Systems I – CS 6013

Computer Architecture and Operating Systems

Lecture 4: Function Calls in ASM

MASTER OF SOFTWARE DEVELOPMENT (MSD) PROGRAM

J. DAVISON DE ST. GERMAIN

SPRING 2024

Miscellaneous

- Questions?
- gcc –o doit doit.c –Wall
  - -g
    - Debug mode
  - -O
    - Optimized mode
  - -S
    - Output assembly code (.s file)
- doit & #What is the ampersand for?

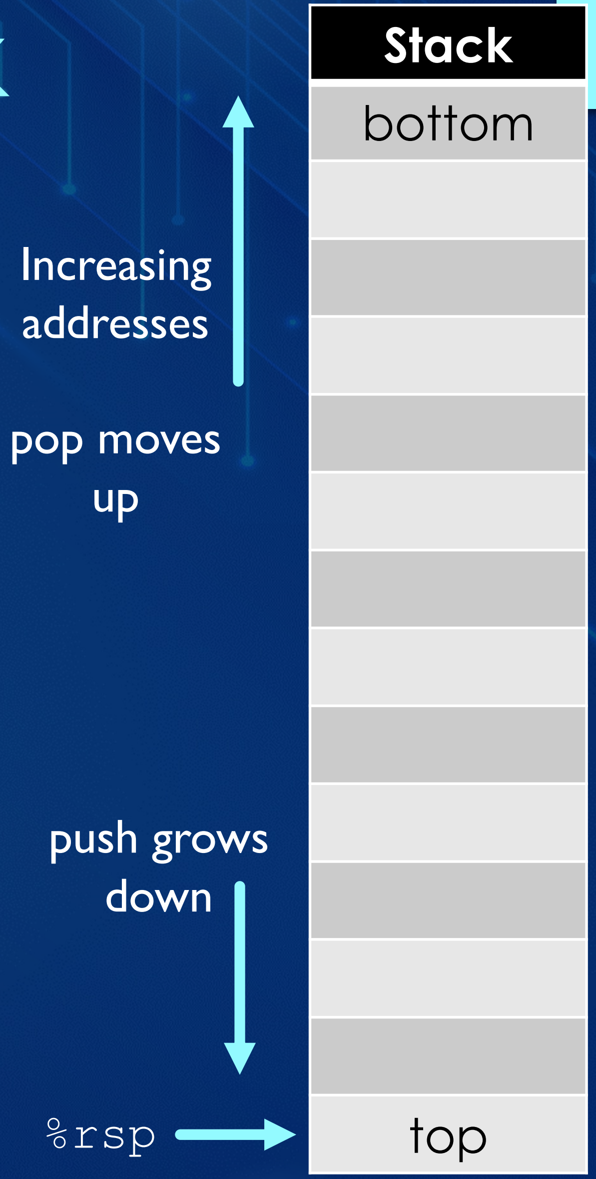
  [1] 7353
  - Runs in the *background*. What is 7353? What is 1?
    - PID – process identification number
    - Background process number.
      - `fg %1`, c-z (control z)
- ./doit #What is the ./ and why is it there?
  - echo $PATH
- Make sure to look at Week 2 reading list
- New HW assignment has been posted
  - [Has old Apple MacBook instructions that do not work with new MacBooks (Apple Silicon / ARM)]
  - Not due for 2 weeks.
- Diego, please see footnote 7 in the reading (Introduction (Chapter 2)). ☺

Lecture 4 – Topics

- Call Stack
- Function Parameters

Function Calls and The Stack

- The call stack (remember this?)
  - While in previous lectures, to match a “physical stack” I’ve displayed the stack as growing upward… in reality, with respect to memory addresses, it grows downward.
- When a function is called, information about it is pushed onto the stack. Space for what information?
  - Parameters
  - Local Variable
  - Return Value(s)
  - Location of the previous stack frame
  - etc.
- What registers can I use?
- Answers to these questions are described by the *Calling Conventions* or *Application Binary Interface* (ABI)



Example: System V ABI for x86_64

- Basically the ABI for Unix – Specific to language / operation system. Note, not the same as the API (Application Programming Interface).
- First 6 int or pointer parameters/arguments to a function must be placed in rdi, rsi, rdx, rcx, r8, r9
- Floating point parameters in xmm0, … xmm7
- Int return values must be placed in rax (and you look for them there when trying to capture return value).
- rbx, rbp, r12-r15 are “callee-saved” registers. If the function uses them, it must restore them before it returns
- Other registers may be “clobbered” by the function
- Other parameters are passed on the stack

Passing / Returning Values in Assembly

- Passing parameters:

```
int main() {
  int x = doit( 9, 12, 45 );
   // mov rdi, 9
   // mov rsi, 12
   // mov rdx, 45
   // call doit
}
```

- Compute and return a value

```
int doit( int a, int b, int c ) {
    int result = a + b + c;
    return result;
    // doit:
    //    mov rbx, 0 ; rbx is result
    //    add rbx, rdi
    //    add rbx, rsi
    //    add rbx, rdx
    //    mov rax, rbx
}
```

- Can do the above without first and last mov.

Passing / Returning Values in Assembly

- Passing parameters:

```
int main() {
  int x = doit( 9, 12, 45 );
  // mov rdi, 9
  // mov rsi, 12
  // mov rdx, 45
  // call doit
}
```

- Compute and return a value

```
int doit( int a, int b, int c ) {
   int result = a + b + c;
   return result;
   // doit:
   //    mov rbx, rdi ; rax is result
   //    add rbx, rsi
   //    add rbx, rdx
}
```

Register Protocols

- Some registers are <u>temporaries</u>
  - Call a function => register value may have changed on return
  - A.k.a. Caller-saved
    - eg: %r10, %rsi
- Some registers are <u>preserved</u>
  - Call a function => register value the same on return
  - a.k.a. callee-saved
    - eg: %rbx, %rsp
- Classification of registers is part of an:
  - Application Binary Interface (ABI)

x86-64 Linux Register Usage

	register	usage
Caller-saved	%rax	*return value*
	%rdi	*1st argument*
	%rsi	*2nd argument*
	%rdx	*3rd argument*
	%rcx	*4th argument*
	%r8	*5th argument*
	%r9	*6th argument*
	%r10	*temporary*
	%r11	*temporary*
Callee-saved	%rbx	*preserved*
	%r12	*preserved*
	%r13	*preserved*
	%r14	*preserved*
	%rbp	*stack frame*
	%rsp	*stack pointer*

Application Binary Interface

An OS-specific ABI defines:

- How arguments are passed to functions
  - So far, only integer and address args
- How results are returned from functions
  - So far, only integer and address results
- Which registers are preserved (and not)
  - There are more registers…
- Other constraints, such as stack alignment
  - x86-64 Linux: stack aligned on call at 8 mod 16
- Optional debugging protocols

Managing the Stack

- Function **must** set up stack before running its body
- Must **clean up** before returning
- Setup is called “prologue”
- Tear down is called “epilogue”

Prologue

- Save the base pointer (bp) to the stack, then overwrite it with sp
- Make room on stack for other function calls by *subtracting* from sp (stack grows downward)
- Copy arguments into locations relative to bp

```
int g( int y ) {
}

int f( int x ) {
    int z = 44;
    return g( z )
}
main() {
    int a = 2;
    f( a );
}
```

Note, "main()" and "a", and "Address" don't actually exist on the Stack. They are implicitly there, but we write them down like this just for our own information.

Stack

			Address
main()	a	2	1004
			1000
			996
			992
			988
			984
			980
			…

Prologue

- Save the base pointer (bp) to the stack, then overwrite it with sp
- Make room on stack for other function calls by *subtracting* from sp (stack grows downward)
- Copy arguments into locations relative to bp

```
int g( int y ) {
}

int f( int x ) {
    int z = 44;
    return g( z )
}
main() {
    int a = 2;
    f( a );
}
```

Registers	
bp	1004
sp	996
ip	<addr>

Stack			
			Address
main()	a	2	1004
			1000
f()	x	2	996
	z	44	992
			988
			984
			980
			…

- Old / simplistic view… But what actually happens?
- Need to handle the prologue…

Prologue

- Save the base pointer (bp) to the stack, then overwrite it with sp
- Make room on stack for other function calls by *subtracting* from sp (stack grows downward)
- Copy arguments into locations relative to bp

```
int g( int y ) {
}

int f( int x ) {
    int z = 44;
    return g( z )
}
main() {
   int a = 2;
   f( a );
}
```

Registers

bp	1004
sp	996
ip	<addr>

Stack

			Address
main()	a	2	1004
			1000
			996
			992
			988
			984
			980
			…

- What does the prologue require?

Prologue

- Save the base pointer (bp) to the stack, then overwrite it with sp
- Make room on stack for other function calls by *subtracting* from sp (stack grows downward)
- Copy arguments into locations relative to bp

```
int g( int y ) {
}

int f( int x ) {
    int z = 44;
    return g( z )
}
main() {
    int a = 2;
    f( a );
}
```

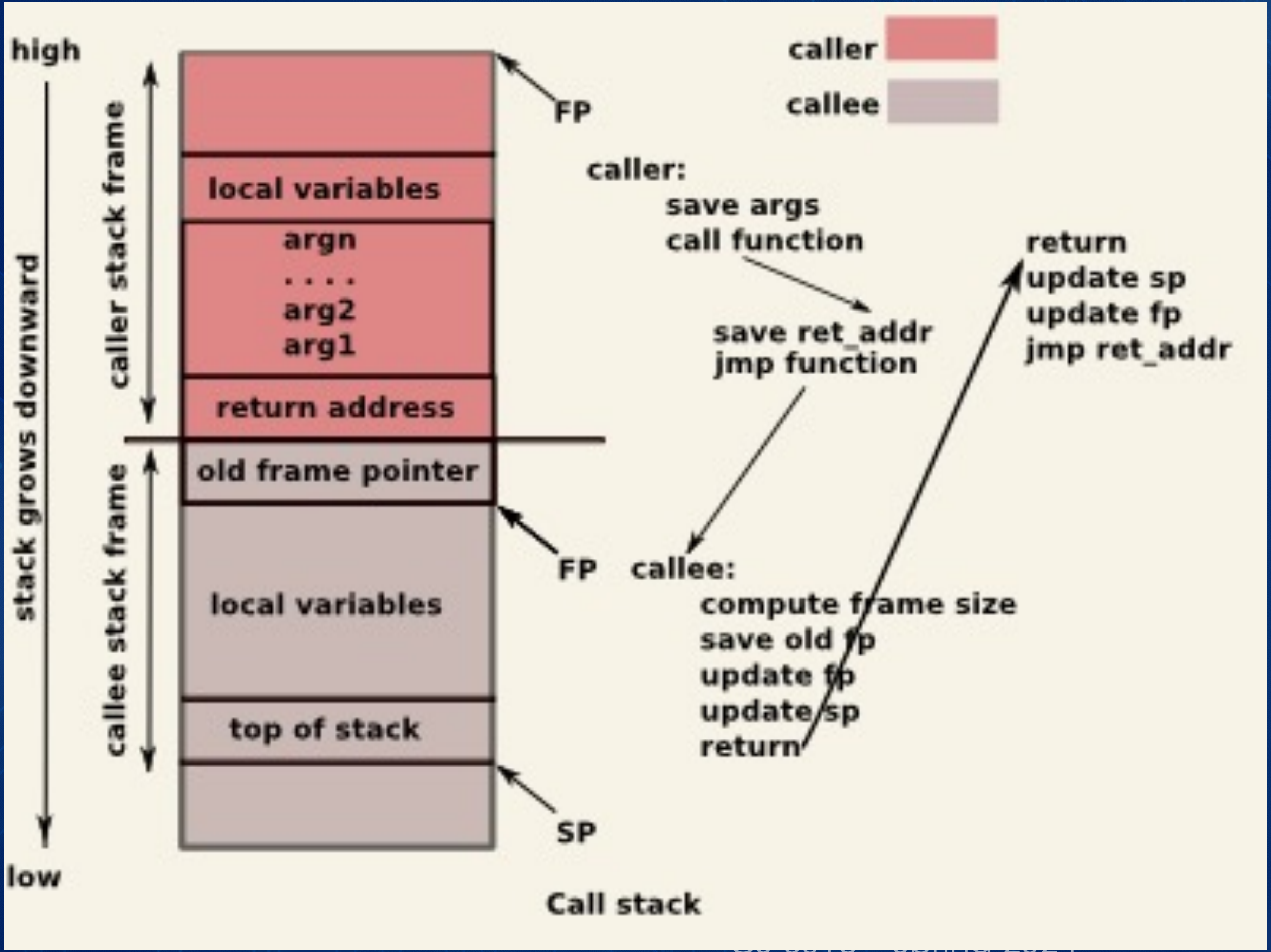
Registers	
bp	996
sp	984
ip	<addr>

Stack			
			Address
main()	a	2	1004
	[ip]	addr	1000
f()	[bp]	1004	996
	x	2	992
	z	44	988
			984
			980
			…

What (Why) is ip?
mov [bp-4], rdi
What does this do?
Copies rdi into the address of base pointer – 4.
What is rdi?
The 1st parameter - in this case x (with the value of 2)
What would it look like after calling g()?

Epilogue

- Store return int into rax. Why?
- Then, undo what we just did in the prologue…
- Restore the base pointer so that when we return, the caller's stack pointer is in the right place.
- pop rbp
  - What is rbp vs bp from previous slide?
  - r == 64 bit version of register.



- Note 1: In this diagram the BP (Base Pointer) is referred to as the FP (Frame Pointer).
- Note 2: SP (Stack Pointer) always points to the top of the stack (remember, stack is growing downward

Special Instructions

-
```
call functionLabel ; save the program counter
                   ; (the special/secret IP
                   ; register) to the stack,
                   ; then jump to functionLabel
```

-
```
leave ; restore rbp from the stack
```

-
```
ret   ; jump to the return address saved by call
      ; instruction and pop it off the stack
```

Examples

- Write some code and see the assembly…
  - Godbolt.org
  - Assembly is close, but not exact to our syntax…

  int main() { return 0; }
  - push, pop, why eax,0

  int main() { int i = 3; return 0; }
  - mov [rbp-4], 3 ; what is this?
    - Save the value of 3 on the stack. (4 bytes)

  int doit() { return 10; }

  int main() { int g = doit(); return g; }
  - Look at assembly output with –O option…

~ *Fin* ~