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

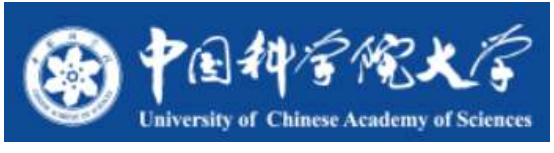
中国科学院大学
University of Chinese Academy of Sciences

操作系统基础
——编程语言、编译与调试

李鹏

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，已获得原作者授权，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。



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

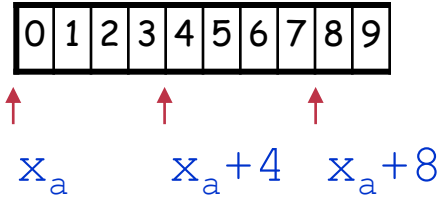
- **数组 Array 的起始地址记为X_{Array}**
- **标识符 Array 是一个指向数组起始位置的常量指针**
  - 其值为X_{Array}

访问数组元素

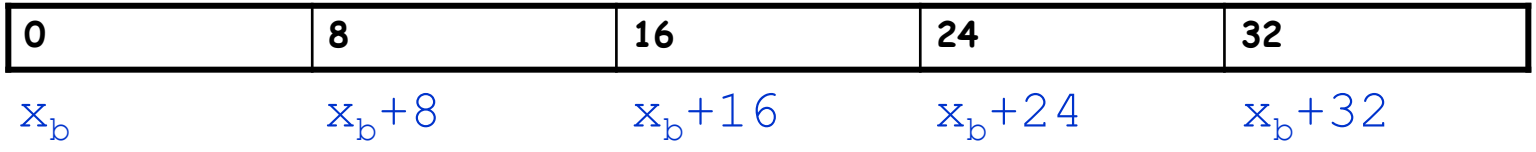
- **在C语言中，数组元素可以被索引访问**
  - 数组索引是一个位于**0**与**N-1**之间的整数
- **第`i`个数组元素被储存在X_{Array} + sizeof(Type) * i**

数组示例

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

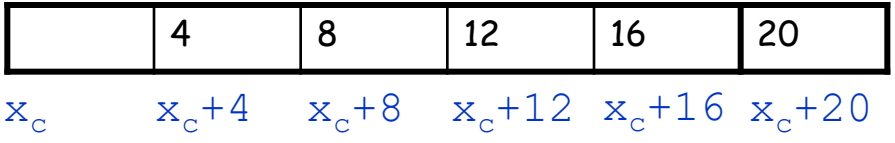



```
char *b[5] ;
```

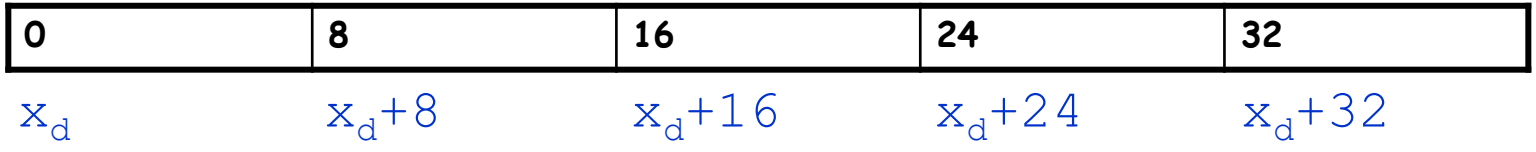


数组示例

```
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

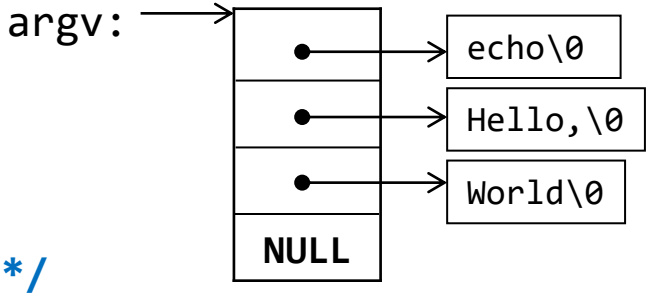
- **指针的值由取地址运算符&获取**
- **指针可以通过运算符 * 进行解引用**
  - 解引用的结果是指针指向的对象值
- **数组与指针有紧密的关联**
  - 数组的名称可以被视作常量指针
  - **ip[0]**与***ip**是等价的

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）

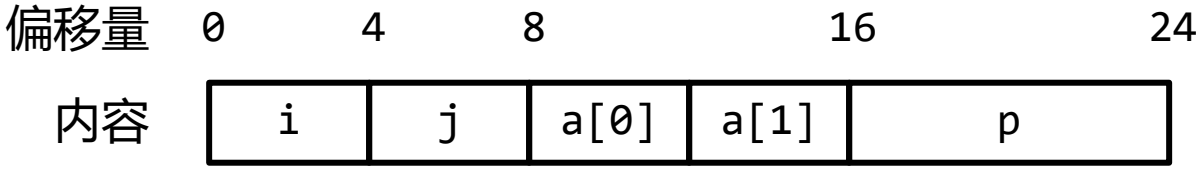
```
long area (struct rect *rp)
{
    return (*rp).width * (*rp).height;
}

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）

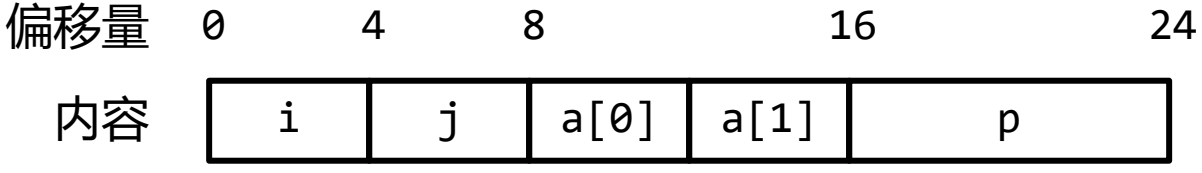
```
struct rec {
    int i;
    int j;
    int a[2];
    int *p;
} *r;
```



结构体（struct）

- 对结构体成员的引用被翻译为偏移量

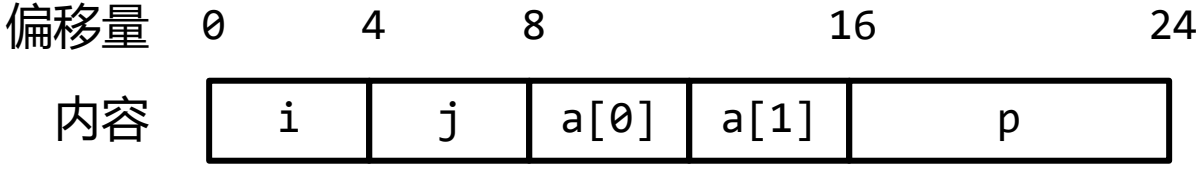
```
r->j = r->i  # 将成员r->i的值赋给成员r->j
# r的地址存储在x0
lw      x1, 0(x0)                # 获取r->i的值
sw      x1, 4(x0)                # 储存到r->j
```



结构体（struct）

```
&(r->a[i])
```

```
# r的地址存储在x0，i存储在x1
slli   x1, x1, 2
mv     x2, [x0, x1, lsl 2]       # &r->a[i]
```



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
```

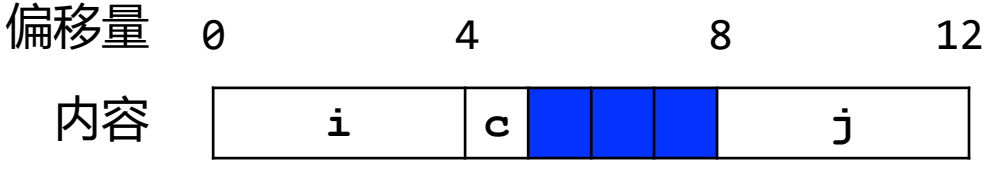
- malloc()返回的泛型指针对齐要求为8字节

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)
	gcc $^ -o $@
%.o:%.c
	gcc  -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

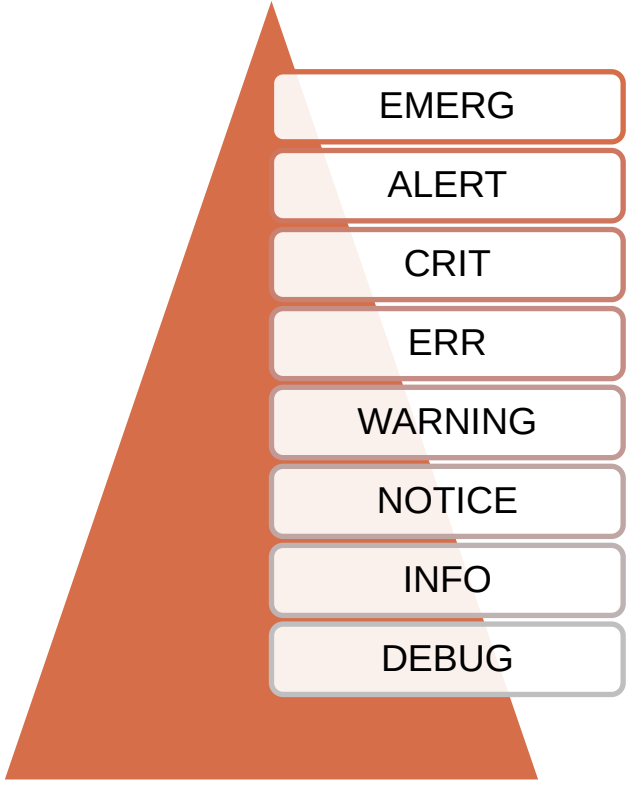
内核调试

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、端点执行、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`