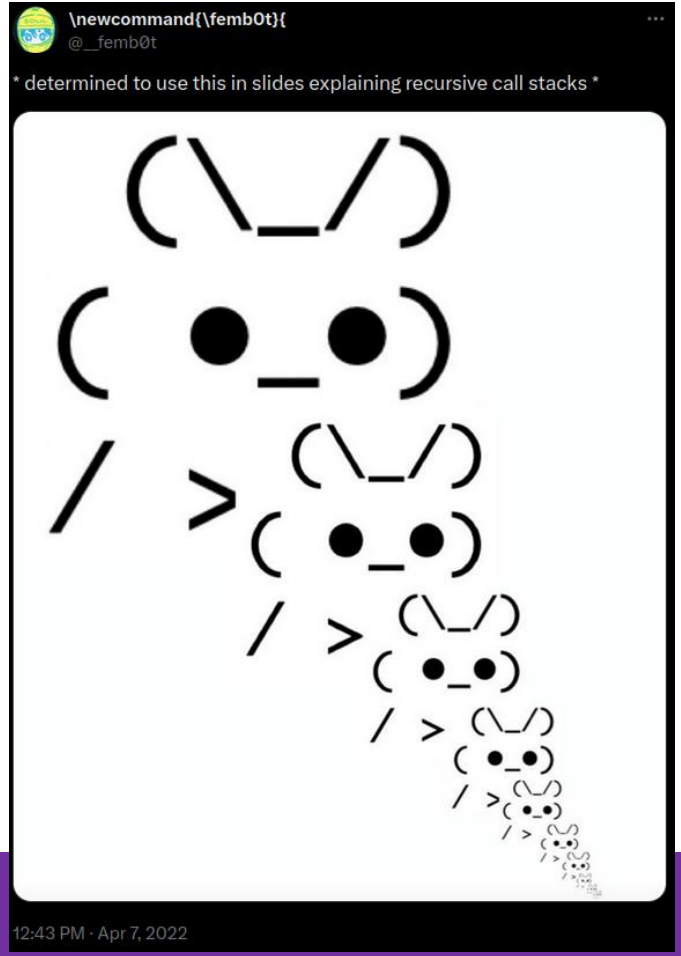
Procedures II & Executables

CSE 351 Summer 2024

Instructor:
Ellis Haker

Teaching Assistants:
Naama Amiel
Micah Chang
Shananda Dokka
Nikolas McNamee
Jiawei Huang



Administrivia

- Today
  - HW9 due (11:59pm)
  - **Quiz 1 due (11:59pm)!!!!!**
- Monday, 7/15
  - RD12 due (1pm)
  - HW10 due (11:59pm)
  - **Midterm Survey out on Canvas**
- Wednesday, 7/17
  - RD13 due (1pm)
  - HW11 due (11:59pm)
  - **Midterm Survey Due (11:59pm)**

please start Lab 2 early!

Aside: Lab2 Extra Credit

- All labs from now on will have some extra credit
- Separate Gradescope assignment
- Not worth a significant amount of credit! ← no points assigned
  - I will *maybe* bump your grade up if you're on the borderline

Lecture Topics

- Stack structure
- Calling conventions
  - Passing control
  - Passing data
  - Managing local data
- **Stack frames**
  - Saved registers
  - **Stack layout**
  - Register saving convention
- Illustration of Recursion
- Executables
  - CALL
  - Object Files

Review Questions

see drawing on next slide

Answer the following questions about when `main()` is run (assume `x` and `y` are stored on the stack):

```c
int main() {
    int i, x = 0;
    for(i=0; i<3; i++)
        x = randSum(x);
    printf("x = %d\n",x);
    return 0;
}
```

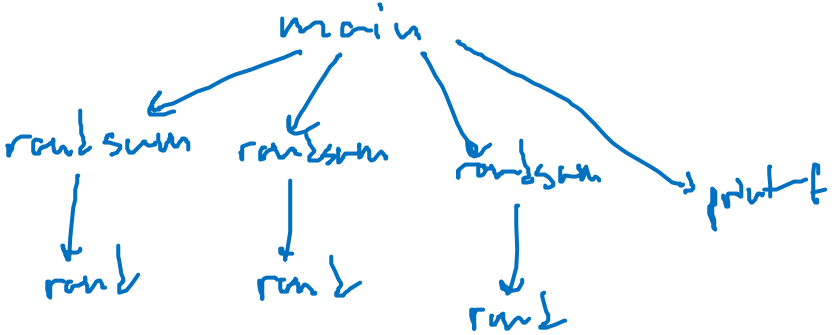
```c
int randSum(int n) {
    int y = rand()%20;
    return n+y;
}
```

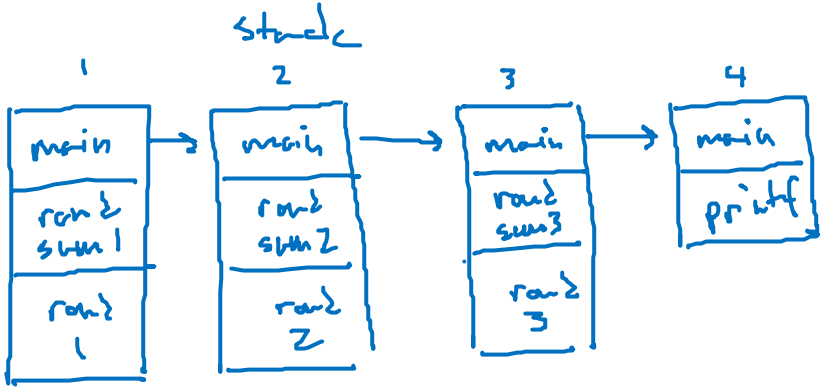
1. How many *total stack frames* are created?
   A) 3 **B)** 5 **C)** 7 **D)** 8

2. What is the maximum *depth* (# of frames) of the Stack?
   A) 1 **B)** 2 **C)** 3 **D)** 4

3. (Not on Ed) Which has a higher address address: **x** or **y**?

Review Questions



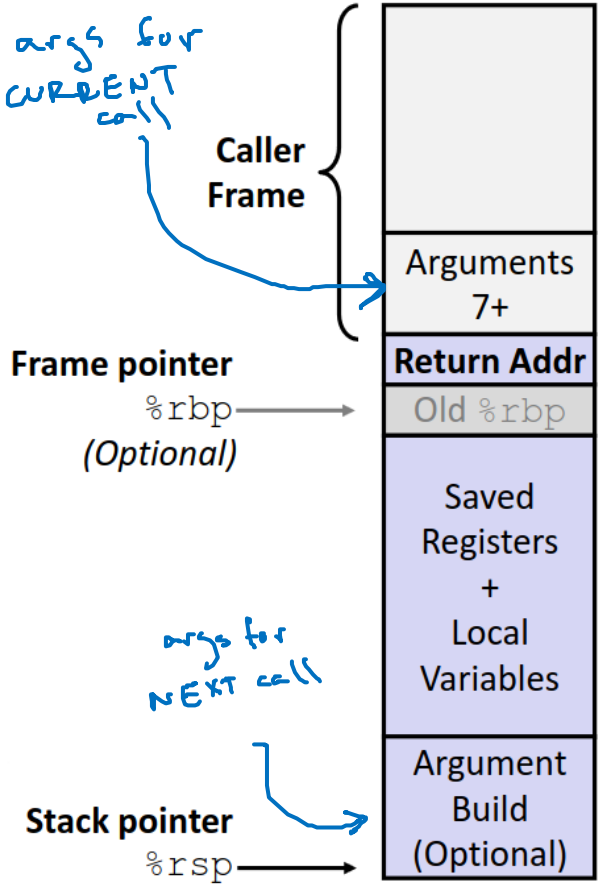


7 function calls total (incl. main) = 7 stack frames

max depth = 3

Recap: x86-64/Linux Stack Frame

- Caller's stack frame
  - **Extra arguments** (if > 6 args) for this call
- Current stack frame
  - **Return address** (pushed by `call`)
  - **Old frame pointer** (optional)
  - **Saved register** content
  - **Local variables** (that can't be saved in registers)
  - **Argument build** - if the current function needs to call another, extra arguments for that call go here



Procedure Call Example: `increment`

```
long increment(long* p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```



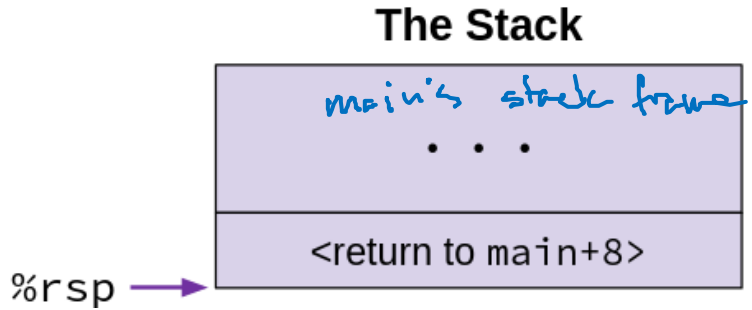
```
increment:
    movq (%rdi), %rax
    addq %rax, %rsi
    movq %rsi, (%rdi)
    ret
```

`%rdi`	p (arg1)
`%rsi`	`val` (arg2), `y`
`%rax`	x (return)

Example: initial state

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

```
call_incr:
    subq $16, %rsp
    movq $351, 8(%rsp)
    movl $100, %esi
    leaq 8(%rsp), %rdi
    call increment
    addq 8(%rsp), %rax
    addq $16, %rsp
    ret
```



- When `main` calls `call_incr`, push the return address onto the stack
  - Marks the beginning of `call_incr`'s stack frame

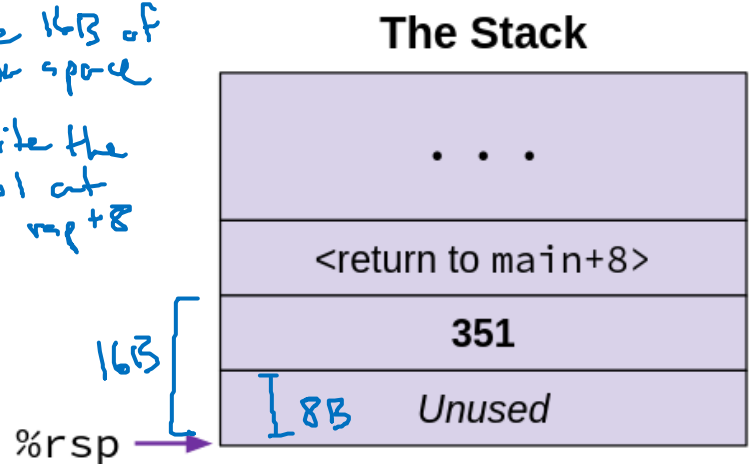
Example: step 1

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

```
call_incr:
    subq $16, %rsp
    movq $351, 8(%rsp)
    movl $100, %esi
    leaq 8(%rsp), %rdi
    call increment
    addq 8(%rsp), %rax
    addq $16, %rsp
    ret
```

subq - create 16B of stack space

movq - write the #351 at address rsp+8



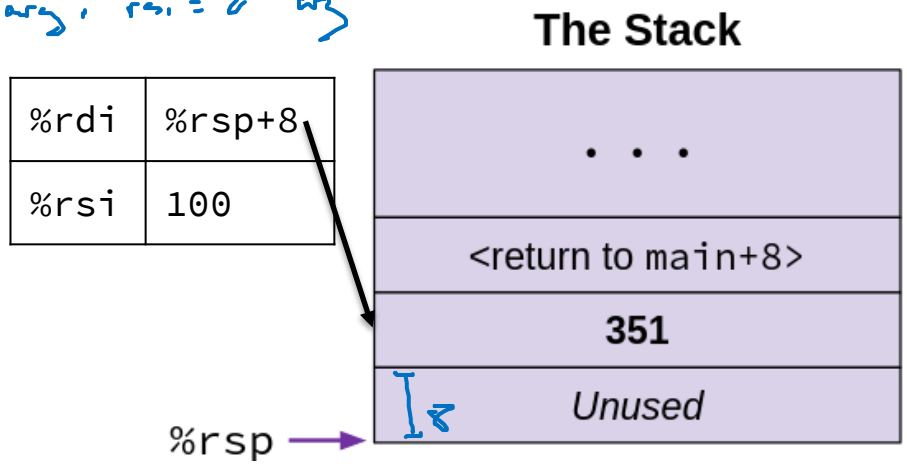
- Stack grows down, so `subq` adds 16B of free space to the stack
- `movq`
  - `$` means `351` is an immediate
  - Destination is memory address `%rsp+8`

Example: step 2

```
long call_incr() {
     long v1 = 351;
     long v2 = increment(&v1, 100);
     return v1 + v2;
}
```

```
call_incr:
     subq $16, %rsp
     movq $351, 8(%rsp)
     movl $100, %esi
     leaq 8(%rsp), %rdi
     call increment
     addq 8(%rsp), %rax
     addq $16, %rsp
     ret
```

rdi = 1st arg, rsi = 2nd arg



- Set up arguments for `increment`

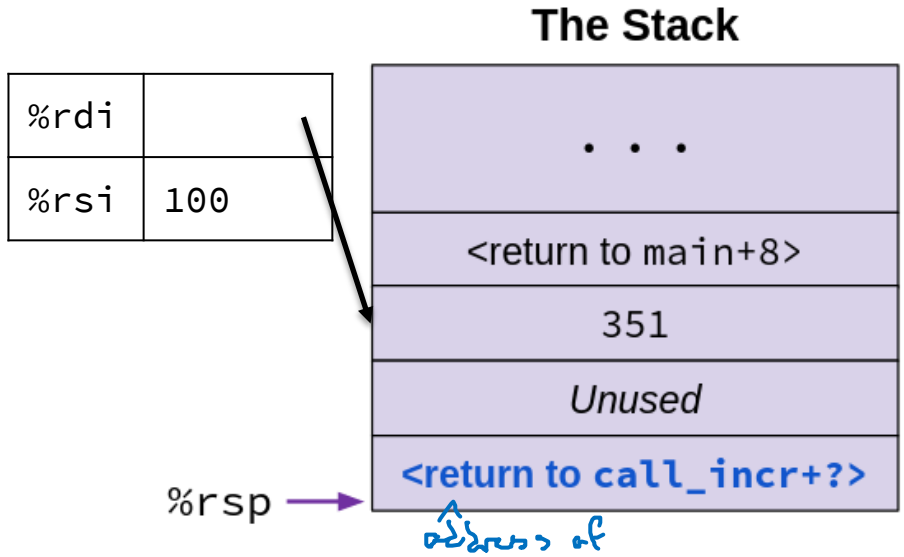
lea — %rdi gets the value %rsp+8, which is the address of v1, NOT the value at that address (351)

Example: step 3

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

```
call_incr:
    subq $16, %rsp
    movq $351, 8(%rsp)
    movl $100, %esi
    leaq 8(%rsp), %rdi
    call increment
    addq 8(%rsp), %rax
    addq $16, %rsp
    ret
```

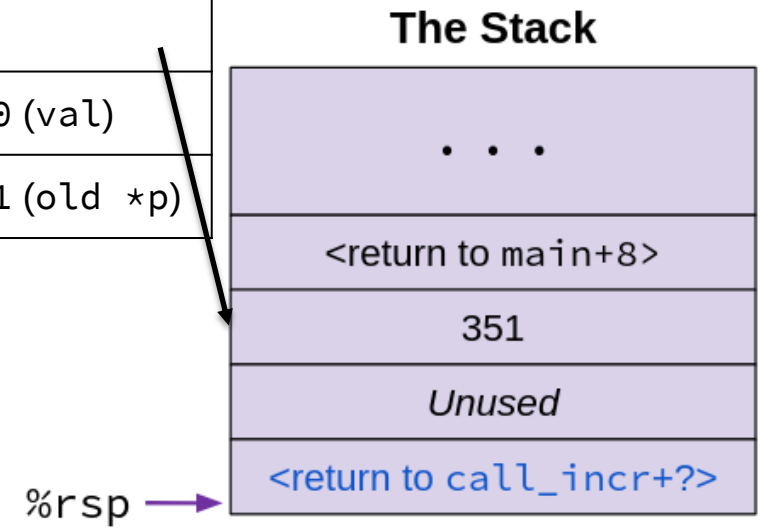
return



- `call` pushes return address (to the `addq` instruction after the `call`) onto the stack
  - Sets `%rip` to point to the beginning of increment

Example: step 4

%rdi	(p)
%rsi	100 (val)
%rax	351 (old *p)




```
long increment(long* p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

```
increment:
    movq (%rdi), %rax
    addq %rax, %rsi
    movq %rsi, (%rdi)
    ret
```

- Store return value into %rax (i.e. x)

Example: step 5

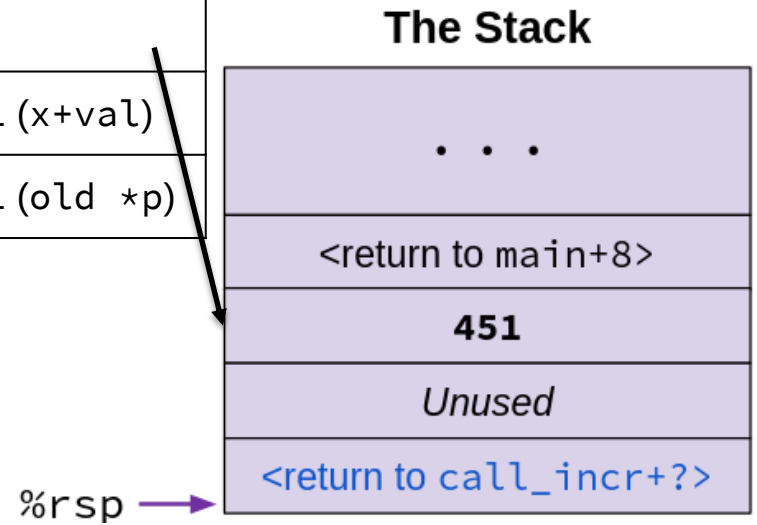
addq: rsi = rsi + rax = 100 + 351 = 451

movq: () means memory operand. store rsi's value at the address in rdi

```
long increment(long* p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

```
increment:
    movq (%rdi), %rax
    addq %rax, %rsi
    movq %rsi, (%rdi)
    ret
```

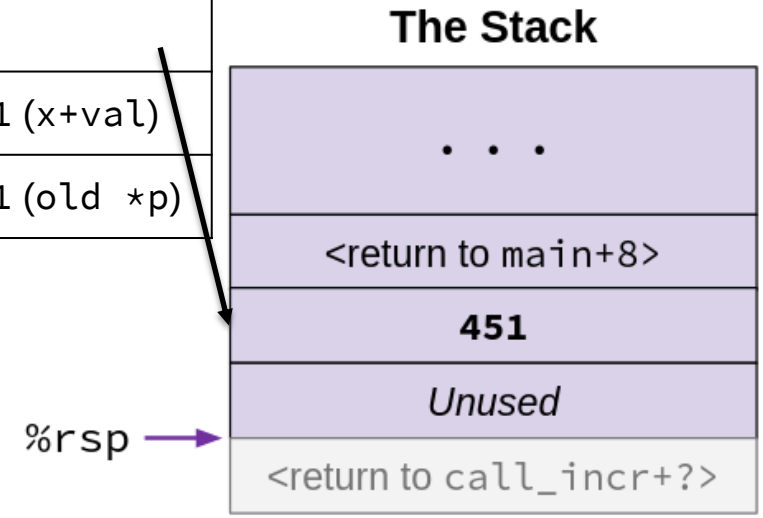
%rdi	(p)
%rsi	451 (x+val)
%rax	351 (old *p)



- Add %rax (x) to %rsi (val)
- Write result to the location location %rdi (p) points to

Example: step 6

%rdi	(p)
%rsi	451 (x+val)
%rax	351 (old *p)




```
long increment(long* p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

```
increment:
    movq (%rdi), %rax
    addq %rax, %rsi
    movq %rsi, (%rdi)
    ret
```

- Remove return address from the stack
  - Set %rip to removed value

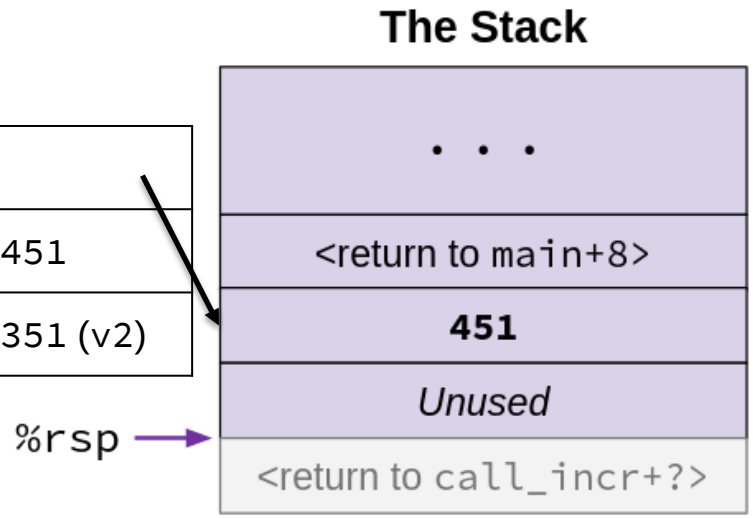
Example: step 6.5

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

```
call_incr:
    subq $16, %rsp
    movq $351, 8(%rsp)
    movl $100, %esi
    leaq 8(%rsp), %rdi
    call increment
    addq 8(%rsp), %rax
    addq $16, %rsp
    ret
```

resume at this instruction

%rdi	
%rsi	451
%rax	351 (v2)

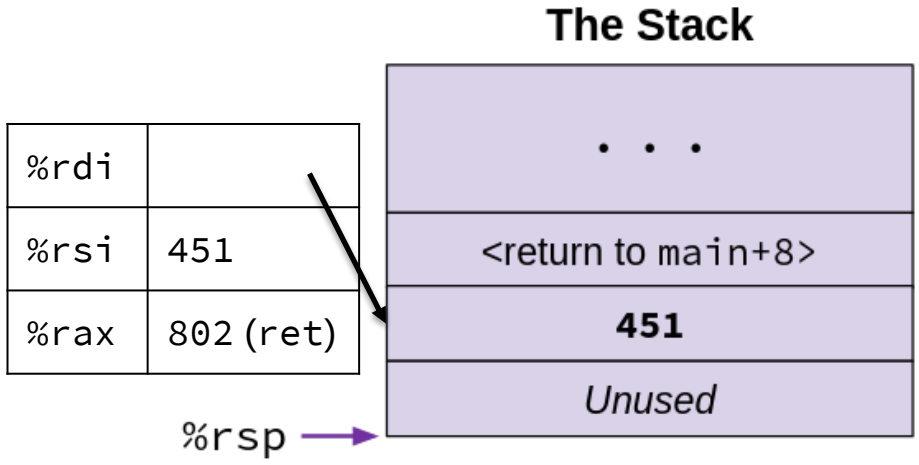


- Resume instruction at the location `%rip` is set to (the `addq` instruction)
- `%rax` stores the value that was returned from `increment`

Example: step 7

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

```
call_incr:
    subq $16, %rsp
    movq $351, 8(%rsp)
    movl $100, %esi
    leaq 8(%rsp), %rdi
    call increment
    addq 8(%rsp), %rax
    addq $16, %rsp
    ret
```



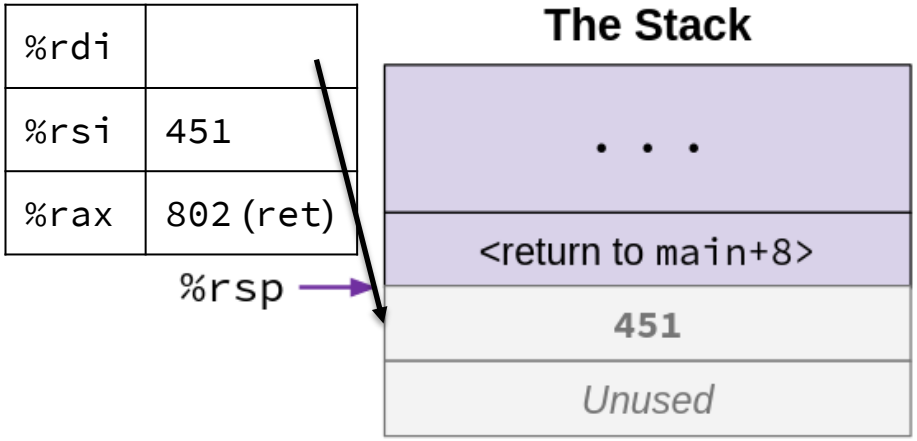
- Update %rax to contain v1+v2

() means memory operand
add the value at address rsp+8 to rax

Example: step 8

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

```
call_incr:
    subq $16, %rsp
    movq $351, 8(%rsp)
    movl $100, %esi
    leaq 8(%rsp), %rdi
    call increment
    addq 8(%rsp), %rax
    addq $16, %rsp
    ret
```



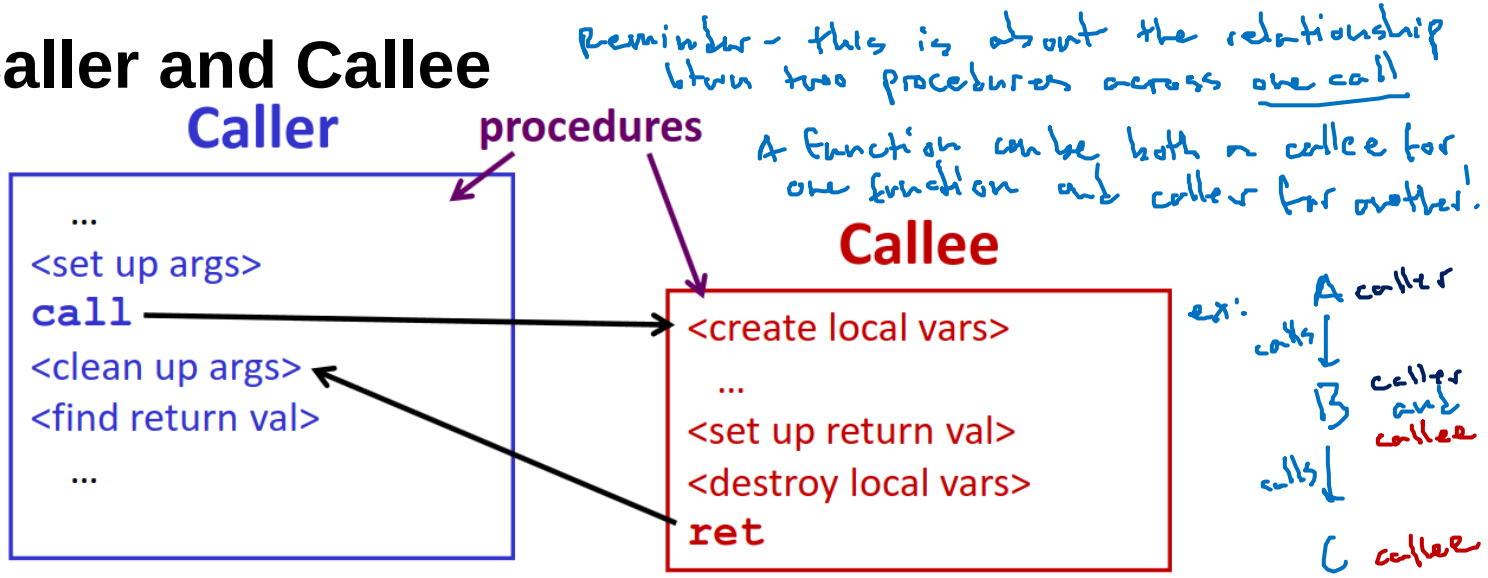
- Clean up the stack
- Return back to `main` (not shown)

addq undoes subq from earlier
ret expects rsp to point to return address

Lecture Topics

- Stack structure
- Calling conventions
  - Passing control
  - Passing data
  - Managing local data
- **Stack frames**
  - Saved registers
  - Stack layout
  - **Register saving convention**
- Illustration of Recursion
- Executables
  - CALL
  - Object Files

Recap: Caller and Callee



- Both use the same registers for their arguments and local variables, so how do we prevent them from overwriting each other's data?
  1. Before writing to a register, **push** the old value onto the stack
  2. **Pop** old value back into the register before returning

Register Saving Conventions

- **Caller-saved registers**
  - It is the **caller**'s responsibility to save these registers' values before calling another procedure
  - **Callee** is free to change these registers
  - **Caller** restores registers after **callee** returns
- **Callee-saved registers**
  - **Callee** guarantees that registers not be modified by this function
  - **Caller** doesn't need to save before calling
  - If the **callee** wants to use these, it must save their old data first, and restore before it returns

Silly Register Convention Analogy

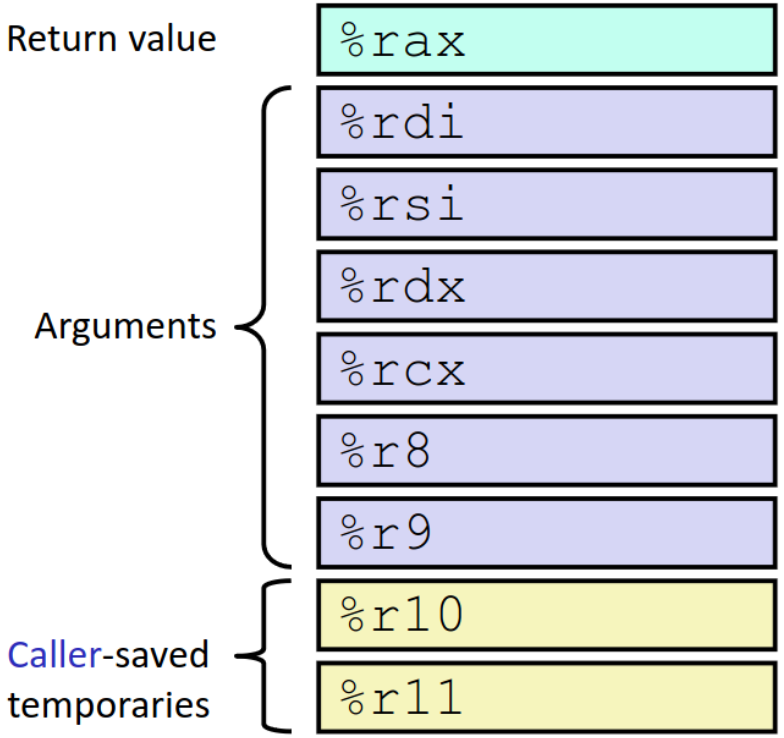
Parents = **caller**, child = **callee**

1. Parents are gone for the weekend and give their child the keys to the house
   - Being suspicious, they hid the valuables from the first floor (**caller-saved**) before leaving
   - They warn the child to leave the second floor untouched: “These rooms better look the same when we return!”
2. Child decides to throw a wild party (*computation*), spanning the entire house
   - To avoid getting in trouble, child moves all of the stuff from the second floor to the backyard shed (callee**-saved**) before the guests trash the house
   - Child cleans up house after the party and moves stuff back to second floor
3. Parents return home
   - Move valuables back into first floor and continue with their lives

x86-64 Linux Registers: Caller-Saved

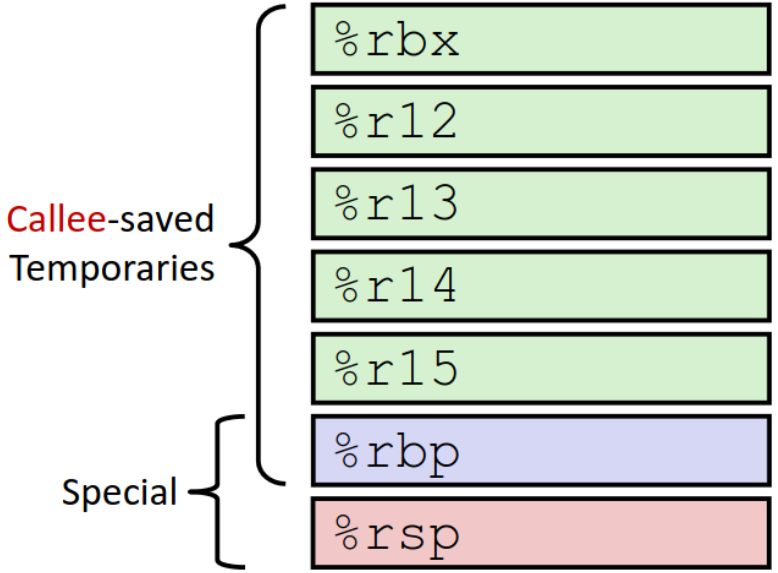
- **`%rax`**
  - Return value from **callee**
- **`%rdi...%r9`**
  - Arguments for **callee**
  - **Caller** saves before putting args there
- **`%r10, %r11`**
  - General-purpose registers

Don't memorize — on reference sheet!



x86-64 Linux Registers: Callee-Saved

- **`%rbx`, `%r12` - `%r15`**
  - General-purpose registers
- **`%rbp`**
  - Base pointer, or general-purpose
    - Can mix and match
- **`%rsp`**
  - Special case
    - Does not explicitly push value
    - Stack should be in the same state on return as it was at the beginning of the call



Why have both **Caller**- *and* **Callee**-Saved?

- We need *one* convention for all functions
- Neither is “best” in all cases
  - If caller isn’t using a register, **caller**-saved is better
  - If callee doesn’t need a register, **callee**-saved is better
  - If “do need to save”, **callee**-saved generally makes faster programs
    - Callee can be called from multiple places
- So… we went with “some of each”
  - Compiler tries to pick registers to minimize saving

Register Saving Conventions Summary

- **Caller-saved**: register values need to be pushed onto the stack before making a procedure call *only if the caller needs that value later*
  - **Callee** may change those register values
  - Popped after call returns
- **Callee-saved**: register values need to be pushed onto the stack *only if the callee intends to use those registers*
  - **Caller** expects unchanged values in those registers upon return
  - Popped before **callee** returns
- *Don't forget to restore/pop the values later!*

Lecture Topics

- Stack structure
- Calling conventions
  - Passing control
  - Passing data
  - Managing local data
- Stack frames
  - Saved registers
  - Stack layout
  - Register saving convention
- **Illustration of Recursion**
- Executables
  - CALL
  - Object Files

Recursive Function

```
/* 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
    jne .L8
    ret
.L8:
    pushq %rbx
    movq %rdi, %rbx
    shrq %rdi
    call pcount_r
    andl $1, %ebx
    addq %rbx, %rax
    popq %rbx
    ret
```

- Counts the number of 1's in the binary representation of x
- Compiler Explorer: https://godbolt.org/z/E943Gz3M5
  - Compiled with -O1 instead of -Og for more natural instruction ordering

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);
}
```

```
pcount_r:
    movl $0, %eax
    testq %rdi, %rdi
    jne .L8
    ret
.L8:
    pushq %rbx
    movq %rdi, %rbx
    shrq %rdi
    call pcount_r
    andl $1, %ebx
    addq %rbx, %rax
    popq %rbx
    ret
```

why movl (4B) when longs are 8B? Remember that 4B instructions also 0 out the rest of the 8B register!

movl - set up ret value

testq - set flags based on rdi & rdi = rdi = x

jne - jump if x != 0

ret - otherwise, return

Recursive Function: Saved Registers

```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
    jne .L8
    ret
.L8:
    pushq %rbx
    movq %rdi, %rbx
    shrq %rdi
    call pcount_r
    andl $1, %ebx
    addq %rbx, %rax
    popq %rbx
    ret
```

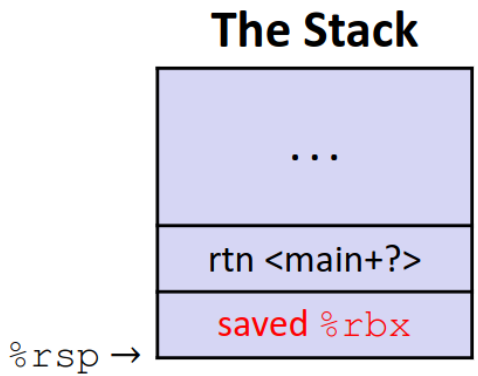
callee saved

- `%rdi` is a **caller**-saved register, needs to be saved before recursive call
  - Rather than saving `%rdi` the stack, compiler put it in `%rbx`, which is **callee**-saved
    - Has to save old `%rbx` value on the stack first

Recursive Function: Saved Registers (pt 2)

```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
    jne .L8
    ret
.L8:
    pushq %rbx
    movq %rdi, %rbx
    shrq %rdi
    call pcount_r
    andl $1, %ebx
    addq %rbx, %rax
    popq %rbx
    ret
```



%rdi	x
%rbx	x

Recursive Function: Call Setup

```
/* 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
    jne .L8
    ret
.L8:
    pushq %rbx
    movq %rdi, %rbx
    shrq %rdi
    call pcount_r
    andl $1, %ebx
    addq %rbx, %rax
    popq %rbx
    ret
```

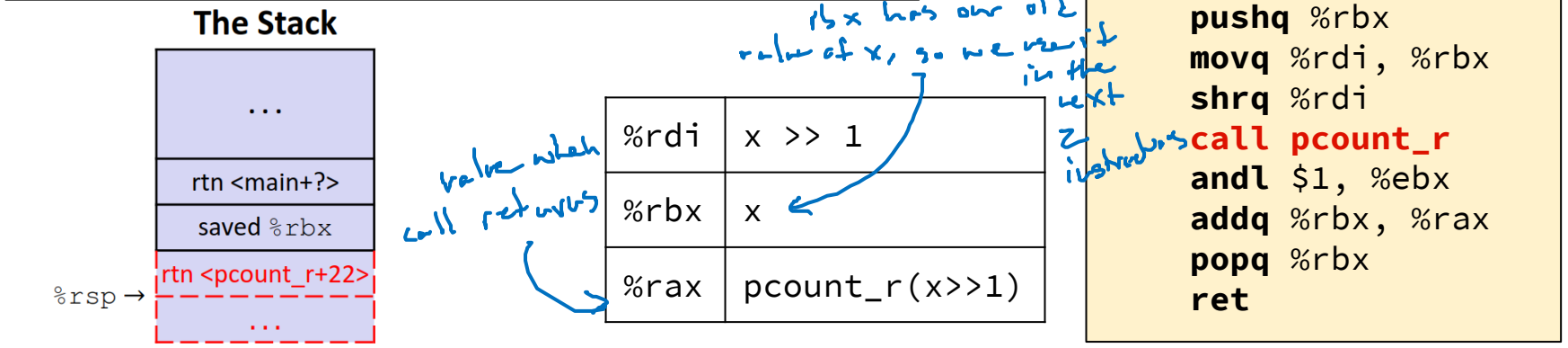
- Shift %rdi by 2 to set up argument for next call

%rdi	x >> 1
%rbx	x

Recursive Function: Call

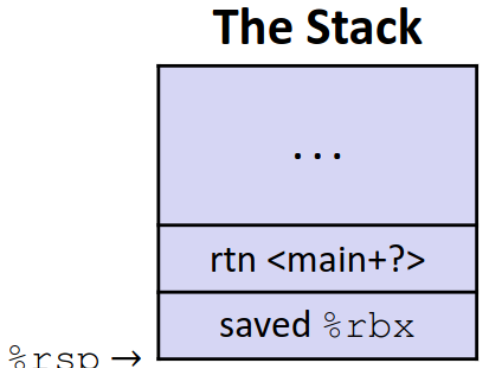
```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + pcount_r(x >> 1);
}
```

```
pcount_r:
    movl $0, %eax
    testq %rdi, %rdi
    jne .L8
    ret
.L8:
    pushq %rbx
    movq %rdi, %rbx
    shrq %rdi
    call pcount_r
    andl $1, %ebx
    addq %rbx, %rax
    popq %rbx
    ret
```



Recursive Function: Result

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

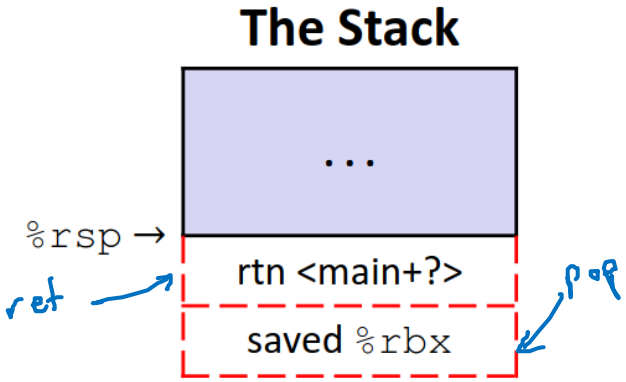


%rdi	x >> 1
%rbx	x & 1
%rax	return value

```
pcount_r:
    movl $0, %eax
    testq %rdi, %rdi
    jne .L8
    ret
.L8:
    pushq %rbx
    movq %rdi, %rbx
    shrq %rdi
    call pcount_r
    andl $1, %ebx
    addq %rbx, %rax
    popq %rbx
    ret
```

Recursive Function: Completion

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



%rdi	x >> 1
%rbx	old %rbx
%rax	return value

```
pcount_r:
    movl $0, %eax
    testq %rdi, %rdi
    jne .L8
    ret
.L8:
    pushq %rbx
    movq %rdi, %rbx
    shrq %rdi
    call pcount_r
    andl $1, %ebx
    addq %rbx, %rax
    popq %rbx
    ret
```

Observations About Recursion

- Works without any special considerations
  - Each call gets its own stack frame for local variables + return address
  - Register saving prevents one function call from corrupting another’s data
  - Stack discipline follows call/return pattern
    - Last-in, first-out (like a stack!)
- The principals work for all functions, not just recursion

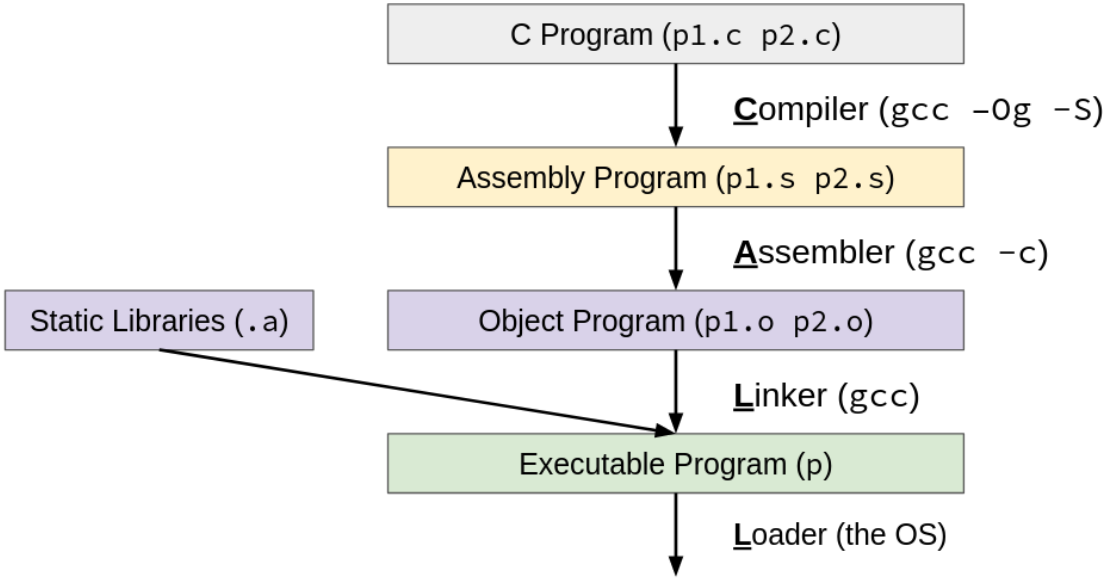
Lecture Topics

- Stack structure
- Calling conventions
  - Passing control
  - Passing data
  - Managing local data
- **Stack frames**
  - Saved registers
  - Stack layout
  - **Register saving convention**
- Illustration of Recursion

- **Executables**
  - **CALL**
  - **Object Files**

CALL: Building an Executable with C

- Code in files **`p1.c p2.c`**
  - Compile with **`gcc -Og p1.c p2.c -o p`**
  - Run with **`./p`**



CALL: Compiler

- Input: Higher-level language code (*e.g.*, C, Java)
  - Ex: `foo.c`
- Output: Assembly language code (*e.g.* x86, ARM)
  - Ex: `foo.s`
- For C, starts with preprocessor to process #directives *before* compilation
  - Macro substitution, etc.
  - If you're curious: http://tigcc.ticalc.org/doc/cpp.html
- Performs optimizations
  - For gcc, specified by `-O` flag (e.g. `-Og`, `-O3`)
  - List of options: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
- *Super complex*, there's a whole course dedicated to these (CSE 401)!

Compiling (into Assembly) Example

Note: this is still “source code” in a sense – human-readable instructions, written out as text.

Example: C code (`sum.c`)

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

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

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

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

CALL: Assembler

- Input: Assembly language code (e.g., x86, ARM)
  - Ex: `foo.s`
- Output: Object files (e.g., ELF, COFF)
  - Ex: `foo.o`
- Very similar to assembly but a little different; Contains object code and **information tables**
- Reads and uses *assembly directives* from source files
  - *e.g.*, `.text`, `.data`, `.quad`
  - x86 directives: https://docs.oracle.com/cd/E26502_01/html/E28388/eoiyg.html
- Produces "**machine language**" (binary instructions)
  - Does it's best, but object file is NOT a complete binary

Producing Machine Language

- **Simple cases**: arithmetic and logical operations, shifts, etc.
  - i.e. Instructions that don't reference addresses
  - Assembler can do this
    - All necessary information is contained in the instruction itself!
- **Complex cases**: jumps, accessing static data (e.g., global variable or jump table), procedure calls
  - Addresses and labels are not generated in the assembly stage
    - May need addresses to things from other files
- So what do we do in the meantime?

Object File Information Tables

Each object file has a **symbol table** and **relocation table**

- **Symbol Table** holds list of “items” that may be used by other files
  - i.e. *“this is what I have and know about”*
  - **Non-local Labels** – function names usable 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)
  - i.e. “*these are the things I need*”
  - Any label or piece of static data referenced in an instruction in this file
    - Both internal and external

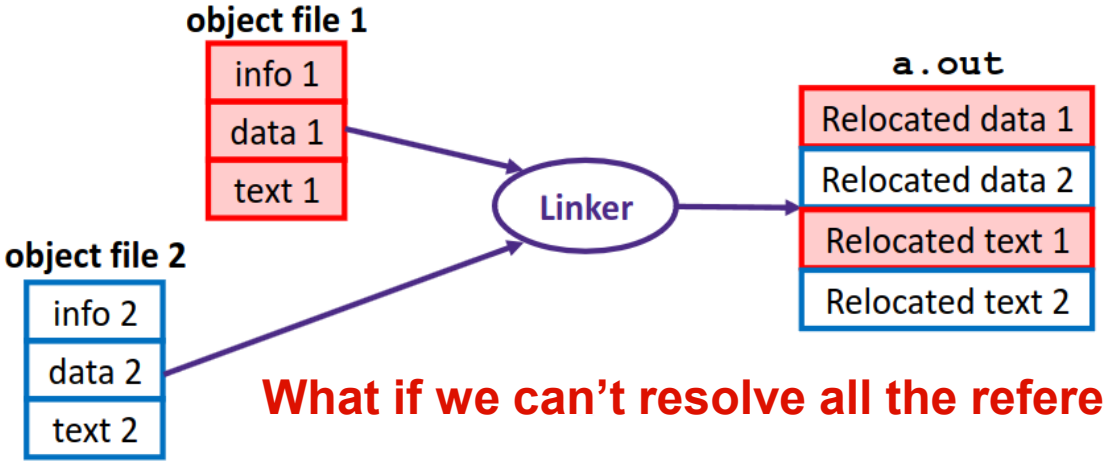
CALL: Linker

- Input: Object files (e.g., ELF, COFF)
  - Ex: `foo.o`
- Output: Executable binary program
  - Ex: `foo` (default is `a.out`)
- Combines (links) several object files
- Enables separate compilation/assembling of files
  - Changes to one file don’t require recompiling the others

Linking Example

text = instructions + literals
data = static data (global vars)

1. Concatenate text and data segments from each `.o` file
2. Go through each entry in relocation tables
   a. Find address based on its location in the text and data segments
   b. Replace label in the code with that address



What if we can't resolve all the references?

Linking Example (pt 2)

1. Concatenate text and data segments from each `.o` file
2. Go through each entry in relocation table
    a. Find address based on its location in the text and data segments
    b. Replace label in the code with that address

```
// tell the compiler that findme
// is in a different file
extern void findme();

int main() {
    findme();
    return 0;
}
```

```
$ gcc findme.c
/usr/bin/ld: /tmp/ccAQ36Zy.o: in function ‘main’:
findme.c:(.text+0xa): undefined reference to
‘findme’
collect2: error: ld returned 1 exit status
```

CALL: Loader

- Input: executable binary program, command-line arguments
  - Ex: `./foo arg1 arg2`
- Output: <program is run>
- Memory sections (Instructions, Static Data, Stack) are set up
- Registers are initialized
- Handled by operating system
  - Want to implement this yourself? Take OS (CSE 451)!

Disassembling

- Approximates of assembly from machine code (object file or executable)
  - Ex: `objdump -d foo`
- Looks similar to assembly file, but we actually have more info!
  - Addresses, all symbols, etc.

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

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

What Can be Disassembled?

- Anything that can be interpreted as executable code! ***However…***
  - Not always accurate
  - Often illegal (for commercial software)
  - Falls under academic misconduct for school assignments (unless we tell you to)

```
% 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:
```

[REDACTED]

Discussion

Discuss in groups of 2-4, and then we'll talk as a class:

- We've seen a few examples of names that are derived from history
  - Ex: a "word" being 2 byte in x86
- Naming/etymology plays a big role in learning
  - Which new terms from CSE 351 been the most intuitive for you to learn vs. the most difficult?
  - What do you think goes into a good vs. bad name (more generally in computer science)?

some fun examples:
- register names – named after intended purpose
- core dumped – named after outdated memory technology
- mantissa – from Latin

Summary

- Stack is organized into **frames**, one for each call
  - Store all the data for that function, return address, and saved registers
- **Register saving convention** prevents data from being lost between calls
  - **Caller-saved**: saved before a function is called, popped after
  - **Callee-saved**: saved before being used, popped before returning
- 4 steps to generate and run a program:
  - **Compile**: generate assembly for each source file
  - **Assemble**: generate object file for each source file
    - Includes **symbol table** and **relocation table**
  - **Link**: combine object files into one executable
  - **Load**: OS sets up memory and registers before running