

# Procedures & Executables

CSE 351 Autumn 2017

## Instructor:

Justin Hsia

## Teaching Assistants:

Lucas Wotton

Michael Zhang

Parker DeWilde

Ryan Wong

Sam Gehman

Sam Wolfson

Savanna Yee

Vinny Palaniappan

# Administrivia

- ❖ Lab 2 due Friday (10/27)
- ❖ Homework 3 released tomorrow (10/24)
- ❖ Lab 1 grading
  - Double-check your total
  - See Piazza for common misconceptions
- ❖ **Midterm** next Monday (10/30, 5pm, KNE 120)
  - Make a cheat sheet! – two-sided letter page, *handwritten*
  - Check Piazza this week for announcements
  - **Review session** 5:30-7:30pm on Friday (10/27) in EEB 105

# Procedures

- ❖ Stack Structure
- ❖ Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
- ❖ **Register Saving Conventions**
- ❖ Illustration of Recursion

# Register Saving Conventions

- ❖ When procedure `yoo` calls `who`:
  - `yoo` is the *caller*
  - `who` is the *callee*
- ❖ Can registers be used for temporary storage?

```
yoo:  
  . . .  
  movq $15213, %rdx  
  call who  
  addq %rdx, %rax  
  . . .  
  ret
```

```
who:  
  . . .  
  subq $18213, %rdx  
  . . .  
  ret
```

- No! Contents of register `%rdx` overwritten by `who`!
- This could be trouble – something should be done. Either:
  - *Caller* should save `%rdx` before the call (and restore it after the call)
  - *Callee* should save `%rdx` before using it (and restore it before returning)

# Register Saving Conventions

## ❖ “*Caller-saved*” registers

- It is the **caller**'s responsibility to save any important data in these registers before calling another procedure (*i.e.* the **callee** can freely change data in these registers)
- **Caller** saves values in its stack frame before calling **Callee**, then restores values after the call

## ❖ “*Callee-saved*” registers

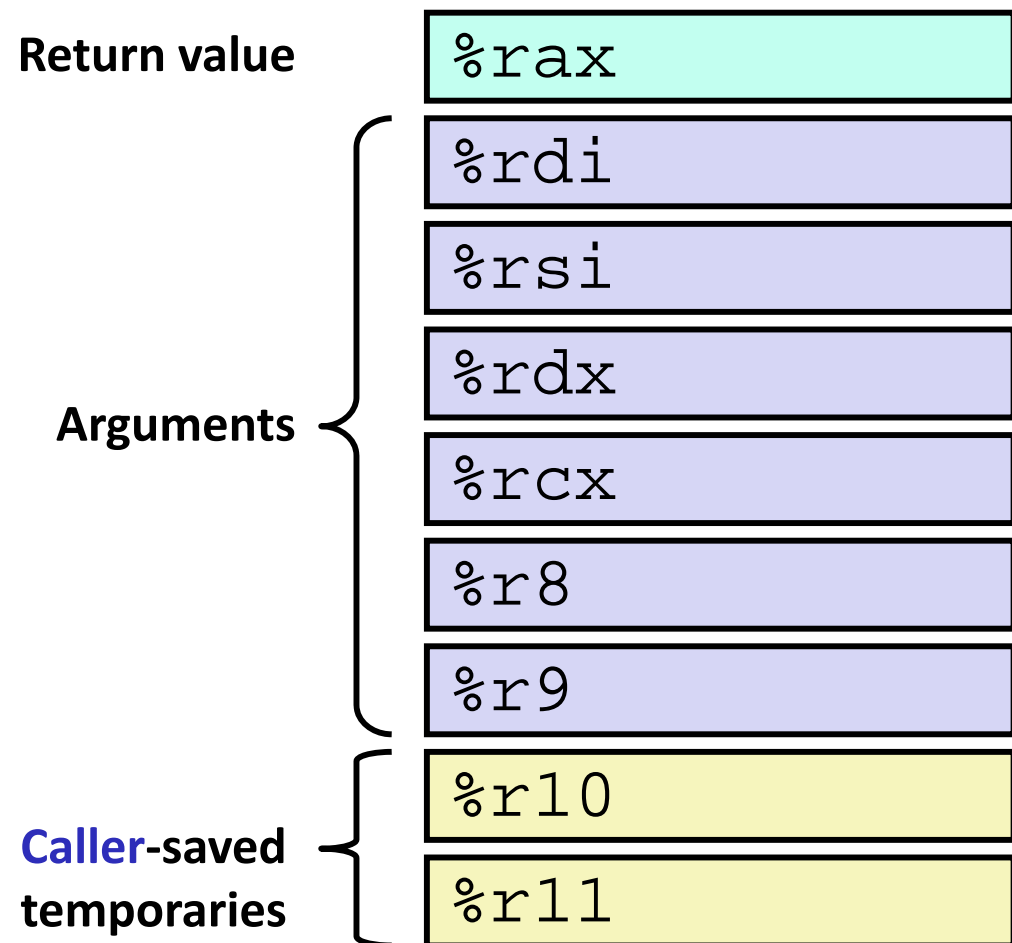
- It is the callee's responsibility to save any data in these registers before using the registers (*i.e.* the **caller** assumes the data will be the same across the **callee** procedure call)
- **Callee** saves values in its stack frame before using, then restores them before returning to **caller**

# Silly Register Convention Analogy

- 1) Parents (*caller*) leave for the weekend and give the keys to the house to their child (*callee*)
  - Being suspicious, they put away/hid the valuables (*caller-saved*) before leaving
  - Warn child to leave the bedrooms 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 being disowned, child moves all of the stuff from the bedrooms to the backyard shed (*callee-saved*) before the guests trash the house
  - Child cleans up house after the party and moves stuff back to bedrooms
- 3) Parents return home and are satisfied with the state of the house
  - Move valuables back and continue with their lives

# x86-64 Linux Register Usage, part 1

- ❖ **%rax**
  - Return value
  - Also **caller**-saved & restored
  - Can be modified by procedure
- ❖ **%rdi, ..., %r9**
  - Arguments
  - Also **caller**-saved & restored
  - Can be modified by procedure
- ❖ **%r10, %r11**
  - **Caller**-saved & restored
  - Can be modified by procedure



# x86-64 Linux Register Usage, part 2

## ❖ `%rbx, %r12, %r13, %r14`

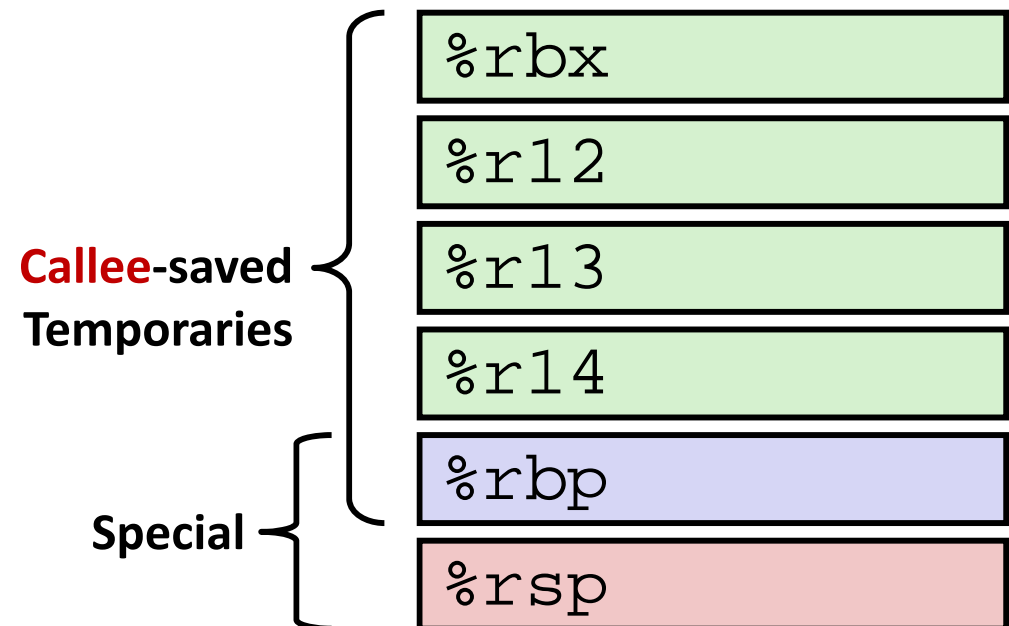
- **Callee**-saved
- **Callee** must save & restore

## ❖ `%rbp`

- **Callee**-saved
- **Callee** must save & restore
- May be used as frame pointer
- Can mix & match

## ❖ `%rsp`

- Special form of **callee** save
- Restored to original value upon exit from procedure





# x86-64 64-bit Registers: Usage Conventions

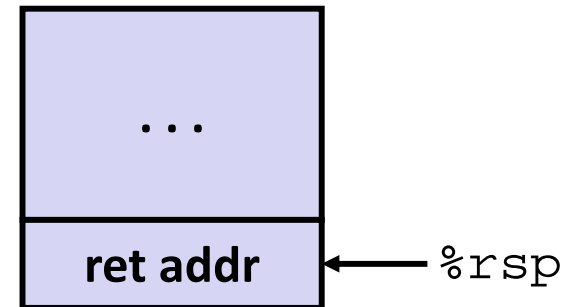
<code>%rax</code>	Return value - <b>Caller</b> saved	<code>%r8</code>	Argument #5 - <b>Caller</b> saved
<code>%rbx</code>	<b>Callee</b> saved	<code>%r9</code>	Argument #6 - <b>Caller</b> saved
<code>%rcx</code>	Argument #4 - <b>Caller</b> saved	<code>%r10</code>	<b>Caller</b> saved
<code>%rdx</code>	Argument #3 - <b>Caller</b> saved	<code>%r11</code>	<b>Caller</b> Saved
<code>%rsi</code>	Argument #2 - <b>Caller</b> saved	<code>%r12</code>	<b>Callee</b> saved
<code>%rdi</code>	Argument #1 - <b>Caller</b> saved	<code>%r13</code>	<b>Callee</b> saved
<code>%rsp</code>	Stack pointer	<code>%r14</code>	<b>Callee</b> saved
<code>%rbp</code>	<b>Callee</b> saved	<code>%r15</code>	<b>Callee</b> saved

# Callee-Saved Example (step 1)

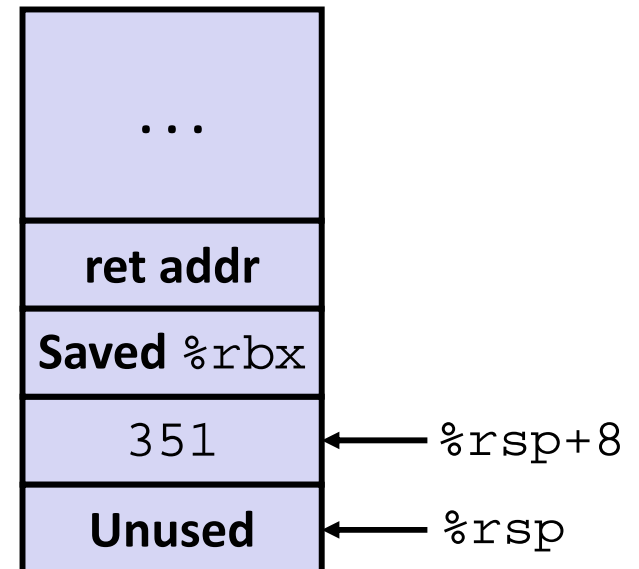
```
long call_incr2(long x) {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return x+v2;
}
```

```
call_incr2:
    pushq    %rbx
    subq    $16, %rsp
    movq    %rdi, %rbx
    movq    $351, 8(%rsp)
    movl    $100, %esi
    leaq    8(%rsp), %rdi
    call    increment
    addq    %rbx, %rax
    addq    $16, %rsp
    popq    %rbx
    ret
```

Initial Stack Structure



Resulting Stack Structure

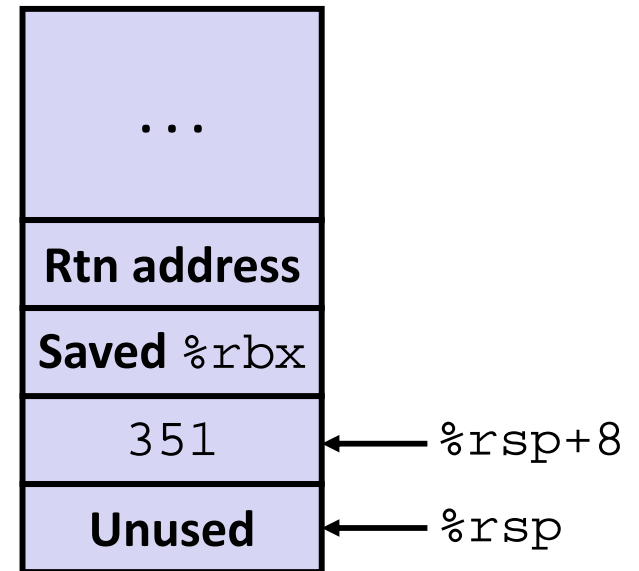


# Callee-Saved Example (step 2)

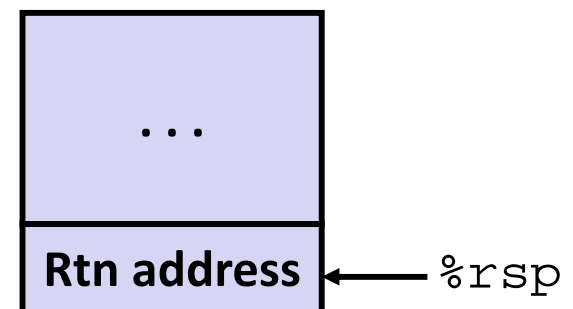
```
long call_incr2(long x) {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return x+v2;
}
```

```
call_incr2:
    pushq    %rbx
    subq    $16, %rsp
    movq    %rdi, %rbx
    movq    $351, 8(%rsp)
    movl    $100, %esi
    leaq    8(%rsp), %rdi
    call    increment
    addq    %rbx, %rax
    addq    $16, %rsp
    popq    %rbx
    ret
```

Stack Structure



Pre-return Stack Structure



# Why Caller *and* Callee Saved?

- ❖ We want *one* calling convention to simply separate implementation details between caller and callee
- ❖ In general, neither caller-save nor callee-save is “best”:
  - If caller isn’t using a register, caller-save is better
  - If callee doesn’t need a register, callee-save is better
  - If “do need to save”, callee-save generally makes smaller programs
    - Functions are called from multiple places
- ❖ So... “some of each” and compiler tries to “pick registers” that minimize amount of saving/restoring

# Register 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
- ❖ **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
- ❖ Don't forget to restore/pop the values later!

# 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!

```
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: Base Case

```

/* 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
%rdi	x	Argument
%rax	Return value	Return value

```

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

```

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



# Recursive Function: Callee Register Save

```

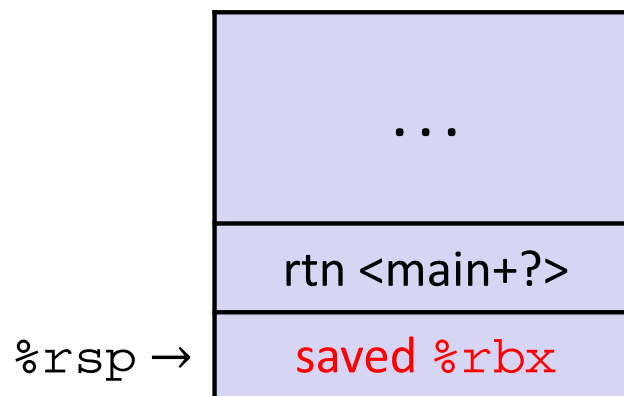
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x&1)+pcount_r(x >> 1);
}
    
```

Register	Use(s)	Type
%rdi	x	Argument

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



```

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
    
```

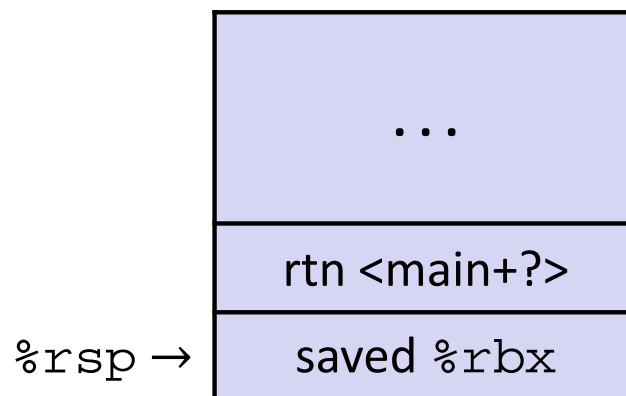
# Recursive Function: Call Setup

```

/* 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
%rdi	x (new)	Argument
%rbx	x (old)	Callee saved

## The Stack



```

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
    
```

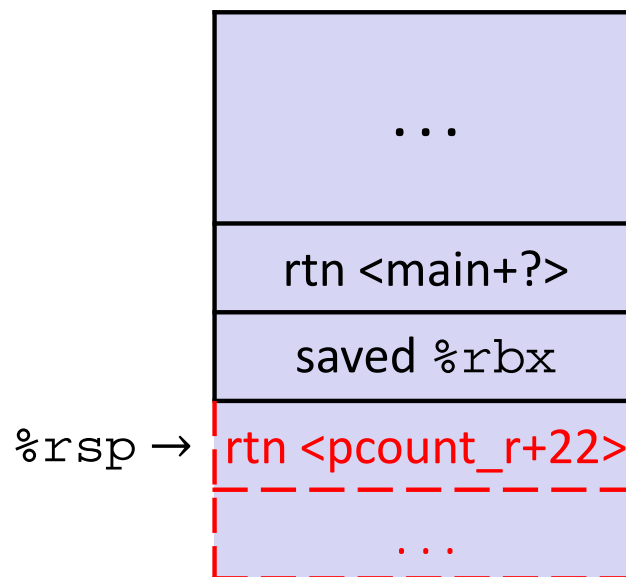
# Recursive Function: Call

```

/* 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
%rax	Recursive call return value	Return value
%rbx	x (old)	Callee saved

## The Stack



```

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
    
```

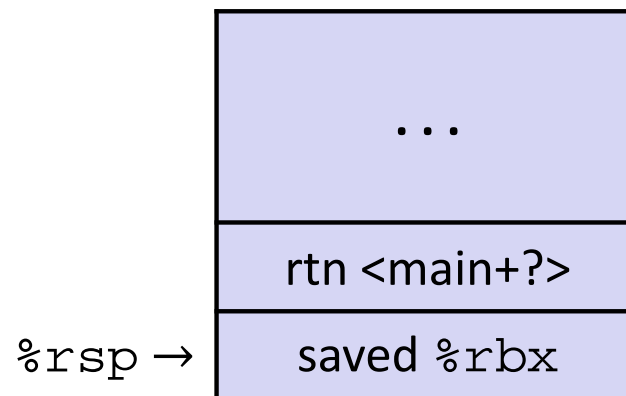
# Recursive Function: Result

```

/* 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
%rax	Return value	Return value
%rbx	x&1	Callee saved

## The Stack



```

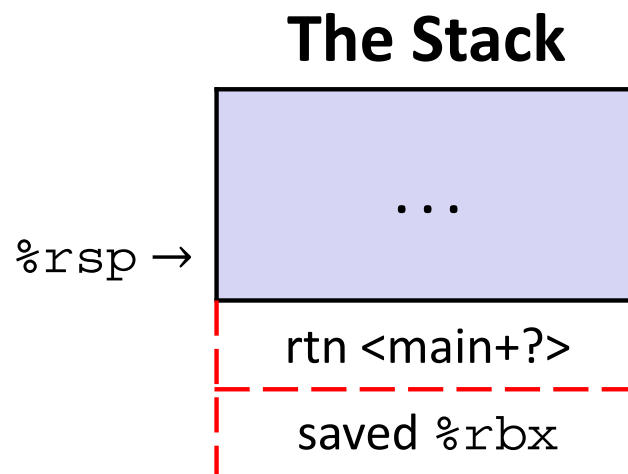
pcount_r:
    movl    $0, %eax
    testq  %rdi, %rdi
    je     .L6
    pushq  %rbx
    movq   %rdi, %rbx
    shrq   %rdi
    call   pcount_r
    andl   $1, %ebx
    addq   %rbx, %rax
    popq   %rbx
.L6:
    rep   ret
    
```

# Recursive Function: Completion

```

/* 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
%rax	Return value	Return value
%rbx	Previous %rbx value	Callee restored



```

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
    
```

# Observations About Recursion

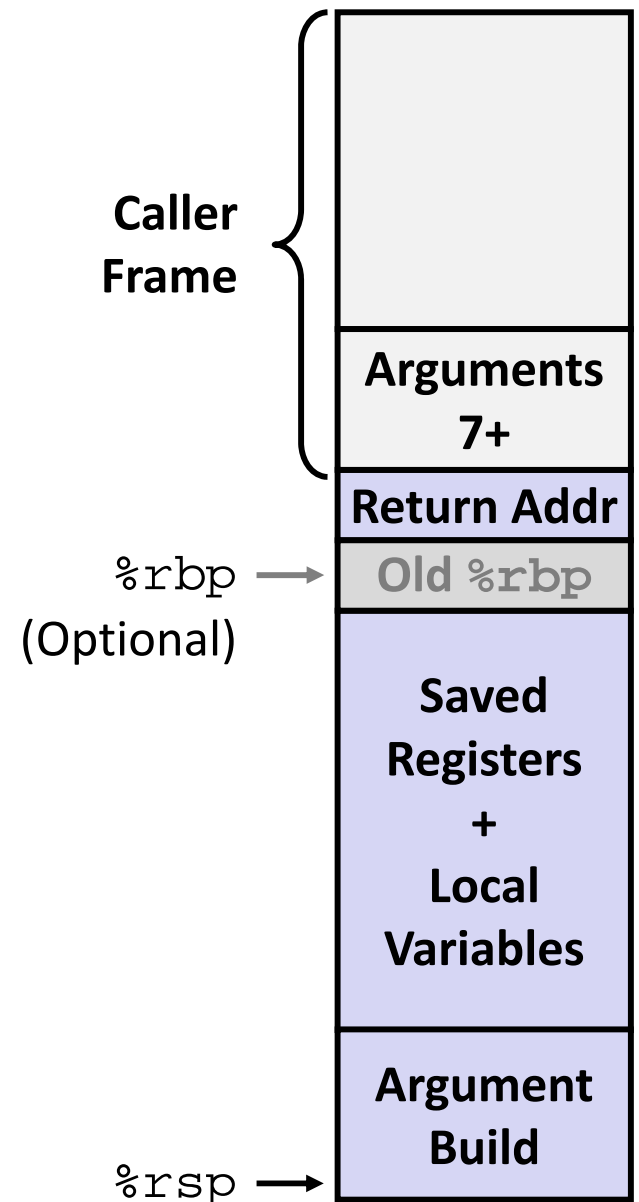
- ❖ Works without any special consideration
  - Stack frames mean that each function call has private storage
    - Saved registers & local variables
    - Saved return pointer
  - Register saving conventions prevent one function call from corrupting another's data
    - Unless the code explicitly does 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();
```

- Memory & data
- Integers & floats
- x86 assembly
- Procedures & stacks
- Executables**
- Arrays & structs
- Memory & caches
- Processes
- Virtual memory
- Memory allocation
- Java vs. C

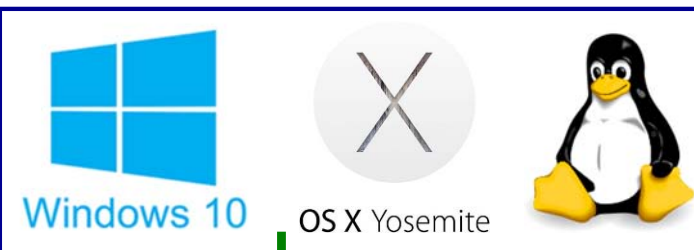
Assembly language:

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

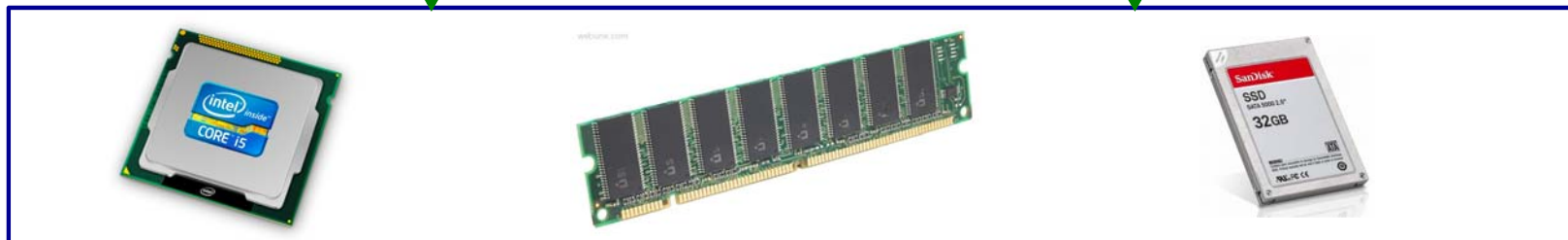
Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:

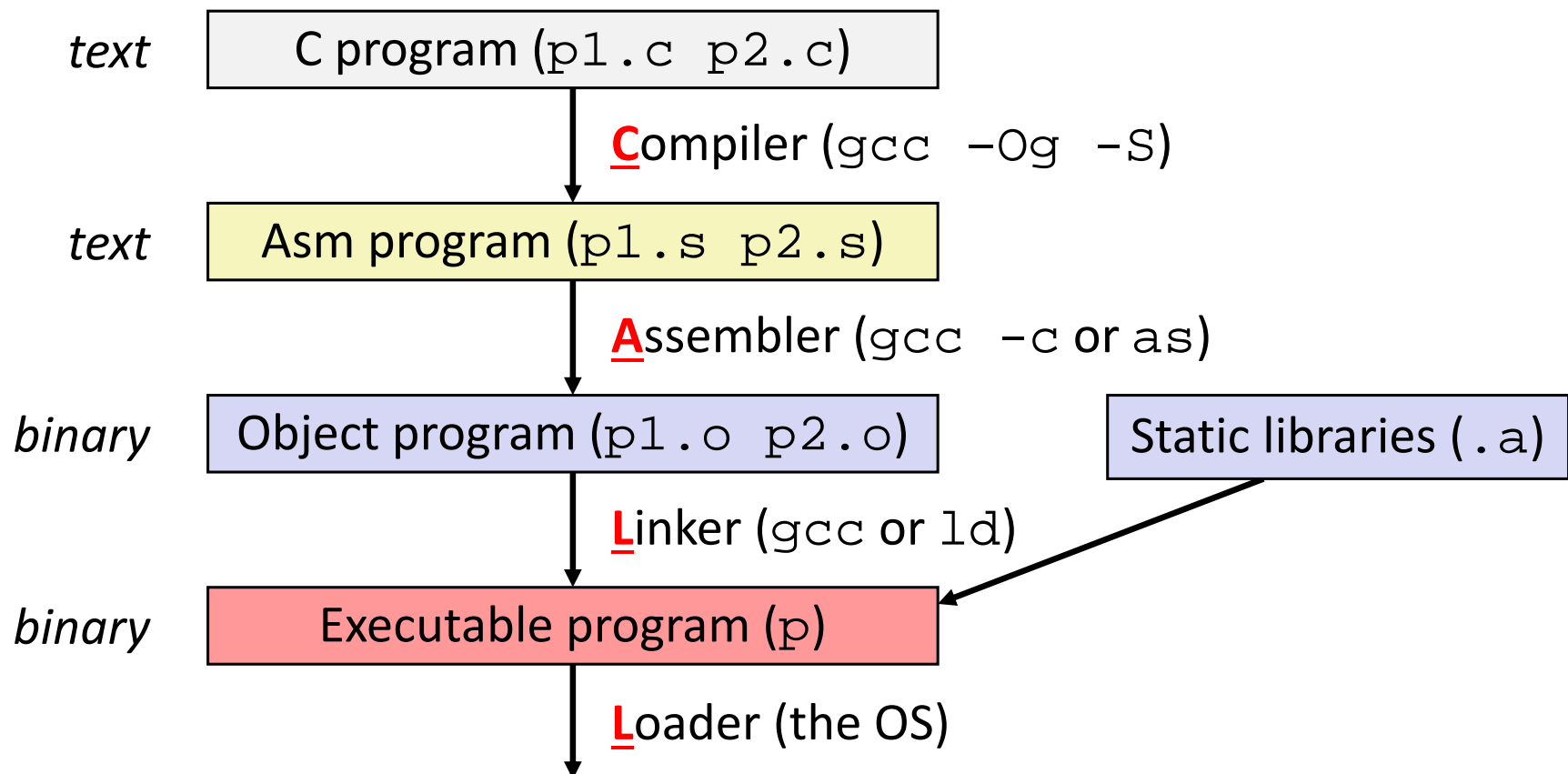


Computer system:



# 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
  - If curious/interested: <http://tigrcc.ticalc.org/doc/cpp.html>
- ❖ Super complex, whole courses devoted to these!
- ❖ Compiler optimizations
  - “Level” of optimization specified by capital ‘O’ flag (*e.g.* `-Og`, `-O3`)
  - Options: <https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>

# Compiling Into Assembly

## ❖ 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)

- Generates file sum.s (see <https://godbolt.org/g/o34FHp>)

```
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`
- ❖ **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
  - 2) text segment: the machine code
  - 3) data segment: data in the source file (binary)
  - 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
- ❖ More info: ELF format
- [http://www.skyfree.org/linux/references/ELF\\_Format.pdf](http://www.skyfree.org/linux/references/ELF_Format.pdf)

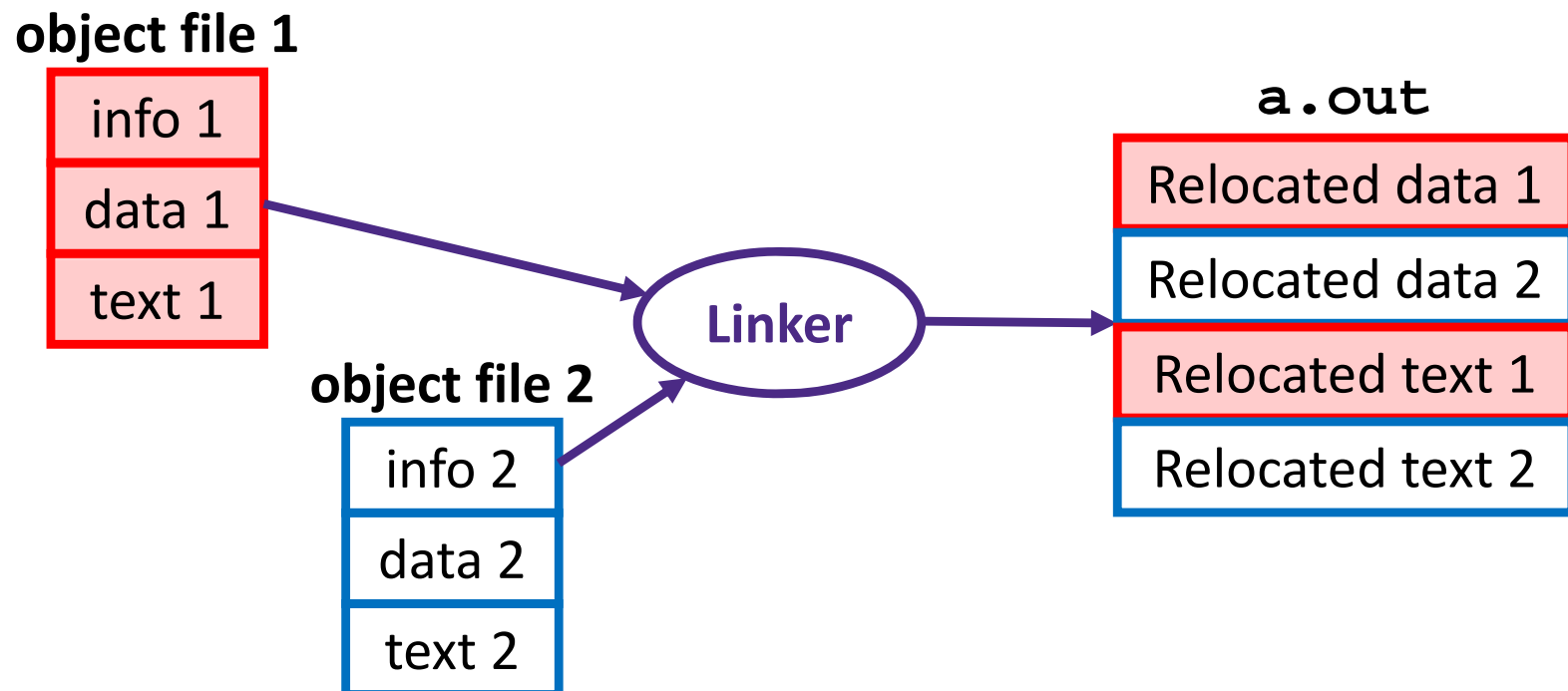


# Linker

- ❖ **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) Take text segment from each `.o` file and put them together
- 2) Take data segment from each `.o` file, put them together, and concatenate this onto end of 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:
```

Reverse engineering forbidden by  
Microsoft End User License Agreement

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

# 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, Stack) are set up
- ❖ Registers are initialized