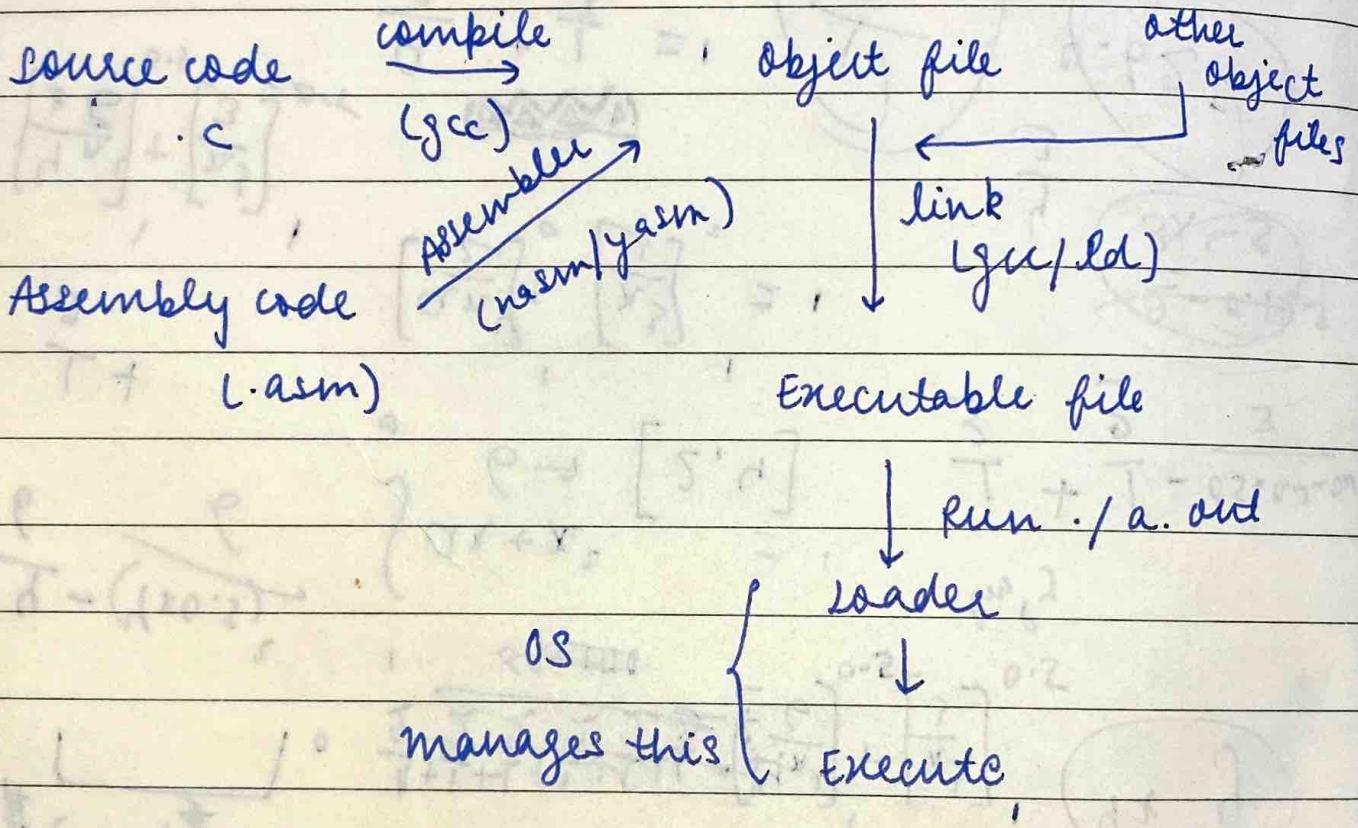
Assembly - Ray Safarth's

Concepts of OS - 3 easy pieces

Utiliza" of OS / System calls Linux Programming API

Linux Kernel programming Robert Love



Wednesday 31.08.21

```
segment .data
a  dq 175
b  dq 4097
segment .text
global main
main: mov rax, [a]
      add rax, [b]
      xor rax, rax
      ret
```

(segment.data → assembler directive; segment.text, global main → directive)

Assembly source code

1) Assembler directive
2) Opcodes
3) Labels
   ↳ memory locations that refer to some part of the code. define within segment.text

dq → quad word (8 bytes)

64 bit processor → highest

- Inline f^n's → copy the code instead of jumping it

xor rax, rax (return 0)

64-bit registers	16-bit registers	32-bit registers
rax	ax	eax
rbx	bx	ebx
⋮	⋮	⋮

Basic Registers

ax, eax, rax

↳ same physical register

Register naming mech in x86-64 processors

PSW → special purpose register that cannot be set by a direct instruction

Program status word sets some flags based on the last program operation

Addressing modes

[MOV] → Intel machines use only this for all addressing modes

↳ make more complex

Complex I.S.

MVI → Move Immediate

MOV rax, a → Direct addressing

[a] → Indirect addressing

In ARM, LD needed for this (Interpret the variable's data as an address and bring the data stored in the address into the register)

MOV a, rax

MOV [rax], a → ST in ARM

opcode doesn't depend on operand ⇒ orthogonal
↳ Intel make assembly prog. simpler
→ Processor architecture & assembler both become more complex

1) Vector operations
→ Specialized inst^n s.t. multiple operations of the same type can be done using a single inst^n.
additional for multimedia & ml workload

2) Floating point
→ Registers & inst^n are different
no orthogonality wrt data types

3) ~~Embedded~~ Embed assembly programs within C language without making it a distinct f^n

Monday 5.09.22

Stack ⇓ → Local variables

malloc, free Heap ⇑ → Dynamic

data (global variables)

text (Labels)

Stack needs to be aligned

↳ support hardware constraints

rsp divisible by 16

yasm -f elf64 print.asm

Calling convention.

1st - rdi
2nd - rsi
3 - rdx
4 - rcx
5 - r8
6 - r9
7 - Stack → rsp - 8

rax → return values (no. of floating point arg.)

xmm0, ... xmm7
1st 8th

tell the assembler that not in file itself but some other library / file

"Hello world!", 0x0a, 0

extern printf (point, array)

load eff. address

lea rdi, [Str]

mov rax, 0

call printf

constants: equ / pseudo ops

times

dw a times 30 0

↳ define array

ld -o print print.asm

Str: "Hello world %d", 0x0a, 0

lea rdi, [Str]
mov rsi, 5
mov rax, 0
call printf

java → java native interface

cpython

user program → python function → C/C++ fn

↓

interface (wrapper fn)

system call → call to the O.S.

↳ not treated as ordinary functions

32 bit

↳ Store parameter for system call in eax register and then use int 0x80

read, write are system calls

$x = x + 100$

↳ wrapper add f^n in python which itself calls a C/C++ f^n

but we

just use call & wrapper f^n

Ld → link w basic C libraries called glibc

Memory leak

↳ responsibility of programmer

valgrind 2 etc

↳ run time

Ubuntu has bash shell default

there are multiple shells - csh, ksh etc, sh
↳ bourne shell

Terminal is used to interact

7.09.2022

Registers rsi, rdi, ... used to send integer args.

Return value rax

(two) volatile — value stored by the calling f^n; can be changed by the called f^n; calling f^n must maintain its data.

Non-volatile (three) → responsibility of saving data on called function

0-34 char

35-39 padding → adding add. memory which helps in alignment

40-47 int

x86-64 processors support backward compatibility (16, 32 bit mode) ↓ during boot process

modes of execution → depends on the value stored in control register (cr0)

OS initially loaded by boot loader
↳ keep instⁿ in locations in RAM

modes in assembly programs -

Privilege level determines this, stored in clo register.

Dynamic memory allocaⁿ in assembly
malloc fⁿ

12.09.22 Monday

Embedding assembly in C

(1) Interact directly w hardware

(2) Performance optimization (relevant only for rare hardware)

Two ways of embedding assembly

1) Basic asm
2) Extended asm

```
__asm__("movq $3, %rax");
```

1) opcodes differ based on type of operands (movq, movl, addq, addl)
2) Sequence of operands is reversed, destination is at the end

gcc → gnu/gas (assembler)

Basic assembly is something where you are not specifying anything except assembly code.

↓ avoid using it cuz compiler is not aware of what you're exactly doing

```
int x = 2;
__asm__("movq $3, %rax");
printf("%d", x);
```

[ ] → optional

Extended assembly

__asm__ [volatile] [goto]

(Assembler Template,
: [output operands]
: [Input operands]
: [clobbers]
: [goto labels]) ;
temp variables

① volatile → Instruction reordering helps optimize execution and is used by compilers.

↓ The compiler should not reorder this (embedded assembly) inst^n if volatile is specified.

② goto → Specify to the compiler that there is a jump instruction

③ Assembler template → Specify the operation, but w/o using the registers

add rax 10

add $10, %rax

X

"addl %1, %2"

④ output operand → "r"(a) "m"(a)
↓ register ↓ memory

write to register → "="
"=r" or "=m"
valid

Both read and write "+" modifier

⑤ Input operands → same as output operands

```
int main(void) {
    int sum, x, y;
    x = 5;
    y = 10;                    // sum = x+y;

__asm__("addq %%1, %%2" : "=r"(sum)
                        : "r"(a), "0"(b));
printf("%d", sum);
```

1 → a and b map into same reg

"0": Is a constraint which asks the compiler to map the 2 variables into the same register.

Exercise: → Copy the content of one variable into another

"addq %1 %2 \t\n
movq %2 %3"

2 or more instructions can be given

⑥ Clobbers : Specify to the assembler that a specific list of memory locations or registers would be changed in assembly.

%rax, %rbx

⑦ Goto labels : Specify the list of labels to which you might jump from assembly.

Absolute core of the O.S. ← has shell, text editor etc

Linux → A specific kernel used by most systems today. Android, Raspberry Pi, etc (except Windows, mac) Artix, Ubuntu

Unix → Traditional OS since 1969

etc AT&T System V, Berkely UNIX, IBM → AIX

Linux was designed to be compatible with other unix systems.

Standardization system → basic options / facilities that all the OS should provide. This Standard is called POSIX.

~~Every System~~

Linux fully posix compliant

Windows is not.

MacOSx → POSIX compliant

Linux kernel is open-source. (download & change things)

Artix, Ubuntu, Fedora ... etc ?

→ Distributions of linux } ⇒ They collect the source code of kernel & system utilities, build them & give them to the users in a more usable form

Gnu: Organization whose aim was to design open-source software so that people could use them more conveniently.

Compiler, GUI tools, text editors etc. were not able to build proper kernel

Linux was used together with GNU tools to make one complete OS.

glibc GNU/Linux systems.

OS is more than the kernel.

Android → created all system tools own GUI, ~~Andr Library en~~ virtual envi called Android runtime, only linux kernel is still same

Installing	Building
just copying the libraries that have been built possibly on some other machine	copying and linking the software

Building software → cross-compilation

Lec 2

64		16	32
rax	→ accumulator	ax	eax
rbx	→ base register	bx	ebx
rcx	→ count register	cx	ecx
rdx	→ data register	dx	edx
rsi	→ source index	si	esi
rdi	→ destination index	di	edi
rbp	→ base pointer	bp	ebp
rsp	→ stack pointer	sp	esp
r8			
r9			
⋮			
r15			

ax, eax & rax are same physical register