IPADS
INSTITUTE OF PARALLEL AND DISTRIBUTED SYSTEMS

上海交通大学
SHANGHAI JIAO TONG UNIVERSITY

操作系统高级教程
——前置知识

李鹏

ISCAS
中国科学院软件研究所
Institute of Software,Chinese Academy of Sciences

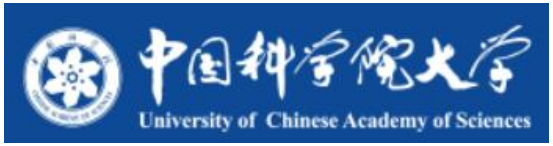
中国科学院大学
University of Chinese Academy of Sciences

开放性问题

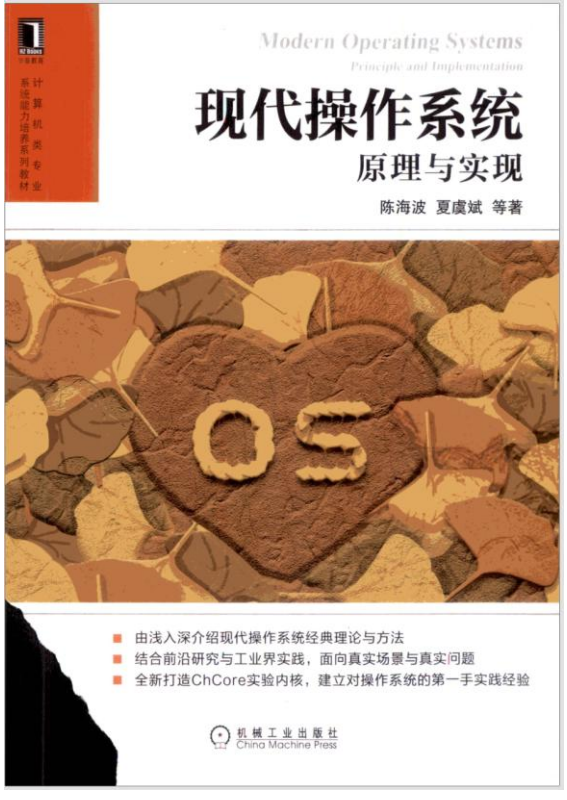
- 操作系统与其他软件有什么区别?
- 为什么要学习操作系统?
- 你认为未来的操作系统应如何演化?

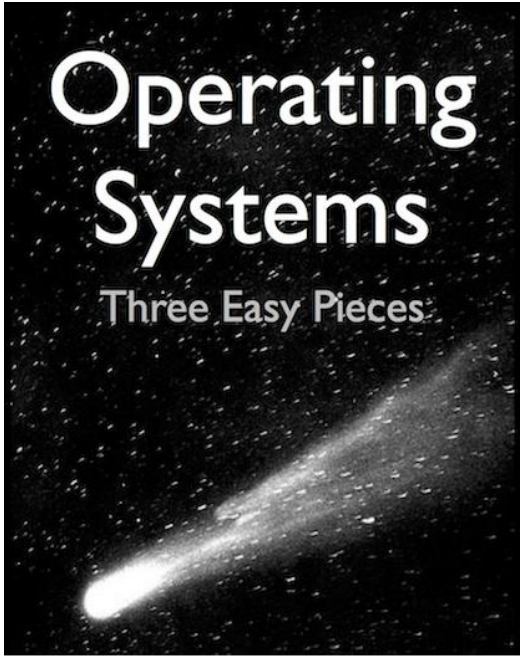
改编声明

- 本课程教学及PPT内容基于**上海交通大学并行与分布式系统研究所**发布的操作系统课程修改，原课程官网：
  - https://ipads.se.sjtu.edu.cn/courses/os/index.shtml
- 本课程修改人为**中国科学院软件研究所**，用于国科大操作系统课程教学。



图书推荐





www.ostep.org

Remzi H. Arpaci-Dusseau
Andrea C. Arpaci-Dusseau

从C语言到汇编语言

为什么硬件不能直接运行C语言的源代码

- **硬件设计**
  - 高级语言的表达能力很强
  - 硬件理解高级语言的复杂度过高、难以高效设计
- **机器指令**
  - 格式相对固定
  - 功能相对简单
  - 二进制编码

C代码示例

```
long mult2(long a, long b)
{
  return a * b;
}
```

```
// C code in mstore.c
long mult2(long, long);

void multstore(long x, long y, long *dest)
{
  long t = mult2(x, y);
  *dest = t;
}
```

编译过程


mstore.c
源程序（文本）
预处理器 (cpp)
mstore.i
经预处理的源程序（文本）
编译器 (cc1)
mstore.s
汇编程序（文本）
汇编器 (as)
mstore.o
可重定向目标程序（二进制）
链接器 (ld)
mstore
可执行目标程序（二进制）

从C程序到二进制编码

```
// C code in mstore.c
long mult2(long, long);

void multstore(long x,
               long y, long *dest)
{
  long t = mult2(x, y);
  *dest = t;
}
```

```
// Assembly file in mstore.s
multstore:
        addi    sp,sp,-16
        sd      ra,8(sp)
        sd      s0,0(sp)
        mv      s0,a2
        call    mult2
        sd      a0,0(s0)
        ld      ra,8(sp)
        ld      s0,0(sp)
        addi    sp,sp,16
        jr      ra
```

```
//Binary code in mstore.o
41 11 06 e4 22 e0 32 84 97 00 00 00
e7 80 00 00 08 e0 a2 60 02 64 41 01
82 80
```

```
//Binary code in mstore
41 11 06 e4 22 e0 32 84 ef f0 3f ff
08 e0 a2 60 02 64 41 01 82 80
```

从C程序到二进制编码

```
// C code in mstore.c
long mult2(long, long);

void multstore(long x,
               long y, long *dest)
{
  long t = mult2(x, y);
  *dest = t;
}
```

```
// Assembly file in mstore.s
multstore:
        addi    sp,sp,-16
        sd      ra,8(sp)
        sd      s0,0(sp)
        mv      s0,a2
        call    mult2
        sd      a0,0(s0)
        ld      ra,8(sp)
        ld      s0,0(sp)
        addi    sp,sp,16
        jr      ra
```

指令

```
//Binary code in mstore.o
41 11 06 e4 22 e0 32
e7 80 00 00 08 e0 a2
82 80
```

```
//Binary code in mstore
e0 32 84 ef f0 3f ff
64 41 01 82 80
```

机器码很难直接理解

从C程序到二进制编码

```
// C code in mstore.c
long mult2(long, long);

void multstore(long x,
               long y, long *dest)
{
  long t = mult2(x, y);
  *dest = t;
}
```

适合阅读

```
// Assembly file in mstore.s
multstore:
        addi    sp,sp,-16
        sd      ra,8(sp)
        sd      s0,0(sp)
        mv      s0,a2
        call    mult2
        sd      a0,0(s0)
        ld      ra,8(sp)
        ld      s0,0(sp)
        addi    sp,sp,16
        jr      ra
```

指令

理解RISC-V汇编

指令集架构与操作系统

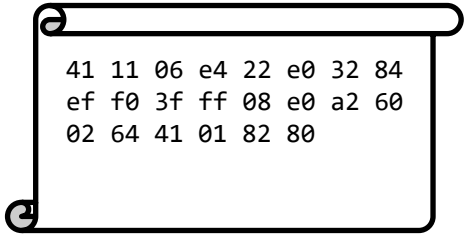
- **ISA (Instruction Set Architecture)**
  - CPU向软件（应用程序和操作系统）提供的接口
    - 理解软件在CPU上的运行（操作系统设计、程序调试等）接口
  - 操作系统中包含体系结构相关的汇编代码
    - 操作系统启动代码（例如栈尚未设置）
    - 部分操作C语言无法表达（例如获取系统状态、刷新TLB）
    - 部分场景下，汇编代码比C代码高效很多（例如memcpy、matmul）

为什么选择RISC-V

- **CPU指令集结构**
  - x86、ARM、RISC-V、SPARC、LoongArch（龙芯）、…
- **RISC-V的优势**
  - 开放架构：其指令集和架构规范是公开可用的，没有专利限制
  - 低成本和低功耗：RISC-V是精简的指令集架构，可以设计出高性能、低成本和低功耗的处理器
  - 生态系统： RISC-V的生态系统正在迅速增长，包括处理器核心、开发工具、操作系统支持和社区支持
  - 稳定性：RISC-V架构规范具有一定的稳定性，以确保向后兼容性，从而保护了现有的RISC-V生态系统投资

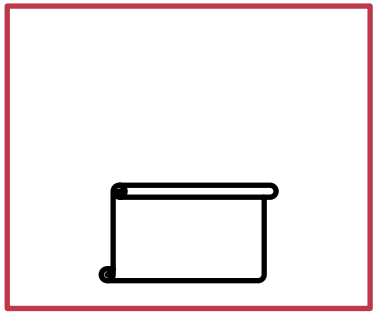
俯瞰指令执行：程序代码在哪

Mulstore程序



CPU

内存

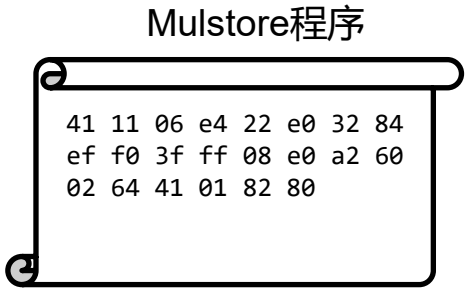


存储（硬盘）

俯瞰指令执行：代码加载

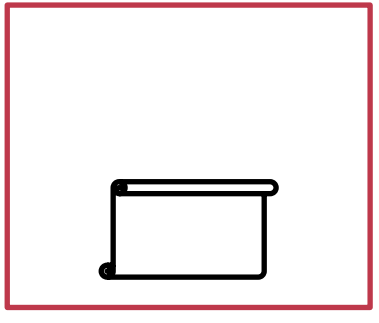
问：谁负责加载程序?

问：谁负责加载OS?



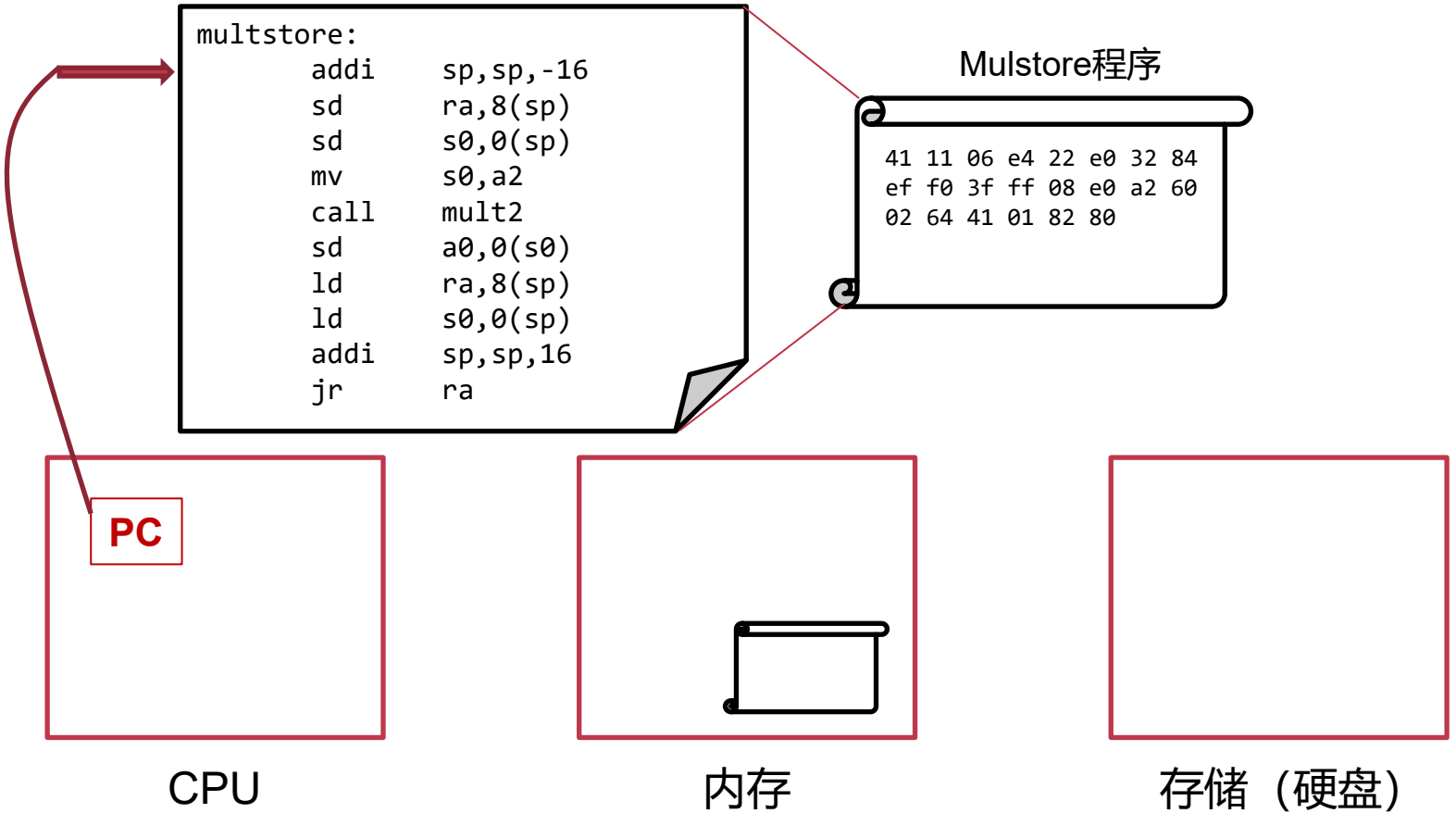
CPU

内存

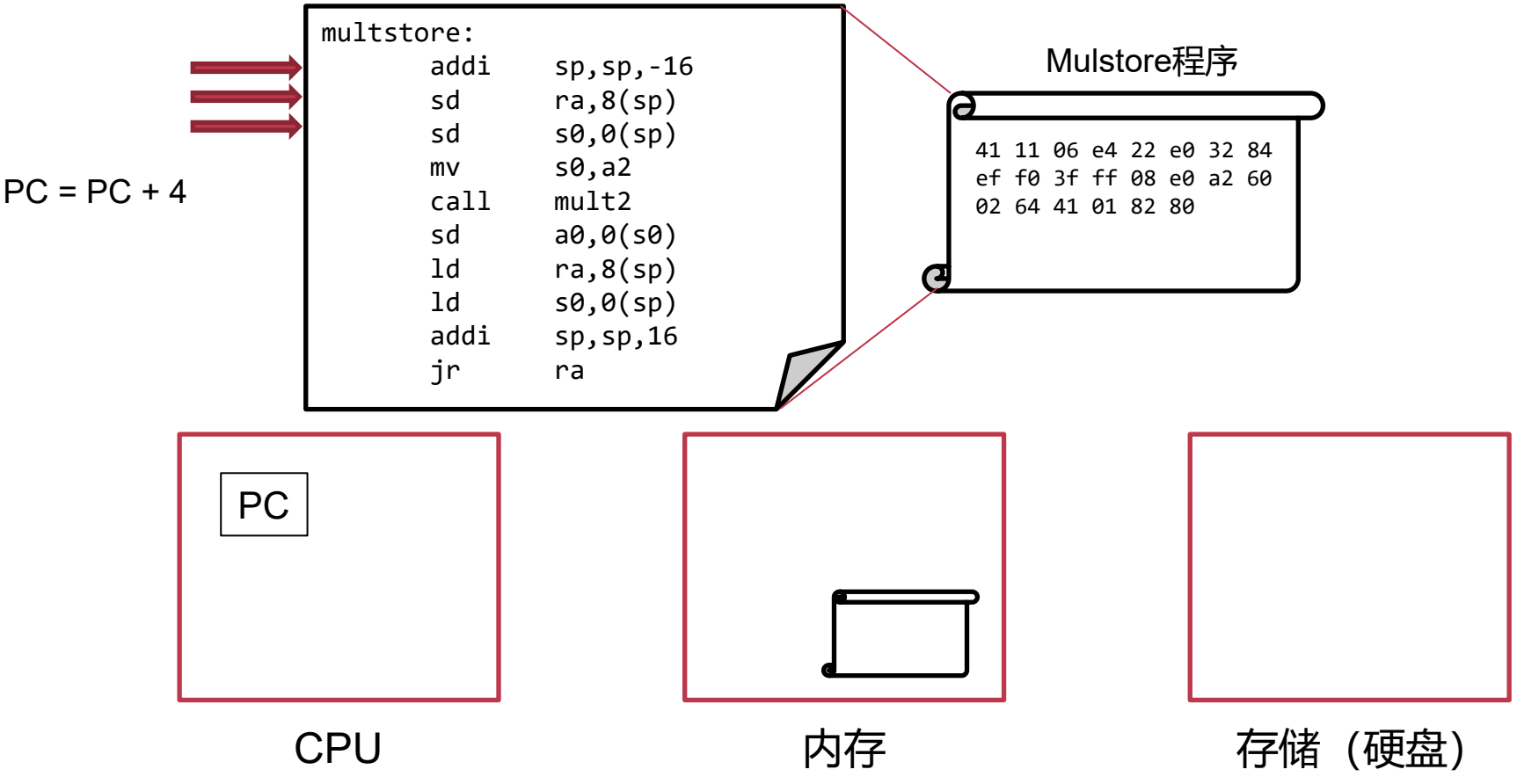


存储（硬盘）

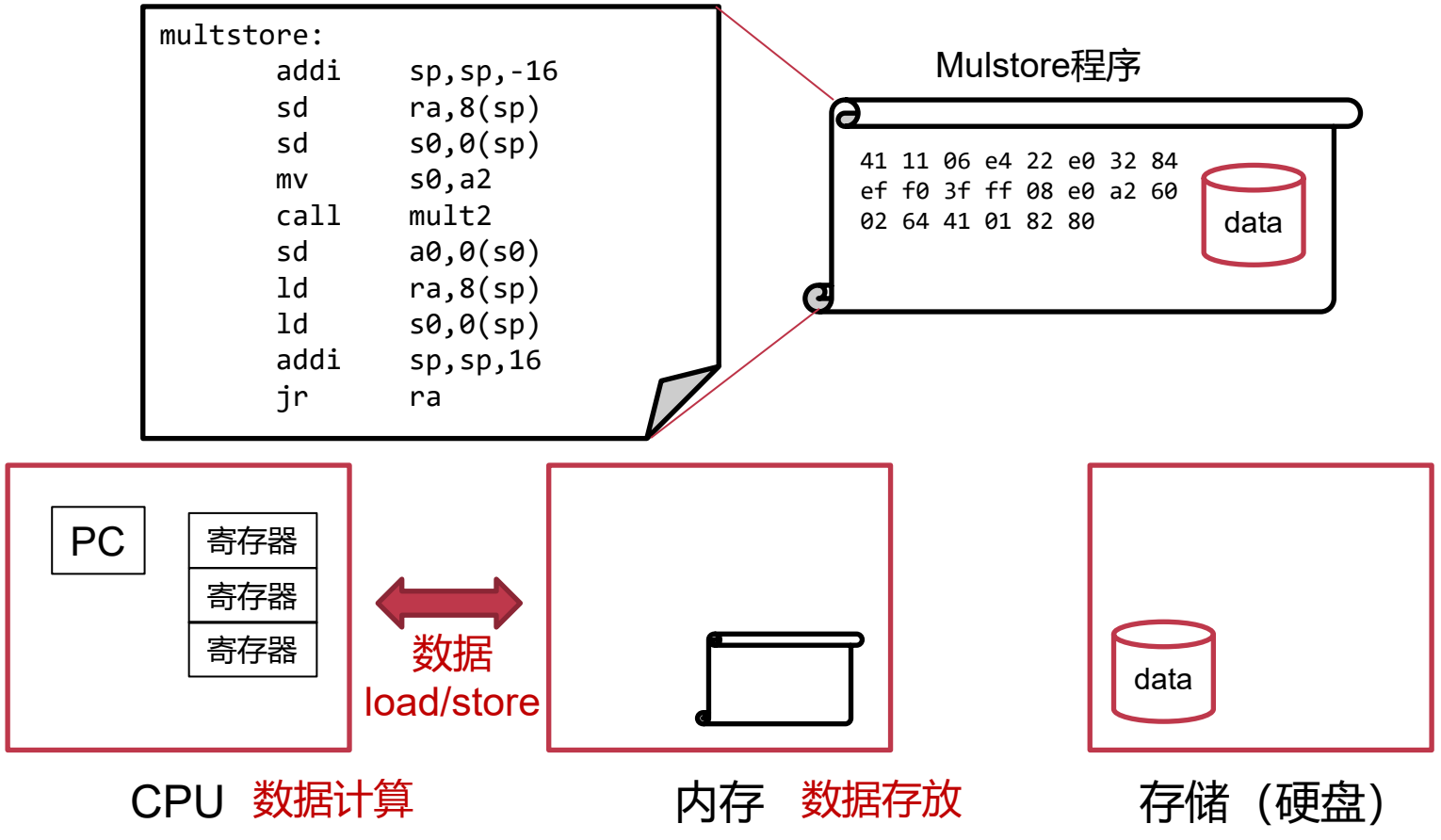
俯瞰指令执行：指令位置


multstore:
addi sp,sp,-16
sd ra,8(sp)
sd s0,0(sp)
mv s0,a2
call mult2
sd a0,0(s0)
ld ra,8(sp)
ld s0,0(sp)
addi sp,sp,16
jr ra
Mulstore程序
41 11 06 e4 22 e0 32 84
ef f0 3f ff 08 e0 a2 60
02 64 41 01 82 80
PC
CPU
内存
存储（硬盘）

俯瞰指令执行：更新PC找到下一条指令



俯瞰指令执行：数据在哪


multstore:
addi sp,sp,-16
sd ra,8(sp)
sd s0,0(sp)
mv s0,a2
call mult2
sd a0,0(s0)
ld ra,8(sp)
ld s0,0(sp)
addi sp,sp,16
jr ra
Mulstore程序
41 11 06 e4 22 e0 32 84
ef f0 3f ff 08 e0 a2 60
02 64 41 01 82 80
data
PC
寄存器
寄存器
寄存器
数据
load/store
data
CPU 数据计算
内存 数据存放
存储（硬盘）

算术与逻辑运算指令

寄存器：处理器内部的高速存储单元

x0/zero 写入忽略，读出永远是 0

Table 1. Integer register convention

Name	ABI Mnemonic	Meaning	Preserved across calls?
x0	zero	Zero	— (Immutable)
x1	ra	Return address	No
x2	sp	Stack pointer	Yes
x3	gp	Global pointer	— (Unallocatable)
x4	tp	Thread pointer	— (Unallocatable)
x5 - x7	t0 - t2	Temporary registers	No
x8 - x9	s0 - s1	Callee-saved registers	Yes
x10 - x17	a0 - a7	Argument registers	No
x18 - x27	s2 - s11	Callee-saved registers	Yes
x28 - x31	t3 - t6	Temporary registers	No

x1 – x31：

31个64位寄存器

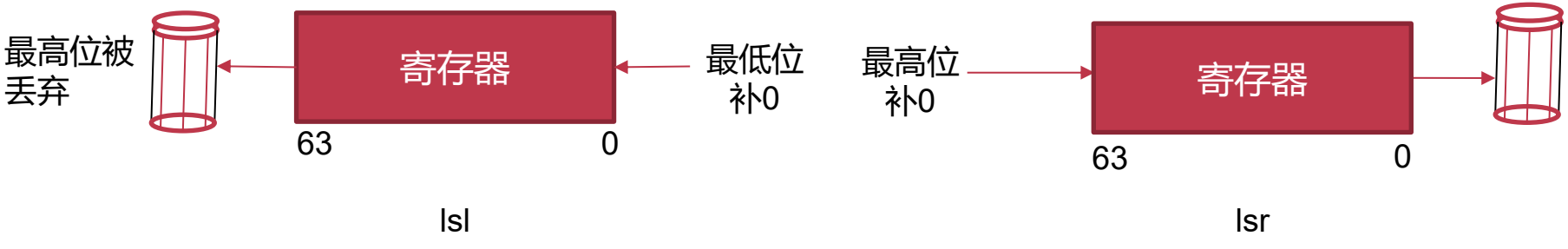
RISC-V (RV64) 寄存器

算术指令

指令	效果	描述
`add  rd,rs1,rs2`	`rd ← rs1 + rs2`	加
`addi rd,rs1,imm`	`rd ← rs1 + imm`	加立即数
`sub  rd,rs1,rs2`	`rd ← rs1 + rs2`	减
`mul  rd,rs1,rs2`	`rd ← rs1 × rs2`	乘 (mul/mulu)
`div  rd,rs1,rs2`	`rd ← rs1 ÷ rs2`	除 (div/divu)

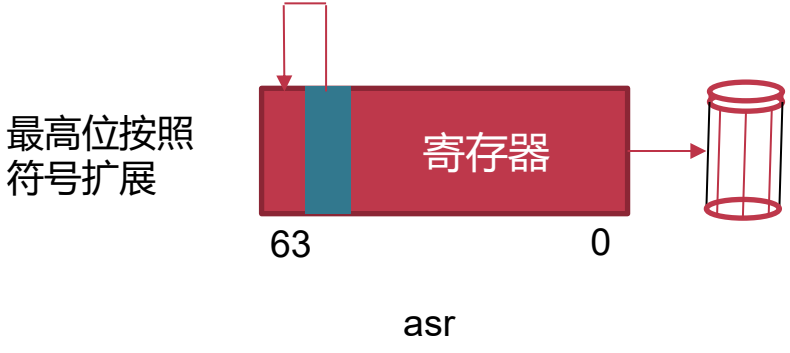
移位指令

指令	效果	描述
`sll  rd,rs1,rs2`	`rd ← rs1 <<L rs2`	(Logical) left shift
`slli rd,rs1,imm`	`rd ← rs1 <<L imm`	(Logical) left shift
`srl  rd,rs1,rs2`	`rd ← rs1 >>L rs2`	Logical right shift
`srli rd,rs1,imm`	`rd ← rs1 >>L imm`	Logical right shift



移位指令

指令	效果	描述
`sll rd,rs1,rs2`	`rd ← rs1 <<L rs2`	(Logical) left shift
`srl rd,rs1,rs2`	`rd ← rs1 >>L rs2`	Logical right shift
`sra rd,rs1,rs2`	`rd ← rs1 >>A rs2`	Arithmetic right shift
`srai rd,rs1,imm`	`rd ← rs1 >>A imm`	Arithmetic right shift



逻辑运算指令

指令	效果	描述
xor rd,rs1,rs2	rd ← rs1 ^ rs2	按位异或
xori rd,rs1,imm	rd ← rs1 ^ imm	按位异或立即数
or rd,rs1,rs2	rd ← rs1 | rs2	按位或
ori rd,rs1,imm	rd ← rs1 | imm	按位或立即数
and rd,rs1,rs2	rd ← rs1 & rs2	按位与
andi rd,rs1,imm	rd ← rs1 & imm	按位与立即数

高位指令

指令	效果	描述
lui rd,imm	rd ← imm << 12	高位立即数
auipc rd,imm	rd ← pc+(imm<<12)	PC 加高位立即数

```
# 加载大立即数到寄存器
lui a0, 0x76543
# a0 = 0x76543000
addi a0, a0, 0x210
# a0 = 0x76543210
```

伪指令：已有指令的多种用法

汇编伪指令	实际指令	描述
`neg rd, rs1`	`sub rd, zero, rs1`	取相反数
`not rd, rs1`	`xori rd, rs1, -1`	按位取反
`mv rd, rs1`	`addi rd, rs1, 0`	寄存器间复制
`li rd, imm`	（根据 imm 不同）	加载立即数到寄存器

练习

寄存器	初始值
a0	0xffffffff00000000
a1	0x1
a2	0x3

指令	目标寄存器	结果
add a2,a2,a1	a2	0x4
addi a1,a1,-1	a1	0x0
mul a0,a1,a2	a0	0x3
xor a2,a1,a0	a2	0xffffffff00000001
srai a0,a0, 1	a0	0xffffffff80000000
srli a0,a0, 4	a0	0x0fffffff0000000

add	加
sub	减
mul	乘
div	除
sll	逻辑左移
srl	逻辑右移
sra	算数右移
xor	按位异或
or	按位或
and	按位与

算术运算汇编代码

```
long arith(long x, long y, long z)
{
    long q1 = x ^ y;
    long q2 = z * 48;
    long q3 = q1 & 0x0F0F0F0F;
    long q4 = q2 - q3;
    return q4;
}
```

```
slli a5, a2, 1      # q2 = z * 2
add a5, a5, a2      # q2 = z * 2 + z = z * 3
slli a5, a5, 4      # q2 = (z * 3) * 16 = z * 48
xor a0, a0, a1      # q1 = x ^ y
li a4, 0x0f0f0f0f   # a4 = 0x0f0f0f0f
and a0, a0, a4      # q3 = q1 & a4
sub a0, a5, a0      # q4 = q2 - q3
ret                 # Return
```

add	加
sub	减
mul	乘
div	除
sll	逻辑左移
srl	逻辑右移
sra	算数右移
xor	按位异或
or	按位或
and	按位与

初始时，寄存器 a0、a1、a2 分别对应变量 x、y、z

32位运算指令

- **部分指令有 32 位版本**
  - addiw, slliw, srliw, sraiw
  - addw, subw, sllw, srlw, sraw
- **功能与不带w后缀的指令相同，但只对操作数低32位运算，结果存入寄存器时高32位为符号扩展**

访存指令

回顾

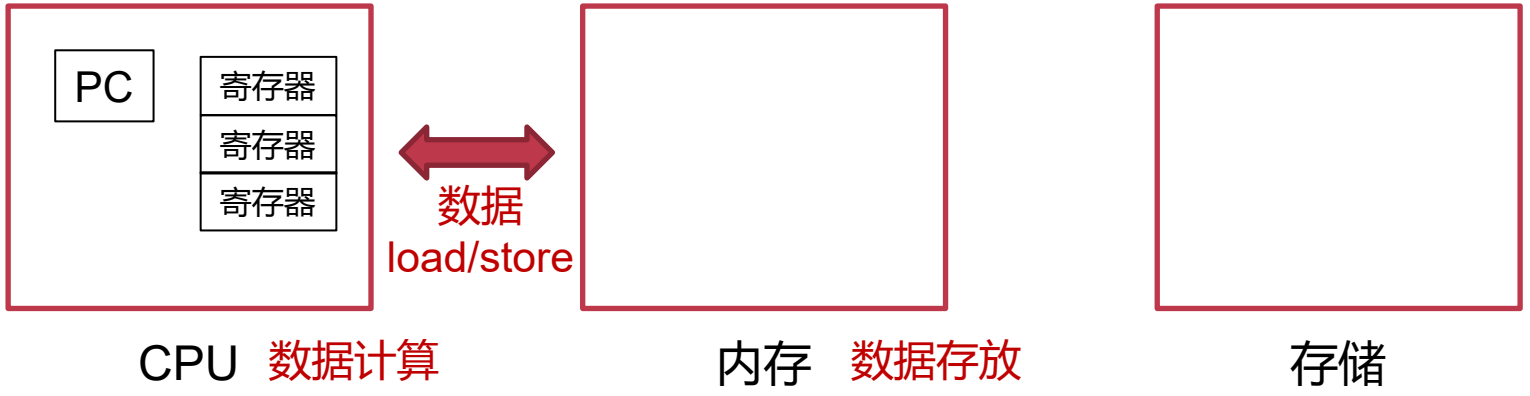
```
multstore:
        addi    sp,sp,-16
        sd      ra,8(sp)
        sd      s0,0(sp)
        mv      s0,a2
        call    mult2
        sd      a0,0(s0)
        ld      ra,8(sp)
        ld      s0,0(sp)
        addi    sp,sp,16
        jr      ra
```

Mulstore程序

```
41 11 06 e4 22 e0 32 84
ef f0 3f ff 08 e0 a2 60
02 64 41 01 82 80
```

data

计算指令
mv指令
操作CPU内部的数据



访存指令

指令	效果	描述
`lb rd, offset(rs1)`	`Rd ← mem[x[rs1]+offset]`	从内存地址x[rs1]+offset加载byte大小的数据到寄存器
`sb rd,offset(rs1)`	`mem[x[rs1]+offset] ← Rd`	从寄存器rd中将数据写入地址为x[rs1]+offset的内存位置

lb也可以是lw/ld/lh/lhu/lbu

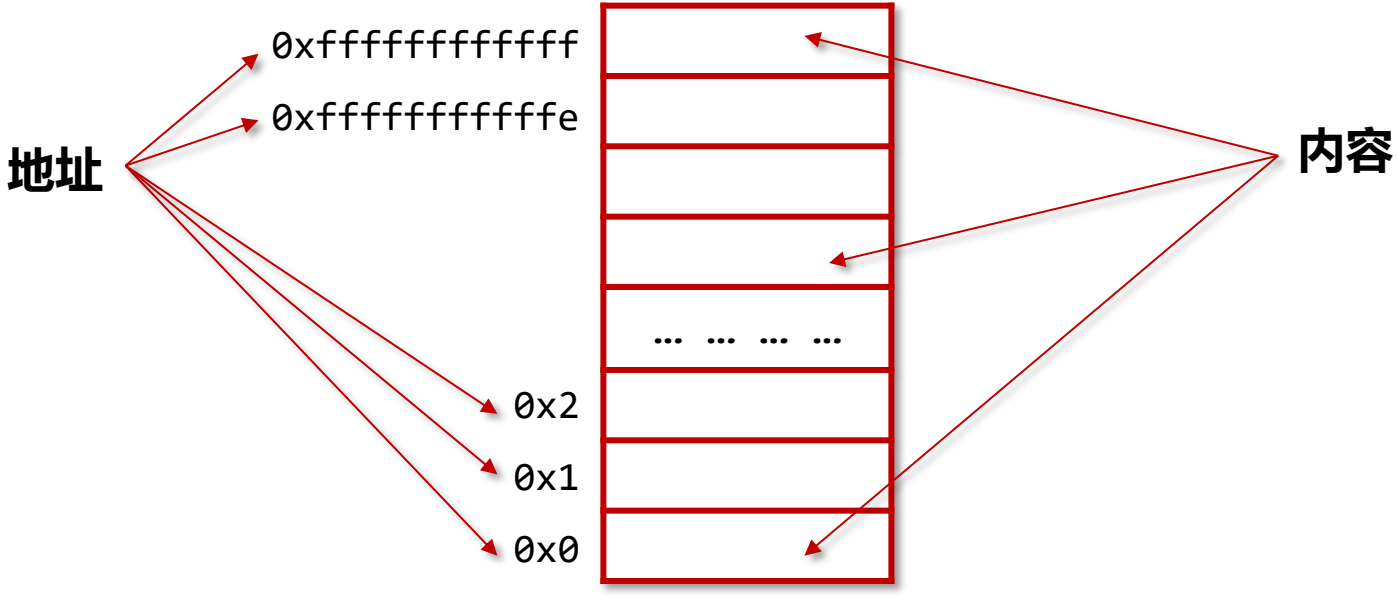
b：byte字节，w：word字，h：halfword半字，u：unsigned无符号

sb也可以是sw/sh

offset为符号位扩展的12位立即数

处理器视角下的内存

- 内存可以被视为一个很大的字节数组
- 数组中每个元素可以由唯一的地址来索引



内存地址

- **将“内存数组”的名称记为 M**
  - 若addr为要访问的内存地址，
    M[addr]即为由addr开始的内存单元的内容
    - addr 在这里被用作“内存数组”的索引
  - 内存单元的大小由上下文决定
- **addr的具体格式由寻址模式决定**

基地址加偏移量模式

- 引用M[r_b, Offset]处的数据
- 基地址寄存器r_b
- RISC-V中偏移量Offset通常是一个立即数

如： lw x1, 100(x2) # 加载寄存器 x1 中的值，该值位于基地址寄存器 x2 加上立即数 100 的内存位置

如果需要使用寄存器中的值作为偏移量，加载或计算偏移量，然后将其与基地址相加。

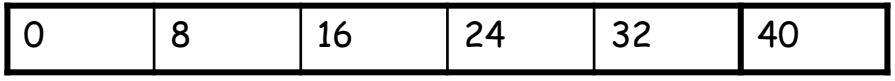
如：add x4, x2, x3 # 计算偏移量

lw x1, 0(x4) # 使用计算得到的偏移量加载数据到 x1

加载寄存器 x1 中的值，该值位于基地址寄存器 x2 加上偏移量寄存器 x3 的内存位置

示例：基地址加偏移量模式

```
long E[6] ;
```



- **假设E是一个整型数组**
  - E的起始地址存放在s2寄存器中
- **访问数组元素E[1]的RISC-V汇编为：**
  - **ld s1,8(s2)**

练习题

内存地址	值
0x100	0xFF
0x108	0xAB

寄存器	值
X0	0x100

操作	（地址）值
X0	0x100
0(X0)	0xFF
8(x0)	0xAB

X86 AT&T汇编

AT&T汇编语言格式

- **汇编语言**
  - 汇编语言是各种CPU提供的机器指令的助记符的集合，使得我们可以用汇编语言直接控制硬件系统进行工作
  - 主要有两种形式的汇编语言
    - Intel
      - DOS, Windows采用
    - AT&T
      - Unix, Linux采用

AT&T汇编语言格式

- **汇编指令操作对象：寄存器、内存地址、IO端口**

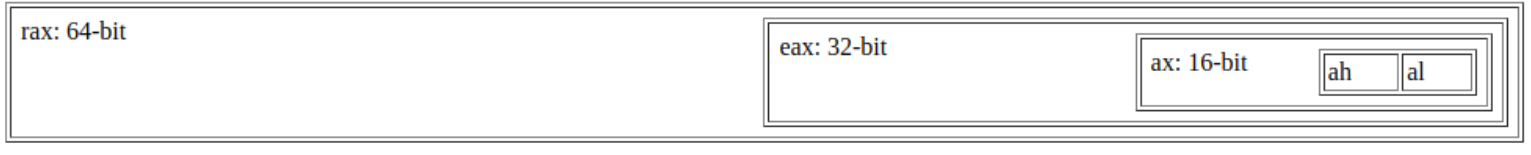
- **寄存器**
  - 逻辑寄存器
    - 架构相关
    - ISA提供给编译器可见，x86共16个通用寄存器
  - 物理寄存器
    - 实际上的硬件实现
    - 现代CPU可能有上百个物理寄存器组成

AT&T汇编语言格式

寄存器	是否有约束	惯例/用途
rax	否	1，系统调用时，调用号；2，函数返回值；3，除法运算中，存放被除数、以及运算结果的商；4，乘法运算中，存放被乘数、以及运算结果；
rbx	是，被调用者保存	基址寄存器，用来存储基础访问地址
rcx	否	1，函数调用时，第4个参数；2，有时用作counter；
rdx	否	1，函数调用时，第3个参数；2，除法运算中，存放运算结果的余数；3，乘法运算中，存放运算结果溢出的部分；
rbp	是，被调用者保存	frame pointer，存放当前函数调用时栈的基地址
rsp	是，被调用者保存	时时刻刻指向栈顶
rdi	否	1，函数调用时，第1个参数；2， rep movsb中的目的寄存器；
rsi	否	1，函数调用时，第2个参数；2， rep movsb中的源寄存器；
r8	否	1，函数调用时，第5个参数
r9	否	1，函数调用时，第6个参数
r10、r11	否	
r12、r13、r14、r15	是，被调用者保存	

AT&T汇编语言格式

- **X86的向后兼容性**
  - 64位：rax
  - 32位：eax
  - 16位：ax



AT&T汇编语言格式

- **AT&T汇编语法格式**
  - 一条汇编语句包含4个部分
    - 标号(可选)
    - 操作码(指令助记符)
    - 操作数(由具体指令决定)
    - 注释(可选)

标号: 操作码　操作数1，操作数2，…　　#可选的注释

AT&T汇编语言格式

- **操作码**
  - AT&T汇编中，操作码名称的最后一个字符指明操作数的宽度
    - b, w, l, q分别表示byte(字节), word(字), long(双字), qword (四字)

```
movl var, %eax
```

把内存地址var处的4个字节载入到寄存器eax中

  - 操作码前缀
    - 用于修饰随后的操作码，常用于重复字符串指令、执行总线锁定操作或指定操作数和地址宽度
    - 如

```
movl $0x0, %eax
movl $0x12, %ecx
rep stosl
```

AT&T汇编语言格式

- **AT&T汇编语法格式**
  - 操作数
    - 一个指令可以含有0~3个操作数，中间用逗号分隔
    - 对于有两个操作数的指令，前者表示源操作数，后者表示目的操作数
      - 如

```
movw %ax, %bx
```

    - 操作数可以是立即数、寄存器、内存
      - 立即操作数前面需加上$
      - 寄存器操作数前面需加上%
      - 内存操作数
        - » 引用内存地址为base+ index*scale+disp的变量

```
disp(base, index, scale)
```

        - » 如表示把内存地址var处的内容载入%eax

```
mov var, %eax
```

AT&T汇编语言格式

- **运算指令**
  - add %r10,%r11
  - add $5,%r10
  - div %r10
  - inc %r10
  - mul %r10
- **拷贝指令**
  - mov %r10,%r11
  - mov $99,%r10
  - mov %r10,(%r11)
  - mov (%r10),%r11
  - push %r10
  - pop %r10

AT&T汇编语言格式

- **流程控制指令**
  - jmp label //跳转到标号
  - cmp %r10,%r11
  - je label //如果相等，跳转到标号
  - jne label
  - jl label
  - jg label
  - call label
  - ret
  - syscall

```
1:    cmpb var,%al
      je 1f
      movb %al,%dl
      jmp 1b
1:    pushw %dx
```

AT&T汇编语言格式

- C语言 内联汇编

```
#include <stdio.h>
int main()
{
        int a = 10, b;
        __asm__("movl %1, %%eax\n\t"
                 "movl %%eax, %0\n\t”
                :"=r"(b) /* output */
                :"r"(a) /* input */
                :“%eax” /* clobbered register*/ );
        printf("Result: %d, %d\n", a, b);
        return 0;
}
```

AT&T汇编语言格式

- **汇编指示符**
  - .byte 0x80, 0xca
  - .word 0x800
  - .text

数组

数组的声明

- **C语言语法：Type Array[N] ;**
- **数组会在内存中占用一块连续的区域**
  - 该区域的大小为`sizeof(Type)*N`字节

数组的起始地址

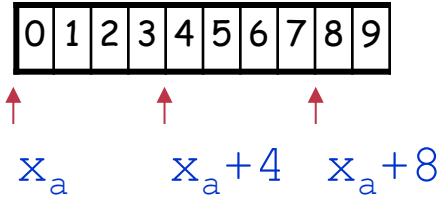
- 数组 Array 的起始地址记为X_{Array}
- 标识符 Array 是一个指向数组起始位置的常量指针
  - 其值为X_{Array}

访问数组元素

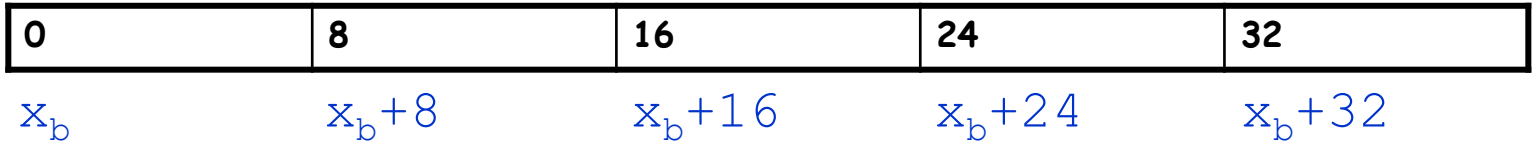
- **在C语言中，数组元素可以被索引访问**
  - 数组索引是一个位于**0**与**N-1**之间的整数
- **第`i`个数组元素被储存在X_{Array} + sizeof(Type) * i**

数组示例

```
char a[10] ;
```

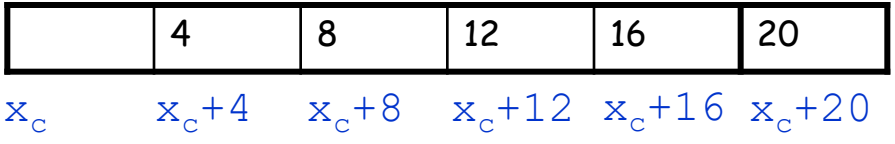



```
char *b[5] ;
```

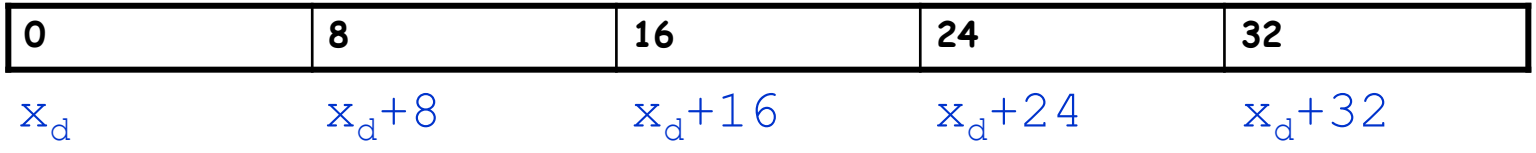


数组示例

```
int c[6] ;
```




```
double *d[5] ;
```



回顾：访存指令

- **假设 E 是一个整型数组（单个元素 4 byte）**
  - E 的起始地址存放在x0寄存器中
  - E 的索引 i 存放在x1寄存器中
- **那么数组元素E[i]的访问在RISC-V汇编中为：**
  - **slli x1, x1,2**
  - **add x0,x1**
  - **ld x2,x0**

指针运算：结果放到 x2

表达式	类型	值	汇编代码
E	int *	x_E	mv x2, x0
E[0]	int	M[x_E]	ld x2, x0

E[i]	int	M[x_E+4i]	slliw x1, x1, 2
			add x0, x1
			ld x2, x0

E+i-1	int *	x_E+4i-4	slliw x1, x1, 2
			add x2, x0, x1
			addi x2, x2,-4

*(E+i-3)	int	M[x_E+4i-12]	addiw x0, x0, -12
			slliw x1, x1, 2
			add x0, x1
			ldr x2, x0

&E[i]-E	char	i	mv x2, x1

指针

指针

- **每个指针有属于自己的类型**
  - 指向类型为**Type**的对象的指针类型为**Type ***
  - 特殊类型**void ***表示泛型指针
    - 堆内存申请函数 malloc 返回的就是这样的泛型指针
- **每个指针有属于自己的值**
  - 指针的值是内存地址

指针

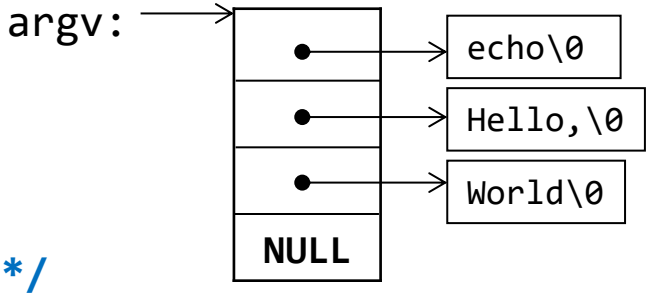
- **指针的值由取地址运算符&获取**
- **指针可以通过运算符 * 进行解引用**
  - 解引用的结果是指针指向的对象值
- **数组与指针有紧密的关联**
  - 数组的名称可以被视作常量指针
  - **ip[0]**与***ip**是等价的

指针运算

- **加减法**
  - 指针与整数相加减结果为**指针**：p+i，p-i
  - 指针与指针相减结果为**整数**：p-q
- **引用（&）与解引用（*）**
  - *p, &E
- **下标索引**
  - A[i]，*(A+i)

命令行参数

```
$ echo hello, world
hello, world
```




```
#include <stdio.h>
/* echo command-line arguments */
int main(int argc, char *argv[])
{
    int i ;
    for (i = 1; i < argc ; i++)
        printf("%s%s", argv[i],
                    (i < argc-1) ? " " : "");
    printf("\n") ;
    return 0;
}
```

函数指针

- **指针可以指向函数**
- **示例：函数指针 void (*f)(int *)**
  - f 是该函数指针的标识符
  - f 指向的函数以int *为参数
  - f 指向的函数返回类型为void
- **函数指针可以被赋值为函数标识符**
  - f = func
- **注意与返回类型为指针的函数区分**
  - void *f(int *)声明的是返回类型为void *的函数

函数指针

```
#include <stdlib.h>

/* numcmp: compare s1 and s2 numerically */
int numcmp(char *s1, char *s2)
{
    double v1, v2;

    v1 = atof(s1);
    v2 = atof(s2);
    if (v1 < v2) return -1;
    else if ( v1 > v2 ) return 1;
    else return 0;
}
```

函数指针

```
#include <stdio.h>
#include <string.h>

{
    ...
    int numeric = 0, (*cmp)(void *, void *);
    char *s1, *s2;
    ...
    if (...) numeric = 1 ;
    ...
    cmp = (int (*)(void *, void *))
                (numeric ? numcmp : strcmp);
    (*cmp)(s1, s2);
    ...
}
```

异质数据结构

结构体（struct）

- 结构体将多个对象统合在同一个结构中

```
struct rect {
    long llx;    /* 左下角的X轴坐标 */
    long lly;    /* 左下角的Y轴坐标 */
    unsigned long width;  /* 宽度（像素） */
    unsigned long height; /* 高度（像素） */
    unsigned long color; /* 颜色代码 */
};
```

结构体（struct）

- 结构体中每个对象以其名称来索引

```
struct rect r;
r.llx = r.lly = 0;
r.color = 0xFF00FF;
r.width = 10;
r.height = 20;
```

结构体（struct）

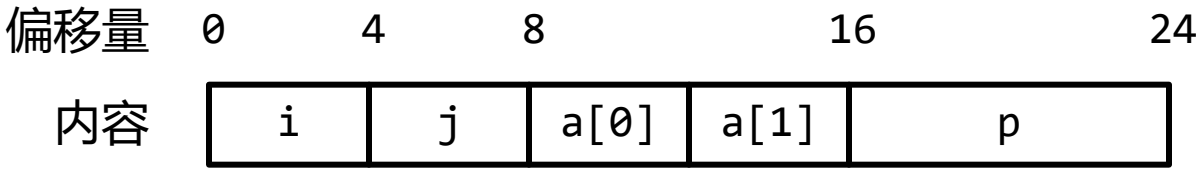
```
long area (struct rect *rp)
{
    return (*rp).width * (*rp).height;
}

void rotate_left (struct rect *rp)
{
    /* 交换宽度和高度 */
    long t = rp->height;
    rp->height = rp->width;
    rp->width = t;
    /* 更新左下角的坐标 */
    rp->llx -= t;
}
```

结构体（struct）

- **结构体在内存中的布局**
  - 结构体的所有成员被储存在**连续的（？）**内存区域
  - 结构体的指针是该区域**第一个字节**的地址

结构体（struct）

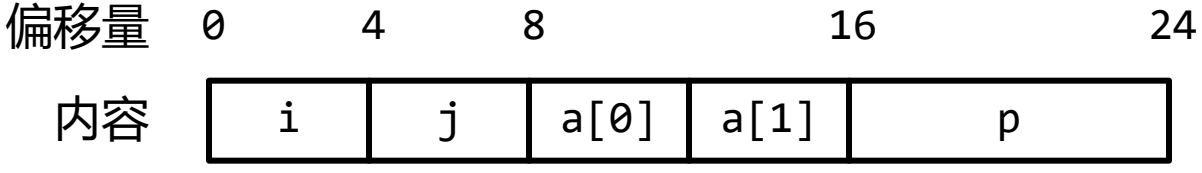
```
struct rec {
    int i;
    int j;
    int a[2];
    int *p;
} *r;
```



结构体（struct）

- 对结构体成员的引用被翻译为偏移量

```
r->j = r->i  # 将成员r->i的值赋给成员r->j
# r的地址存储在x0
lw      x1, 0(x0)                # 获取r->i的值
sw      x1, 4(x0)                # 储存到r->j
```



联合（union）

- 一个联合对象可以以**不同的数据类型**访问
- 联合的声明语法与结构体类似，但语义有很大区别
- 联合中所有成员的引用都指向**同一块**内存区域

联合（union）

```
struct S3 {
    char c;
    int i[2];
    double v;
};

union U3 {
    char c;
    int i[2];
    double v;
};
```

以下成员的偏移量及S3与U3占用的空间大小为:

类型	c	i	v	大小
S3	0	4	16	24
U3	0	0	0	8

联合（union）

```
// On a little-endian machine (e.g., ARM)
double uu2double(unsigned word0, unsigned word1)
{
    union {
           double d;
           unsigned u[2];
    } temp;

    temp.u[0] = word0;
    temp.u[1] = word1;
    return temp.d;
}
```

对齐

对齐（alignment）

- **对齐是对某些对象存储地址的限制**
  - 某些类型的地址必须是**某个值k**的倍数
  - **k**一般为2, 4或8
- **对齐简化了处理器与内存系统之间硬件接口的设计**

对齐（alignment）

- ARM硬件在数据没有对齐的情况下**仍能正确地工作**
- 数据对齐能够提高内存系统的性能

对齐（alignment）

- **Linux的对齐限定**
  - 1字节数据结构地址没有限制
  - 2字节数据结构地址必须是2的倍数
  - 4字节数据结构地址必须是4的倍数
  - 更大的数据结构地址必须是8的倍数

对齐（alignment）

- 通过确保每个数据类型的对象在组织与分配的时候满足对齐的要求来加强数据对齐

```
.align 8
```

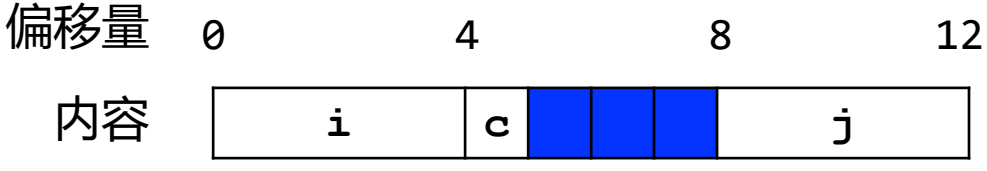
- malloc()返回的泛型指针对齐要求为8字节

对齐（alignment）

- **对齐对结构体的限制**
  - 结构体有时需要在**成员间**插入空隙
  - 结构体有时需要在**末尾处**增加填充

对齐的例子

```
struct S1 {
    int i;
    char c;
    int j;
};
```



MAKEFILE

Makefile的作用

- 编译内核
  - `make menuconfig`
  - `make`
  - 发生了什么?
- Linux 内核文件数目将近2万
  - 内核中的哪些文件将被编译?
  - 它们是怎样被编译的?
  - 编译、连接时的顺序如何确定?
  - 如果修改某些源文件，哪些文件将被重编译、重链接?

Makefile三要素

目标:依赖

(tab)命令

```
myapp:main.o a.o b.o c.o
	gcc main.o a.o b.o c.o -o myapp
main.o:main.c
	gcc -c main.c
a.o:a.c
	gcc -c a.c
b.o:b.c
	gcc -c b.c
c.o:c.c
	gcc -c c.c
```

```
SOURCES=$(wildcard *.c)
OBJECTS=$(subst %c,%o,$(SOURCES))

myapp:$(OBJECTS)
	$(LD) $^ -o $@
%.o:%.c
	$(CC)  -c $^ -o $@
```

如何简化？——通配符、模式与变量

- **内置变量**
  - $(CC)、$(LD)、$(MAKE)
- **自动变量**
  - $@指代当前目标
  - $^ 指代所有依赖
  - $< 指代第一个依赖
- **自定义变量**
  - txt = Hello World
- **模式匹配**
  - “%.o:%.c”
    - gcc $^ -o $@
- **通配符**
  - SOURCES=$(wildcard *.c)
  - OBJECTS=$(subst %c,%o,$(SOURCES))

Linux内核中的Makefile

- **官方文档：Documentation/kbuild/makefiles.rst（makefile.txt）**
- **五部分**
  - Makefile：顶层Makefile文件
  - .config:系统配置文件
  - arch/$(SRCARCH)/Makefile
  - scripts/Makefile.* 所有 kbuild Makefiles的共有规则
  - kbuild Makefiles 每个子目录中的Makefile
- **四种角色**
  - 用户、一般开发者、Arch开发者、Kbuild开发者

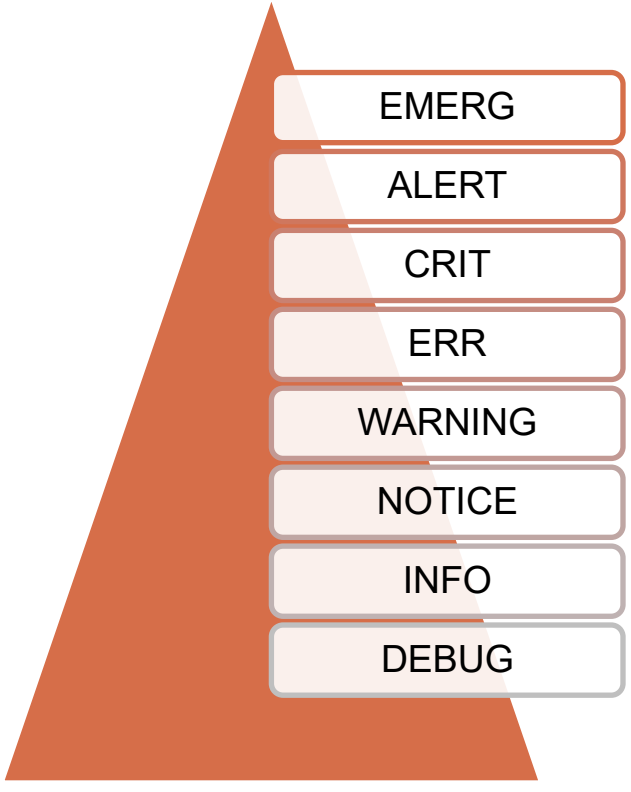
内核调试

内核调试工具

- **打印（是否执行、调用栈、调用顺序、参数值）**
  - printk
  - pr_*, dev_*
  - early_printk
  - WARN_ON
- **KGDB**
- **Ftrace**
- **大脑** 🧠

Printk

- printk(KERN_INFO “Hello World\n”);
  - 8种打印级别
  - config可以控制是否打印时间戳，精确到毫秒
  - 输出到内核环形缓冲区
    - dmesg
- pr_info,pr_err,dev_info,…



打印

- early_printk:用于控制台驱动加载成功之前的调试，通常通过单独的串口驱动
  - 更早的阶段可以通过主板支持的其他手段：指示灯、八段管、蜂鸣器等
- Kernel Panic时会打印出当时PC的值
  - addr2line vmlinx找到对应的文件名、函数名和行号信息
- WARN_ON可打印当前函数的调用栈

KGDB

- 使用串口线（kgdboc）或以太网线（kgdboe）将被调试计算机同上位机相连
  - 配置内核编译选项
  - 启动命令行中加入命令
    - kgdboc= ttyS0,115200
  - 启动kgdb
    - echo g > /proc/sysrq-trigger
    - 单步执行、断点执行、查看函数调用栈、查看修改变量、查看修改内存、查看修改寄存器

```
CONFIG_DEBUG_INFO=y
CONFIG_FRAME_POINTER=y
CONFIG_MAGIC_SYSRQ=y
CONFIG_MAGIC_SYSRQ_SERIAL=y
CONFIG_KGDB_SERIAL_CONSOLE=y
CONFIG_KGDB_KDB=y
CONFIG_KGDB=y
```

FTrace

- 记录内核函数调用的流程(kernel/Documentation/trace/ftrace.txt)
  - 学习、调试、调优操作系统
- 在编译时在每个函数的入口地址放置一个probe点
  - 调用一个probe函数（默认调用名为mcount的函数）并打印log
- 用法
  - `mount -t tracefs none /sys/kernel/tracing`
  - `cat /sys/kernel/tracing/available_tracers`
  - `echo 0 > tracing_on`
  - `echo function_graph > current_tracer`
  - `echo *dma* > set_ftrace_filter`
  - `echo 1 > tracing_on`
  - `some operation`
  - `echo 0 > tracing_on`
  - `cat trace`


欢迎进入
操作系统的世界