Lecture 03 – Our target architecture: x86-64

A brief introduction

Stephen Checkoway

Outline for today

Why study attacks

Processes and the kernel (a CSCI 241 review)

Our target architecture: x86-64

Why study attacks?

Identify vulnerabilities so they can be fixed

Create incentives for vendors to be careful

Learn about new classes of threats

- Determine what we need to defend against
- Help designers build stronger systems
- Help users more accurately evaluate risk

The processor executes code

Review from CSCI 241

Code running on the processor falls into one of two camps

- User mode (applications)
- Kernel mode (operating system, device drivers)

Kernel mode has direct access to hardware

User mode only has access to memory that has been assigned to it; must ask the kernel to perform actions on its behalf (system calls)

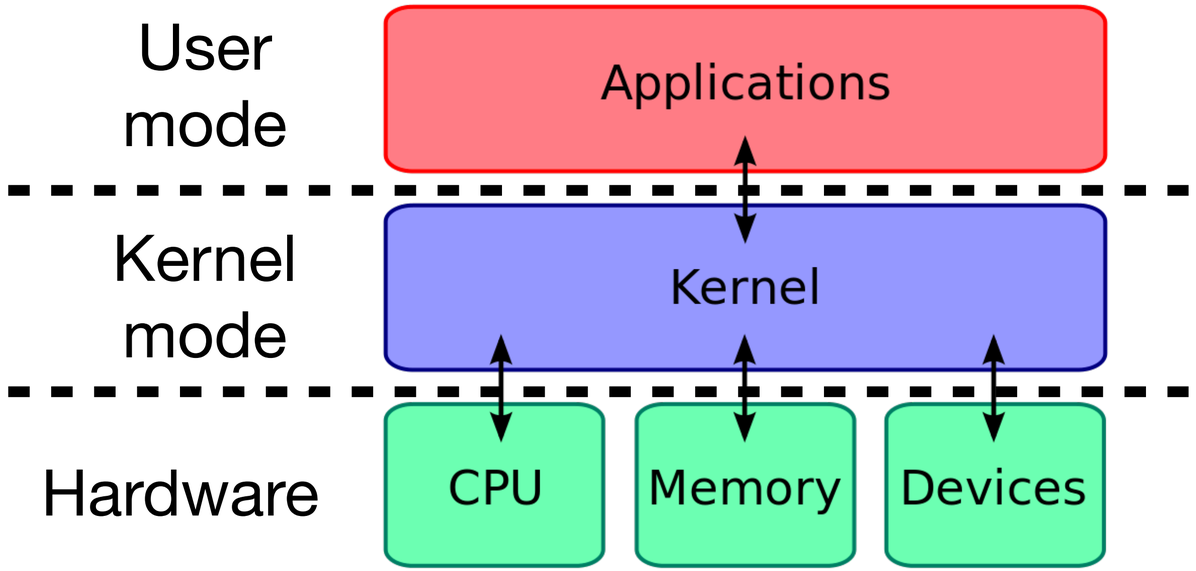
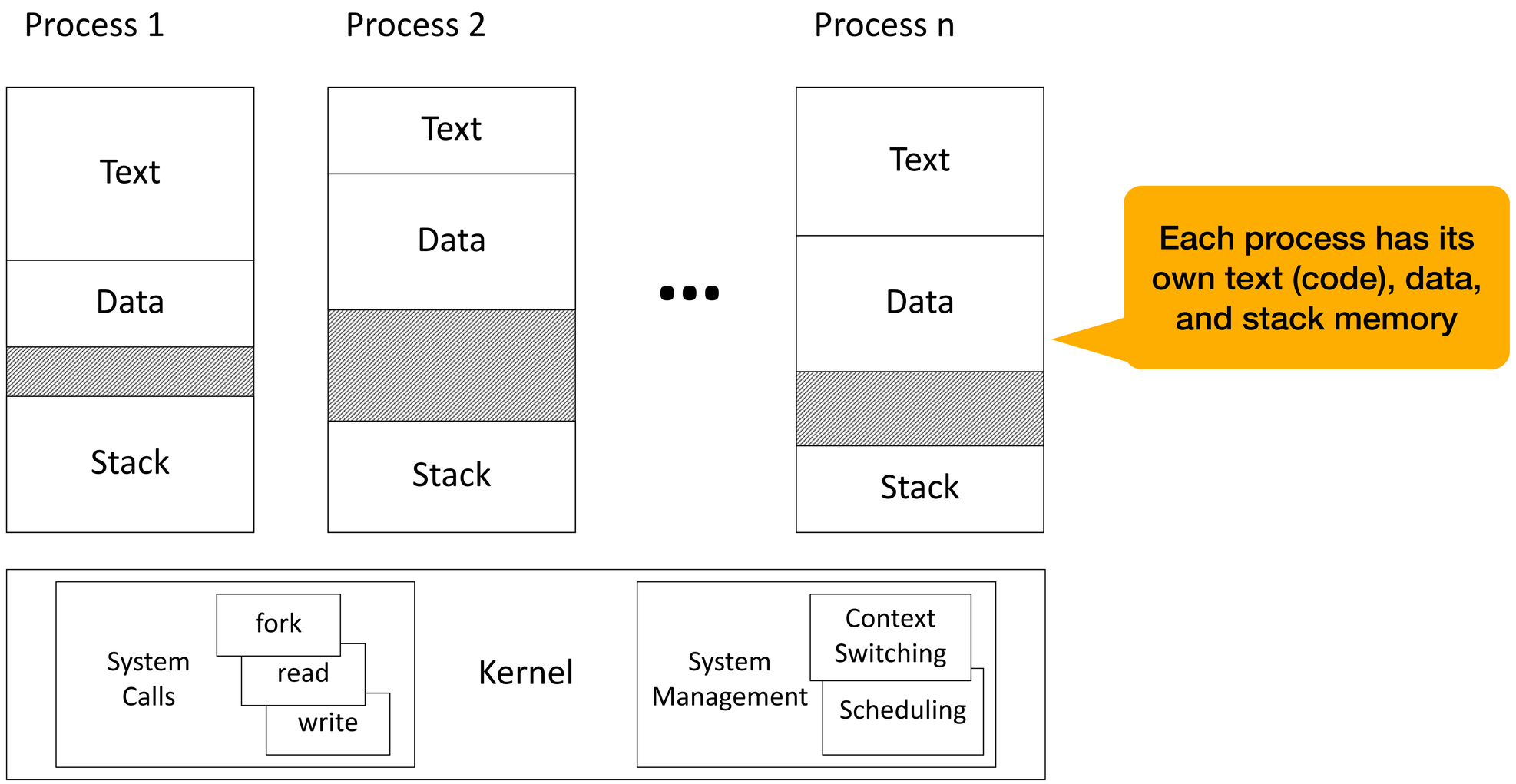


Image: https://en.wikipedia.org/wiki/Kernel_(operating_system)

Process and Kernel Model



Virtual memory

Each running process has its own virtual memory space

- Your computer has a bunch of RAM (random access memory)
- RAM is an array of bytes indexed from 0
- It would be bad if any process could read/write any byte of memory (which is how DOS and early Windows and classic Mac OS worked!)
- The operating system carves up physical memory and gives chunks of it to each process
- Hardware enforces separation; i.e., process A cannot modify the memory of process B (without intentional cooperation)
- This memory forms the virtual memory space of the process

Image: https://en.wikipedia.org/wiki/RAM

Virtual address space

OS presents each process with the fiction that it has access to the entire range of memory from index 0 to the maximum index (determined by hardware and OS)

It does this by mapping virtual addresses used by processes to physical addresses used by the hardware

(It can even write “pages” of memory to disk so that more all of the processes can collectively use more memory than there is physical RAM; this is paging)

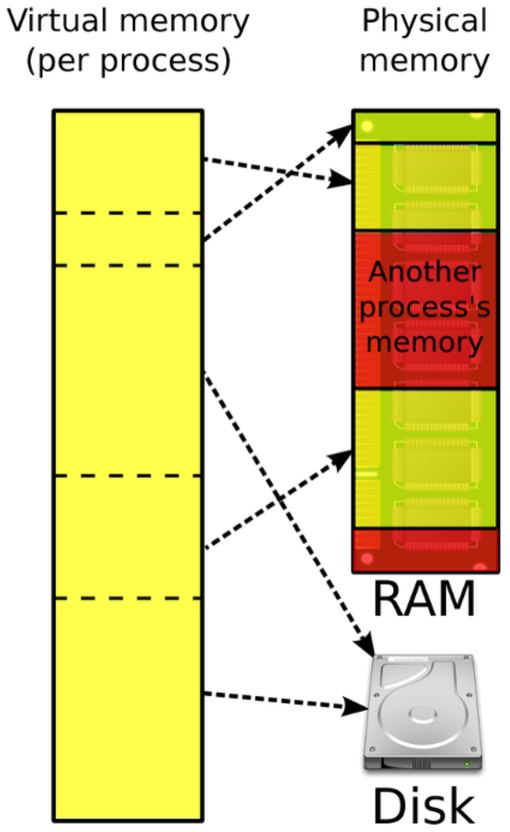
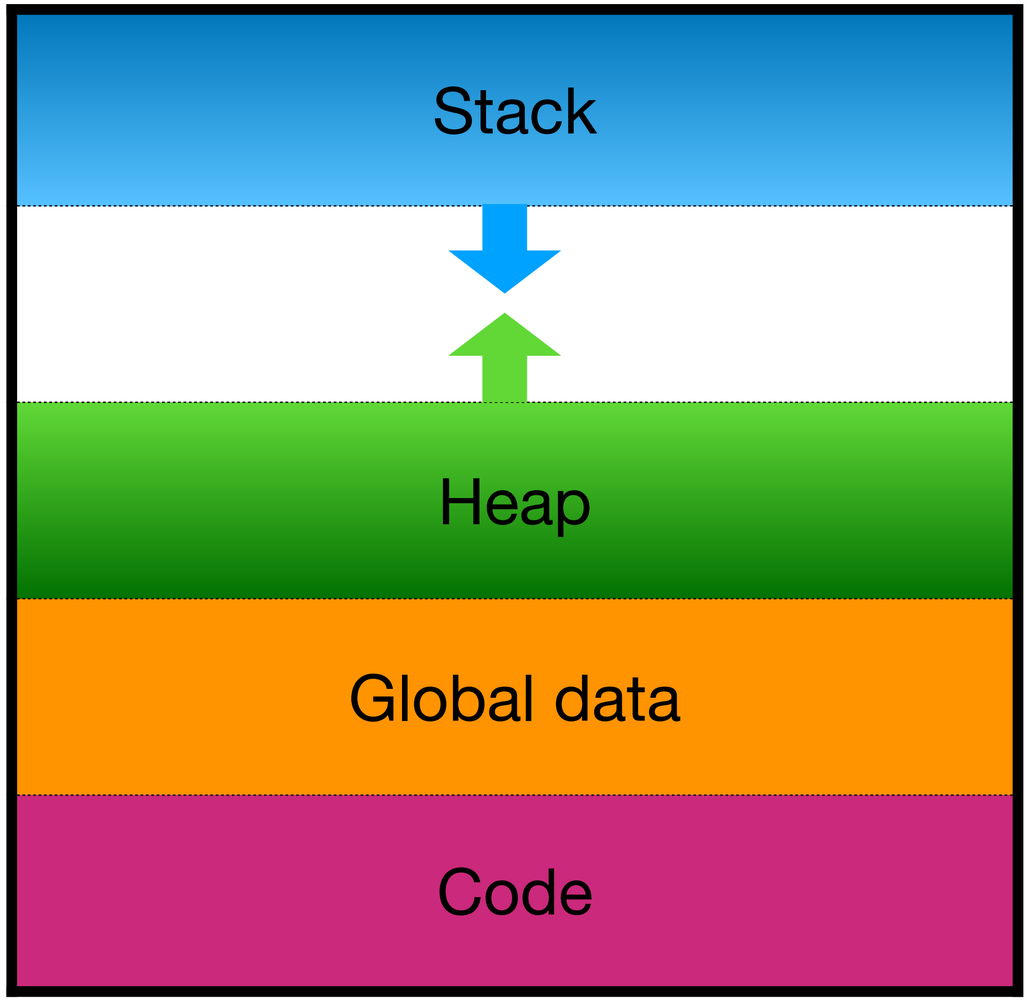


Image: https://en.wikipedia.org/wiki/Virtual_memory

Virtual address space layout (simplified)

High memory address



Low memory address

Stack grows down

Heap grows up

Global data and
Code are fixed size

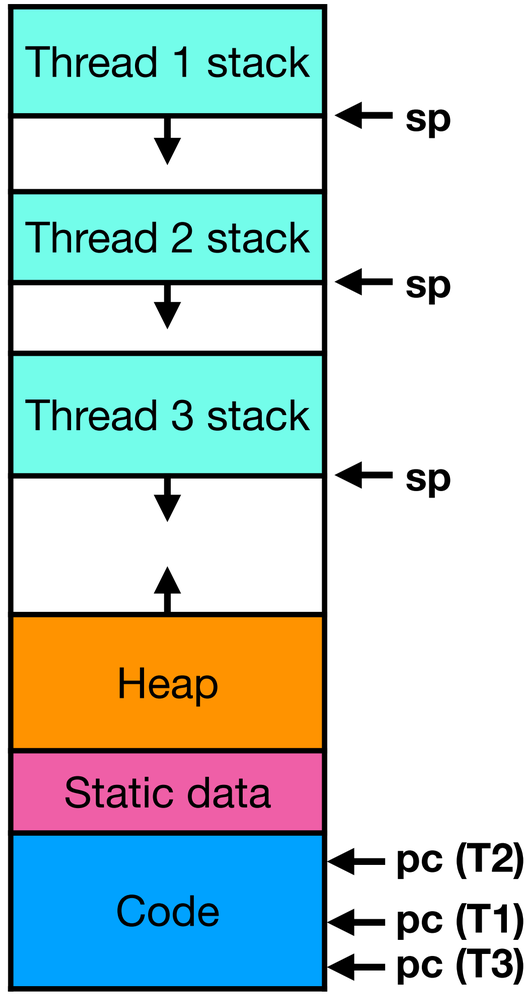
Aside: multi-threaded programs

Each thread in a process shares all of the process's

- memory (data and code)
- open files
- permissions (e.g., to access the file system)
- user ID, group ID, process ID

Each thread in a process has its own

- function call stack with a stack pointer (sp)
- program counter (pc) indicating the next instruction to execute



Virtual memory addresses on Linux

/proc/<pid>/maps

```
$ cat /proc/$$/maps | egrep 'bash|heap|stack|libc\.'
587dbdf4c000-587dbdf7c000 r--p 00000000 fc:00 786900                     /usr/bin/bash
587dbdf7c000-587dbe06b000 r-xp 00030000 fc:00 786900                     /usr/bin/bash
587dbe06b000-587dbe0a0000 r--p 0011f000 fc:00 786900                     /usr/bin/bash
587dbe0a0000-587dbe0a4000 r--p 00154000 fc:00 786900                     /usr/bin/bash
587dbe0a4000-587dbe0ad000 rw-p 00158000 fc:00 786900                     /usr/bin/bash
587debcf3000-587debe8d000 rw-p 00000000 00:00 0                          [heap]
71e7a4000000-71e7a4028000 r--p 00000000 fc:00 805771                     /usr/lib/x86_64-linux-gnu/libc.so.6
71e7a4028000-71e7a41b0000 r-xp 00028000 fc:00 805771                     /usr/lib/x86_64-linux-gnu/libc.so.6
71e7a41b0000-71e7a41ff000 r--p 001b0000 fc:00 805771                     /usr/lib/x86_64-linux-gnu/libc.so.6
71e7a41ff000-71e7a4203000 r--p 001fe000 fc:00 805771                     /usr/lib/x86_64-linux-gnu/libc.so.6
71e7a4203000-71e7a4205000 rw-p 00202000 fc:00 805771                     /usr/lib/x86_64-linux-gnu/libc.so.6
7ffd01367000-7ffd01388000 rw-p 00000000 00:00 0                          [stack]
```

Note how the stack lives at the highest addresses

Dynamic libraries are mapped into Bash's address space (only libc shown)

The heap lives between the code and the dynamic libraries

Each range of memory has its own permissions **r**ead, **w**rite, and e**x**ecute; we'll come back to this later

Stack layout

Each function called in a program is allocated a *stack frame* on the function call stack (*the* stack)

Each frame stores

- The return address
- Local variables
- Arguments to functions it calls (if needed)
- Saved register values

Software uses one or two registers to keep track of the stack

- The stack pointer points to the top (lowest address) of the stack
- The frame pointer points to a fixed location in the stack frame (not always used)

On function calls

When a function is called, the called function

- Allocates space for its own stack frame by adjusting the stack pointer
- Saves the value of the caller's frame pointer and sets up its own frame pointer (if used)
- Saves the values of caller-saved registers is uses

When the function returns, it

- Restores the values of caller-saved registers from the stack frame (including the frame pointer)
- Adjusts the stack pointer back to what it was when the function was called
- Returns to the function that called it

x86-64 (a.k.a. amd64)

64-bit variant of x86

16 64-bit general purpose registers (GPRs) (although some do have fixed purposes for specific instructions)

- rax, rbx, rcx, rdx, rsi, rdi, rsp, rbp, r8, r9, r10, r11, r12, r13, r14, and r15 (these are analogous to the integers registers in MIPS, $t0–$t9, $s0-$s7, $sp, etc.
- rsp is the stack pointer ($sp in MIPS)
- rbp is the frame pointer ($fp in MIPS) but often used as a general purpose integer register

Instruction pointer rip ($pc in MIPS) holds the address of the next instruction to execute

rflags register has bits like the zero flag or the carry flag that are set by arithmetic and logical operations; used for conditional control flow (no analog in MIPS which uses slt/beq/bne)

x86-64

Each 64-bit GPR can be used as a 32-, 16-, or 8-bit register by accessing the low-order 32-, 16-, or 8-bits of the register

Monikers					Description
64-bit	32-bit	16-bit	8 high bits of lower 16 bits	8-bit	
RAX	EAX	AX	AH	AL	Accumulator
RBX	EBX	BX	BH	BL	Base
RCX	ECX	CX	CH	CL	Counter
RDX	EDX	DX	DH	DL	Data (commonly extends the A register)
RSI	ESI	SI	N/A	SIL	Source index for string operations
RDI	EDI	DI	N/A	DIL	Destination index for string operations
RSP	ESP	SP	N/A	SPL	Stack Pointer
RBP	EBP	BP	N/A	BPL	Base Pointer (meant for stack frames)
R8	R8D	R8W	N/A	R8B	General purpose
R9	R9D	R9W	N/A	R9B	General purpose
R10	R10D	R10W	N/A	R10B	General purpose
R11	R11D	R11W	N/A	R11B	General purpose
R12	R12D	R12W	N/A	R12B	General purpose
R13	R13D	R13W	N/A	R13B	General purpose
R14	R14D	R14W	N/A	R14B	General purpose
R15	R15D	R15W	N/A	R15B	General purpose

Table: https://wiki.osdev.org/CPU_Registers_x86-64

x86-64 assembly language

There are two main flavors of x86 (and x86-64) assembly language: Intel format and AT&T format

I’m going to try to stick with Intel format since it’s what’s in the Intel manual

For some reason, the default on Windows is to use Intel and the default on every other OS (macOS, BSD, Linux, etc.) is to use AT&T

Most tools that work with assembly can use either syntax

Annoyingly, the operand order for instructions in AT&T is the opposite of the order in Intel format, there are other syntactic differences as well

Some x86-64 instructions

mov dest, src Copies data from src to dest; performs the role of move, lw, and sw from MIPS

Arithmetic and bit operations

- add dest, src Computes dest = dest + src
- sub dest, src Computers dest = dest - src
- or dest, src Computes dest = dest | src
- Similar for AND and xor

Multiplication and division use specific registers

- mul src Unsigned multiplication rdx:rax = rax * src
- div src Unsigned division of rdx:rax by src; stores quotient in rax and remainder in rdx
- imul/idiv Signed multiplication/division; multiple formats each with slightly different behavior; read the manual for details!

Some examples

When the e– version of the register is used, the upper 32 bits are set to 0

mov eax, 100000 Sets rax to 100000

mov rdi, r8 Sets rdi to r8

xor ecx, ecx Sets rcx = 0 (ecx = ecx XOR ecx combined with zeroing upper 32)

sub rsp, 128 Decrements the stack pointer by 128

Stack operations

Unlike MIPS, there are specific stack manipulation instructions

push src	Decrements rsp by 8, writes src to stack
pop dest	Loads the top of the stack into dest, increments rsp by 8

Control flow instructions

Comparison

- cmp src1, src2 — Computes src1 - src2 and sets rflags bits
- test src1, src2 — Computes src1 & src2 and sets rflags bits

Control flow

- jz label — Jump to label if the zero flag is set
- jnz label — Jump to label if the zero flag is not set
- jc label — Jump to label if the carry flag is set
- jnc label — Jump to label if the carry flag is not set
- jmp label — Jump to label unconditionally

Function call instructions

Function calls

- call label Calls the function label and pushes the address of the next instruction (rip) onto the stack
- ret Pops the top of the stack into rip (returns from the function)
- leave Equivalent to mov rsp, rbp followed by pop rbp

leave is sometimes used just before ret when a frame pointer (rbp) is in use;
sometimes the compiler will use multiple instructions instead like
mov rbp, [rsp + 32] [rsp + 32] means move from memory at address rsp+32
add rsp, 40
ret

Operand differences between MIPS and x86-64

x86-64 is a CISC architecture, it's very different from the RISC architecture you saw in CSCI 210

In MIPS, almost every instruction used only register or immediate operands, r-type or i-type instructions

In MIPS r-type instructions take 3 operands like add $t0, $t2, $s0; in x86-64, the equivalent instructions only take 2 operands and one of which is both a source and a destination

MIPS uses explicit load/store instructions, e.g., lw $t0, 32($s4) to mean load the 32 bits from address $s4 + 32 into register $t0

x86-64 allows up to one operand of almost every instruction to be a memory address

x86-64 operands

Different instructions have different operand restrictions but generally speaking

- Up to 1 operand (either source or destination) can be a memory location
- Up to 1 source operand can be an 8- or 32-bit immediate (constant)

Memory operands come in multiple forms, each specifies an address

- A literal number — The number is the address (more common in 32-bit)
- [rax] — Address is the value stored in register rax
- [rax + 8*rdx] — Address is the value rax + 8*rdx
- [rax + 4*rsi + 900] — Address is the value rax + 4*rsi + 900

In the base + scaled index and base + scaled index + offset operands, the scale value is one of 1, 2, 4, or 8

You can replace rax, rdx, and rsi with any other GPR, those were just examples above

Operand size

In the Intel assembly format, it’s not always obvious if the operand should be 8, 16, 32, or 64 bits

This is especially true for memory operands. E.g., does [rdi + 8] refer to the 8, 16, 32, or 64 bits whose address is rdi + 8?

The Intel format uses additional descriptors to make it clear

- BYTE PTR [rdi + 24] means 8-bits
- WORD PTR [rdi + 24] means 16-bits
- DWORD PTR [rdi + 24] means 32-bits
- QWORD PTR [rdi + 24] means 64-bits

One additional weird instruction

Load effective address

The x86-64 hardware needs to compute the address for memory operands prior to loading/storing values in memory

The most general of which is the base + scaled index + offset, e.g.,

- [rax + 8*rdx + 32]

The lea reg, mem instruction exposes this hardware to the assembly programmer; it computes the “effective address” and stores the result **without reading/writing memory** into the destination register

- lea r8, [rax + 8*rdx + 32] computes r8 = rax + 8*rdx + 32

Example

Consider the C code

```
int foo(int *p, int a) {
  int x = p[0] + p[1] * a - 85;
  return x;
}
```

gcc compiles this code (with optimization) to

```
foo:
        imul    esi, DWORD PTR [rdi+4]
        add     esi, DWORD PTR [rdi]
        lea     eax, [rsi-85]
        ret
```

rdi is the first argument p

rsi is the second argument a but since we're only using 32-bit ints, the compiler used esi to refer to the lower 32-bits of rsi

imul esi, [rdi + 4] computes esi = esi * [rdi + 4]

lea eax, [rsi-85] computes eax = rsi - 85

rax holds the return value

x86-64 calling convention

There are two different x86-64 calling conventions

- Microsoft's x64 calling convention which we won't use or discuss
- **System V AMD64 ABI** which is what everything but Windows uses

The first 6 integer or pointer arguments to a function go in registers rdi, rsi, rdx, rcx, r8, and r9, in that order [no, I don't ever remember this, I look it up every time]; these are analogous to MIPS's $a0, $a1, $a2, and $a3

Any remaining integer/pointer arguments are passed on the stack (just like MIPS)

Floating point arguments go in registers xmm0 through xmm7; we won't need to deal with floating point in this class

x86-64 calling convention continued

An integer/pointer return value is returned in register rax (analogous to $v0 in MIPS)

Callee-saved registers (if the called function wants to use the below registers, it must save their original values on the stack and restore them before returning)

- rbx, rsp, rbp, r12, r13, r14, and r15
- These are analogous to the saved registers $s0–$s7, $sp, and $fp in MIPS

Other registers may be freely used by the called function. If the **calling** function wants to keep their values intact, it must save them prior to calling the function

- I.e., rax, rcx, rdx, rdi, rsi, r8, r9, r10, and r11 are caller-saved
- These are analogous to the temporary registers $t0–$t9, $a0–$a3, $v0, $v1

That's a *lot* of information

Do I need to memorize that all?

NO!

https://en.wikipedia.org/wiki/X86_calling_conventions

But even better, most of the time, we're going to be reading assembly and not writing it

- The compiler will have already gotten the details correct
- We mostly care about understanding existing code and that's **much** easier

Use the Compiler Explorer https://godbolt.org/ to see how compilers compile code