CS202 (003): Operating Systems
Process I

Instructor: Jocelyn Chen

Most of the materials covered in this slide come from the lecture notes of Mike Walfish’s CS202

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

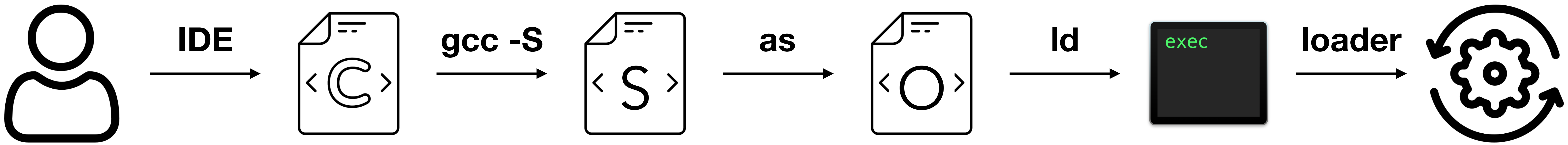
Multiple users use the computer simultaneously

We want to use the resources more efficiently

Increase CPU utilization

Reduce latency

Steps towards creating a process



IDE → source code → gcc -S → assembly code → as → machine code → ld → executable binary → loader → process

source code: a lower-level language that is human-readable

assembly code: a computer process can understand directly but not yet a full program

executable binary: compilation finished ready to execute

process: executable loaded to memory for execution

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”

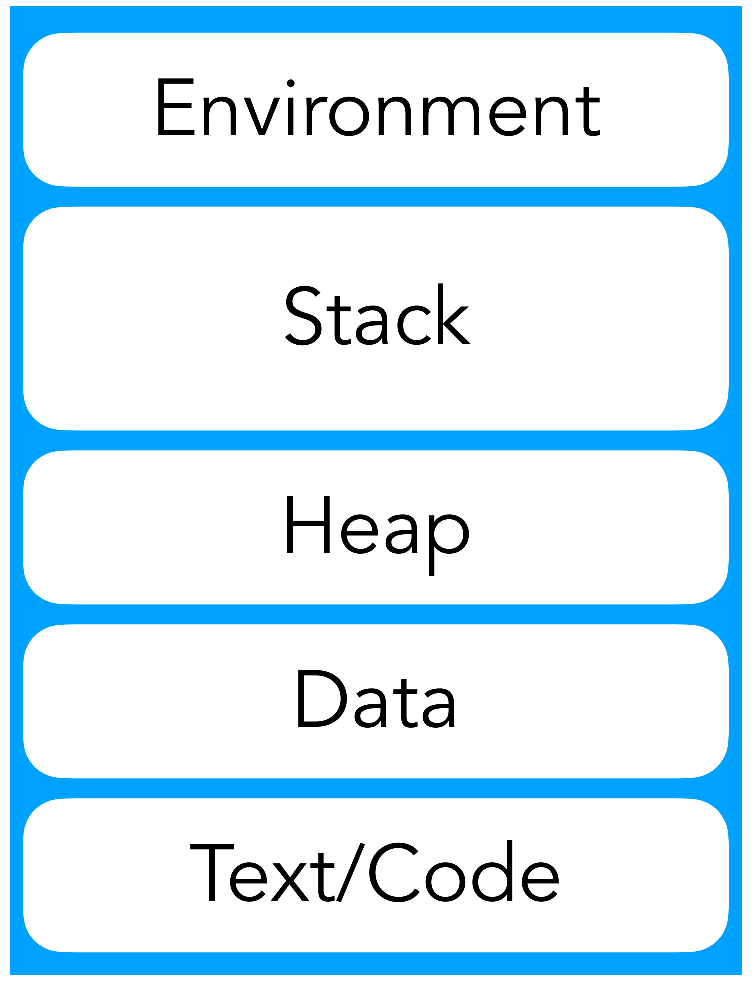
Memory

“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)*

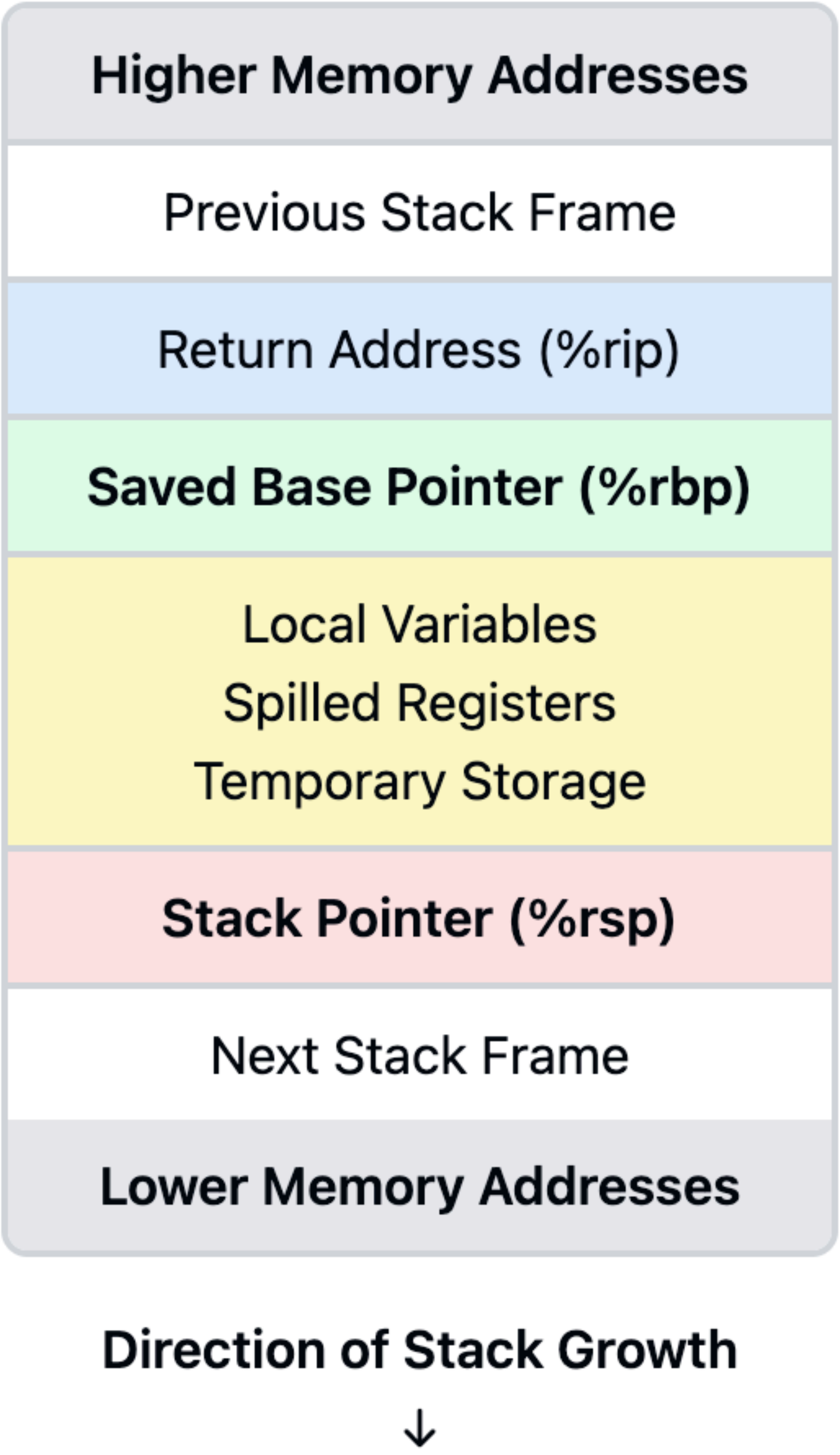
`popq %rax`	`movq (%rsp), %rax`
	`addq $8, %rsp`
		Move the value at the top of the stack to %rax
		Increases the stack pointer by 8 (which means?)

`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



Higher Memory Addresses

Previous Stack Frame

Return Address (%rip)

Saved Base Pointer (%rbp)

Local Variables
Spilled Registers
Temporary Storage

Stack Pointer (%rsp)

Next Stack Frame

Lower Memory Addresses

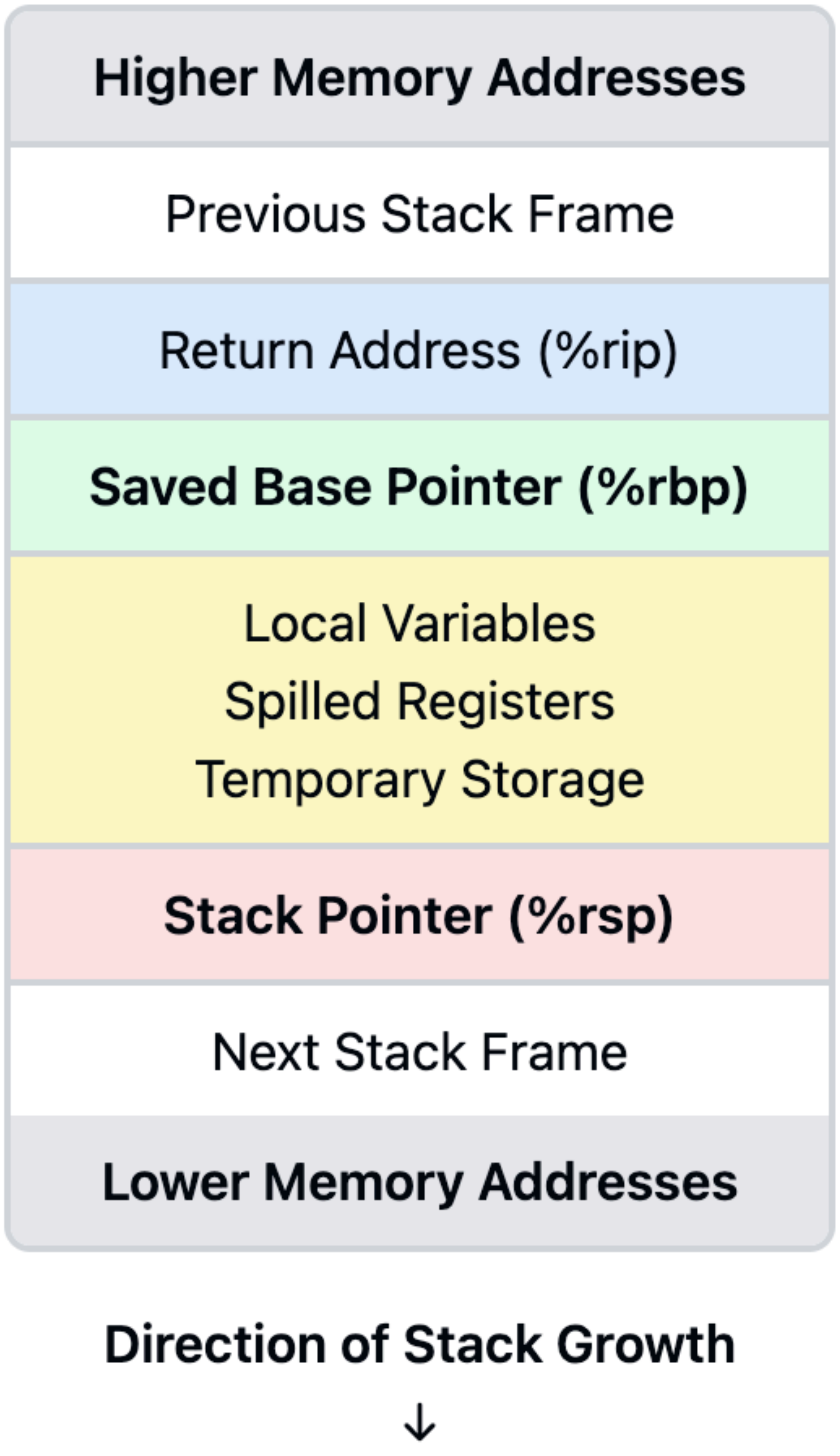
Direction of Stack Growth

↓

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



Higher Memory Addresses

Previous Stack Frame

Return Address (%rip)

Saved Base Pointer (%rbp)

Local Variables
Spilled Registers
Temporary Storage

Stack Pointer (%rsp)

Next Stack Frame

Lower Memory Addresses

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

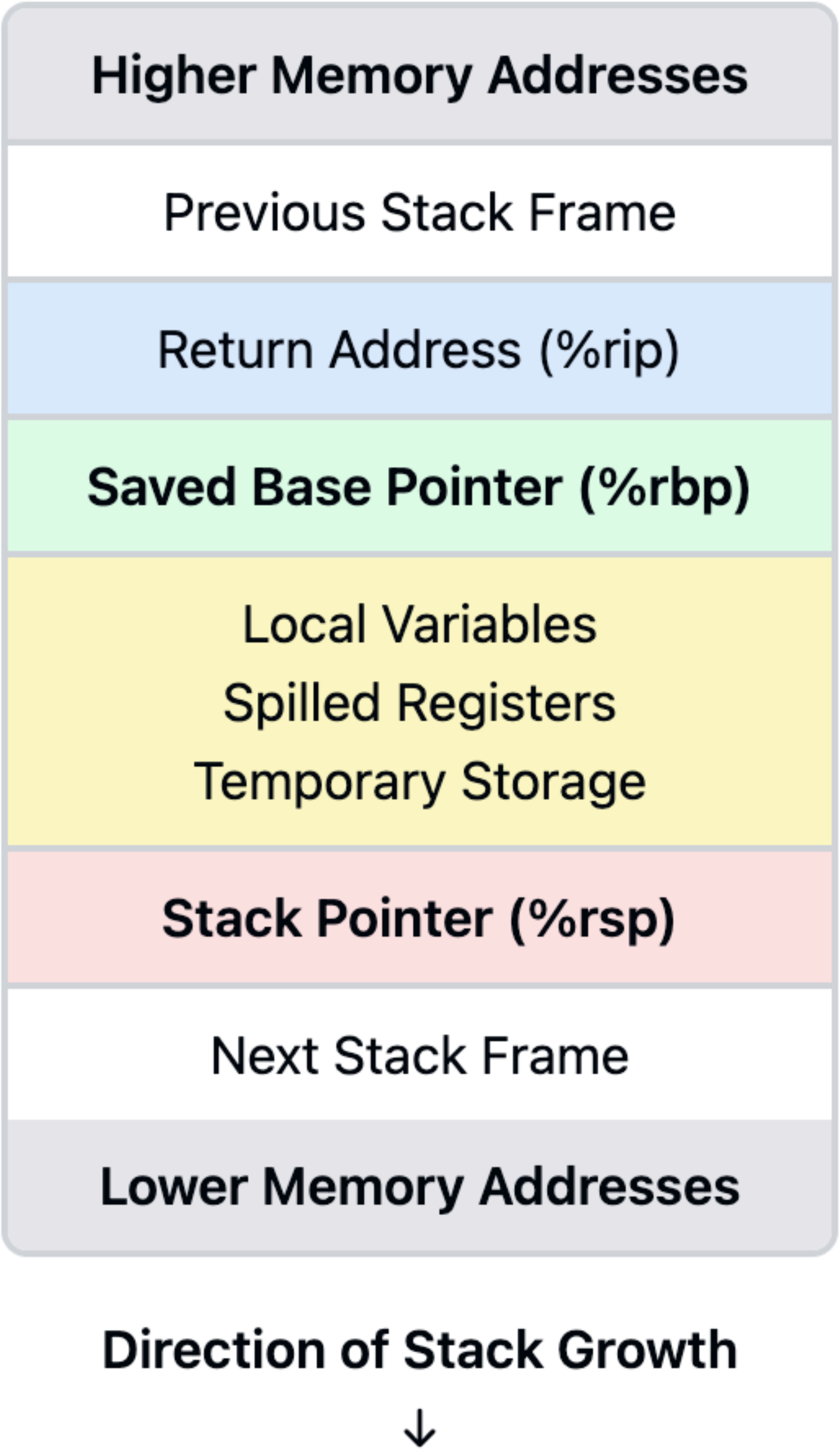
Stack Frames (continued)

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



Higher Memory Addresses

Previous Stack Frame

Return Address (%rip)

Saved Base Pointer (%rbp)

Local Variables
Spilled Registers
Temporary Storage

Stack Pointer (%rsp)

Next Stack Frame

Lower Memory Addresses

Direction of Stack Growth
↓

```
/* CS202 -- handout 1
 *   compile and run this code with:
 *   $ gcc -g -Wall -o example example.c
 *   $ ./example
 *
 *   examine its assembly with:
 *   $ gcc -O0 -S example.c
 *   $ [editor] example.s
*/

#include <stdio.h>
#include <stdint.h>

uint64_t f(uint64_t* ptr);
uint64_t g(uint64_t a);
uint64_t* q;

int main(void)
{
    uint64_t x = 0;
    uint64_t arg = 8;

    x = f(&arg);

    printf("x: %lu\n", x);
    printf("dereference q: %lu\n", *q);

    return 0;
}

uint64_t f(uint64_t* ptr)
{
    uint64_t x = 0;
    x = g(*ptr);
    return x + 1;
}

uint64_t g(uint64_t a)
{
    uint64_t x = 2*a;
    q = &x;    // <-- THIS IS AN ERROR (AKA BUG)
    return x;
}
```

```
2. A look at the assembly...

    To see the assembly code that the C compiler (gcc) produces:
        $ gcc -O0 -S example.c
    (then look at example.s.)
    NOTE: what we show below is not exactly what gcc produces. We have
    simplified, omitted, and modified certain things.

    main:
        pushq   %rbp                # prologue: store caller's frame pointer
        movq    %rsp, %rbp          # prologue: set frame pointer for new frame

        subq    $16, %rsp           # prologue: make stack space

        movq    $0, -8(%rbp)        # x = 0 (x lives at address rbp - 8)
        movq    $8, -16(%rbp)       # arg = 8 (arg lives at address rbp - 16)

        leaq   -16(%rbp), %rdi      # load the address of (rbp-16) into %rdi
                                    # this implements "get ready to pass (&arg)
                                    # to f"

        call    f                   # invoke f

        movq    %rax, -8(%rbp)      # x = (return value of f)

        # eliding the rest of main()

    f:
        pushq   %rbp                # prologue: store caller's frame pointer
        movq    %rsp, %rbp          # prologue: set frame pointer for new frame

        subq   $32, %rsp            # prologue: make stack space
        movq   %rdi, -24(%rbp)      # Move ptr to the stack
                                    # (ptr now lives at rbp - 24)
        movq   $0, -8(%rbp)         # x = 0 (x's address is rbp - 8)

        movq    -24(%rbp), %r8      # move 'ptr' to %r8
        movq    (%r8), %r9          # dereference 'ptr' and save value to %r9
        movq    %r9, %rdi           # Move the value of *ptr to rdi,
                                    # so we can call g

        call    g                   # invoke g

        movq   %rax, -8(%rbp)       # x = (return value of g)
        movq   -8(%rbp), %r10       # compute x + 1, part I
        addq    $1, %r10            # compute x + 1, part II
        movq    %r10, %rax          # Get ready to return x + 1

        movq    % rbp  %rsp         # epilogue: undo stack frame
        popq    %rbp                # epilogue: restore frame pointer from caller
        ret                         # return

    g:
        pushq   %rbp                # prologue: store caller's frame pointer
        movq    %rsp, %rbp          # prologue: set frame pointer for new frame
        subq    $0x8, %rsp          # prologue: make stack space

        ....

        movq    %rbp, %rsp          # epilogue: undo stack frame
        popq    %rbp                # epilogue: restore frame pointer from caller
        ret                         # return
```

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 this Friday!