Systems Programming Concepts

Hic sunt dracones

MAJ John Rollinson

2023-03-08

Army Cyber Institute

Introductions

My Expectations

- **Ask questions**

My Expectations

- **Ask questions**
    - If you’re confused now, so is someone else.

My Expectations

- **Ask questions**
    - If you're confused now, so is someone else.
    - If you're confused now, you will still be on the next slide.

My Expectations

- **Ask questions**
  - If you're confused now, so is someone else.
  - If you're confused now, you will still be on the next slide.
  - If you're confused now, the practical portions will hurt.

My Expectations

- **Ask questions**
    - If you're confused now, so is someone else.
    - If you're confused now, you will still be on the next slide.
    - If you're confused now, the practical portions will hurt.
- **Don't "cheat"**

My Expectations

- **Ask questions**
  - If you're confused now, so is someone else.
  - If you're confused now, you will still be on the next slide.
  - If you're confused now, the practical portions will hurt.
- **Don't "cheat"**
  - There's no grade

My Expectations

- **Ask questions**
  - If you're confused now, so is someone else.
  - If you're confused now, you will still be on the next slide.
  - If you're confused now, the practical portions will hurt.
- **Don't "cheat"**
  - There's no grade
  - Defeats the purpose of being here

My Expectations

- **Ask questions**
  - If you're confused now, so is someone else.
  - If you're confused now, you will still be on the next slide.
  - If you're confused now, the practical portions will hurt.
- **Don't "cheat"**
  - There's no grade
  - Defeats the purpose of being here
  - Won't work later

Course Overview

Themes

It's all ~~code~~ logic – not magic.

Abstractions leave gaps.

Own your tools.

- MAJ Chuck Suslowicz (possibly stolen from Roy Ragsdale)

Themes

Manpages are your friend.[1]

- A previous C3T coach...

[1] You're documenting your own code, right?

What to Expect

- Lots of information
- Lecture followed by “hands-on”
- Open-ended exercises (may not have “right” answers)

Course Objectives

- Understand the hardware/software interface
- Apply virtual memory concepts to userspace code
- Describe core OS concepts
- Examine code for concurrency issues
- Apply synchronization primitives & deadlock prevention techniques
- Understand distributed systems concepts

Course Schedule

- Day 1: CPUs & Instructions
- Day 2: Virtual Memory
- Day 3: Operating Systems Overview
- Day 4: Concurrency
- Day 5: Distributed Systems Concepts

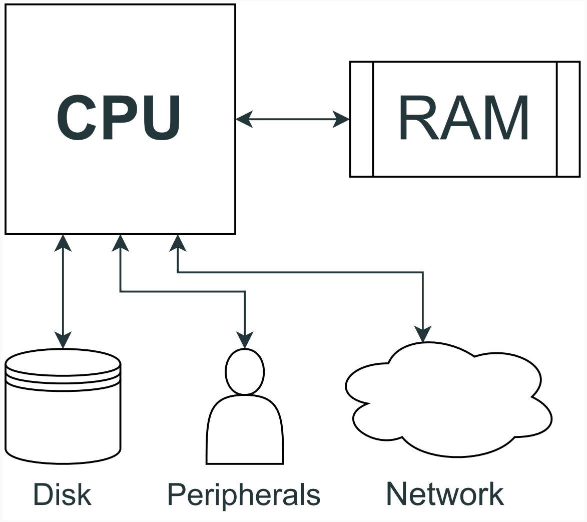
CPUs & Instructions

Anatomy of a computer

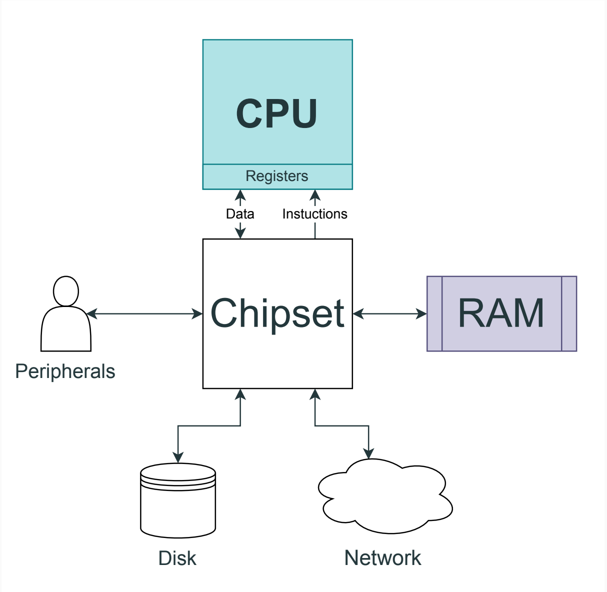
What *is* a computer?

What are the *components* of a computer?

Anatomy of a computer: Naive CPU



Anatomy of a computer: 2-Stage CPU



Anatomy of a computer: 5-Stage CPU

1. Fetch

Anatomy of a computer: 5-Stage CPU

1. Fetch
2. Decode

Anatomy of a computer: 5-Stage CPU

1. Fetch
2. Decode
3. Execute

Anatomy of a computer: 5-Stage CPU

1. Fetch
2. Decode
3. Execute
4. Memory

Anatomy of a computer: 5-Stage CPU

1. Fetch
2. Decode
3. Execute
4. Memory
5. Write-back

Anatomy of a computer: Stages Example

Initial state:

```
rip          = 0x40000
rsi          = 0x41000
mem[0x41000] = 0xDEADBEEF
```

1. **Fetch**: `48 8b 06` (@ address of *rip*)

Anatomy of a computer: Stages Example

Initial state:

```
rip          = 0x40000
rsi          = 0x41000
mem[0x41000] = 0xDEADBEEF
```

1. **Fetch**: `48 8b 06` (@ address of *rip*)
2. **Decode**: `mov rax, [rsi]`

Anatomy of a computer: Stages Example

Initial state:

```
rip          = 0x40000
rsi          = 0x41000
mem[0x41000] = 0xDEADBEEF
```

1. **Fetch**: `48 8b 06` (@ address of *rip*)
2. **Decode**: `mov rax, [rsi]`
3. **Execute**: `mov rax, [0x41000]`

Anatomy of a computer: Stages Example

Initial state:

```
rip          = 0x40000
rsi          = 0x41000
mem[0x41000] = 0xDEADBEEF
```

1. **Fetch**: `48 8b 06` (@ address of *rip*)
2. **Decode**: `mov rax, [rsi]`
3. **Execute**: `mov rax, [0x41000]`
4. **Memory**: `mov rax, 0xDEADBEEF`

Anatomy of a computer: Stages Example

Initial state:

```
rip          = 0x40000
rsi          = 0x41000
mem[0x41000] = 0xDEADBEEF
```

1. **Fetch**: `48 8b 06` (@ address of *rip*)
2. **Decode**: `mov rax, [rsi]`
3. **Execute**: `mov rax, [0x41000]`
4. **Memory**: `mov rax, 0xDEADBEEF`
5. **Writeback**: $rax = 0xDEADBEEF$

Anatomy of a computer: Pipelining Example

CPU

Fetch	
Decode	
Execute	
Memory	
Writeback	

Data (0x1000)

```
1
42
1337
0
```

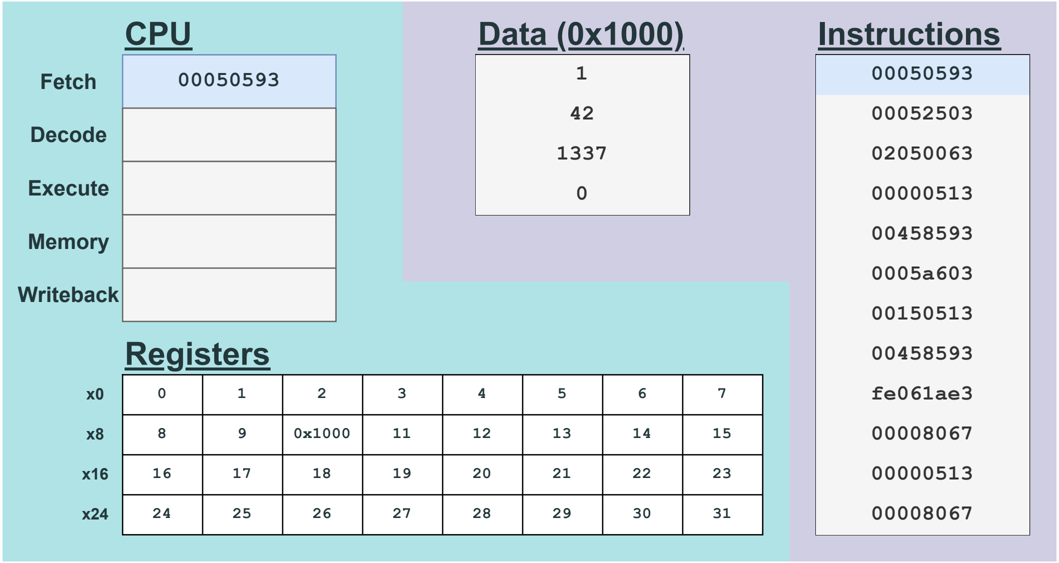
Instructions

```
00050593
00052503
02050063
00000513
00458593
0005a603
00150513
00458593
fe061ae3
00008067
00000513
00008067
```

Registers

x0	0	1	2	3	4	5	6	7
x8	8	9	0x1000	11	12	13	14	15
x16	16	17	18	19	20	21	22	23
x24	24	25	26	27	28	29	30	31

Anatomy of a computer: Pipelining Example



x0	0	1	2	3	4	5	6	7
x8	8	9	0x1000	11	12	13	14	15
x16	16	17	18	19	20	21	22	23
x24	24	25	26	27	28	29	30	31

Anatomy of a computer: Pipelining Example

CPU

Fetch	00052503
Decode	mv a1, a0
Execute	
Memory	
Writeback	

Data (0x1000)

1
42
1337
0

Instructions

00050593
00052503
02050063
00000513
00458593
0005a603
00150513
00458593
fe061ae3
00008067
00000513
00008067

Registers

x0	0	1	2	3	4	5	6	7
x8	8	9	0x1000	11	12	13	14	15
x16	16	17	18	19	20	21	22	23
x24	24	25	26	27	28	29	30	31

Anatomy of a computer: Pipelining Example

CPU

Fetch	02050063
Decode	`lw a0, 0(a0)`
Execute	`mv a1, 0x1000`
Memory	
Writeback	

Data (0x1000)

1
42
1337
0

Instructions

00050593
00052503
02050063
00000513
00458593
0005a603
00150513
00458593
fe061ae3
00008067
00000513
00008067

Registers

x0	0	1	2	3	4	5	6	7
x8	8	9	0x1000	11	12	13	14	15
x16	16	17	18	19	20	21	22	23
x24	24	25	26	27	28	29	30	31

Anatomy of a computer: Pipelining Example

CPU

Fetch	STALL
Decode	beqz a0, 28
Execute	lw a0, 0(0x1000)
Memory	mv a1, 0x1000
Writeback	

Data (0x1000)

1
42
1337
0

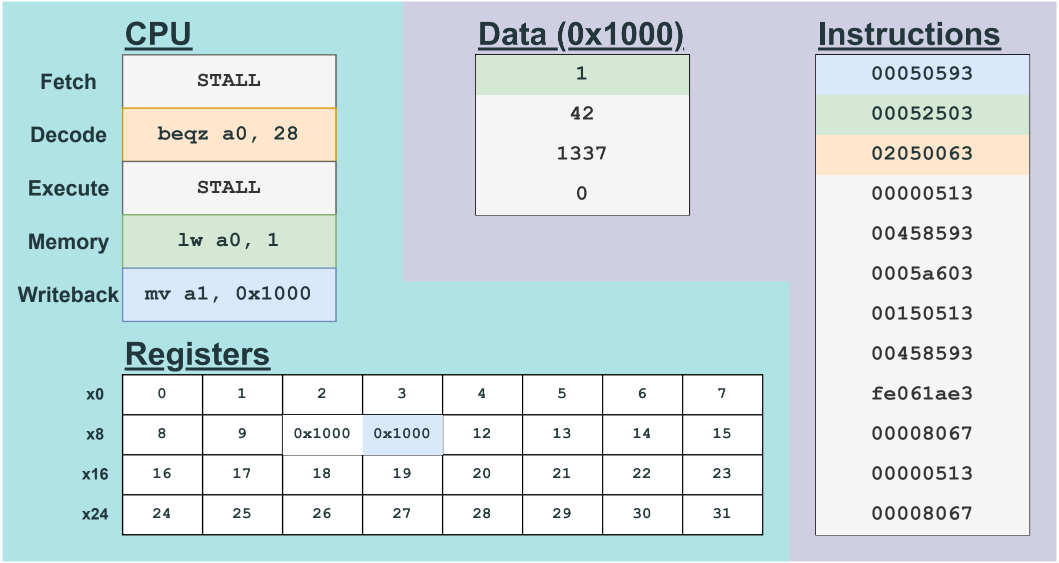
Instructions

00050593
00052503
02050063
00000513
00458593
0005a603
00150513
00458593
fe061ae3
00008067
00000513
00008067

Registers

x0	0	1	2	3	4	5	6	7
x8	8	9	0x1000	11	12	13	14	15
x16	16	17	18	19	20	21	22	23
x24	24	25	26	27	28	29	30	31

Anatomy of a computer: Pipelining Example



CPU

Fetch	STALL
Decode	beqz a0, 28
Execute	STALL
Memory	lw a0, 1
Writeback	mv a1, 0x1000

Data (0x1000)

1
42
1337
0

Instructions

```
00050593
00052503
02050063
00000513
00458593
0005a603
00150513
00458593
fe061ae3
00008067
00000513
00008067
```

Registers

x0	0	1	2	3	4	5	6	7
x8	8	9	0x1000	0x1000	12	13	14	15
x16	16	17	18	19	20	21	22	23
x24	24	25	26	27	28	29	30	31

Anatomy of a computer: Pipelining Example

CPU

Fetch	00000513
Decode	STALL
Execute	beqz 1, 28
Memory	STALL
Writeback	lw a0, 1

Data (0x1000)

1
42
1337
0

Instructions

00050593
00052503
02050063
00000513
00458593
0005a603
00150513
00458593
fe061ae3
00008067
00000513
00008067

Registers

x0	0	1	2	3	4	5	6	7
x8	8	9	1	0x1000	12	13	14	15
x16	16	17	18	19	20	21	22	23
x24	24	25	26	27	28	29	30	31

Anatomy of a computer: Check on Learning

```
      00:   00050593            mv   a1, a0
      04:   00052503            lw   a0, 0(a0)
      08:   02050063            beqz a0, 16 <.LBB0_4>
      0c:   00000513            li   a0, 0
      10:   00458593            addi a1, a1, 4
.LBB0_2:
      14:   0005a603            lw   a2, 0(a1)
      18:   00150513            addi a0, a0, 1
      1c:   00458593            addi a1, a1, 4
      20:   fe061ae3            bnez a2, -6 <.LBB0_2>
      24:   00008067            ret
.LBB0_4:
      28:   00000513            li  a0, 0
      2c:   00008067            ret
```

Instruction Sets

Instruction Set Architecture (ISA): The abstract model of a computer. It encompasses the instructions, registers, memory, and I/O interfaces.

Machine Code: The machine readable representation of executable instructions.

Assembly Language: A human-friendly representation of machine code.

Instruction Sets: RISC & CISC

Reduced Instruction Set Computers (RISC): each instruction performs a single operation

Complex Instruction Set Computers (CISC): some instructions combine two or more operations to improve code density

Note: Reality is more complicated than this.

Instruction Sets: Virtual Machines

ISAs may not target “real” hardware and may target a software-defined runtime (e.g., JVM or eBPF).

The compiled code is often referred to as *bytecode*.

Sometimes called “managed” code (i.e., CLR on Windows).

Instruction Sets: Writing Assembly

inline.c

```c
bool checked_add(int64_t *result,
                 int64_t  left,
                 int64_t  right) {
    int overflow = 0;
    asm volatile (
        "add %0, %2;\n"
        "adox %1, %1;\n"
        : "+r" (left), "+r" (overflow)
        : "r" (right));
    *result = left;
    return overflow;
}
```

separate.S

```asm
.global checked_add
checked_add:
    xor rax, rax
    add rsi, rdx
    adox eax, eax
    mov [rdi], rsi
    ret
```

Instruction Sets: Documentation

- Intel Software Developer Manuals
- AMD64 Architecture Manual
- ARM Documentation
- RISC-V Specification
- JVM Specifications

x86_64 Assembly

The AMD64 ISA will be the focus of the remainder of this class when specifics are needed. Other ISAs differ in important details but most of the principles are transferable.

x86_64 Assembly: Move

Registers: `mov rbp, rsp`

Immediates: `mov rax, ffh`

Memory:

- `push rbp`
- `pop rbp`
- `mov [rdi], rdx`
- `mov rax, [rbp - 8]`
- `mov rax, [rdi + rsi*8]`
- `mov rax, [rdi + rsi*8 + 24]`

x86_64 Assembly: Arithmetic

Generic form: `OP DST, SRC` $\implies dst = op(dst, src)$

Special considerations:

- `lock` prefix required for atomic update of a memory address
- `lea rax, [rdi + rsi*8 + 24]` $\implies rax = rdi + 8 * rsi + 24$
- *EFLAGS* register has special flags that get updated.

x86_64 Assembly: Control flow

Relative control flow changes:

- `jmp 8`
- `call 88h`
- `jnz -8`

Absolute control flow changes:

- `jmp rax`
- `call rax`
- `call [rdx]`
- `ret`

x86_64 Assembly: Arrays & Structs

Arrays:

```
int array[8];
array[x] = 42;
// mov [rdi + rsi*4], 42
//     array + x * sizeof(int)
```

Structs:

```
struct {
    uint16_t a;
    uint8_t b;
    uint8_t c;
} s;
s.c = 42;
// mov [rdi + 3], 42
//      &s + offsetof(s, c)
```

Exercise 1.1

Reading/writing assembly (demo).

Anatomy of a function

- Prologue
- Epilogue
- Calling conventions
  - Args
  - Return
  - Caller-saved
  - Callee-saved
  - *Alignment*

Anatomy of a function: Prologue

```
; Create the stack frame (present most of the time)
push rbp
mov  rbp, rsp
sub  rsp, 64
; Save Callee-saved variables
mov  [rbp - 40], rbx
mov  [rbp - 48], r12
; Debugging support
mov  [rbp -  8], rdi
mov  [rbp - 16], rsi
mov  [rbp - 24], rdx
mov  [rbp - 32], rcx
```

Anatomy of a function: Epilogue

```
mov r12, [rbp - 48]
mov rbx, [rbp - 40]
leave
; leave is equivalent to `mov rsp, rbp; pop rbp'`
ret
```

Anatomy of a function: Calling Convention: Windows

Arguments:

- `rcx`
- `rdx`
- `r8`
- `r9`
- 32-bytes reserved stack
- Stack “right-to-left”

Return:

- `rax`

Volatile:

- All arguments
- `rax`
- `r10`
- `r11`

Anatomy of a function: Calling Convention: AMD64

Arguments:

- `rdi`
- `rsi`
- `rdx`
- `rcx`
- `r8`
- `r9`
- Stack “right-to-left”

Return:

- `rax`
- `rdx`

Volatile:

- All arguments
- `rax`
- `r10`
- `r11`

Exercise 1.2

1. Write `fibonacci` in recursive assembly ($f(n) = f(n-1) + f(n-2)$).
2. Write a function that prints the address after the *call-site* (calls printf).
3. Write a function that prints the value of all registers at the call-site.
4. Write a function that prints as many return addresses in the call-stack as it can. (It should not segfault)

See `checker12.c` in `ex12.tar.gz` for a simple test harness.

Traps, Interrupts, & Exceptions

Trap: deliberate transition into the kernel

Interrupt: external events (timers, I/O, etc.)

Exception: error occurred

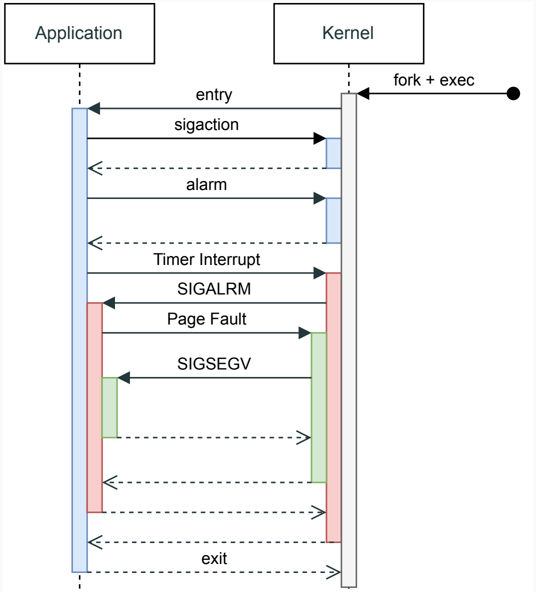
Signal: kernel concept

Traps, Interrupts, & Exceptions: Alarm: Code

```c
void alarm_handler(int unused) {
    int *p_bad = (int *)0x1;
    *p_bad = 42;
}

void segfault_handler(int unused, siginfo_t *info, ucontext_t *uctx) {
    /* Platform-specific code to change address of `p_bad` */
}
```

Traps, Interrupts, & Exceptions: Alarm: Diagram



Exercise 1.3

Writing syscalls directly:

1. Give code that opens "/etc/passwd", reads it in, and writes it to STDOUT
2. Exec a program to print the flag
3. Fork and execute a program in the background (e.g. `socat`).

Privilege Levels

- Rings / kernel mode
- Why are they necessary/helpful?
- What do they protect? (I/O & memory)

Next Time

Speaking of memory. . .