Machine-Level Programming: Data

石元峰

2019年10月10日

Outline

One Dimension Array

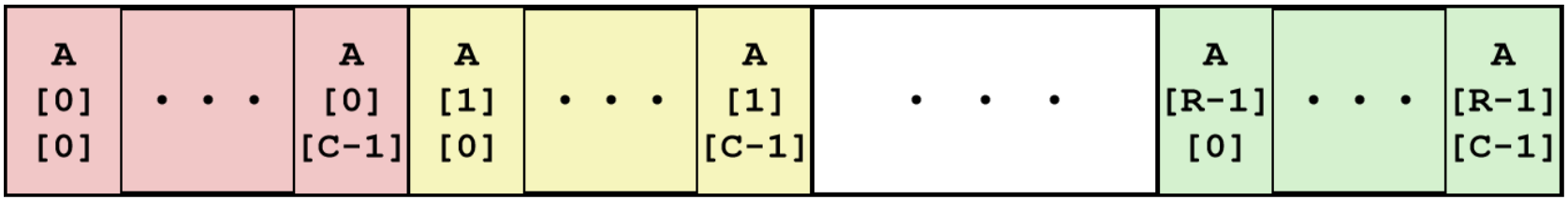
- T A[N];
- 其首先在内存中分配了一个$L * N$字节的连续区域，这里L为数据类型T的大小
- 标识符A可以用来作为指向数组开头的指针，记其值为x_A，则数组元素$A[i]$会被存放在地址为$x_A + L * i$的地方

One Dimension Array

- int A[N];
- A[i]为int类型，其值为$M[x_A + 4i]$，用汇编代码实现为：movl (%rdi, %rsi, 4),%eax
- A+i-1为int*类型，其值为$x_A + 4i - 4$，用汇编代码实现为：leaq -4(%rdi, %rsi, 4),%rax
- *(A+i-1)等价于A[i-1]，表示数组第i个元素

Multidimensional (Nested) Array

- T A[R][C];
- 数组元素在内存中遵循“行优先”的顺序排列
- $D[i][j]$的内存地址为$\&D[i][j] = x_D + (C * i + j) * sizeof(T)$

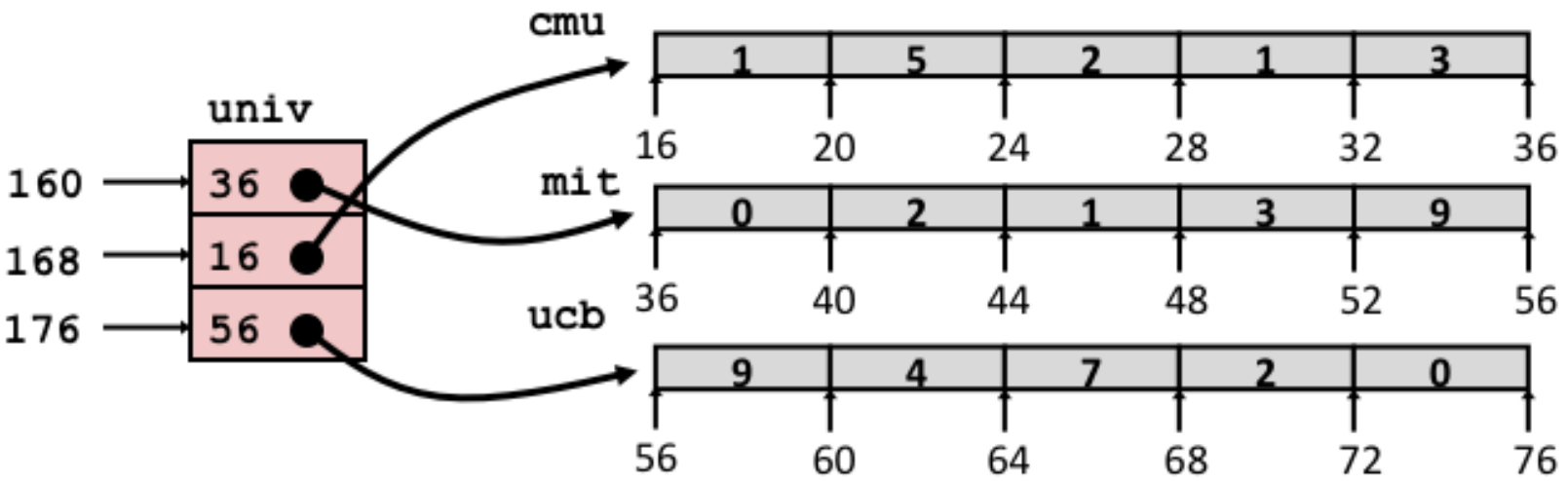


Multi-Level Array

```
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```

```
#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, ucb};
```

- univ[N][M]的值为$M[M[x_{univ} + 8N] + 4M]$
- 在地址计算上，其与多维数组是不同的



Fixed and Variable Dimensions

对于定长数组：int A[16][16];

求A[i][j]地址可以使用一系列移位和加法计算$x_A + 64i + 4j$

```
# A in %rdi, i in %rsi, j in %rdx
salq    $6, %rsi                    # 64*i
addq    %rsi, %rdi                  # x_A+64*i
movl    (%rdi,%rdx,4),%eax          # M[x_A+64*i+4*j]
ret
```

Fixed and Variable Dimensions

而对于变长数组：size_t n; int A[n][n];

求$A[i][j]$地址必须使用乘法指令对i伸缩n倍

```
# n in %rdi, A in %rsi, i in %rdx, j in %rcx
imulq   %rdx, %rdi                  # n*i
leaq    (%rsi ,%rdi ,4),%rax        # x_A+4*n*i
movl    (%rax,%rcx,4),%eax          # M[x_A+4*n*i+4*j]
ret
```

Declaration

- struct声明，将不同类型的对象聚合到一个对象中，用名字引用各个组成部分
- 结构的所有组成部分都存放在内存中的一段连续的区域内，而指向结构的指针就是结构第一个字节的位置
- 编译器以每个字段的字节偏移为内存引用指令中的位移，从而产生对结构元素的引用

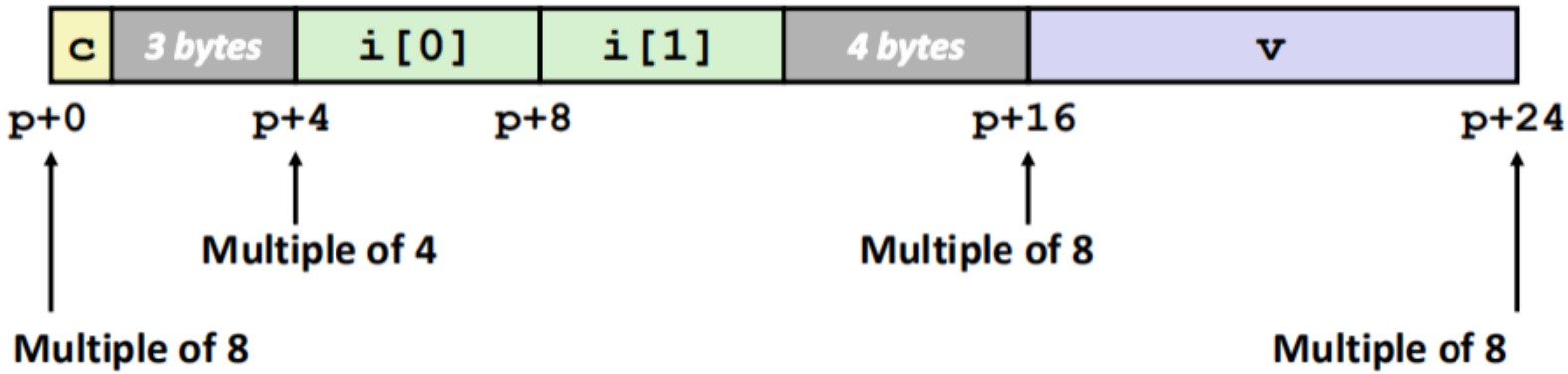
Alignment Principle

- 许多计算机系统要求某种类型对象的地址必须是K(一般是2、4或8)的倍数。其简化了处理器和内存系统间的接口设计。
- 任何K字节的基本对象的地址必须是K的倍数
  - K=1: char
  - K=2: short
  - K=3: int, float
  - K=4: long, double, char*
- 对于包含结构的代码，编译器可能需要在字段的分配中插入间隙，以保证每个结构元素都满足它的对齐要求
- 结构对象本身对其起始地址也有一些对齐要求

Alignment and Structure

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

结构中元素最大的对齐要求决定了该结构整体的对齐要求



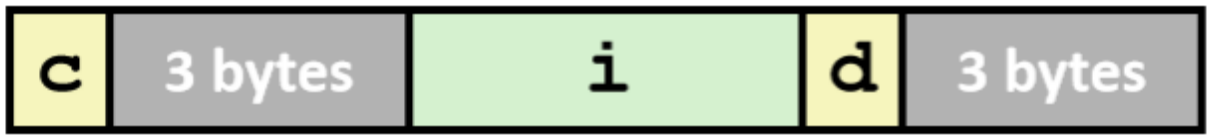
结构对象组成的数组仍然要满足对齐要求，仍包含插入的间隙

Saving Space

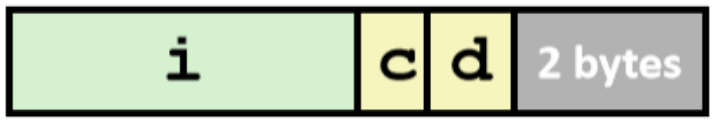
把较大数据类型的元素放在前面

```
struct S4 {
  char c;
  int i;
  char d;
} *p;
```

```
struct S5 {
  int i;
  char c;
  char d;
} *p;
```



12 bytes



8 bytes

XMM Registers

- 浮点数储存在16个YMM寄存器中：%ymm0-%ymm15，每个都是32字节。每个XMM寄存器是对应的YMM寄存器的低16字节
- 对于浮点数，参数按顺序放置在%xmm0,%xmm1,· · · (多于16个参数，多余的参数依次放置于栈中)
- return value放置在%xmm0中
- 所有XMM寄存器都是调用者保存(caller-saved)的，被调用者可以不用保存就覆盖其中任意一个寄存器

Operations

- 只涉及寄存器的FP move指令和涉及内存的FP move指令是不同的
  - movapd %xmm0, %xmm1 为从寄存器传送数据到寄存器
  - movsd (%rdi), %xmm0 为从内存传送数据到寄存器
  - movsd %xmm1, (%rdi) 为从内存传送数据到寄存器
- 浮点数的比较指令ucomiss S_1, S_2(单精度为uncomiss，双精度为uncomisd):$S_2 - S_1$
  - S_2必须在XMM寄存器中，S_1可以在XMM寄存器中，也可以在内存中
  - 设置三个条件码：ZF、CF、PF。
  - PF：对于浮点比较，两个操作数任一个是NaN，即设置该位

Constant FP Value

- AVX浮点数不能以立即数为操作数
  - 编译器必须为所有的常量值分配和初值化储存空间，之后代码把这些值从内存读入
  - vmulsd　.LC2(%rip), %xmm0, %xmm0　从标号为.LC2的内存位置读出值
- 唯一的例外：xorpd　%xmm0, %xmm0　将寄存器XMM0中的值设置为0

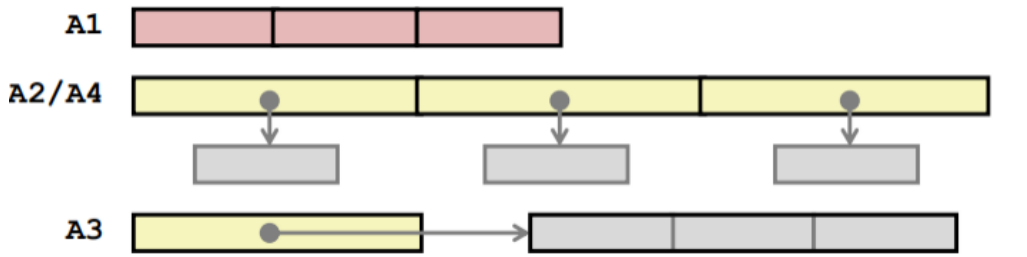
Pointers and Arrays

Decl	An			*An		
	Cmp	Bad	Size	Cmp	Bad	Size
int A1[3]	Y	N	12	Y	N	4
int *A2	Y	N	8	Y	Y	4

- 使用单独的指针存在有bad reference的可能
- 而数组标识符表示的指针永远指向数组开头

Pointers and Arrays

Decl	An			*An			**An		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
int A1[3]	Y	N	12	Y	N	4	N	-	-
int *A2[3]	Y	N	24	Y	N	8	Y	Y	4
int (*A3)[3]	Y	N	8	Y	Y	12	Y	Y	4
int (*A4[3])	Y	N	24	Y	N	8	Y	Y	4



- int *A[N]为指向单个整数的指针组成的数组
- int (*A)[N]为指向指向整数数组开头的指针的指针

Thanks for attention!