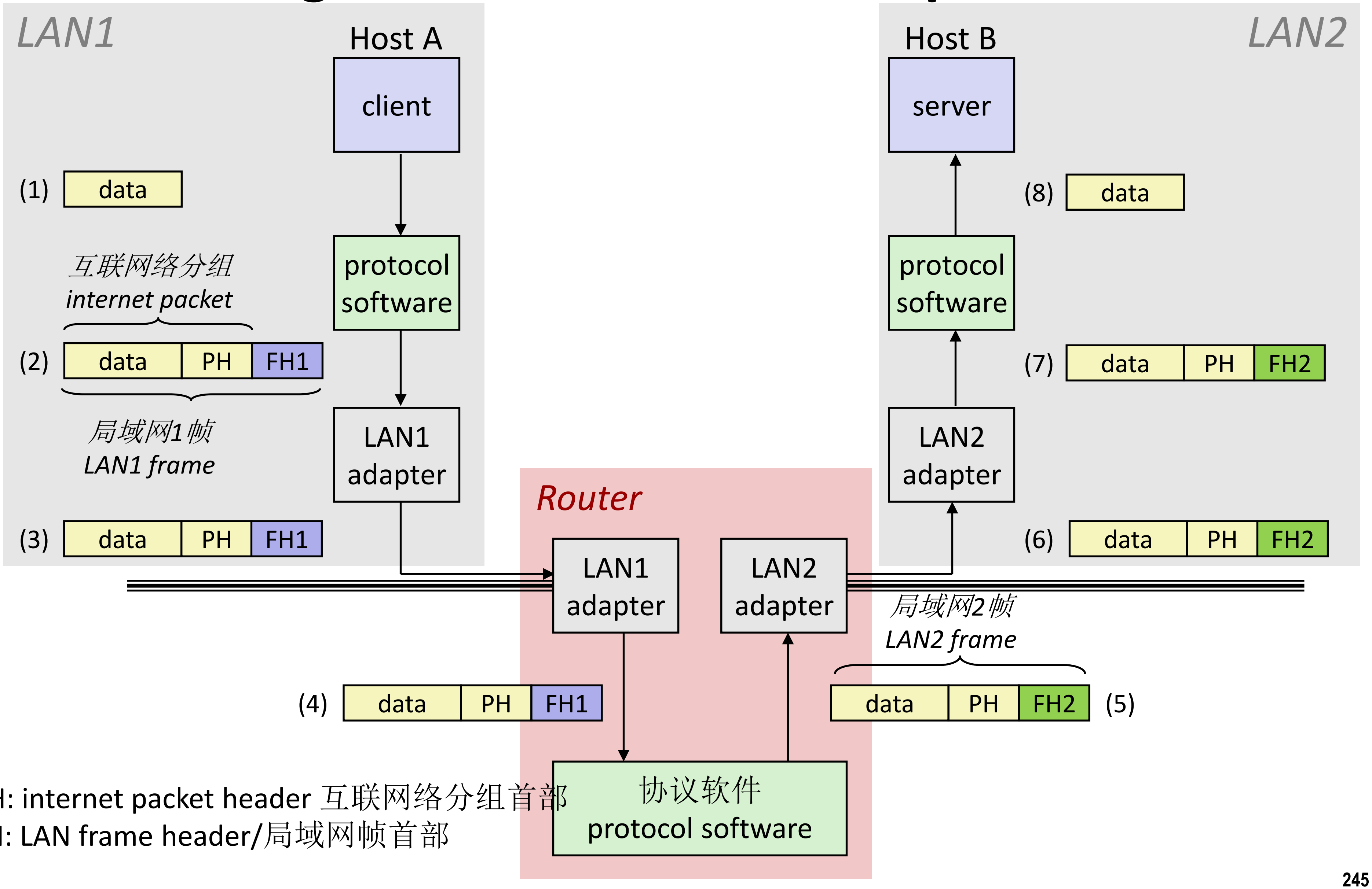
[✓ C.] 不要使用scanf或rioreadlineb来读二进制文件。

[✓ D.] 对磁盘和终端设备，标准I/O是首选。

网络编程

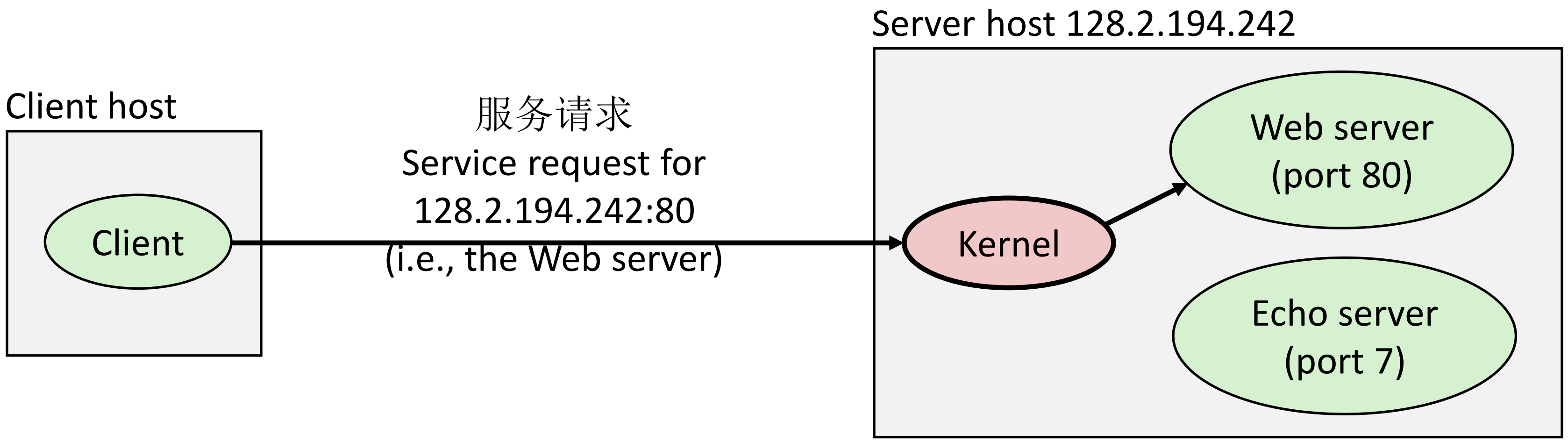
- C/S模型
- 网络基础
- 套接字
- WEB服务器

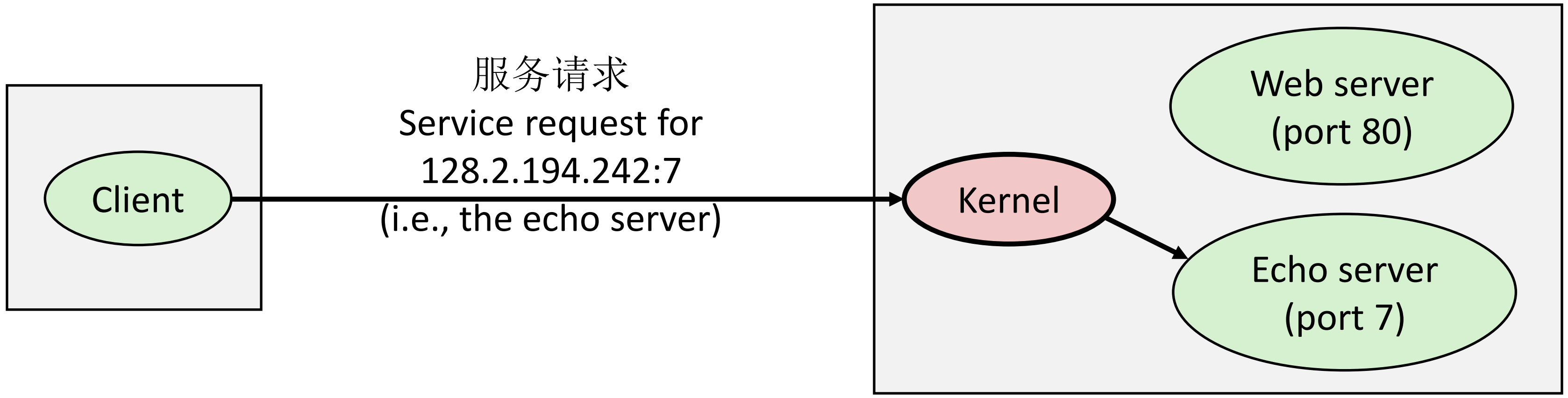
通过封装传输互联网络（网际）数据
Transferring internet Data Via Encapsulation



PH: internet packet header 互联网络分组首部
FH: LAN frame header/局域网帧首部

使用端口标识服务
Using Ports to Identify Services

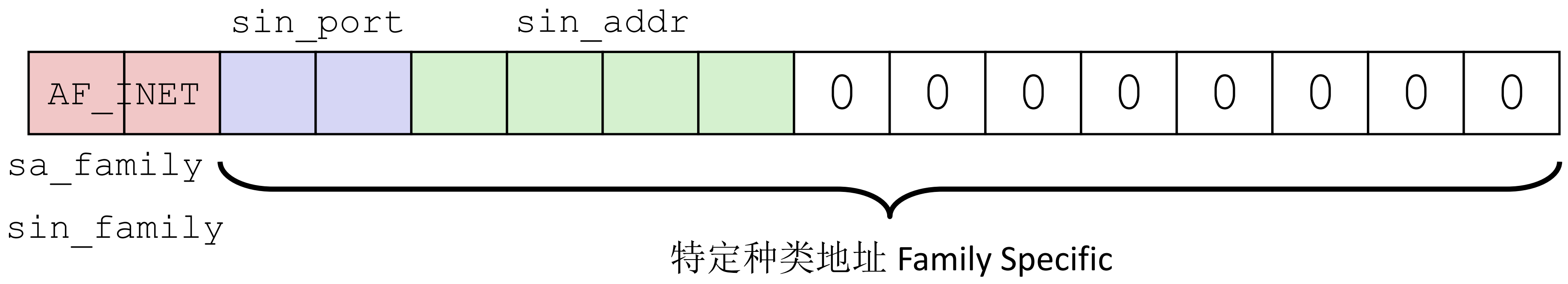




套接字地址结构 Socket Address Structures

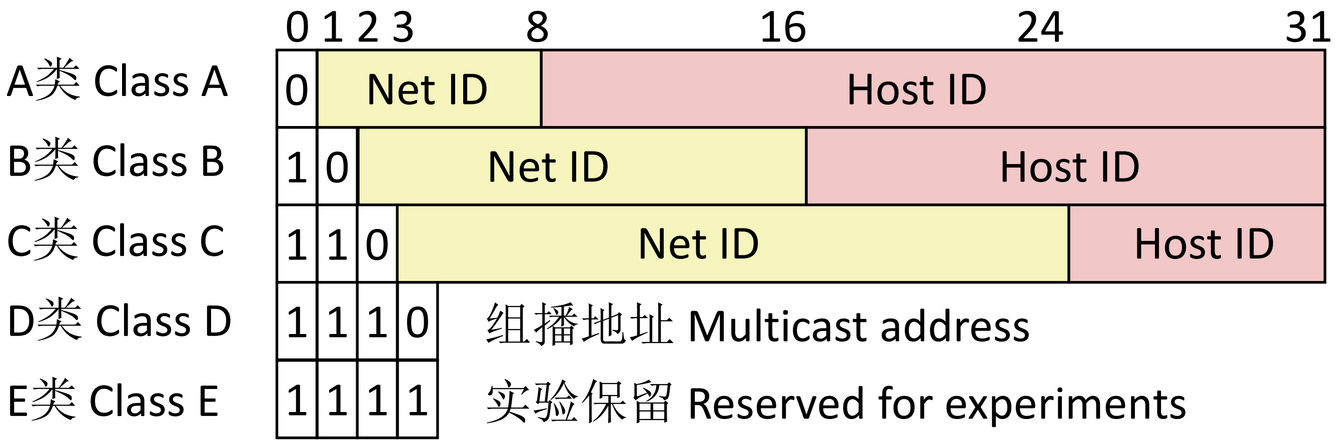
- **互联网（IPv4）特定套接字地址 Internet (IPv4) specific socket address:**
  - 必须强制转换，在函数使用套接字地址参数的时候 Must cast (`struct sockaddr_in *`) to (`struct sockaddr *`) for functions that take socket address arguments.

```
struct sockaddr_in  {
  uint16_t        sin_family;  /* Protocol family (always AF_INET) */
  uint16_t        sin_port;    /* Port num in network byte order */
  struct in_addr  sin_addr;    /* IP addr in network byte order */
  unsigned char   sin_zero[8]; /* Pad to sizeof(struct sockaddr) */
};
```



IP地址结构 IP Address Structure

- **IP(V4)地址空间分成类：IP (V4) Address space divided into classes:**



- **网络号记为w.x.y.z/n形式 Network ID Written in form w.x.y.z/n**
  - n为主机地址部分的位数 n = number of bits in host address
  - 例如CMU记为： 128.2.0.0/16 E.g., CMU written as 128.2.0.0/16
    - B类地址 Class B address
- **不能路由的（私有）IP地址 Unrouted (private) IP addresses:**

  10.0.0.0/8 172.16.0.0/12 192.168.0.0/16

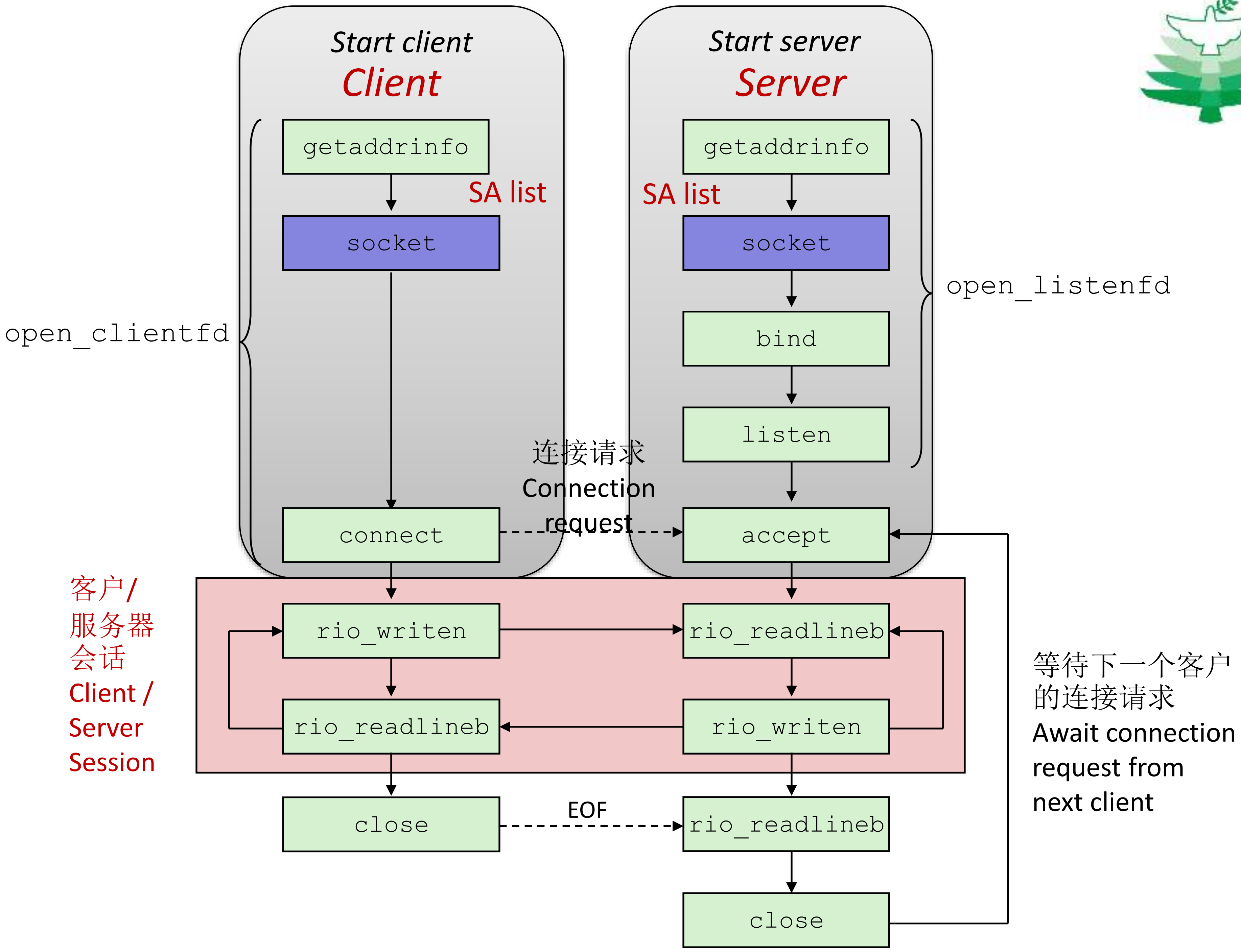
协议栈 Protocol Stacks

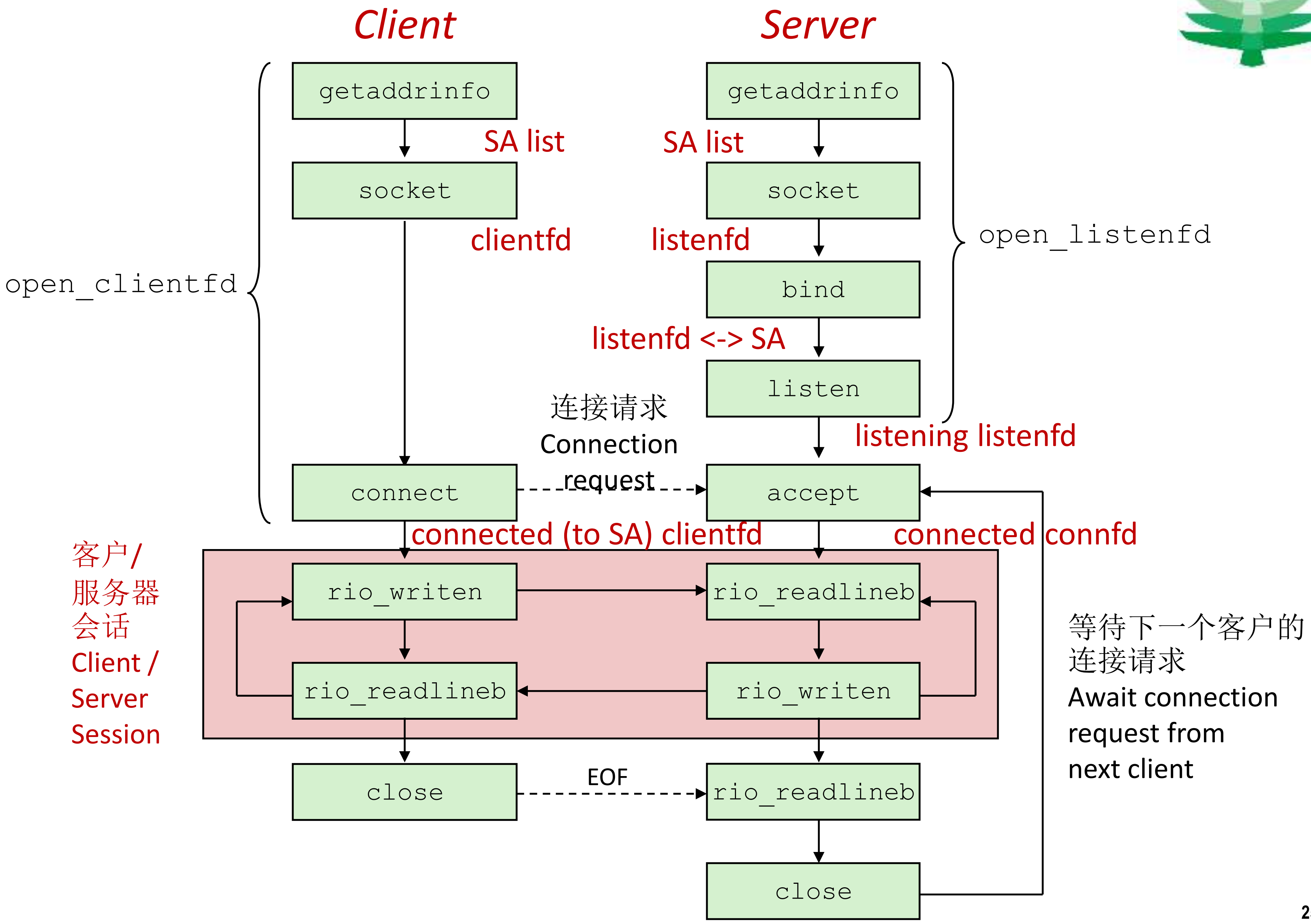
OSI模型 OSI Model

Application
Presentation
Session
Transport
Network
Data Link
Physical

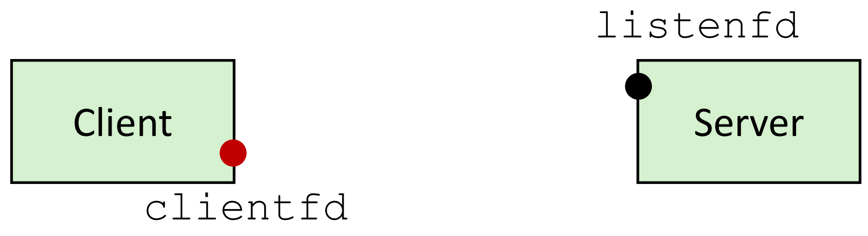
互联网模型 Internet Model

Application	HTTP	SMTP	SSH	DNS
Security	TLS			
Transport	TCP			UDP
Addressing	IP			
Physical Link	Ethernet	WiFi	SDH	


Start client
Client
getaddrinfo
SA list
socket
connect
open_clientfd
Start server
Server
getaddrinfo
SA list
socket
bind
listen
accept
open_listenfd
连接请求
Connection request
客户/服务器会话
Client / Server Session
rio_writen
rio_readlineb
rio_readlineb
rio_writen
close
EOF
rio_readlineb
close
等待下一个客户的连接请求
Await connection request from next client

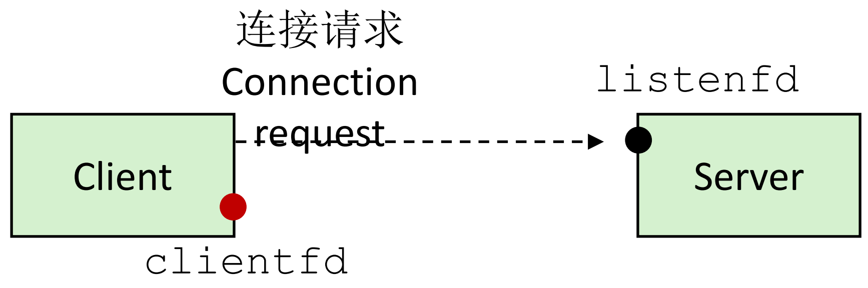

Client
Server
getaddrinfo
SA list
socket
clientfd
open_clientfd
getaddrinfo
SA list
socket
listenfd
bind
listenfd <-> SA
listen
open_listenfd
listening listenfd
连接请求
Connection request
connect
accept
connected (to SA) clientfd
connected connfd
客户/服务器会话
Client / Server Session
rio_writen
rio_readlineb
rio_readlineb
rio_writen
close
EOF
rio_readlineb
close
等待下一个客户的连接请求
Await connection request from next client

connect/accept Illustrated-说明



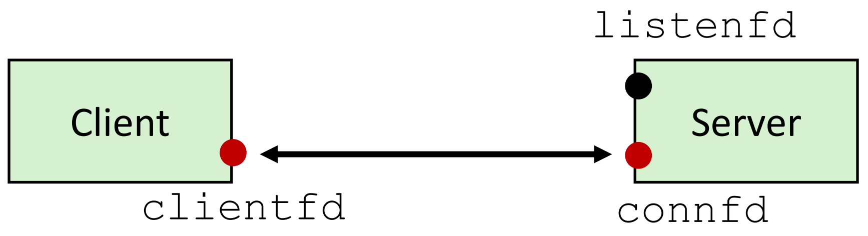
1.服务器阻塞在accept函数，等待侦听描述符listenfd上的连接请求

1. Server blocks in `accept`, waiting for connection request on listening descriptor `listenfd`



2.客户端通过阻塞式调用connect来发出连接请求

2. Client makes connection request by calling and blocking in `connect`

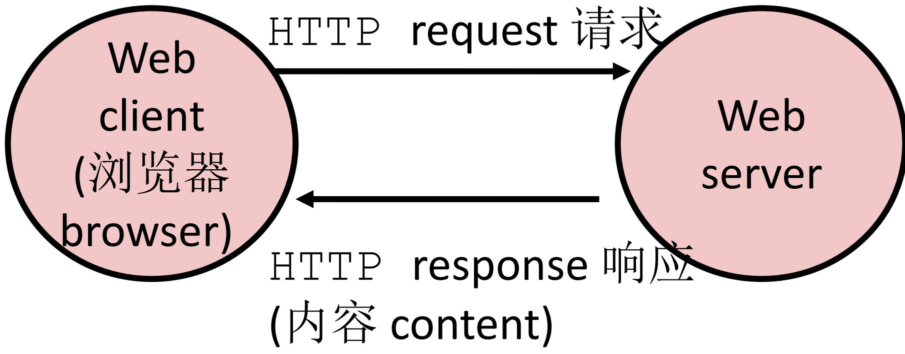


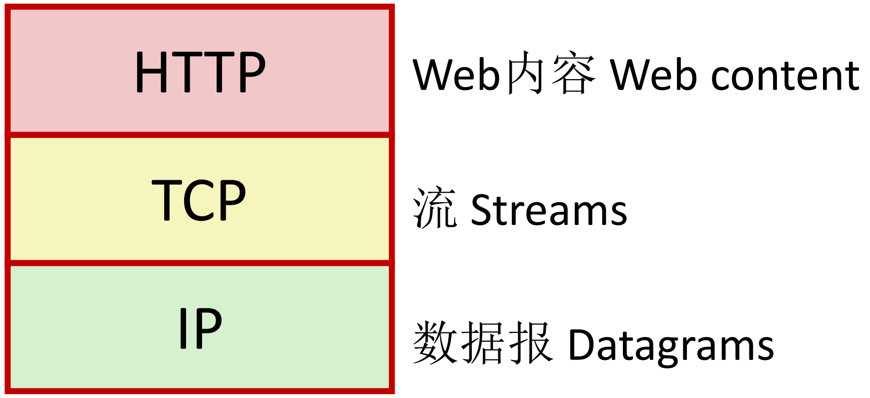
3.服务器从accept返回connfd，客户端从connect返回。现在已在clientfd和connfd之间建立连接

3. Server returns `connfd` from `accept`. Client returns from `connect`. Connection is now established between `clientfd` and `connfd`

Web服务器基础Web Server Basics

- **客户和服务器使用超文本传送协议（HTTP）通信 Clients and servers communicate using the HyperText Transfer Protocol (HTTP)**
  - 客户和服务器建立TCP连接 Client and server establish TCP connection
  - 客户请求内容 Client requests content
  - 服务器对请求内容进行响应 Server responds with requested content
  - 客户和服务器关闭连接（最终) Client and server close connection (eventually)
- **当前版本是HTTP/2.0，但是HTTP/1.1仍在广泛使用 Current version is HTTP/2.0 but HTTP/1.1 widely used still**
  - RFC 2616, June, 1999.


Web client (浏览器 browser)
HTTP request 请求
Web server
HTTP response 响应 (内容 content)

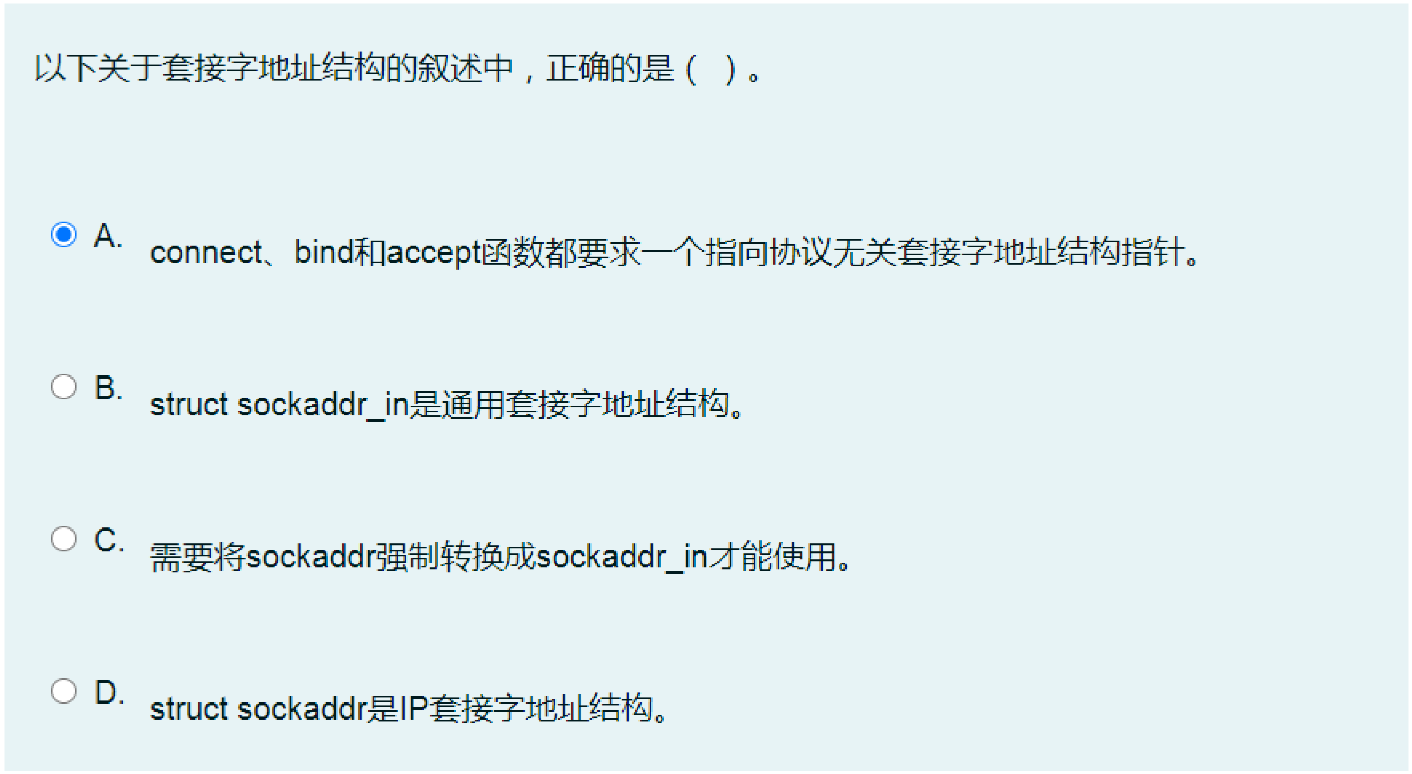


`http://www.w3.org/Protocols/rfc2616/rfc2616.html`

静态和动态内容 Static and Dynamic Content

- **HTTP响应中返回的内容可以是*静态*的，也可以是*动态*的 The content returned in HTTP responses can be either *static* or *dynamic***
  - 静态内容：存储在文件中并响应HTTP请求检索的内容 *Static content*: content stored in files and retrieved in response to an HTTP request
    - 示例：HTML文件、图像、音频剪辑、Javascript程序 Examples: HTML files, images, audio clips, Javascript programs
    - 请求标识哪个内容文件 Request identifies which content file
  - 动态内容：响应HTTP请求而动态生成的内容 *Dynamic content*: content produced on-the-fly in response to an HTTP request
    - 示例：由服务器代表客户端执行的程序生成的内容 Example: content produced by a program executed by the server on behalf of the client
    - 请求标识包含可执行代码的文件 Request identifies file containing executable code
- **Web内容关联一个文件，该文件由服务器管理 *Web content associated with a file that is managed by the server***

例题



以下关于套接字地址结构的叙述中，正确的是（ ）。

[✓ A.] connect、bind和accept函数都要求一个指向协议无关套接字地址结构指针。

B. struct sockaddr_in是通用套接字地址结构。

C. 需要将sockaddr强制转换成sockaddr_in才能使用。

D. struct sockaddr是IP套接字地址结构。

并发编程

- 进程并发编程、I/O多路复用、线程并发
- 基本同步技术
  - 变量实例到内存映射、线程交叉执行、进度图、信号量
- 线程并行
  - 典型问题并行分治策略、加速比、并行效率、阿姆达尔定律等
- 并发问题
  - 数据竞争、死锁等概念

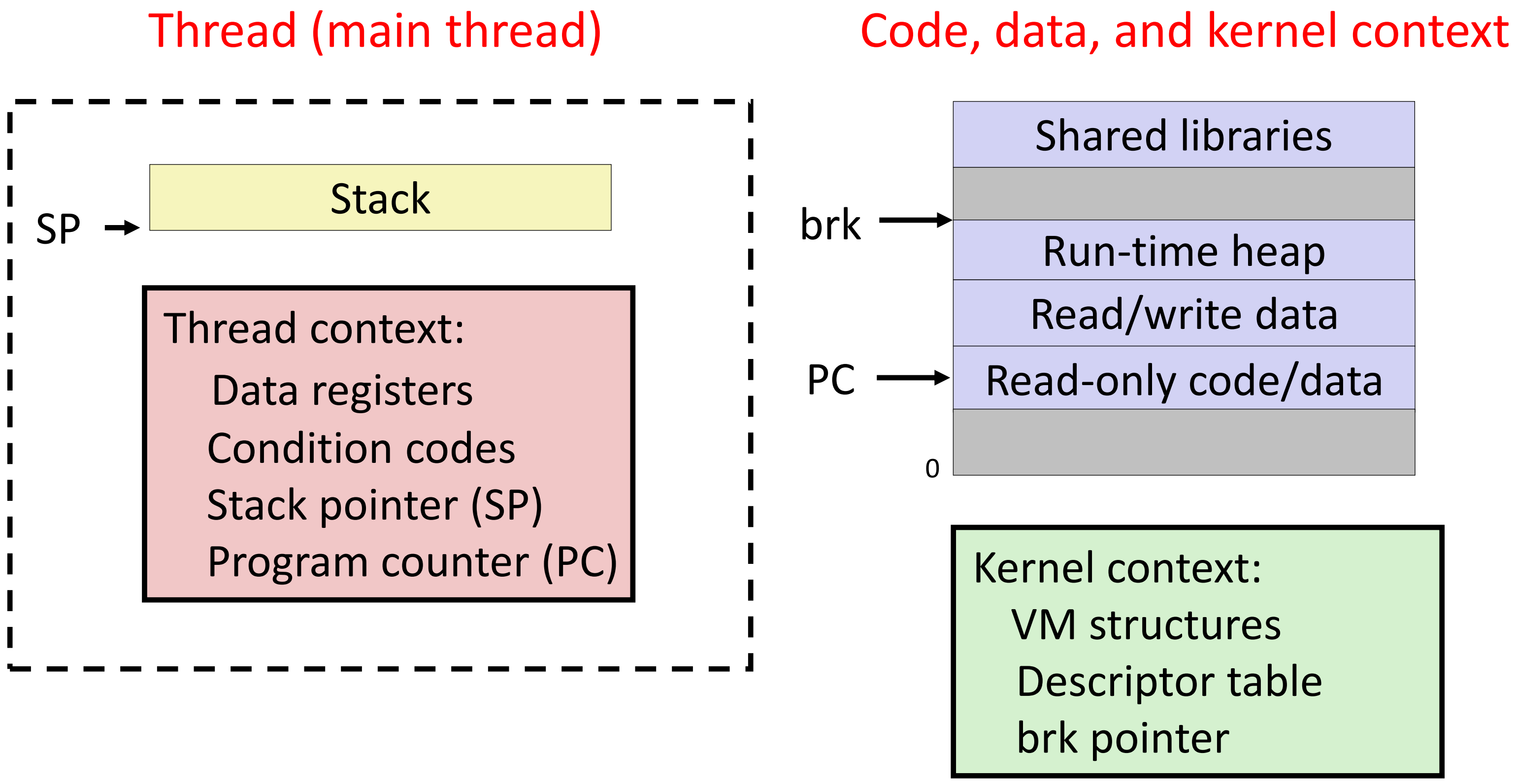
实现并发服务器的方案/Approaches for Writing Concurrent Servers

服务器可以并发处理多个客户端/Allow server to handle multiple clients concurrently

1. **基于进程的/Process-based**
   - 内核自动调度多个逻辑流/Kernel automatically interleaves multiple logical flows
   - 每个流都有自己的私有地址空间/Each flow has its own private address space
2. **基于事件的/Event-based**
   - 程序员手动管理控制流/Programmer manually interleaves multiple logical flows
   - 所有流共享同一个地址空间/All flows share the same address space
   - 通过I/O多路复用技术/Uses technique called *I/O multiplexing*.
3. **基于线程的/Thread-based**
   - 内核自动调度逻辑流/Kernel automatically interleaves multiple logical flows
   - 每个流共享同样的地址空间/Each flow shares the same address space
   - 基于线程和基于事件的混合/Hybrid of of process-based and event-based.

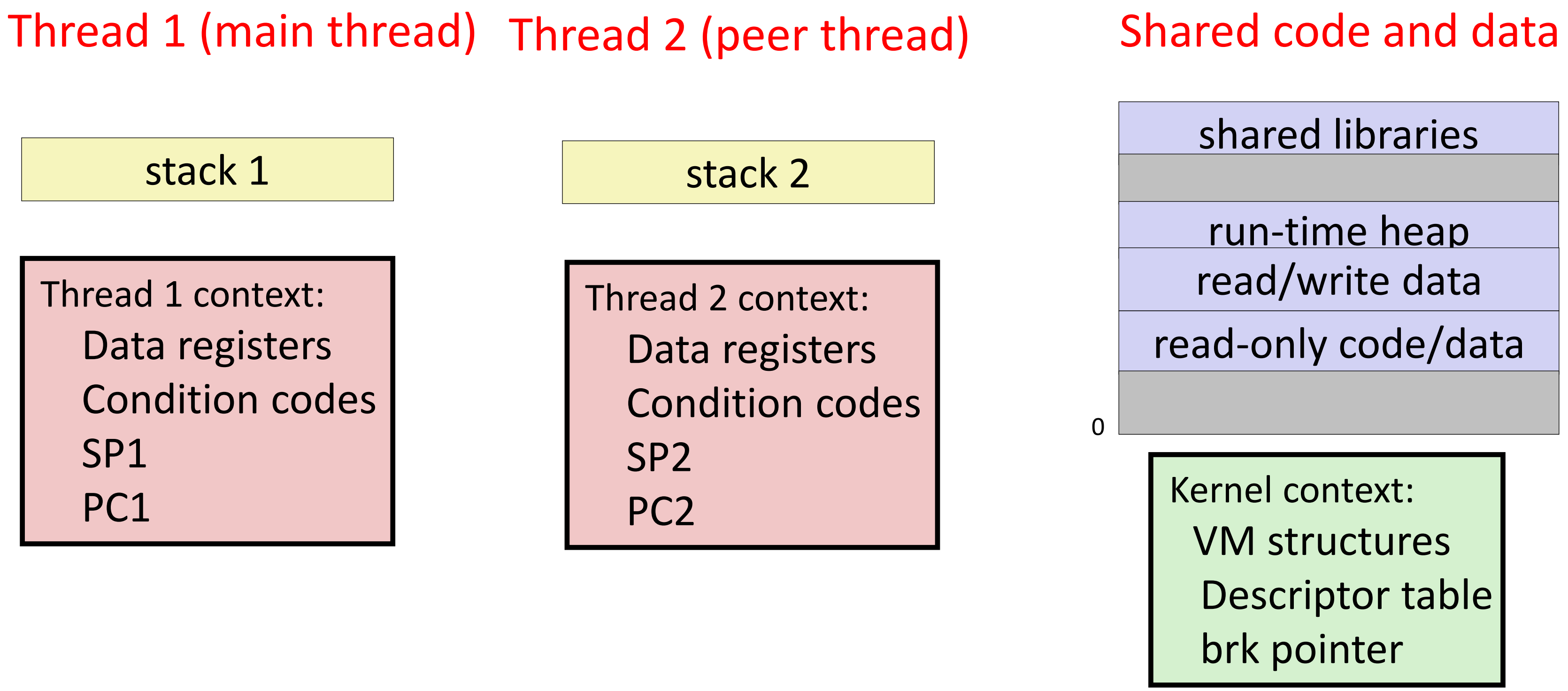
进程的另一个视图/Alternate View of a Process

- **Process = thread + code, data, and kernel context**



多线程进程/A Process With Multiple Threads

- 一个进程可以有多个线程/**Multiple threads can be associated with a process**
  - 每个线程有自己的逻辑控制结构/Each thread has its own logical control flow
  - 每个线程共享同样的代码、数据和内核上下文/Each thread shares the same code, data, and kernel context
  - 每个线程有局部变量的本地栈/Each thread has its own stack for local variables
    - 但是不受保护/but not protected from other threads
  - 每个线程有自己的ID/Each thread has its own thread id (TID)



映射变量实例到内存/Mapping Variable Instances to Memory

- 全局变量/Global variables
  - *定义/Def:* 声明在函数外面的变量/Variable declared outside of a function
  - **全局变量在虚拟内存中只有一个实例/Virtual memory contains exactly one instance of any global variable**
- 局部变量/Local variables
  - *定义/Def:* 在函数内声明的非static变量/Variable declared inside function without `static` attribute
  - **每个线程栈有一个局部变量实例/Each thread stack contains one instance of each local variable**
- 局部静态变量/Local static variables
  - *定义/Def:* 在函数内声明的static变量/Variable declared inside function with the `static` attribute
  - **虚拟内存中只有一个局部静态变量实例/Virtual memory contains exactly one instance of any local static variable.**

并发执行/ Concurrent Execution (cont)

- 不正确的顺序：两个线程增加计数器，但是结果是1而不是2/Incorrect ordering: two threads increment the counter, but the result is 1 instead of 2

i (thread)	$instr_i$	$\%rdx_1$	$\%rdx_2$	cnt
1	H_1	-	-	0
1	L_1	0	-	0
1	U_1	1	-	0
2	H_2	-	-	0
2	L_2	-	0	0
1	S_1	1	-	1
1	T_1	1	-	1
2	U_2	-	1	1
2	S_2	-	1	1
2	T_2	-	1	1

Oops!

信号量/Semaphores

- ***信号量：/Semaphore:*** **非负全局整型同步变量，由P和V操作/non-negative global integer synchronization variable. Manipulated by *P* and *V* operations.**
- **P(s)**
  - 如果s非0，则减1之后立即返回/If *s* is nonzero, then decrement *s* by 1 and return immediately.
    - 测试和自减操作原子性完成/Test and decrement operations occur atomically (indivisibly)
  - 如果s是0，则挂起线程直到s变为非0并由v操作重启动/If *s* is zero, then suspend thread until *s* becomes nonzero and the thread is restarted by a V operation.
  - 重启动后，P操作自减并将控制权返回给调用者/After restarting, the P operation decrements *s* and returns control to the caller.
- ***V(s):***
  - 自增1/Increment *s* by 1.
    - 自增操作原子性完成/Increment operation occurs atomically
  - 如果有任意线程在P操作中阻塞等待s变为非0，则重启其中一个线程，完成对应的p操作并递减s/If there are any threads blocked in a P operation waiting for *s* to become non-zero, then restart exactly one of those threads, which then completes its P operation by decrementing *s*.

- **信号量不变式/Semaphore invariant: *(s >= 0)***

表征并行程序性能/Characterizing Parallel Program Performance

- ***p*表示处理器核数, T_k 表示使用k个核运行的时间/*p* processor cores, T_k is the running time using *k* cores**
- 加速比定义/***Def.*** ***Speedup:*** $S_p = T_1 / T_p$
  - S_p表示相对加速比，如果T_1是并行版本代码在1个核上的运行时间/S_p is *relative speedup* if T_1 is running time of parallel version of the code running on 1 core.
  - S_p表示绝对加速比，如果T_1是串行版本代码在1个核上的运行时间/S_p is *absolute speedup* if T_1 is running time of sequential version of code running on 1 core.
  - 绝对加速比能够更加真实的表示并行加速收益/Absolute speedup is a much truer measure of the benefits of parallelism.
- 并行效率定义/***Def.*** ***Efficiency:*** $E_p = S_p /p = T_1/(pT_p)$
  - 是(0, 100]之间的一个百分比/Reported as a percentage in the range (0, 100].
  - 测度的是并行带来的额外开销/Measures the overhead due to parallelization

阿姆达尔定律/Amdahl’s Law

- Gene Amdahl (Nov. 16, 1922 – Nov. 10, 2015)
- **描述了并行化的困难/Captures the difficulty of using parallelism to speed things up.**
- **问题概述/Overall problem**
  - T Total sequential time required/串行运行时间
  - p Fraction of total that can be sped up ($0 \leq p \leq 1$)/可并行加速比例
  - k Speedup factor/加速因子
- **最终的性能/Resulting Performance**
  - $T_k = pT/k + (1-p)T$
    - 并行部分被加速k倍/Portion which can be sped up runs k times faster
    - 串行部分保持不动/Portion which cannot be sped up stays the same
  - 最短时间/Least possible running time:
    - $k = \infty$
    - $T_\infty = (1-p)T$

例题1

- **计算题/填空题：一个应用可并行部分占比0.8，使用4个处理器加速，加速因子为2，加速比=________，并行效率=_________。**

- **解析**
  - 并行加速后的时间为：0.2 + 0.8 / 2 = 0.6
  - 加速比为：1 / 0.6 = 1.67
  - 并行效率为：1.67 / 4 * 100= 42%

例题2

- 多选题：可能会被多个线程并发访问的变量实例包括：
  - A：局部变量实例
  - B：全局变量实例
  - C：静态变量实例
  - D：寄存器

- 解析：**B、C**

例题3

- 判断题：多个线程对受保护的同一个共享浮点变量实例进行多次累加操作，输入和程序都不变的情况下，多次计算的结果总是相同的。
- 解析：错。浮点操作具有不可结合性，改变累加顺序可能改变结果。