Machine-Level Programming: Structs and Y86-64

CSCI 237: Computer Organization
19th Lecture, Wednesday, October 23

Kelly Shaw

1

Administrative Details

- Midterm in lab today
- Read CSAPP Ch. 4.1-4.2
- Snack and Gab today 4:10-4:30 in CS commons
- Gourd Decorating tomorrow at 3:30 in CS commons

2

Last Time: Machine-Level Programming: Arrays and Structs

- Arrays (Ch 3.8)
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- Structures (Ch 3.9)
  - Allocation
  - Access
  - Alignment

3

Today: Arrays, Structs and Y86-64 ISA

- Arrays (Ch 3.8)
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- Structures (Ch 3.9)
  - Allocation
  - Access
  - Alignment
- Y86-64 Instruction Set Architecture
  - Similar state and instructions as x86-64
  - Simpler encodings

4

```
get_herd_value:
.LFB0:
        movslq  %esi, %rsi
        movslq  %edi, %rdi
        leaq    (%rdi,%rdi,4), %rax
        addq    %rax, %rsi
        movl    herd(,%rsi,4), %eax
        ret
herd:
        .long   1
        .long   5
        .long   2
        .long   0
        .long   6
        .long   1
        .long   5
        .long   2
        .long   1
        .long   3
        .long   1
        . . .
```


5

Questions about Nested Arrays?

- Allocated contiguously
- We can locate any element using math
- We will see later that these arrays are "cache friendly"

- However, there are other ways to make 2-D arrays.
  - What if we wanted to assemble an array out of existing arrays?

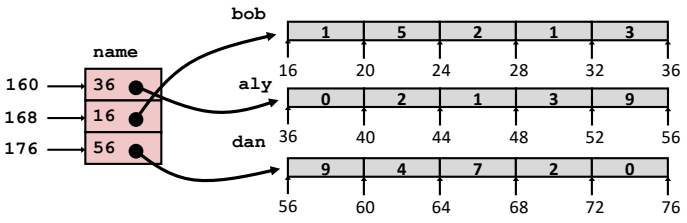
6

Multi-Level Array Example

```
eph_val bob = { 1, 5, 2, 1, 3 };
eph_val aly = { 0, 2, 1, 3, 9 };
eph_val dan = { 9, 4, 7, 2, 0 };
```

```
#define COUNT 3
int *name[COUNT] = {aly, bob, dan};
```

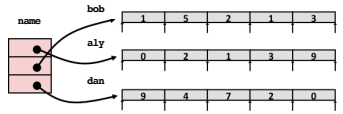
- Variable `name` denotes array of 3 elements
- Each element is a pointer
  - 8 bytes
- Each pointer points to array of `int`'s



7

Element Access in Multi-Level Array

```
int get_name_value
  (size_t index, size_t val)
{
  return name[index][val];
}
```




```
salq    $2, %rsi            # 4*val
addq    name(,%rdi,8), %rsi # p = name[index] + 4*val
movl    (%rsi), %eax        # return *p
ret
```

- Computation
  - Element access **`Mem[Mem[name+8*index]+4*val]`**
  - Must do two memory reads
    - First get pointer to row array
    - Then access element within array

8

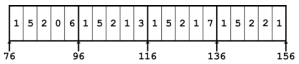
Array Element Accesses Comparison

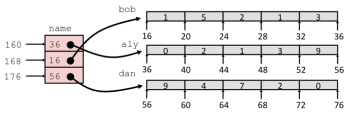
Nested array

```
int get_herd_value
  (size_t index, size_t val)
{
  return herd[index][val];
}
```

Multi-level array

```
int get_name_value
  (size_t index, size_t val)
{
  return name[index][val];
}
```





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

`Mem[herd+20*index+4*val]`

`Mem[Mem[name+8*index]+4*val]`

9

9

Practice on Your Own

- How are elements in a 3 dimensional organized in memory?
- Consider running this code to see:

```
int arr[2][3][4];
for(int i = 0; i < 2; i++){
      for(int j = 0; j < 3; j++){
        for(int k = 0; k < 4; k++){
           printf("arr[%d][%d][%d]: %lx\n", i, j, k,
                    (unsigned long) &arr[i][j][k]);
        }
      }
  }
```

10

10

(Quick) Struct Overview

- Structs are a way to make "composite types" in C
- Syntax:

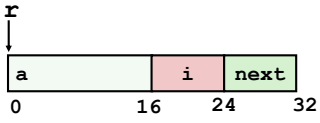
```
struct type_name {
     type_0 name_0;
     type_1 name_1;
     type_2 name_2;
     …
};

…

struct type_name var;
var.name_0 = val;
var.name_2 = another_val;
…
```

11

11

Ch 3.9 - Structure Representation

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



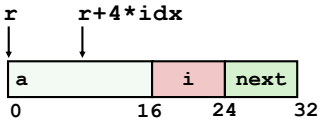
- Structure represented as block of memory
  - **Big enough to hold all of the fields**
- Fields ordered according to declaration
  - **Even if another ordering could yield a more compact representation**
- Compiler determines overall size + positions of fields
  - **Machine-level program has no understanding of the structures in the source code**

12

12

Generating Pointer to Structure Member

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



- Generating Pointer to Array Element
  - Offset of each structure member determined at compile time
  - 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
```

13

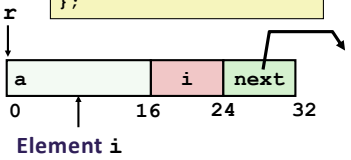
13

Following Linked List

- C Code

```
void set_val
  (struct rec *r, int val)
{
  while (r) {
    int i = r->i;
    r->a[i] = val;
    r = r->next;
  }
}
```

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



Register	Value
%rdi	r
%rsi	val

```
.L11:                           # loop:
  movslq  16(%rdi), %rax        #   i = M[r+16]
  movl    %esi, (%rdi,%rax,4)   #   M[r+4*i] = val
  movq    24(%rdi), %rdi        #   r = M[r+24]
  testq   %rdi, %rdi            #   Test r
  jne     .L11                  #   if !=0 goto loop
```

14

14

Structures & Alignment

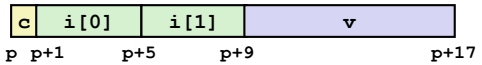
- Aligned Data
  - Primitive data type requires K bytes
  - Address must be multiple of K
  - Required on some machines; advised on x86-64
- Motivation for Aligning Data
  - Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
    - Inefficient to load or store data that spans quad word boundaries
    - *Virtual memory* trickier when data spans 2 pages
- Compiler
  - Inserts gaps in structure to ensure correct alignment of fields

15

15

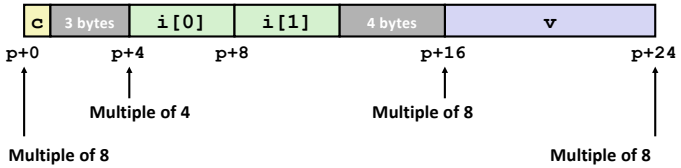
Structures & Alignment

- Unaligned Data




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

- Aligned Data
  - Primitive data type requires K bytes
  - Address must be multiple of K



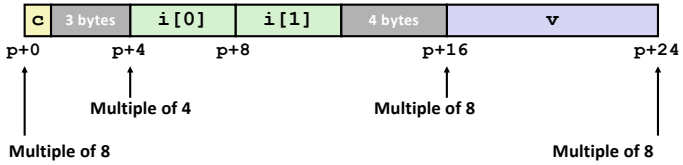
16

16

Satisfying Alignment with Structures

- Within structure:
  - Must satisfy each element's alignment requirement
- Overall structure placement
  - Each structure has alignment requirement K
    - K = Largest alignment of any element
  - Initial address & structure length must be multiples of K
- Example:
  - K = 8, due to **double** element

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



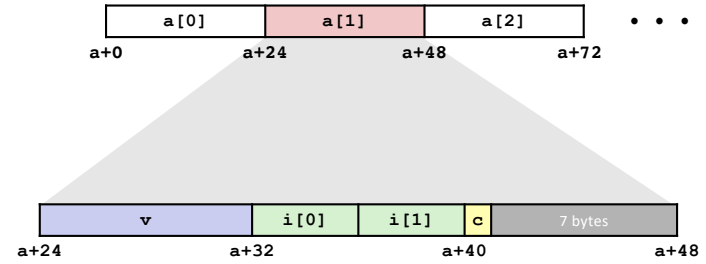
17

17

Arrays of Structures

- Overall structure length multiple of K
- Satisfy alignment requirement for every element

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



18

18

Ch 3.8 & 3.9 Summary

- Arrays
  - Elements packed into contiguous region of memory
  - Use index arithmetic to locate individual elements
- Structures
  - Elements packed into single region of memory
  - Access using offsets determined by compiler
  - Possible require internal and external padding to ensure alignment

19

19

Practice on Your Own

- What is the size of the following struct (assuming alignment requirements are adhered to)?

```
struct item{
    char str[10];
    long id;
    int num;
};
```

- If the address of an instance of this struct is stored in %rdi, write the assembly code to read the value of the `id` field into register %rsi.

20

20