

CS202 (003): Operating Systems

Process I

Instructor: Jocelyn Chen

Any other questions?

Last time...

“an **instance** of
running program”

Process is the key abstraction of a OS!

We want our computer to do multiple
things at the same time

Writing code and listening to music

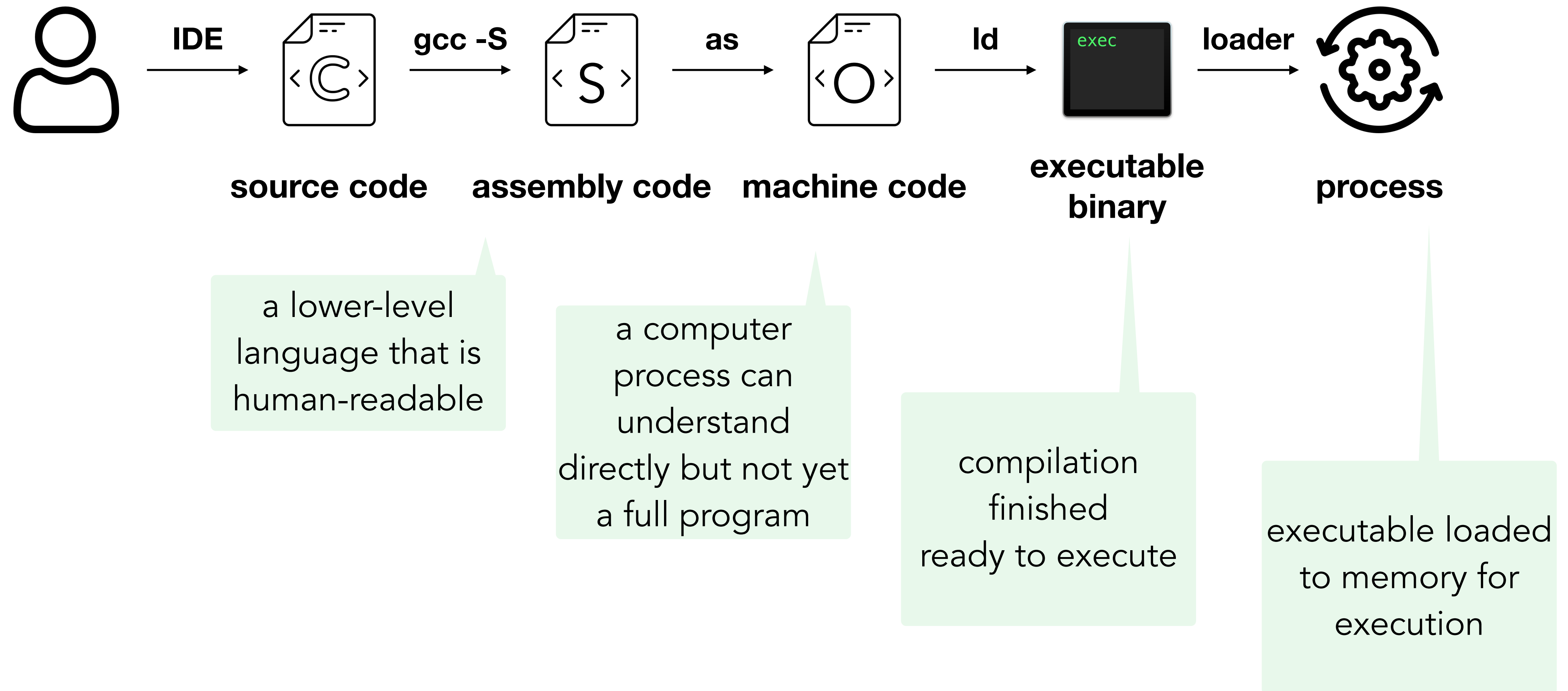
**Multiple users use the computer
simultaneously**

We want to use the resources more
efficiently

Increase CPU utilization

Reduce latency

Steps towards creating a process



To understand process...

How process see an abstract machine?

How OS implement the process abstraction?

Let's first refresh our memory a bit...

Basic elements in a machine

CPU (a CPU core)

Execution units
(e.g. ALUs)

Perform computations
according to the
instructions

Registers

Can be read by execution
units very quickly

**General-purpose (16 on
x86-64)**

RAX, RBX, RCX, RDX, RSI,
RDI, R8-R15, **RSP** and **RBP**

Special-purpose

RIP, ...

Memory

Stores information, such as data
and programs, for immediate use

Takes more time to access than
register (2~X00 cycles)

“Hierarchies of memory”, but we
don't emphasize on that in this class

Disk

GPUs

..... (peripherals)

Three Aspects to a Process

CPU (a processor)

“Each process has its own registers”

Memory

“Each process has its own view of memory”

Others

signal state, UID, signal mask, controlling terminal, priority, etc...

Process thinks memory as a contiguous array

Environment

command line args, ...

Stack

local variables, params, return addresses

Heap

malloc()

Data

Store global variables and constants

Text/Code

Store program itself

Lower Address

Do you still remember assembly code?

`movq PLACE1, PLACE2`

Move 64-bit quantity from PLACE1 to PLACE2

Places can be registers, memory addresses, or immediates (constants)

`pushq %rax`

`subq $8, %rsp`
`movq %rax, (%rsp)`

Allocates 8 bytes of space on the stack (why 8?)

Remember: the stack grows downward, that's why we do subtract

The stack pointer (%rsp) is automatically adjusted

Do you still remember assembly code?

`movq PLACE1, PLACE2`

Move 64-bit quantity from PLACE1 to PLACE2

Places can be registers, memory addresses, or immediates (constants)

`popq %rax`

`movq (%rsp), %rax
addq $8, %rsp`

Move the value at the top of the stack to %rax
Increases the stack pointer by 8 (which means?)

`call 0x12345`

Pseudo-code:
`pushq %rip
movq $0x12345, %rip`

Pushes the %rip onto the stack (which means?)
Sets the %rip to the address of the called function

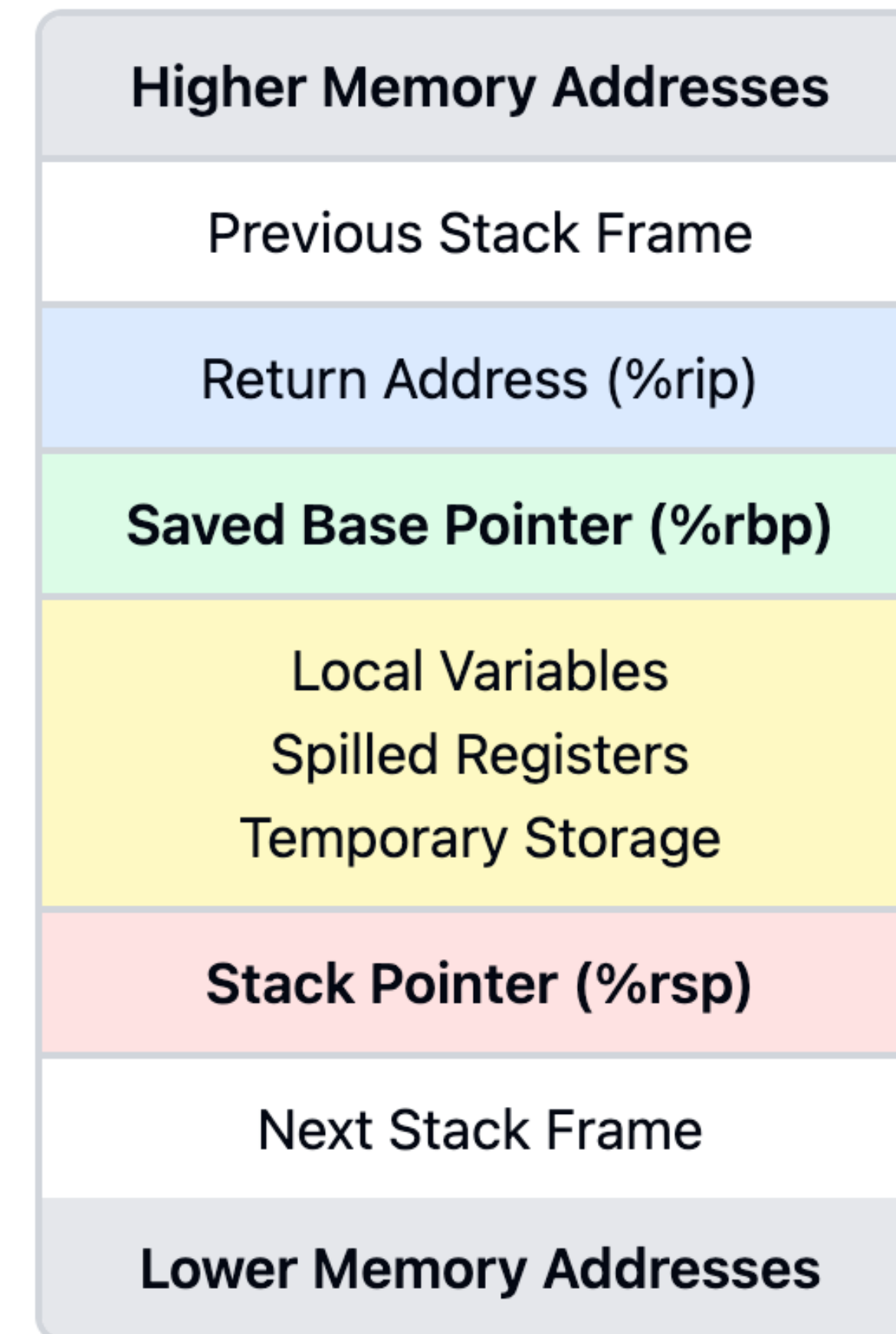
`ret`

Pseudo-code:
`popq %rip`

Pops the top value from the stack into %rip

Stack Frames

- Stack is partitioned into frames (one per function)
- Current function's frame: from base pointer (%rbp) to stack pointer (%rsp)
- Implements functional scope in languages like C
 - Allows different variables with the same name in different function invocations
 - Programmer writes functions with local variables
 - Compiler implements this using stack frames



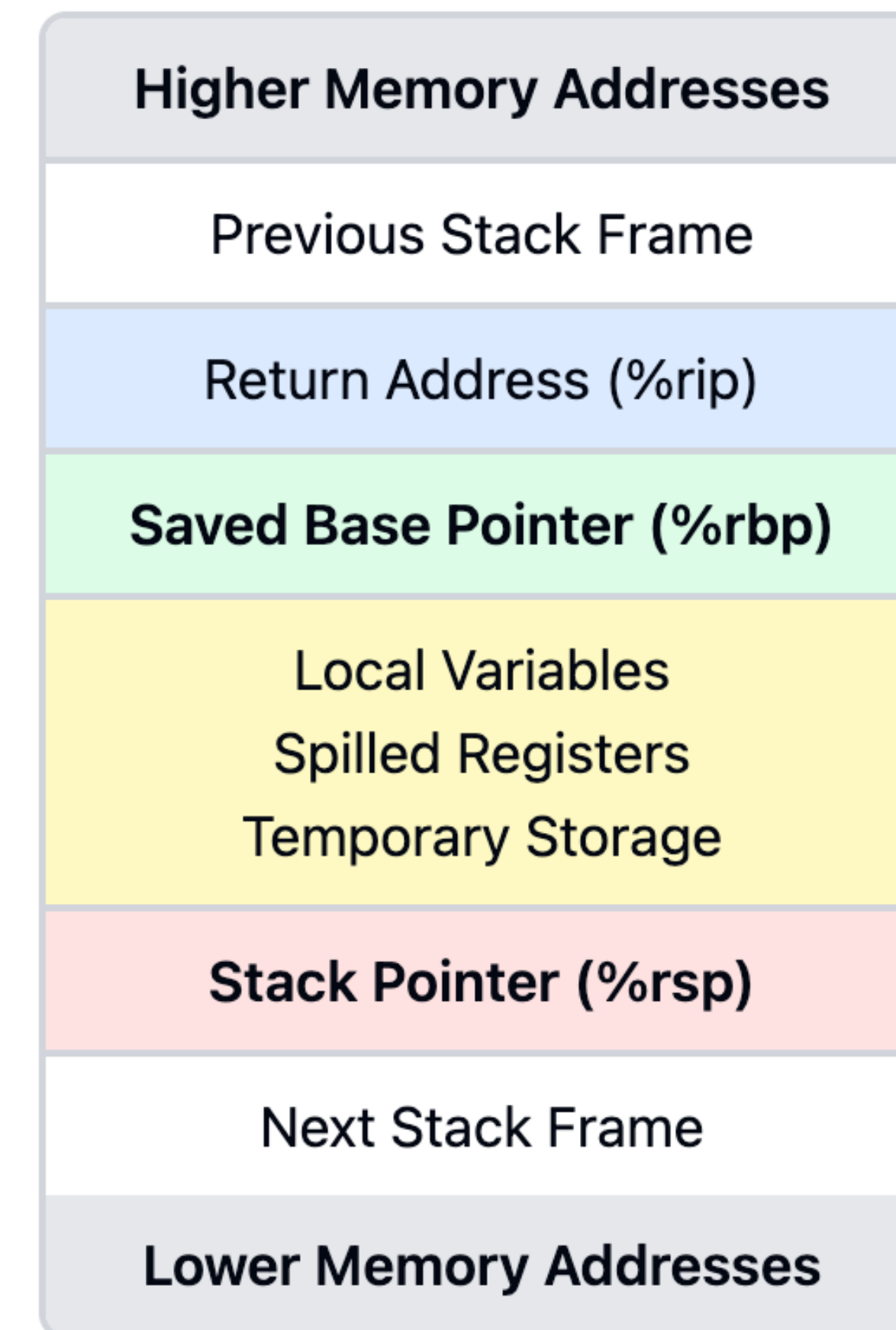
Direction of Stack Growth



Stack Frames (continued)

Function Prologue and Epilogue

- The prologue and epilogue are responsible for maintaining the correct stack frame structure:
- **Prologue:** Saves the old frame pointer, sets up new frame
- **Epilogue:** Restores the old frame pointer
- These operations ensure that when a function returns, the caller's frame pointer is intact



Direction of Stack Growth



Function Calls and Register Management

- Function state (registers) may need to be saved during calls
- This is a **compiler convention**, not hardware architecture

Key Points on Function Calls

Requires agreement between caller and callee on:

- How arguments are passed
- Who is responsible for saving/restoring registers

Stack Frames (continued)

Call-Preserved vs Call-Clobbered Registers

- Call-preserved: Function must save and restore these if used
- Call-clobbered: Caller must save these if their values are needed after the call

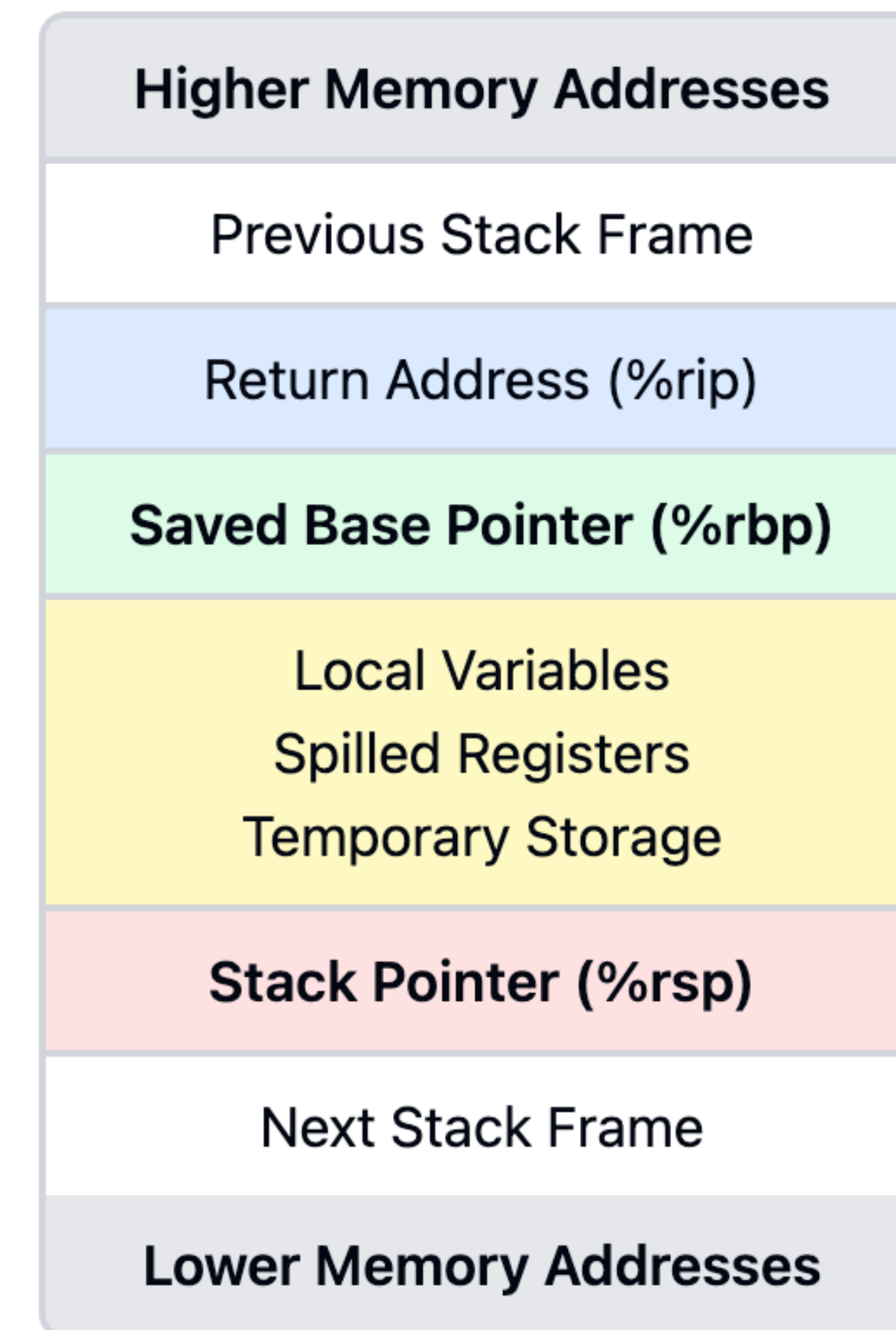
x86-64 Calling Conventions:

Arguments are passed in registers: %rdi, %rsi, %rdx, %rcx

Return value is in register %rax

Call-preserved (**callee-save**) registers: %rbx, %rbp, %r12-%r15

Call-clobbered (**caller-save**) registers: everything else



Direction of Stack Growth



Jan 24, 24 0:24	example.c	Page 1/1
	<pre>1 /* CS202 -- handout 1 2 * compile and run this code with: 3 * \$ gcc -g -Wall -o example example.c 4 * \$./example 5 * 6 * examine its assembly with: 7 * \$ gcc -O0 -S example.c 8 * \$ [editor] example.s 9 */ 10 11 #include <stdio.h> 12 #include <stdint.h> 13 14 uint64_t f(uint64_t* ptr); 15 uint64_t g(uint64_t a); 16 uint64_t* q; 17 18 int main(void) 19 { 20 uint64_t x = 0; 21 uint64_t arg = 8; 22 23 x = f(&arg); 24 25 printf("x: %lu\n", x); 26 printf("dereference q: %lu\n", *q); 27 28 return 0; 29 } 30 31 uint64_t f(uint64_t* ptr) 32 { 33 uint64_t x = 0; 34 x = g(*ptr); 35 return x + 1; 36 } 37 38 uint64_t g(uint64_t a) 39 { 40 uint64_t x = 2*a; 41 q = &x; // <-- THIS IS AN ERROR (AKA BUG) 42 return x; 43 }</pre>	

Jan 24, 24 0:24	as.txt	Page 1/1
	<pre>1 2. A look at the assembly... 2 3 To see the assembly code that the C compiler (gcc) produces: 4 \$ gcc -O0 -S example.c 5 (then look at example.s.) 6 NOTE: what we show below is not exactly what gcc produces. We have 7 simplified, omitted, and modified certain things. 8 9 main: 10 pushq %rbp # prologue: store caller's frame pointer 11 movq %rsp, %rbp # prologue: set frame pointer for new frame 12 13 subq \$16, %rsp # prologue: make stack space 14 15 movq \$0, -8(%rbp) # x = 0 (x lives at address rbp - 8) 16 movq \$8, -16(%rbp) # arg = 8 (arg lives at address rbp - 16) 17 18 leaq -16(%rbp), %rdi # load the address of (rbp-16) into %rdi 19 # this implements "get ready to pass (&arg) 20 # to f" 21 22 call f # invoke f 23 24 movq %rax, -8(%rbp) # x = (return value of f) 25 26 # eliding the rest of main() 27 28 f: 29 pushq %rbp # prologue: store caller's frame pointer 30 movq %rsp, %rbp # prologue: set frame pointer for new frame 31 32 subq \$32, %rsp # prologue: make stack space 33 movq %rdi, -24(%rbp) # Move ptr to the stack 34 # (ptr now lives at rbp - 24) 35 movq \$0, -8(%rbp) # x = 0 (x's address is rbp - 8) 36 37 movq -24(%rbp), %r8 # move 'ptr' to %r8 38 movq (%r8), %r9 # dereference 'ptr' and save value to %r9 39 movq %r9, %rdi # Move the value of *ptr to rdi, 40 # so we can call g 41 42 call g # invoke g 43 44 movq %rax, -8(%rbp) # x = (return value of g) 45 movq -8(%rbp), %r10 # compute x + 1, part I 46 addq \$1, %r10 # compute x + 1, part II 47 movq %r10, %rax # Get ready to return x + 1 48 49 movq %rbp, %rsp # epilogue: undo stack frame 50 popq %rbp # epilogue: restore frame pointer from caller 51 ret # return 52 53 g: 54 pushq %rbp # prologue: store caller's frame pointer 55 movq %rsp, %rbp # prologue: set frame pointer for new frame 56 subq \$0x8, %rsp # prologue: make stack space 57 58 59 60 movq %rbp, %rsp # epilogue: undo stack frame 61 popq %rbp # epilogue: restore frame pointer from caller 62 ret # return</pre>	

Demystifying Pointers and Memory Regions

A pointer (e.g., "int* foo") is simply a variable that stores a memory address.

- **Stack:** Temporary, function-local storage
 - Example: Local variables, function parameters
 - Automatically allocated/deallocated
- **Heap:** Dynamically allocated memory
 - Example: Memory allocated with malloc(), new
 - Manually managed (allocation/deallocation)
- **Text Section:** Read-only program code and constants
 - Example: String literals, const global variables
 - Typically read-only, attempting to modify can cause errors

Pointer Lifetime and Stack Frames

It's a bug to pass or return a pointer to a variable in a prior stack frame.

Quizz Time!

Lab 0 due next Monday!