Building an Executable
CSE 351 Summer 2018

Instructor:
Justin Hsia

Teaching Assistants:
Josie Lee    Natalie Andreeva    Teagan Horkan

http://xkcd.com/1790/
Administrivia

- Lab 2 due Monday (7/16)
- Homework 3 due 7/23

- **Midterm** Wednesday (7/18, in lecture)
  - Make a cheat sheet! – two-sided letter page, *handwritten*
  - Check Piazza for announcements
  - **Review session** 5:00-6:30 pm on Monday (7/16) in EEB 105
Procedures

- Stack Structure
- Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
- Register Saving Conventions
- Illustration of Recursion
Recursive Function

/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0) 
        return 0;
    else 
        return (x&1)+pcount_r(x >> 1);
}

Compiler Explorer:
https://godbolt.org/g/W8DxeR
• Compiled with -O1 for brevity
  instead of -Og
• Try -O2 instead!

movl  $0, %eax
testq %rdi, %rdi
je   .L6
pushq %rbx
movq %rdi, %rbx
shrq %rdi
call pcount_r
andl $1, %ebx
addq %rbx, %rax
popq %rbx
.L6:
rep ret
Recursive Function: Base Case

```c
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x&1)+pcount_r(x >> 1);
}
```

```assembly
pcount_r:
    movl $0, %eax
    testq %rdi, %rdi
    je .L6
    pushq %rbx
    movq %rdi, %rbx
    shrq %rdi
    call pcount_r
    andl $1, %ebx
    addq %rbx, %rax
    popq %rbx
.L6:
    rep ret
```

**Register Use(s) Type**

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>x</td>
<td>Argument</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
<td>Return value</td>
</tr>
</tbody>
</table>

Trick because some AMD hardware doesn’t like jumping to `ret`

(don’t worry about it)
Recursive Function: **Callee Register Save**

```c
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x&1)+pcount_r(x >> 1);
}
```

Need original value of `x` *after* recursive call to `pcount_r`.

“Save” by putting in `%rbx` (*callee saved*), but need to save old value of `%rbx` before you change it.

The Stack

```
... 
%rbx
saved, %rbx 
%rsp →
```

---

### pcount_r:

```
.movl $0, %eax
.testq %rdi, %rdi
.je .L6
.pushq %rbx
.movq %rdi, %rbx
.shrq %rdi
.call pcount_r
.andl $1, %ebx
.addq %rbx, %rax
.popq %rbx
.L6:
    rep ret
```
Recursive Function: Call Setup

```c
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x&1)+pcount_r(x >> 1);
}
```

**Register Use(s) Type**

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>x (new)</td>
<td>Argument</td>
</tr>
<tr>
<td>%rbx</td>
<td>x (old)</td>
<td>Callee saved</td>
</tr>
</tbody>
</table>

**The Stack**

```
%rsp →
  saved %rbx
  rtn <main+?>
  ...  
```

**pcount_r:**

- `movl $0, %eax`
- `testq %rdi, %rdi`
- `je .L6`
- `pushq %rbx`
- `movq %rdi, %rbx`
- `shrq %rdi`
- `call pcount_r`
- `andl $1, %ebx`
- `addq %rbx, %rax`
- `popq %rbx`

```
.L6:
  rep ret
```
Recursive Function: Call

```c
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x&1)+pcount_r(x >> 1);
}
```

### Register Use(s) Type

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Recursive call return value</td>
<td>Return value</td>
</tr>
<tr>
<td>%rbx</td>
<td>x (old)</td>
<td>Callee saved</td>
</tr>
</tbody>
</table>

### The Stack

- **frames**
- **main**
- **pcount_r**, \%rdi = 0b1
- **pcount_r**, \%rdi = 0b10
- **pcount_r**, \%rdi = 0b11
- **pcount_r**, \%rdi = 0b110
- **saved**, \%rbx = 0b1
- **pcount_r**, \%rdi = 0b110
- **pcount_r**, \%rdi = 0b111
- **pcount_r**, \%rdi = 0b1110
- **pcount_r**, \%rdi = 0b1111

### pcount_r:
```
  movl $0, %eax
  testq %rdi, %rdi
  je .L6
  pushq %rbx
  movq %rdi, %rbx
  shrq %rdi
  call pcount_r
  andl $1, %ebx
  addq %rbx, %rax
  popq %rbx
.L6:
  rep ret
```
Recursive Function: Result

```c
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x&1)+pcount_r(x >> 1);
}
```

The Stack

```
%rsp →
saved %rbx
rtn <main+?>
...
```

PCOUNT_R:
```
movl $0, %eax
testq %rdi, %rdi
je .L6
pushq %rbx
movq %rdi, %rbx
shrq %rdi
call pcount_r
andl $1, %ebx
addq %rbx, %rax
popq %rbx
.L6:
    rep ret
```
Recursive Function: Completion

```c
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x&1)+pcount_r(x >> 1);
}
```

```
pcount_r:
movl $0, %eax
testq %rdi, %rdi
je .L6
pushq %rbx
movq %rdi, %rbx
shrq %rdi
call pcount_r
andl $1, %ebx
addq %rbx, %rax
popq %rbx
popq %rbx

.L6:
rep ret
```

**Register Use(s) Type**

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Return value</td>
<td>Return value</td>
</tr>
<tr>
<td>%rbx</td>
<td>Previous %rbx</td>
<td>Callee restored</td>
</tr>
</tbody>
</table>

**The Stack**

```
%rsp →
...
```

- rtn <main+?>
- saved %rbx

- move $0, %eax
- testq %rdi, %rdi
- je .L6
- pushq %rbx
- movq %rdi, %rbx
- shrq %rdi
- call pcount_r
- andl $1, %ebx
- addq %rbx, %rax
- popq %rbx
- popq %rbx

.L6: restore before returning
- rep ret
Observations About Recursion

- Works without any special consideration
  - Stack frames mean that each function call has private storage
    - Saved registers & local variables
    - Saved return pointer
  - Register saving conventions prevent one function call from corrupting another’s data
    - Unless the code explicitly does so (e.g. buffer overflow)
  - Stack discipline follows call / return pattern
    - If P calls Q, then Q returns before P
    - Last-In, First-Out (LIFO)

- Also works for mutual recursion (P calls Q; Q calls P)
x86-64 Stack Frames

- Many x86-64 procedures have a minimal stack frame
  - Only return address is pushed onto the stack when procedure is called

- A procedure *needs* to grow its stack frame when it:
  - Has too many local variables to hold in *caller*-saved registers
  - Has local variables that are arrays or structs
  - Uses `&` to compute the address of a local variable
  - Calls another function that takes more than six arguments
  - Is using *caller*-saved registers and then calls a procedure
  - Modifies/uses *callee*-saved registers
x86-64 Procedure Summary

- Important Points
  - Procedures are a combination of instructions and conventions
    - Conventions prevent functions from disrupting each other
  - Stack is the right data structure for procedure call/return
    - If P calls Q, then Q returns before P
  - Recursion handled by normal calling conventions
- Heavy use of registers
  - Faster than using memory
  - Use limited by data size and conventions
- Minimize use of the Stack
Roadmap

C:

car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);

Java:

Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg = c.getMPG();

Assembly language:

get_mpg:
  pushq %rbp
  movq %rsp, %rbp
  ...
  popq %rbp
  ret

Machine code:

0111010000011000
1000110100000100000000101000100111000010110000011111101000011111

Computer system:

Memory & data
Integers & floats
x86 assembly
Procedures & stacks
Executables
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C
Building an Executable from a C File

- Code in files p1.c p2.c
- Compile with command: gcc -Og p1.c p2.c -o p
  - Put resulting machine code in file p
- Run with command: ./p
Compiler

- **Input:** Higher-level language code (e.g. C, Java)
  - foo.c

- **Output:** Assembly language code (e.g. x86, ARM, MIPS)
  - foo.s

- First there’s a preprocessor step to handle `#`directives
  - Macro substitution, plus other specialty directives

- Super complex, whole courses devoted to these!

- Compiler optimizations
  - “Level” of optimization specified by capital ‘O’ flag (e.g. `-Og`, `-O3`)
Compiling Into Assembly

- C Code (sum.c)

```c
void sumstore(long x, long y, long *dest) {
    long t = x + y;
    *dest = t;
}
```

- x86-64 assembly (gcc -Og -S sum.c)

  ```
  sumstore(long, long, long*):
  addq %rdi, %rsi
  movq %rsi, (%rdx)
  ret
  ```

**Warning:** You may get different results with other versions of gcc and different compiler settings
Assembler

- **Input:** Assembly language code (e.g. x86, ARM, MIPS)
  - foo.s

- **Output:** Object files (e.g. ELF, COFF)
  - foo.o
  - Contains *object code* and *information tables*

- Reads and uses *assembly directives*
  - e.g. `.text`, `.data`, `.quad`
  - x86: [https://docs.oracle.com/cd/E26502_01/html/E28388/eoiyg.html](https://docs.oracle.com/cd/E26502_01/html/E28388/eoiyg.html)

- Produces “machine language”
  - Does its best, but object file is *not* a completed binary

- **Example:** `gcc -c foo.s`
Producing Machine Language

- **Simple cases**: arithmetic and logical operations, shifts, etc.
  - All necessary information is contained in the instruction itself

- What about the following?
  - Conditional jump
  - Accessing static data (e.g. global var or jump table)
  - call `addr/label`

- Addresses and labels are problematic because final executable hasn’t been constructed yet!
  - So how do we deal with these in the meantime?
Object File Information Tables

- **Symbol Table** holds list of “items” that may be used by other files
  - *Non-local labels* – function names for call
  - *Static Data* – variables & literals that might be accessed across files

- **Relocation Table** holds list of “items” that this file needs the address of later (currently undetermined)
  - Any *label* or piece of *static data* referenced in an instruction in this file
    - Both internal and external

- Each file has its own symbol and relocation tables
Object File Format

1. **object file header**: size and position of the other pieces of the object file “table of contents”

2. **text segment**: the machine code (Instructions)

3. **data segment**: data in the source file (binary) (Static Data & Literals)

4. **relocation table**: identifies lines of code that need to be “handled”

5. **symbol table**: list of this file’s labels and data that can be referenced

6. **debugging information**: (info for GDB)

- More info: ELF format
  - [http://www.skyfree.org/linux/references/ELF_Format.pdf](http://www.skyfree.org/linux/references/ELF_Format.pdf)
Linker

- **Input:** Object files (e.g. ELF, COFF)
  - foo.o
- **Output:** executable binary program
  - a.out
- Combines several object files into a single executable (linking)
- Enables separate compilation/assembling of files
  - Changes to one file do not require recompiling of whole program
Linking

1) Put together text segments from each .o file
2) Put together data segments from each .o file and concatenate this onto the end of the text segments
3) Resolve References
   ▪ Go through Relocation Table; handle each entry
Disassembling Object Code

- Disassembled:

```
0000000000400536 <sumstore>:
  400536:  48 01 fe     add  %rdi,%rsi
  400539:  48 89 32    mov  %rsi,(%rdx)
  40053c:  c3          retq
```

- **Disassembler** *(objdump -d sum)*
  - Useful tool for examining object code *(man 1 objdump)*
  - Analyzes bit pattern of series of instructions
  - Produces approximate rendition of assembly code
  - Can run on either a.out (complete executable) or .o file
What Can be Disassembled?

% objdump -d WINWORD.EXE

WINWORD.EXE: file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000:
30001001:
30001003:
30001005:
3000100a:

- Anything that can be interpreted as executable code
- Disassembler examines bytes and attempts to reconstruct assembly source

Reverse engineering forbidden by Microsoft End User License Agreement
Loader

- **Input:** executable binary program, command-line arguments
  - `/a.out arg1 arg2`
- **Output:** `<program is run>`

- Loader duties primarily handled by OS/kernel
  - More about this when we learn about processes
- Memory sections (Instructions, Static Data, Literals, Stack) are set up
- Registers are initialized
Roadmap

C:
```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:
```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
c.getMPG();
```

Assembly language:
```
get_mpg:
    pushq %rbp
    movq %rsp, %rbp
    ...
    popq %rbp
    ret
```

Machine code:
```
0111010000011000
100011010000010000000010
1000100111000010
1100000111111101000011111
```

OS:
```
Windows 10
OS X Yosemite
```

Memory & data
Integers & floats
x86 assembly
Procedures & stacks
Executables
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C
Data Structures in Assembly

- **Arrays**
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level

- **Structs**
  - Alignment

- **Unions**
Array Allocation

- **Basic Principle**
  - `T A[N];` → array of data type `T` and length `N`
  - *Contiguously* allocated region of `N*sizeof(T)` bytes
  - Identifier `A` returns address of array (type `T*`)

```c
char msg[12];
int val[5];
double a[3];
char* p[3];
(or char *p[3];)
```
Array Access

- **Basic Principle**
  - `T A[N];` → array of data type `T` and length `N`
  - Identifier `A` returns address of array (type `T*`)

- **Reference**

<table>
<thead>
<tr>
<th>x[4]</th>
<th>int</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>int*</td>
<td>a</td>
</tr>
<tr>
<td>x+1</td>
<td>int*</td>
<td>a+4</td>
</tr>
<tr>
<td>&amp;x[2]</td>
<td>int*</td>
<td>a+8</td>
</tr>
<tr>
<td>x[5]</td>
<td>int</td>
<td>??  (whatever’s in memory at addr x+20)</td>
</tr>
<tr>
<td>*(x+1)</td>
<td>int</td>
<td>7</td>
</tr>
<tr>
<td>x+i</td>
<td>int*</td>
<td>a+4*i</td>
</tr>
</tbody>
</table>
**Array Example**

```c
typedef int zip_dig[5];
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig uw = { 9, 8, 1, 9, 5 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```

- **typedef**: Declaration "`zip_dig uw`" equivalent to "`int uw[5]`"

**Initialization**

- `typedef`: Declaration "`zip_dig uw`" equivalent to "`int uw[5]`"
Array Example

typedef int zip_dig[5];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zid_dig uw = { 9, 8, 1, 9, 5 };
zid_dig ucb = { 9, 4, 7, 2, 0 };

- Example arrays happened to be allocated in successive 20 byte blocks
  - Not guaranteed to happen in general
    - Could have allocated variables in-between
Array Accessing Example

```c
typedef int zip_dig[5];

int get_digit(zip_dig z, int digit)
{
    return z[digit];
}
```

- Register `%rdi` contains starting address of array
- Register `%rsi` contains array index
- Desired digit at `%rdi+4*%rsi`, so use memory reference (%rdi, %rsi, 4)
# Referencing Examples

```c
typedef int zip_dig[5];
```

```c
zip_dig cmu;
zip_dig uw;
zip_dig ucb;
```  

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value</th>
<th>Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>uw[3]</td>
<td>36 + 3*4 = 48</td>
<td>9</td>
<td>Yes</td>
</tr>
<tr>
<td>uw[6]</td>
<td>36 + 6*4 = 60</td>
<td>4</td>
<td>No</td>
</tr>
<tr>
<td>uw[-1]</td>
<td>36 + (-1)*4 = 32</td>
<td>3</td>
<td>No</td>
</tr>
<tr>
<td>cmu[15]</td>
<td>16 + 15*4 = 76</td>
<td>?</td>
<td>No</td>
</tr>
</tbody>
</table>

- No bounds checking
- Example arrays happened to be allocated in successive 20 byte blocks
  - Not guaranteed to happen in general
Array Loop Example

\[
\begin{align*}
zi &= 10 \times 0 + 9 = 9 \\
zi &= 10 \times 9 + 8 = 98 \\
zi &= 10 \times 98 + 1 = 981 \\
zi &= 10 \times 981 + 9 = 9819 \\
zi &= 10 \times 9819 + 5 = 98195
\end{align*}
\]

```c
typedef int zip_dig[5];

int zd2int(zip_dig z)
{
    int i;
    int zi = 0;
    for (i = 0; i < 5; i++) {
        zi = 10 * zi + z[i];
    }
    return zi;
}
```
Array Loop Example

- **Original:**

- **Transformed:**
  - Eliminate loop variable \( i \), use pointer \( zend \) instead
  - Convert array code to pointer code
    - Pointer arithmetic on \( z \)
  - Express in do-while form (no test at entrance)

```
int zd2int(zip_dig z)
{
    int zi = 0;
    int *zend = z + 5;
    do {
        zi = 10 * zi + *z;
        z++;
    } while (z < zend);
    return zi;
}
```
Array Loop Implementation

- Registers:
  - `%rdi` `z`
  - `%rax` `zi`
  - `%rcx` `zend`

- Computations
  - `10^z_i + *z` implemented as `*z + 2(5z_i)`
  - `z++` increments by 4 bytes (size of `int`)

```c
int zd2int(zip_dig z)
{
    int zi = 0;
    int *zend = z + 5;
    do {
        zi = 10 * zi + *z;
        z++;
    } while (z < zend);
    return zi;
}
```

```assembly
# %rdi = z
leaq 20(%rdi),%rcx
movl $0,%eax
.L17:
lea (%rax,%rax,4),%edx
movl (%rdi),%eax
lea (%rax,%rdx,2),%eax
addq $4,%rdi
cmpq %rdi,%rcx
jne .L17
```

```
# %rdi = z
leaq 20(%rdi),%rcx
movl $0,%eax
.L17:
lea (%rax,%rax,4),%edx
movl (%rdi),%eax
lea (%rax,%rdx,2),%eax
addq $4,%rdi
cmpq %rdi,%rcx
jne .L17
```

```
# %rdi = z
leaq 20(%rdi),%rcx
movl $0,%eax
.L17:
lea (%rax,%rax,4),%edx
movl (%rdi),%eax
lea (%rax,%rdx,2),%eax
addq $4,%rdi
cmpq %rdi,%rcx
jne .L17
```
C Details: Arrays and Pointers

- Arrays are (almost) identical to pointers
  - `char *string` and `char string[]` are nearly identical declarations
  - Differ in subtle ways: initialization, `sizeof()`, etc.

- An array name looks like a pointer to the first (0th) element
  - `ar[0]` same as `*ar`; `ar[2]` same as `*(ar+2)`

- An array variable is read-only (no assignment)
  - Cannot use "`ar = <anything>`"
C Details: Arrays and Functions

- Declared arrays only allocated while the scope is valid:

  ```
  char* foo() {
      char string[32]; ...;
      return string;
  }
  ```

- An array is passed to a function as a pointer:
  - Array size gets lost!
  ```
  int foo(int ar[], unsigned int size) {
      ... ar[size-1] ...
  }
  ```

- Must explicitly pass the size!

- An array is passed to a function as a pointer: array size gets lost!

- Really int *ar (%rdi can only fit 8 bytes)

- Data will get overwritten by future stack frames

- Returns stack addr that is < %rsp