outlineL22-w11TR-student

Thursday, November 17, 2022 12:57 PM

outlineL22-w11TR-st...

CS 354 - Machine Organization & Programming
Tuesday Nov15, Thursday Nov 17, 2022

Exam Results expected by Friday Nov 18
Homework hw6: DUE on or before Monday Nov 21
Homework hw7: DUE on or before Monday Nov 28
Project p5: DUE on or before Friday Dec 2

Last Week

Instructions - SET Instructions - Jumps Encoding Targets Converting Loops	The Stack from a Programmer's Perspective The Stack and Stack Frames Instructions - Transferring Control Register Usage Conventions Function Call-Return Example

This Week

Function Call-Return Example (L20 p7) Recursion Stack Allocated Arrays in C Stack Allocated Arrays in Assembly Stack Allocated Multidimensional Arrays	Stack Allocated Structs Alignment Alignment Practice Unions

Next Week: Pointers in Assembly, Stack Smashing, and Exceptions
B&O 3.10 Putting it Together: Understanding Pointers
3.12 Out-of-Bounds Memory References and Buffer Overflow

8.1 Exceptions
8.2 Processes
8.3 System Call Error Handling
8.4 Process Control through p719

Recursion

Use a stack trace to determine the result of the call `fact(3)`:

```
int fact(int n) {
   int result;
   if (n <= 1) result = 1;
   else        result = n * fact(n - 1);
   return result;
}
```

direct recursion

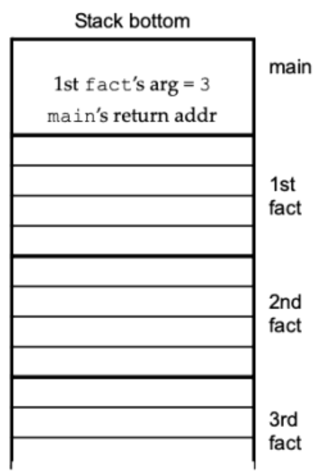
recursive case

base case

"infinite" recursion

Assembly Trace

```
fact:
   pushl %ebp
   movl %esp, %ebp
   pushl %ebx
   subl $4,%esp

   movl 8(%ebp),%ebx
   movl $1,%eax

   cmpl $1,%ebx
   jle .L1

   leal -1(%ebx),%eax
   movl %eax,(%esp)
   call fact

   imull %ebx,%eax

.L1:
   addl $4,%esp
   popl %ebx
   popl %ebp
   ret
```



- *"Infinite" recursion causes*

- *When tracing functions in assembly code*

Stack Allocated Arrays in C

Recall Array Basics

`T A[N];` where *T* is the element datatype of size *L* bytes and *N* is the number of elements

1. Contiguous region of stack L * N bytes

2. Identifier associated with the start of the array

❋ *The elements of* `A` Are accessed using address arithmetic

Recall Array Indexing and Address Arithmetic

L = element size
X(a) = start address of the array

`&A[i]` $\equiv$ A + i = x(a) + L * i

(index)

→ For each array declarations below, what is L (element size), the address arithmetic for the ith element, and the total size of the array?

C code	L	address of ith element	total array size
1. `int I[11]`	4	X(i) + 4i	4y
2. `char C[7]`			
3. `double D[11]`			
4. `short S[42]`	2	X(s)+2i	84
5. `char *C[13]`			
6. `int **I[11]`	4	X(i) + 4i	44
7. `double *D[7]`			

Stack Allocated Arrays in Assembly

Arrays on the Stack

→ How is an array laid out on the stack? Option 1 or 2:

higher addresses	
earlier frames	
1.	**2.**
A[0]	A[N-1]
A[1]	...
...	A[1]
A[N-1]	A[0]
Stack Top	

Stack grows always to the lower address

❋ *The first element (index 0) of an array*

Is closest to the top of the stack.

Accessing 1D Arrays in Assembly

IA-32 is designed to simplify array access

Assume array's start address in %edx and index is in %ecx

`movl (%edx, %ecx, 4), %eax` ≡ M[x(a) + 4*i) = A[i]

x_A i

In C: *(A+i)

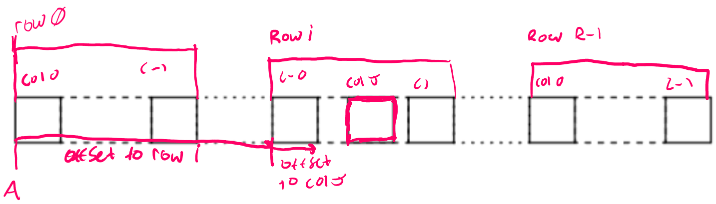
→ Assume `I` is an `int` array, `S` is a `short int` array, for both the array's start address is in %edx, and the index `i` is in %ecx. Determine the element type and instruction for each:

C code	type	assembly instruction to move C code's value into %eax
1. `I`	Int*	X(I) — Movl %edx, %eax
2. `I[0]`	int	M[x_i] Movl (%eax, %edx)
3. `*I`		
4. `I[i]`		
✱ 5. `&I[2]`	int*	x_i+(2·4) leal 8(%edx,) %eax
✱ 6. `I+i-1`	int*	x_I+(4·i)-4 cast correct datatyp leal -4(%edx,%ecx,4) %eax
7. `*(I+i-3)`		
8. `S[3]`		
9. `S+1`		
10. `&S[i]`		
11. `S[4*i+1]`	short M[x_S+2(4·i)+2] movws 2(%edx,%ecx,8), %eax	
12. `S+i-5`		

Stack Allocated Multidimensional Arrays

Recall 2D Array Basics

T `A[R][C];` where *T* is the element datatype of size *L* bytes,
R is the number of rows and *C* is the number of columns



✱ *Recall that 2D arrays are stored on the stack* In row major order

```
int A[5][3];          typedef int row_t[3];    ← col per row
                      row_t A[5];              rows
```

Accessing 2D Arrays in Assembly

`&A[i][j]` ≡ Start address + offset to row i + offset to col J.

$$X_A + (C * i * L) + (L * J)$$

Given array A as declared above, if x_A in %eax, i in %ecx, j in %edx
then `A[i][j]` in assembly is:

```
                 i+i·2
leal (%ecx, %ecx, 2), %ecx        3i in %ecx
sall $2, %edx                     4j
addl %eax, %edx               (X_A + 4j)
movl (%edx, %ecx, 4), %eax  [X_A + (3i·4) + 4j] → %eax
```

Compiler Optimizations

- If only accessing part of array

  Compiler points to that part of the array

  Then offset from there

- If taking a fixed stride through the array

  Compiler uses stride * element size as offset

Stack Allocated Structures

Structures on the Stack

```
struct iCell {
  int x;
  int y;
  int c[3];
  int *v;
};
```

# byte	offset	Hex
4	0	0x0
4	4	0x4
12	8	0x8
4	20	0x14

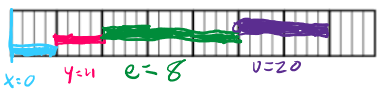
higher addresses

earlier frames	
1.	**2.**
v	x
c[2]	y
c[1]	c[0]
c[0]	c[1]
y	c[2]
x	v

Stack Top

→ How is a structure laid out on the stack? Option 1 or 2:

The compiler
- associates data member names with the offset of the struct, from the start of the struct



- use address arithmetic with offset to access data members

❋ *The first data member of a structure* is closest to the top of the stack

Accessing Structures in Assembly

Given:

```
struct iCell ic = //assume ic is initialized
void function(iCell *ip) {
```

→ Assume `ic` is at the top of the stack, %edx stores `ip` and %esi stores `i`.
Determine for each the assembly instruction to move the C code's value into %eax:

C code	assembly
1. `ic.v`	movl 20(%esp), %eax
2. `ic.c[i]`	
3. `ip->x`	movw (%edx) ,%eax
4. `ip->y`	
5. `&ip->c[i]`	leal 8(%edx, %esi, 4), %eax

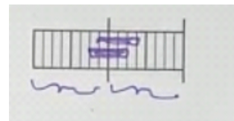
❋ *Assembly code to access a structure* does not have data member names, only offsets!

Alignment

What? Most systems restrict the addresses where primitive data can be stored.

Why? Primarily for performance + cheaper hardware

Example: Assume cpu reads 8 byte words
f is a misaligned float



It's slow: misalignment can take multiple reads
2 reads, extracts, combines

Restrictions

IA-32 has no alignment restrictions

Linux: short Must be multiple of two
int, float, pointer, double Must be multiple of four

Windows: same as Linux except
double Must be multiple of eight

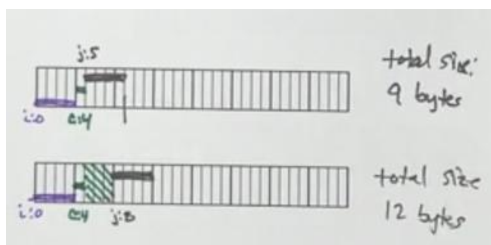
Implications

Going to add padding to make the alignment correct by the compiler

Structure Example

```
struct s1 {
   int i;
   char c;
   int j;
};
```

Padding vs. no Padding



⁕ *The total size of a structure* Is typically a multiple of it's largest data member size
-In this diagram it would be four, so it's a multiple of four

Alignment Practice

→ For each structure below, complete the memory layout and determine the total bytes allocated.

```
1) struct sA {
     int i;
     int j;
     char c;
   };
```

```
2) struct sB {
     char a;
     char b;
     char c;
   };
```

```
3) struct sC {
     char c;
     short s;
     int i;
     char d;
   };
```

```
4) struct sD {
     short s;
     int i;
     char c;
   };
```

```
5) struct sE {
     int i;
     short s;
     char c;
   };
```

❋ *The order that a structure's data members are listed*

Can affect memory utilization

Unions

What? A *union* is

- Like a struct except fields share same memory
- Allocates only enough memory for largest field

Why?

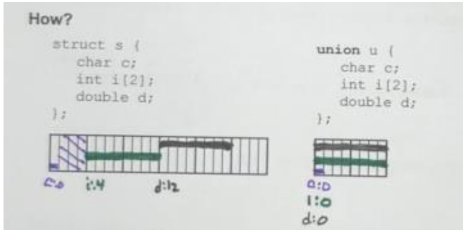
- Allows data to be accessed as different types
- Used to access hardware
-

How? Implement "polymorphism"

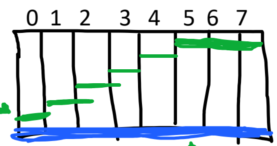
How?

◆

How? Implement "polymorphism"



Example




```
typedef union {
   unsigned char cntrlrByte;
   struct {
      unsigned char playbutn   : 1;
      unsigned char pausebutn  : 1;
      unsigned char ctrlbutn   : 1;
      unsigned char fire1butn  : 1;
      unsigned char fire2butn  : 1;
      unsigned char direction  : 3;
   } bits;
} CntrlrReg;
```

CntrlrReg c1;
readctrlReg(c1.cntrlrByte); //this reads all 8 bits into c1, hypothetically
If (c1.bits.fire1butn) { … fireshot();} //we are using the union here to make it convenient for different uses