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

<table>
<thead>
<tr>
<th>Instructions - SET</th>
<th>The Stack from a Programmer’s Perspective</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions - Jumps</td>
<td>The Stack and Stack Frames</td>
</tr>
<tr>
<td>Encoding Targets</td>
<td>Instructions - Transferring Control</td>
</tr>
<tr>
<td>Converting Loops</td>
<td>Register Usage Conventions</td>
</tr>
<tr>
<td></td>
<td>Function Call-Return Example</td>
</tr>
</tbody>
</table>

### This Week

<table>
<thead>
<tr>
<th>Function Call-Return Example (L20 p7)</th>
<th>Stack Allocated Structs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recursion</td>
<td>Alignment</td>
</tr>
<tr>
<td>Stack Allocated Arrays in C</td>
<td>Alignment Practice</td>
</tr>
<tr>
<td>Stack Allocated Arrays in Assembly</td>
<td>Unions</td>
</tr>
<tr>
<td>Stack Allocated Multidimensional Arrays</td>
<td></td>
</tr>
</tbody>
</table>

**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):

```c
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

\[ T \ A[N]; \quad \text{where } T \text{ is the element datatype of size } L \text{ bytes and } N \text{ is the number of elements} \]

1.

2.

* The elements of A

Recall Array Indexing and Address Arithmetic

\&A[i]

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

<table>
<thead>
<tr>
<th>C code</th>
<th>( L )</th>
<th>address of ( i )th element</th>
<th>total array size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. int I[11]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. char C[7]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. short S[42]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. char *C[13]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6. int **I[11]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7. double *D[7]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Stack Allocated Arrays in Assembly

Arrays on the Stack

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

∧ The first element (index 0) of an array

<table>
<thead>
<tr>
<th>higher addresses</th>
<th>earlier frames</th>
</tr>
</thead>
<tbody>
<tr>
<td>2. A[1]</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

Stack Top

Accessing 1D Arrays in Assembly

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

movl (%edx, %ecx, 4), %eax

→ 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:

<table>
<thead>
<tr>
<th>C code</th>
<th>type</th>
<th>assembly instruction to move C code’s value into %eax</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. I</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. I[0]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. *I</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. I[i]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. &amp;I[2]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6. I+i-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7. *(I+i-3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8. S[3]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9. S+1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10. &amp;S[i]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11. S[4*i+1]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12. S+i-5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

\[ \begin{array}{cccccccc}
  & & & & & & & \\
  & & & & & & & \\
  & & & & & & & \\
  & & & & & & & \\
  & & & & & & & \\
  & & & & & & & \\
\end{array} \]

\* Recall that 2D arrays are stored on the stack

\[
\text{int A}[5][3]; \quad \text{typedef int row_t}[3]; \\
\text{row_t A}[5];
\]

Accessing 2D Arrays in Assembly

\&A[i][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:

\[
\begin{align*}
\text{lea}l & \ (%ecx, \ %ecx, \ 2), \ %ecx \\
\text{sal} & l \ %edx \\
\text{add}l & \ %eax, \ %edx \\
\text{mov}l & \ (%edx, \ %ecx, \ 4), \ %eax
\end{align*}
\]

Compiler Optimizations

\* If only accessing part of array

\* If taking a fixed stride through the array
Stack Allocated Structures

Structures on the Stack

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

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

The compiler

The first data member of a structure

Accessing Structures in Assembly

Given:

```c
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`:

<table>
<thead>
<tr>
<th>C code</th>
<th>assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ic.v</td>
<td>v</td>
</tr>
<tr>
<td>2. ic.c[i]</td>
<td>x, c[i]</td>
</tr>
<tr>
<td>3. ip-&gt;x</td>
<td>x</td>
</tr>
<tr>
<td>4. ip-&gt;y</td>
<td>y</td>
</tr>
<tr>
<td>5. &amp;ip-&gt;c[i]</td>
<td>v, c[i]</td>
</tr>
</tbody>
</table>

Assembly code to access a structure

Copyright © 2016-2022 Jim Skrentny
Alignment

What?

Why?

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

Restrictions

Linux: short
      int, float, pointer, double

Windows: same as Linux except
double

Implications

Structure Example

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

※ The total size of a structure
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
Unions

What? A **union** is

- 

- 

Why?

- 

- 

How?

```c
struct s {
    char c;
    int i[2];
    double d;
};
union u {
    char c;
    int i[2];
    double d;
};
```

Example

```c
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;
```