

x86-64 (2)

# LEA

like a `mov` — but stop at finding the memory address

**never** accesses memory

`lea (%rax), %rbx` is `mov %rax, %rbx`

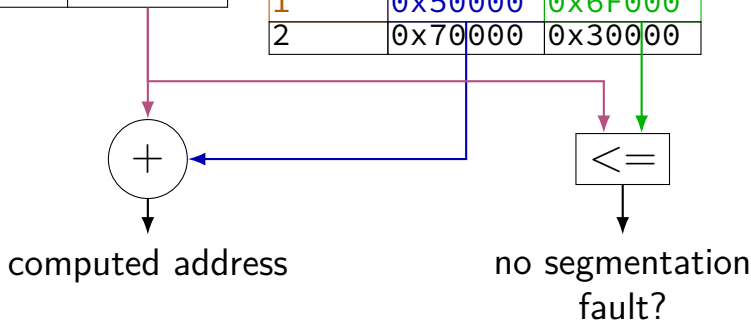
# segmentation

before virtual memory, there was **segmentation**

address

segment #:	offset:
0x1	0x23456

seg #	base	limit
0	0x14300	0x60000
1	0x50000	0x6F000
2	0x70000	0x30000



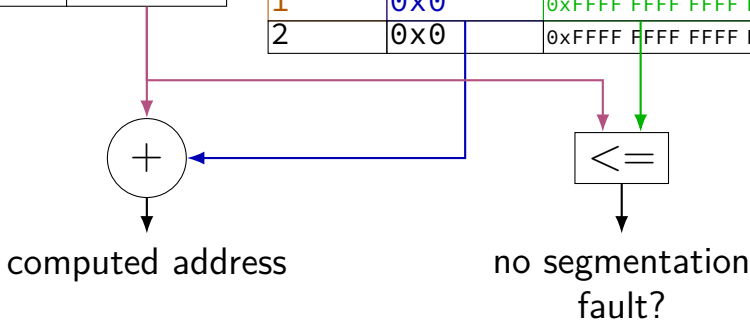
# segmentation

before virtual memory, there was **segmentation**

address

segment #:	offset:
0x1	0x23456

seg #	base	limit
0	0x0	0xFFFF FFFF FFFF FFFF
1	0x0	0xFFFF FFFF FFFF FFFF
2	0x0	0xFFFF FFFF FFFF FFFF



# x86 segmentation

addresses you've seen are the **offsets**

but every access uses a segment number!

segment numbers come from registers

- CS — code segment number (jump, call, etc.)

- SS — stack segment number (push, pop, etc.)

- DS — data segment number (mov, add, etc.)

- ES — addt'l data segment (string instructions)

- FS, GS — extra segments (never default)

instructions can have a **segment override**:

```
movq $42, %fs:100(%rsi)
```

```
    // move 42 to segment (# in FS),
```

```
    // offset 100 + RSI
```

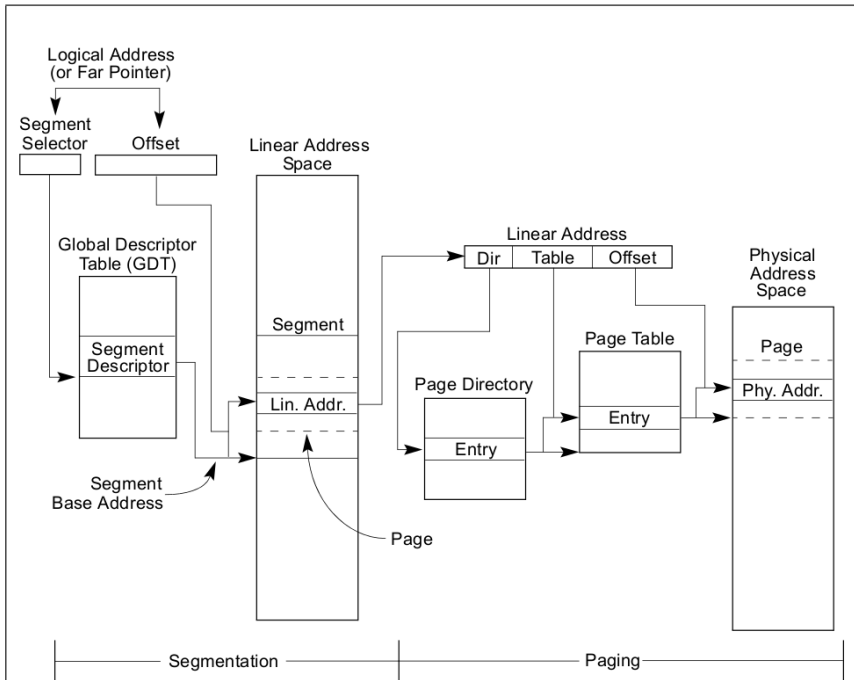


Figure 3.1 Segmentation and Paging

# program address

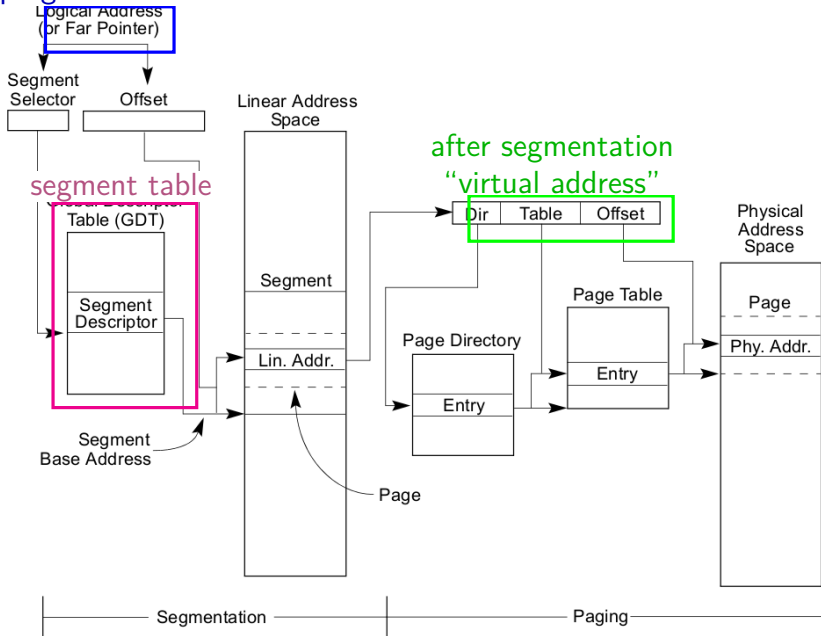


Figure 3.1 Segmentation and Paging

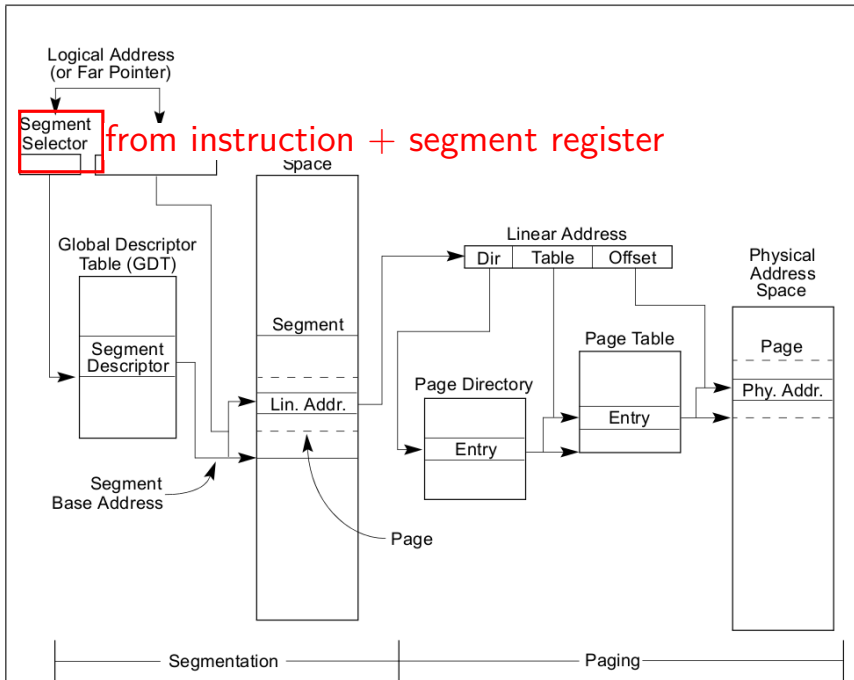
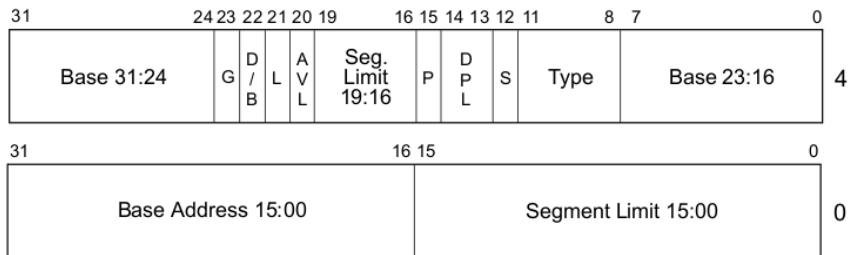


Figure 2.1 Segmentation and Paging

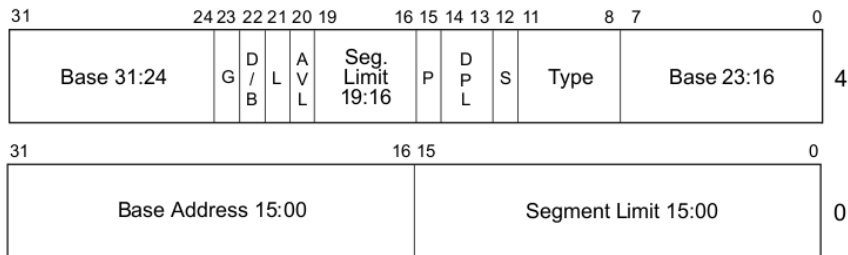


# x86 segment descriptor



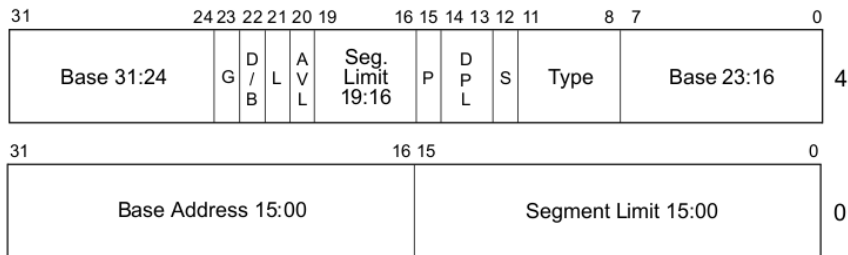
- L — 64-bit code segment (IA-32e mode only)
- AVL — Available for use by system software
- BASE — Segment base address
- D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
- DPL — Descriptor privilege level
- G — Granularity
- LIMIT — Segment Limit
- P — Segment present
- S — Descriptor type (0 = system; 1 = code or data)
- TYPE — Segment type

# x86 segment descriptor



- L — 64-bit code segment (IA-32e mode only)
- AVL — Available for use by system software
- BASE — Segment base address
- D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
- DPL — Descriptor privilege level** user or kernel mode? (if code)
- G — Granularity
- LIMIT — Segment Limit
- P — Segment present
- S — Descriptor type (0 = system; 1 = code or data)
- TYPE — Segment type

# x86 segment descriptor



**L** — 64-bit code segment (IA-32e mode only)

**AVL** — Available for use by system software

**BASE** — Segment base address

**D/B** — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)

**DPL** — Descriptor privilege level

**64-bit or 32-bit or 16-bit mode? (if code)**

**LIMIT** — Segment Limit

**P** — Segment present

**S** — Descriptor type (0 = system; 1 = code or data)

**TYPE** — Segment type

# 64-bit segmentation

in 64-bit mode:

limits are ignored

base addresses are ignored

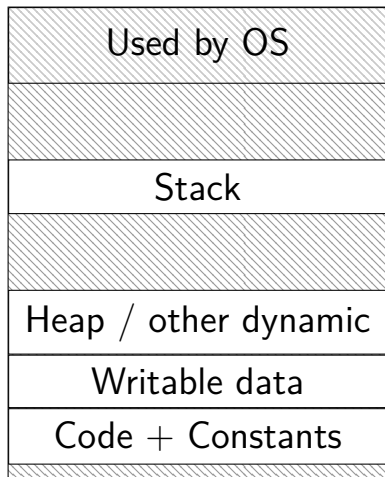
...except for %fs, %gs

when explicit segment override is used

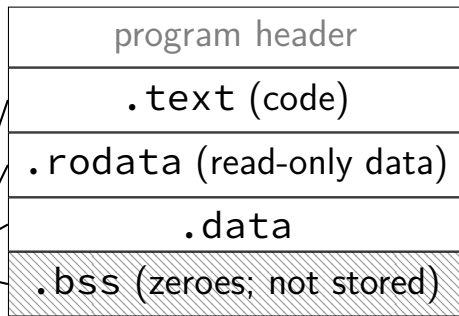
effectively: extra pointer register

# memory v. disk

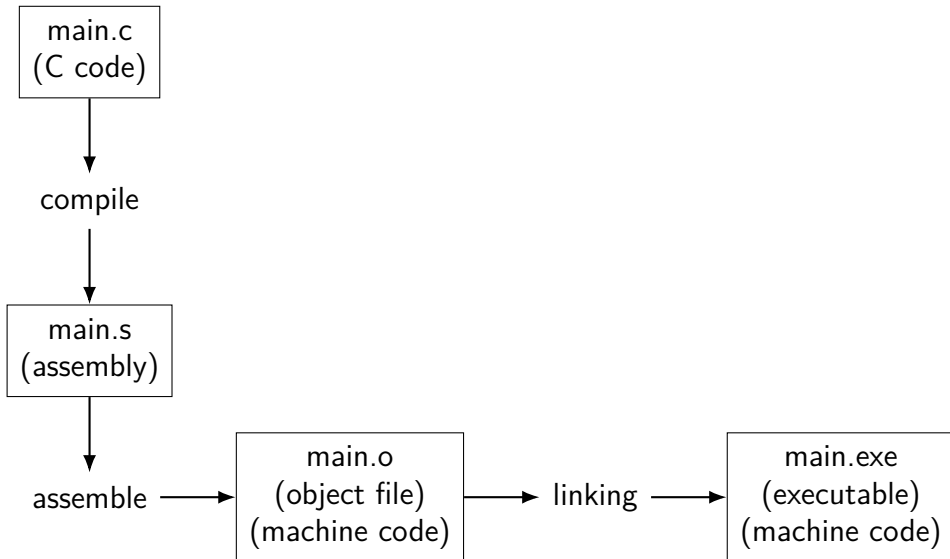
(virtual) memory



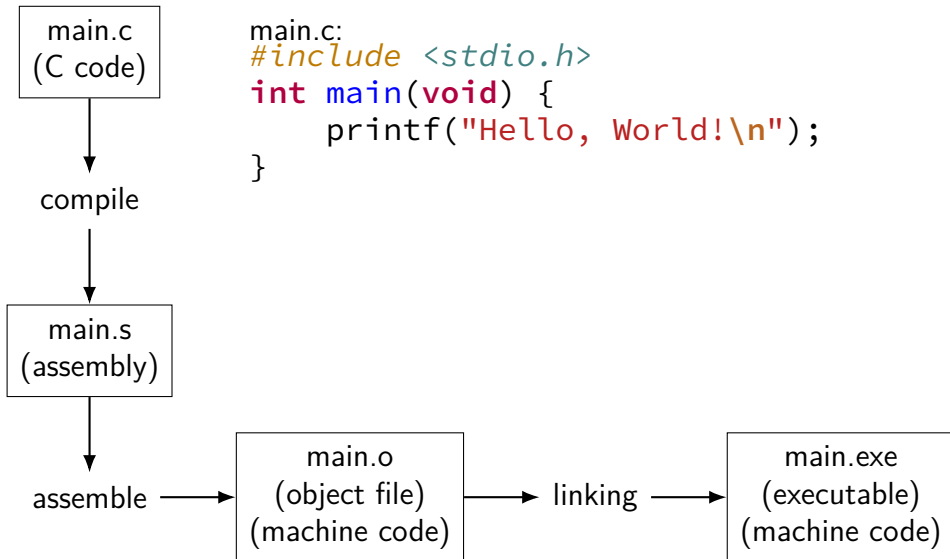
program on disk



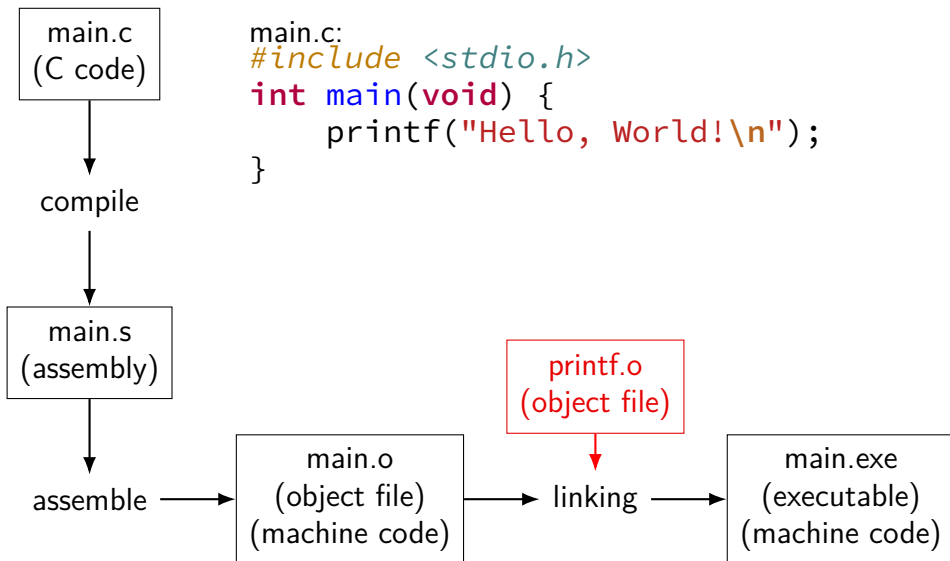
# compilation pipeline



# compilation pipeline



# compilation pipeline





# compilation commands

<b>compile:</b>	<code>gcc -S file.c</code>	$\Rightarrow$	<code>file.s</code> (assembly)
<b>assemble:</b>	<code>gcc -c file.s</code>	$\Rightarrow$	<code>file.o</code> (object file)
<b>link:</b>	<code>gcc -o file file.o</code>	$\Rightarrow$	<code>file</code> (executable)
<code>c+a:</code>	<code>gcc -c file.c</code>	$\Rightarrow$	<code>file.o</code>
<code>c+a+l:</code>	<code>gcc -o file file.c</code>	$\Rightarrow$	<code>file</code>
<code>...</code>			

# what's in those files?


hello.c

```
#include <stdio.h>
int main(void) {
    puts("Hello, World!");
    return 0;
}
```

# what's in those files?

hello.c

```
#include <stdio.h>
int main(void) {
    puts("Hello, World!");
    return 0;
}
```



hello.s

```
.text
.global main
main:
    sub    $8, %rsp
    mov    $.Lstr, %rdi
    call   puts
    xor    %eax, %eax
    add    $8, %rsp
    ret

.data
.Lstr: .string "Hello, World!"
```

# what's in those files?

hello.c

```
#include <stdio.h>
int main(void) {
    puts("Hello, World!");
    return 0;
}
```

hello.s

```
.text
.global main
main:
    sub    $8, %rsp
    mov    $.Lstr, %rdi
    call   puts
    xor     %eax, %eax
    add    $8, %rsp
    ret

.data
.Lstr: .string "Hello, World!"
```

hello.s (Intel syntax)

```
.text
main:
    sub    RSP, 8
    mov    RDI, .Lstr
    call   puts
    xor     EAX, EAX
    add    RSP, 8
    ret

.data
.Lstr: .string "Hello, World!"
```

# what's in those files?

hello.c

```
#include <stdio.h>
int main(void) {
    puts("Hello, World!");
    return 0;
}
```

hello.s

```
.text
.global main
main:
    sub    $8, %rsp
    mov    $.Lstr, %rdi
    call   puts
    xor    %eax, %eax
    add    $8, %rsp
    ret

.data
.Lstr: .string "Hello, World!"
```

Linux x86-64  
calling convention:  
stack addr. must be  
multiple of 16

# what's in those files?

hello.c

```
#include <stdio.h>
int main(void) {
    puts("Hello, World!");
    return 0;
}
```

hello.s

```
.text
.global main
main:
    sub    $8, %rsp
    mov    $.Lstr, %rdi
    call   puts
    xor     %eax, %eax
    add    $8, %rsp
    ret

.data
.Lstr: .string "Hello, World!"
```

sets eax to 0  
(shorter machine  
code than mov)

# what's in those files?

hello.c

```
#include <stdio.h>
int main(void) {
    puts("Hello, World!");
    return 0;
}
```

hello.s

```
.text
.global main
main:
    sub    $8, %rsp
    mov    $.Lstr, %rdi
    call   puts
    xor     %eax, %eax
    add     $8, %rsp
    ret

.data
.Lstr: .string "Hello, World!"
```

mark used by other files

# what's in those files?

hello.c

```
#include <stdio.h>
int main(void) {
    puts("Hello, World!");
    return 0;
}
```

hello.s

```
.text
.global main
main:
    sub    $8, %rsp
    mov    $.Lstr, %rdi
    call   puts
    xor     %eax, %eax
    add    $8, %rsp
    ret

.data
.Lstr: .string "Hello, World!"
```

hello.o

**text** (code) segment:

```
48 83 EC 08 BF 00 00 00 00 E8 00 00
00 00 31 C0 48 83 C4 08 C3
```



# what's in those files?

hello.c

```
#include <stdio.h>
int main(void) {
    puts("Hello, World!");
    return 0;
}
```

hello.s

```
.text
.global main
main:
    sub    $8, %rsp
    mov    $.Lstr, %rdi
    call   puts
    xor     %eax, %eax
    add    $8, %rsp
    ret

.data
.Lstr: .string "Hello, World!"
```

hello.o

**text** (code) segment:

```
48 83 EC 08 BF 00 00 00 00 E8 00 00
00 00 31 C0 48 83 C4 08 C3
```

**data** segment:

```
48 65 6C 6C 6F 2C 20 57 6F 72 6C 00
```

# what's in those files?

hello.c

```
#include <stdio.h>
int main(void) {
    puts("Hello, World!");
    return 0;
}
```

hello.s

```
.text
.global main
main:
    sub $8, %rsp
    mov $.Lstr, %rdi
    call puts
    xor %eax, %eax
    add $8, %rsp
    ret

.data
.Lstr: .string "Hello, World!"
```

hello.o

**text** (code) segment:

```
48 83 EC 08 BF 00 00 00 00 E8 00 00
00 00 31 C0 48 83 C4 08 C3
```

**data** segment:

```
48 65 6C 6C 6F 2C 20 57 6F 72 6C 00
```

# what's in those files?

hello.c

```
#include <stdio.h>
int main(void) {
    puts("Hello, World!");
    return 0;
}
```

hello.s

```
.text
.global main
main:
    sub $8, %rsp
    mov $.Lstr, %rdi
    call puts
    xor %eax, %eax
    add $8, %rsp
    ret
```

hello.o

**text (code) segment:**

```
48 83 EC 08 BF 00 00 00 00 E8 00 00
00 00 31 C0 48 83 C4 08 C3
```

**data segment:**

```
48 65 6C 6C 6F 2C 20 57 6F 72 6C 00
```

**relocations:**

take 0s at            and replace with  
text, byte 5 (|)      data segment, byte 0  
text, byte 10 (|)     address of puts

```
.data
.Lstr: .string "Hello, World!"
```

# what's in those files?

hello.c

```
#include <stdio.h>
int main(void) {
    puts("Hello, World!");
    return 0;
}
```

hello.s

```
.text
.global main
main:
    sub $8, %rsp
    mov $.Lstr, %rdi
    call puts
    xor %eax, %eax
    add $8, %rsp
    ret
```

hello.o

**text (code) segment:**

```
48 83 EC 08 BF 00 00 00 00 E8 00 00
00 00 31 C0 48 83 C4 08 C3
```

**data segment:**

```
48 65 6C 6C 6F 2C 20 57 6F 72 6C 00
```

**relocations:**

take 0s at                      and replace with

text, byte 5 ( )	data segment, byte 0
text, byte 10 ( )	address of puts

**symbol table:**

main    text byte 0

```
.data
.Lstr: .string "Hello, World!"
```

# what's in those files?

hello.c

```
#include <stdio.h>
int main(void) {
    puts("Hello, World!");
    return 0;
}
```

hello.s

```
.text
.global main
main:
    sub $8, %rsp
    mov $.Lstr, %rdi
    call puts
    xor %eax, %eax
    add $8, %rsp
    ret
```

hello.o

**text (code) segment:**

```
48 83 EC 08 BF 00 00 00 00 E8 00 00
00 00 31 C0 48 83 C4 08 C3
```

**data segment:**

```
48 65 6C 6C 6F 2C 20 57 6F 72 6C 00
```

**relocations:**

take 0s at	and replace with
text, byte 5 ( )	data segment, byte 0
text, byte 10 ( )	address of puts

**symbol table:**

```
main    text byte 0
```

```
.data
.Lstr: .string "Hello, World!"
```

.Lstr location specified w/o name  
and not usable by other files  
so no symbol table entry needed

(convention: .L...labels always local)

# what's in those files?

hello.c

```
#include <stdio.h>
int main(void) {
    puts("Hello, World!");
    return 0;
}
```

hello.s

```
.text
.global main
main:
    sub $8, %rsp
    mov $.Lstr, %rdi
    call puts
    xor %eax, %eax
    add $8, %rsp
    ret

.data
.Lstr: .string "Hello, World!"
```

hello.o

**text (code) segment:**

48 83 EC 08 BF 00 00 00 00 E8 00 00  
00 00 31 C0 48 83 C4 08 C3

**data segment:**

48 65 6C 6C 6F 2C 20 57 6F 72 6C 00

**relocations:**

take 0s at            and replace with  
text, byte 5 (|)      data segment, byte 0  
text, byte 10 (|)     address of puts

**symbol table:**

main    text byte 0

+ stdio.o

hello.exe

(actually binary, but shown as hexadecimal) ...

48 83 EC 08 BF A7 02 04 00  
E8 08 4A 00 00 31 C0 48  
83 C4 08 C3 ...  
...(code from stdio.o) ...  
48 65 6C 6C 6F 2C 20 57 6F  
72 6C 00 ...  
...(data from stdio.o) ...

# hello.s

```
.LC0:      .section          .rodata.str1.1,"aMS",@progb+
           .string "Hello, World!"
           .text
           .globl  main

main:
           subq    $8, %rsp
           movl    $.LC0, %edi
           call    puts
           movl    $0, %eax
           addq    $8, %rsp
           ret
```

# exercise (1)

main.c:

```
1  #include <stdio.h>
2  void sayHello(void) {
3      puts("Hello, World!");
4  }
5  int main(void) {
6      sayHello();
7  }
```

Which files contain the **memory address** of sayHello?

- |                          |                   |
|--------------------------|-------------------|
| A. main.s (assembly)     | D. B and C        |
| B. main.o (object)       | E. A, B and C     |
| C. main.exe (executable) | F. something else |



## exercise (2)

main.c:

```
1  #include <stdio.h>
2  void sayHello(void) {
3      puts("Hello, World!");
4  }
5  int main(void) {
6      sayHello();
7  }
```

Which files contain the **literal ASCII string** of Hello, World!?

- |                          |                   |
|--------------------------|-------------------|
| A. main.s (assembly)     | D. B and C        |
| B. main.o (object)       | E. A, B and C     |
| C. main.exe (executable) | F. something else |

# dynamic linking (very briefly)

*dynamic linking* — done **when application is loaded**

idea: don't have  $N$  copies of `printf` on disk

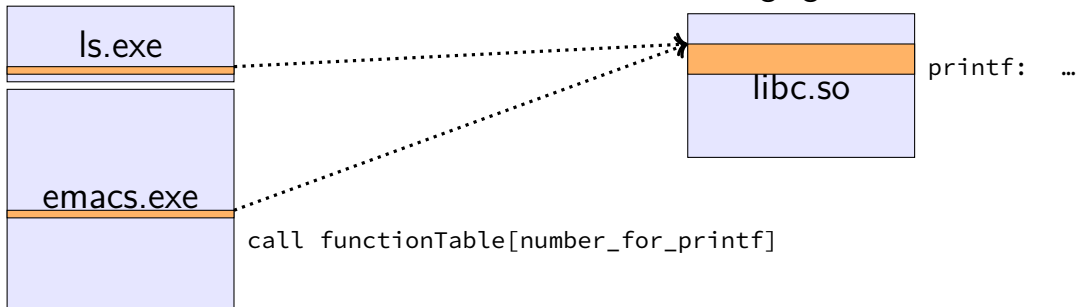
other type of linking: *static* (`gcc -static`)

load executable file + its libraries into memory when app starts

often extra indirection:

`call functionTable[number_for_printf]`

linker fills in `functionTable` instead of changing calls



# ldd /bin/ls

```
$ ldd /bin/ls
linux-vdso.so.1 => (0x00007ffcca9d8000)
libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1
(0x00007f851756f000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
(0x00007f85171a5000)
libpcre.so.3 => /lib/x86_64-linux-gnu/libpcre.so.3
(0x00007f8516f35000)
libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2
(0x00007f8516d31000)
/lib64/ld-linux-x86-64.so.2 (0x00007f8517791000)
libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0
(0x00007f8516b14000)
```

# relocation types

machine code doesn't always use addresses as is

“call function 4303 bytes later”

linker needs to compute “4303”

extra field on relocation list

# ELF (executable and linking format)

Linux (and some others) executable/object file format

<b>header:</b> machine type, file type, etc.
<b>program header:</b> “ <b>segments</b> ” to load (also, some other information)
<b>segment 1 data</b>
<b>segment 2 data</b>
<b>section header:</b> list of “ <b>sections</b> ”(mostly for linker)

# segments versus sections?

note: ELF terminology; may not be true elsewhere!

sections — **object files** (and usually executables), used by **linker**

- have information on intended purpose

- linkers combine these to create executables

- linkers might omit unneeded sections

segments — executables, used to actually load program

- program loader is **dumb** — doesn't know what segments are for

# ELF example

`objdump -x /bin/busybox` (on my laptop)

`-x`: output all headers

`/bin/busybox:` file format `elf64-x86-64`

`/bin/busybox`

architecture: `i386:x86-64`, flags `0x00000102`:

`EXEC_P, D_PAGED`

start address `0x000000000000401750`

Program Header:

[...]

Sections:

[...]

# ELF example

`objdump -x /bin/busybox` (on my laptop)

`-x`: output all headers

```
/bin/busybox:      file format elf64-x86-64
/bin/busybox
architecture: i386:x86-64, flags 0x000000102:
EXEC_P, D_PAGED
start address 0x000000000000401750
```

Program Header:  
[...]

Sections:  
[...]



# a program header (1)

Program Header:

```
[...]  
LOAD off      0x00000000 vaddr 0x04000000 paddr 0x04000000 align 2**21  
      filesz 0x01db697 memsz 0x01db697 flags r-x  
LOAD off      0x01dbea8 vaddr 0x07dbea8 paddr 0x07dbea8 align 2**21  
      filesz 0x00021ee memsz 0x0007d18 flags rw-  
[...]
```

load 0x1db697 bytes:

from 0x0 bytes into the file  
to memory at 0x40000  
readable and executable

load 0x21ee bytes:

from 0x1dbea8  
to memory at 0x7dbea8  
plus (0x7d18-0x21ee) bytes of zeroes  
readable and writable

# a program header (1)

Program Header:

```
[...]
LOAD off      0x00000000 vaddr 0x04000000 paddr 0x04000000 align 2**21
      filesz 0x01db697 memsz 0x01db697 flags r-x
LOAD off      0x01dbea8 vaddr 0x07dbea8 paddr 0x07dbea8 align 2**21
      filesz 0x00021ee memsz 0x0007d18 flags rw-
[...]
```

load **0x1db697** bytes:

from 0x0 bytes into the file  
to memory at 0x40000  
readable and executable

load 0x21ee bytes:

from 0x1dbea8  
to memory at 0x7dbea8  
plus (0x7d18-0x21ee) bytes of zeroes  
readable and writable

# a program header (1)

Program Header:

```
[...]  
LOAD off      0x00000000 vaddr 0x04000000 paddr 0x04000000 align 2**21  
      filesz 0x01db697 memsz 0x01db697 flags r-x  
LOAD off      0x01dbea8 vaddr 0x07dbea8 paddr 0x07dbea8 align 2**21  
      filesz 0x00021ee memsz 0x0007d18 flags rw-  
[...]
```

load 0x1db697 bytes:

from 0x0 bytes into the file  
to memory at 0x40000  
readable and executable

load 0x21ee bytes:

from 0x1dbea8  
to memory at 0x7dbea8  
plus (0x7d18-0x21ee) bytes of zeroes  
readable and writable

# a program header (1)

Program Header:

```
[...]
LOAD off      0x00000000 vaddr 0x04000000 paddr 0x04000000 align 2**21
      filesz 0x01db697 memsz 0x01db697 flags r-x
LOAD off      0x01dbea8 vaddr 0x07dbea8 paddr 0x07dbea8 align 2**21
      filesz 0x00021ee memsz 0x0007d18 flags rw-
[...]
```

load 0x1db697 bytes:

from 0x0 bytes into the file  
to memory at 0x40000  
readable and executable

load 0x21ee bytes:

from 0x1dbea8  
to memory at 0x7dbea8  
plus (0x7d18-0x21ee) bytes of zeroes  
readable and writable

## a program header (2)

Program Header:

```
[...]
NOTE off      0x0000190 vaddr 0x0400190 paddr 0x0400190 align 2**2
      filesz 0x0000044 memsz 0x0000044 flags r--
  TLS  off      0x01dbea8 vaddr 0x07dbea8 paddr 0x07dbea8 align 2**3
      filesz 0x0000030 memsz 0x000007a flags r--
STACK off      0x0000000 vaddr 0x0000000 paddr 0x0000000 align 2**4
      filesz 0x0000000 memsz 0x0000000 flags rw-
RELRO off      0x01dbea8 vaddr 0x07dbea8 paddr 0x07dbea8 align 2**0
      filesz 0x0000158 memsz 0x0000158 flags r--
[...]
```

NOTE — comment

TLS — thread-local storage region (used via %fs)

STACK — indicates stack is read/write

RELRO — make this read-only after runtime linking

# section headers

Sections:

Idx	Name	Size	VMA	LMA	File off	Algn
0	.note.ABI-tag	00000020	0000000000400190	0000000000400190	00000190	2**2
	CONTENTS, ALLOC, LOAD, READONLY, DATA					
1	.note.gnu.build-id	00000024	00000000004001b0	00000000004001b0	000001b0	2**2
	CONTENTS, ALLOC, LOAD, READONLY, DATA					
2	.rela.plt	00000210	00000000004001d8	00000000004001d8	000001d8	2**3
	CONTENTS, ALLOC, LOAD, READONLY, DATA					
3	.init	0000001a	00000000004003e8	00000000004003e8	000003e8	2**2
	CONTENTS, ALLOC, LOAD, READONLY, CODE					
4	.plt	00000160	0000000000400410	0000000000400410	00000410	2**4
	CONTENTS, ALLOC, LOAD, READONLY, CODE					
5	.text	0017ff1d	0000000000400570	0000000000400570	00000570	2**4
	CONTENTS, ALLOC, LOAD, READONLY, CODE					
6	__libc_freeres_fn	00002032	0000000000580490	0000000000580490	00180490	2**4
	CONTENTS, ALLOC, LOAD, READONLY, CODE					
7	__libc_thread_freeres_fn	0000021b	00000000005824d0	00000000005824d0	001824d0	2**4
	CONTENTS, ALLOC, LOAD, READONLY, CODE					
8	.fini	00000009	00000000005826ec	00000000005826ec	001826ec	2**2
	CONTENTS, ALLOC, LOAD, READONLY, CODE					
9	.rodata	00044ac8	0000000000582700	0000000000582700	00182700	2**6
	CONTENTS, ALLOC, LOAD, READONLY, DATA					
10	__libc_subfreeres	000000c0	00000000005c71c8	00000000005c71c8	001c71c8	2**3
	CONTENTS, ALLOC, LOAD, READONLY, DATA					
11	.stapsdt.base	00000001	00000000005c7288	00000000005c7288	001c7288	2**0
	CONTENTS, ALLOC, LOAD, READONLY, DATA					
12	__libc_atexit	00000008	00000000005c7290	00000000005c7290	001c7290	2**3
	CONTENTS, ALLOC, LOAD, READONLY, DATA					
13	__libc_thread_subfreeres	00000018	00000000005c7298	00000000005c7298	001c7298	2**3
	CONTENTS, ALLOC, LOAD, READONLY, DATA					
14	.eh_frame	000141dc	00000000005c72b0	00000000005c72b0	001c72b0	2**3
	CONTENTS, ALLOC, LOAD, READONLY, DATA					
15	.gcc_except_table	0000020b	00000000005db48c	00000000005db48c	001db48c	2**0

# sections

tons of “sections”

not actually needed/used to run program

size, file offset, flags (code/data/etc.)

location in executable *and* in memory

some sections aren't stored (no “CONTENTS” flag)  
just all zeroes

# selected sections

.text	program code
.bss	initially zero data (block started by symbol)
.data	other writeable data
.rodata	read-only data
.init/.fini	global constructors/destructors
.got/.plt	linking related
.eh_frame	try/catch related



# other executable formats

PE (Portable Executable) — Windows

Mach-O — MacOS X

broadly similar to ELF

differences:

- whether segment/section distinction exists
- how linking/debugging info represented
- how program start info represented

# simple executable startup

copy segments into memory

jump to start address

# executable startup code

Linux: executables don't start at `main`

why not?


- need to initialize `printf`, `cout`, `malloc`, etc. data structures

- `main` needs to return somewhere

compiler links in startup code

# linking

callq printf



A vertical arrow points from the 'callq printf' box to the 'callq 0x458F0' box, indicating the resolution of the symbol 'printf' to the memory address '0x458F0'.

callq 0x458F0

# static v. dynamic linking

static linking — linking to create executable

dynamic linking — linking when executable is run

# static v. dynamic linking

static linking — linking to create executable

dynamic linking — linking when executable is run

conceptually: no difference in how they work

reality — very different mechanisms

# linking data structures

symbol table: `name`  $\Rightarrow$  (section, offset)

example: `main:` in assembly adds symbol table entry for `main`

relocation table: offset  $\Rightarrow$  (name, kind)

example: `call printf` adds relocation for name `printf`

kind depends on how instruction encodes address

# hello.s

```
.data
string: .asciz "Hello, World!"
.text
.globl main
main:
    movq $string, %rdi
    call puts
    ret
```



# hello.o

## SYMBOL TABLE:

000000000000000000	l	d	.text	000000000000000000	.text
000000000000000000	l	d	.data	000000000000000000	.data
000000000000000000	l	d	.bss	000000000000000000	.bss
000000000000000000	l		.data	000000000000000000	string
000000000000000000	g		.text	000000000000000000	main
000000000000000000			*UND*	000000000000000000	puts

## RELOCATION RECORDS FOR [.text]:

OFFSET	TYPE	VALUE
000000000000000003	R_X86_64_32S	.data
000000000000000008	R_X86_64_PC32	puts-0x0000000000000004

# hello.o

## SYMBOL TABLE:

000000000000000000	l	d	.text	000000000000000000	.text
000000000000000000	l	d	.data	000000000000000000	.data
000000000000000000	l	d	.bss	000000000000000000	.bss
000000000000000000	l		.data	000000000000000000	string
000000000000000000	g		.text	000000000000000000	main
000000000000000000			*UND*	000000000000000000	puts

## RELOCATION RECORDS FOR [ .text ]:

OFFSET	TYPE	VALUE
000000000000000003	R_X86_64_32S	.data
000000000000000008	R_X86_64_PC32	puts-0x0000000000000004

undefined symbol: look for puts elsewhere

# hello.o

## SYMBOL TABLE:

000000000000000000	l	d	.text	000000000000000000	.text
000000000000000000	l	d	.data	000000000000000000	.data
000000000000000000	l	d	.bss	000000000000000000	.bss
000000000000000000	l		.data	000000000000000000	string
000000000000000000	g		.text	000000000000000000	main
000000000000000000			*UND*	000000000000000000	puts

## RELOCATION RECORDS FOR [ .text ]:

OFFSET	TYPE	VALUE
000000000000000003	R_X86_64_32S	.data
000000000000000008	R_X86_64_PC32	puts-0x0000000000000004

insert address of puts, format for call

# hello.o

## SYMBOL TABLE:

000000000000000000	l	d	.text	000000000000000000	.text
000000000000000000	l	d	.data	000000000000000000	.data
000000000000000000	l	d	.bss	000000000000000000	.bss
000000000000000000	l		.data	000000000000000000	string
000000000000000000	g		.text	000000000000000000	main
000000000000000000			*UND*	000000000000000000	puts

## RELOCATION RECORDS FOR [ .text ]:

OFFSET	TYPE	VALUE
000000000000000003	R_X86_64_32S	<b>.data</b>
000000000000000008	R_X86_64_PC32	puts-0x0000000000000004

insert address of string, format for movq

# hello.o

## SYMBOL TABLE:

000000000000000000	l	d	.text	000000000000000000	.text
000000000000000000	l	d	.data	000000000000000000	.data
000000000000000000	l	d	.bss	000000000000000000	.bss
000000000000000000	l		.data	000000000000000000	string
000000000000000000	g		.text	000000000000000000	main
000000000000000000			*UND*	000000000000000000	puts

## RELOCATION RECORDS FOR [ .text ]:

OFFSET	TYPE	VALUE
000000000000000003	R_X86_64_32S	.data
000000000000000008	R_X86_64_PC32	puts-0x0000000000000004

different ways to represent address

32S — signed 32-bit value

PC32 — 32-bit difference from current address

# hello.o

## SYMBOL TABLE:

000000000000000000	l	d	.text	000000000000000000	.text
000000000000000000	l	d	.data	000000000000000000	.data
000000000000000000	l	d	.bss	000000000000000000	.bss
000000000000000000	l		.data	000000000000000000	string
000000000000000000	<b>g</b>		.text	000000000000000000	main
000000000000000000			*UND*	000000000000000000	puts

## RELOCATION RECORDS FOR [ .text ]:

OFFSET	TYPE	VALUE
000000000000000003	R_X86_64_32S	.data
000000000000000008	R_X86_64_PC32	puts-0x0000000000000004

g: global — used by other files  
l: local

# hello.o

## SYMBOL TABLE:

000000000000000000	l	d	.text	000000000000000000	.text
000000000000000000	l	d	.data	000000000000000000	.data
000000000000000000	l	d	.bss	000000000000000000	.bss
000000000000000000	l		.data	000000000000000000	string
000000000000000000	g		.text	000000000000000000	main
000000000000000000		*	UND*	000000000000000000	puts

## RELOCATION RECORDS FOR [ .text ]:

OFFSET	TYPE	VALUE
000000000000000003	R_X86_64_32S	.data
000000000000000008	R_X86_64_PC32	puts-0x0000000000000004

.text segment beginning plus 0 bytes

# interlude: strace

`strace` — system call tracer

- on Linux, some other Unices

- OS X approx. equivalent: `dtruss`

- Windows approx. equivalent: Process Monitor

indicates what system calls (operating system services) used by a program



# statically linked hello.exe

```
gcc -static -o hello-static.exe hello.s
```

```
strace ./hello-static.exe:
```

```
execve("./hello-static.exe", ["./hello-static.exe"], [/* 46 vars */]) = 0
uname(sysname="Linux", nodename="reiss-lenovo", ...) = 0
brk(NULL) = 0x20a5000
brk(0x20a61c0) = 0x20a61c0
arch_prctl(ARCH_SET_FS, 0x20a5880) = 0
readlink("/proc/self/exe", "/home/cr4bd/spring2017/cs4630/sl"... , 4096) =
brk(0x20c71c0) = 0x20c71c0
brk(0x20c8000) = 0x20c8000
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or direc
fstat(1, st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...) = 0
write(1, "Hello, World!\n", 14) = 14
exit_group(14) = ?
+++ exited with 14 +++
```

# statically linked hello.exe

```
gcc -static -o hello-static.exe hello.s
```

```
strace ./hello-static.exe:
```

```
execve("./hello-static.exe", ["/hello-static.exe"], [/* 46 vars */]) = 0
uname(sysname="Linux", nodename="reiss-lenovo", ...) = 0
brk(NULL) = 0x20a5000
brk(0x20a61c0) = 0x20a61c0
arch_prctl(ARCH_SET_FS, 0x20a5880) = 0
readlink("/proc/self/exe", "/home/cr4bd/spring2017/cs4630/sl"... , 4096) =
brk(0x20c71c0) = 0x20c71c0
brk(0x20c8000) = 0x20c8000
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or direc
fstat(1, st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...) = 0
write(1, "Hello, World!\n", 14) = 14
exit_group(14) = ?
+++ exited with 14 +++
```

standard library startup

# statically linked hello.exe

```
gcc -static -o hello-static.exe hello.s
```

```
strace ./hello-static.exe:
```

```
execve("./hello-static.exe", ["/hello-static.exe"], [/* 46 vars */]) = 0
uname(sysname="Linux", nodename="reiss-lenovo", ...) = 0
brk(NULL) = 0x20a5000
brk(0x20a61c0) = 0x20a61c0
arch_prctl(ARCH_SET_FS, 0x20a5880) = 0
readlink("/proc/self/exe", "/home/cr4bd/spring2017/cs4630/sl"... , 4096) =
brk(0x20c71c0) = 0x20c71c0
brk(0x20c8000) = 0x20c8000
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or direc
fstat(1, st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...) = 0
write(1, "Hello, World!\n", 14) = 14
exit_group(14) = ?
+++ exited with 14 +++
```

memory allocation

# statically linked hello.exe

```
gcc -static -o hello-static.exe hello.s
```

```
strace ./hello-static.exe:
```

```
execve("./hello-static.exe", ["/hello-static.exe"], [/* 46 vars */]) = 0
uname(sysname="Linux", nodename="reiss-lenovo", ...) = 0
brk(NULL) = 0x20a5000
brk(0x20a61c0) = 0x20a61c0
arch_prctl(ARCH_SET_FS, 0x20a5880) = 0
readlink("/proc/self/exe", "/home/cr4bd/spring2017/cs4630/sl"... , 4096) =
brk(0x20c71c0) = 0x20c71c0
brk(0x20c8000) = 0x20c8000
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or direc
fstat(1, st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...) = 0
write(1, "Hello, World!\n", 14) = 14
exit_group(14) = ?
+++ exited with 14 +++
```

implementation of puts

# statically linked hello.exe

```
gcc -static -o hello-static.exe hello.s
```

```
strace ./hello-static.exe:
```

```
execve("./hello-static.exe", ["/hello-static.exe"], [/* 46 vars */]) = 0
uname(sysname="Linux", nodename="reiss-lenovo", ...) = 0
brk(NULL) = 0x20a5000
brk(0x20a61c0) = 0x20a61c0
arch_prctl(ARCH_SET_FS, 0x20a5880) = 0
readlink("/proc/self/exe", "/home/cr4bd/spring2017/cs4630/sl"... , 4096) =
brk(0x20c71c0) = 0x20c71c0
brk(0x20c8000) = 0x20c8000
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or direc
fstat(1, st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...) = 0
write(1, "Hello, World!\n", 14) = 14
exit_group(14) = ?
+++ exited with 14 +++
```

standard library shutdown

# dynamically linked hello.exe

```
gcc -o hello.exe hello.s
```

```
strace ./hello.exe:
```

```
execve("./hello.exe", [ "./hello.exe" ], [ /* 46 vars */ ]) = 0
...
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, st_mode=S_IFREG|0644, st_size=137808, ...) = 0
...
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\t\2\0\0\0\0\0"...
fstat(3, st_mode=S_IFREG|0755, st_size=1864888, ...) = 0
mmap(NULL, 3967392, PROT_READ|PROT_EXEC, ..., 3, 0) = 0x7fdfee54d000
mprotect(0x7fdfee70c000, 2097152, PROT_NONE) = 0
mmap(0x7fdfee90c000, 24576, PROT_READ|PROT_WRITE, ..., 3, 0x1bf000) = 0x7fdfee912000
mmap(0x7fdfee912000, 14752, PROT_READ|PROT_WRITE, ..., -1, 0) = 0x7fdfee912000
close(3) = 0
...
write(1, "Hello, World!\n", 14) = 14
exit_group(14) = ?
+++ exited with 14 +++
```

# dynamically linked hello.exe

```
gcc -o hello.exe hello.s
```

```
strace ./hello.exe:
```

```
execve("./hello.exe", [ "./hello.exe" ], [ /* 46 vars */ ]) = 0
```

```
...
```

```
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0
```

```
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
```

```
open("/etc/ld.so.preload", O_RDONLY) = -1 ENOENT (No such file or directory)
```

```
fstat(3, st)
```

the standard C library (includes puts)

```
...
```

```
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
```

```
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\t\2\0\0\0\0\0"...
```

```
fstat(3, st_mode=S_IFREG|0755, st_size=1864888, ...) = 0
```

```
mmap(NULL, 3967392, PROT_READ|PROT_EXEC, ..., 3, 0) = 0x7fdfee54d000
```

```
mprotect(0x7fdfee70c000, 2097152, PROT_NONE) = 0
```

```
mmap(0x7fdfee90c000, 24576, PROT_READ|PROT_WRITE, ..., 3, 0x1bf000) = 0x7fdfee912000
```

```
mmap(0x7fdfee912000, 14752, PROT_READ|PROT_WRITE, ..., -1, 0) = 0x7fdfee912000
```

```
close(3) = 0
```

```
...
```

```
write(1, "Hello, World!\n", 14) = 14
```

```
exit_group(14) = ?
```

```
+++ exited with 14 +++
```

# dynamically linked hello.exe

```
gcc -o hello.exe hello.s
```

```
strace ./hello.exe:
```

```
execve("./hello.exe", ["/etc/ld.so.preload"], [/* 46 vars */]) = 0
```

```
...
```

```
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
```

```
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or direc
```

```
open("/etc/ld.so.preload",
```

```
fstat(3, st
```

memory allocation (different method)

```
...
```

```
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
```

```
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\t\2\0\0\0\0\0"...
```

```
fstat(3, st_mode=S_IFREG|0755, st_size=1864888, ...) = 0
```

```
mmap(NULL, 3967392, PROT_READ|PROT_EXEC, ..., 3, 0) = 0x7fdfee54d000
```

```
mprotect(0x7fdfee70c000, 2097152, PROT_NONE) = 0
```

```
mmap(0x7fdfee90c000, 24576, PROT_READ|PROT_WRITE, ..., 3, 0x1bf000) = 0x7fdfee90c000
```

```
mmap(0x7fdfee912000, 14752, PROT_READ|PROT_WRITE, ..., -1, 0) = 0x7fdfee912000
```

```
close(3) = 0
```

```
...
```

```
write(1, "Hello, World!\n", 14) = 14
```

```
exit_group(14) = ?
```

```
+++ exited with 14 +++
```



# dynamically linked hello.exe

```
gcc -o hello.exe hello.s
```

```
strace ./hello.exe:
```

```
execve("./hello.exe", ["/.hello.exe"], [/* 46 vars */]) = 0
```

```
...
```

```
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
```

```
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or direc
```

```
open("/etc/ld.so
```

```
fstat(3, st_mode
```

read standard C library header

```
...
```

```
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
```

```
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\t\2\0\0\0\0\0"...
```

```
fstat(3, st_mode=S_IFREG|0755, st_size=1864888, ...) = 0
```

```
mmap(NULL, 3967392, PROT_READ|PROT_EXEC, ..., 3, 0) = 0x7fdfee54d000
```

```
mprotect(0x7fdfee70c000, 2097152, PROT_NONE) = 0
```

```
mmap(0x7fdfee90c000, 24576, PROT_READ|PROT_WRITE, ..., 3, 0x1bf000) = 0x7f
```

```
mmap(0x7fdfee912000, 14752, PROT_READ|PROT_WRITE, ..., -1, 0) = 0x7fdfee91
```

```
close(3) = 0
```

```
...
```

```
write(1, "Hello, World!\n", 14) = 14
```

```
exit_group(14) = ?
```

```
+++ exited with 14 +++
```

# dynamically linked hello.exe

```
gcc -o hello.exe hello.s
```

```
strace ./hello.exe:
```

```
execve("./hello.exe", [ "./hello.exe" ], [ /* 46 vars */ ]) = 0
```

```
...
```

```
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0
```

```
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
```

```
open("/etc/ld.so.preload", O_RDONLY) = -1 ENOENT (No such file or directory)
```

```
fstat(3, {st_mode=S_IFREG|0755, st_size=1864888, ...}) = 0
```

load standard C library (3 = opened file)

```
...
```

```
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
```

```
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\t\2\0\0\0\0\0"...
```

```
fstat(3, {st_mode=S_IFREG|0755, st_size=1864888, ...}) = 0
```

```
mmap(NULL, 3967392, PROT_READ|PROT_EXEC, ..., 3, 0) = 0x7fdfee54d000
```

```
mprotect(0x7fdfee70c000, 2097152, PROT_NONE) = 0
```

```
mmap(0x7fdfee90c000, 24576, PROT_READ|PROT_WRITE, ..., 3, 0x1bf000) = 0x7fdfee912000
```

```
mmap(0x7fdfee912000, 14752, PROT_READ|PROT_WRITE, ..., -1, 0) = 0x7fdfee912000
```

```
close(3) = 0
```

```
...
```

```
write(1, "Hello, World!\n", 14) = 14
```

```
exit_group(14) = ?
```

```
+++ exited with 14 +++
```

# dynamically linked hello.exe

```
gcc -o hello.exe hello.s
```

```
strace ./hello.exe:
```

```
execve("./hello.exe", ["/.hello.exe"], [/* 46 vars */]) = 0
```

```
...
```

```
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
```

```
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or direc
```

```
open
```

```
fstat
```

allocate zero-initialized data segment for C library

```
...
```

```
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
```

```
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\t\2\0\0\0\0\0"...
```

```
fstat(3, st_mode=S_IFREG|0755, st_size=1864888, ...) = 0
```

```
mmap(NULL, 3967392, PROT_READ|PROT_EXEC, ..., 3, 0) = 0x7fdfee54d000
```

```
mprotect(0x7fdfee70c000, 2097152, PROT_NONE) = 0
```

```
mmap(0x7fdfee90c000, 24576, PROT_READ|PROT_WRITE, ..., 3, 0x1bf000) = 0x7f
```

```
mmap(0x7fdfee912000, 14752, PROT_READ|PROT_WRITE, ..., -1, 0) = 0x7fdfee91
```

```
close(3) = 0
```

```
...
```

```
write(1, "Hello, World!\n", 14) = 14
```

```
exit_group(14) = ?
```

```
+++ exited with 14 +++
```

# dynamic linking

load and link (find address of puts) runtime

advantages:

- smaller executables

- easier upgrades

- less memory usage (load one copy of library for multiple programs)

disadvantages:

- library upgrades breaking programs

- programs less compatible between OS versions

- possibly slower

# where's the linker

Where's the code that calls  
`open("../libc.so.6")`?

Could check `hello.exe` — it's not there!

# where's the linker

Where's the code that calls  
`open("...libc.so.6")`?

Could check `hello.exe` — it's not there!

instead: “interpreter”  
`/lib64/ld-linux-x86-64.so.2`

on Linux: contains loading code instead of core OS  
OS loads it instead of program

# objdump — the interpreter

excerpt from `objdump -sx hello.exe`:

Program Header:

```
...  
  INTERP off      0x00000238 vaddr 0x0400238 paddr 0x0400238 align 2**0  
        filesz 0x0000001c memsz 0x0000001c flags r--  
...
```

Contents of section `.interp`:

```
400238 2f6c6962 36342f6c 642d6c69 6e75782d  /lib64/ld-linux-  
400248 7838362d 36342e73 6f2e3200  x86-64.so.2.
```

# dynamic linking information

**symbol table** — works the same, but in executable  
could use same relocations — but these are **expensive**  
rather just copy data from disk without changes  
solutions: global lookup table!



# dynamically linked puts

```
00000000000400400 <puts@plt>:
  400400:      ff 25 12 0c 20 00                jmpq    *0x200c12(%rip)
                        /* 0x200c12+RIP = _GLOBAL_OFFSET_TABLE_+0x18 */
... later in main: ...
  40052d:      e8 ce fe ff ff                callq   400400 <puts@plt>
                        /* instead of call puts */
```

replace puts with **stub** puts@plt

plt = procedure linkage table

stub: jump to \*\_GLOBAL\_OFFSET\_TABLE[3]

dynamic linker changes **table instead of code**

could change code — just would be less efficient

# lazy binding

```
00000000000400400 <puts@plt>:
 400400:      ff 25 12 0c 20 00          jmpq    *0x200c12(%rip)
          /* 0x200c12+RIP = _GLOBAL_OFFSET_TABLE_+0x18 */
 400406:      68 00 00 00 00          pushq   $0x0
 40040b:      e9 e0 ff ff ff          jmpq    4003f0 <_init+0x28>
```

could fill global offset table immediately

alternative: fill on demand

extra code (pushq then jmpq) runs “fixup code”

- reads symbol tables to find function

- edits global offset table

- jumps to function

called “lazy binding”

# lazy binding pro/con

## advantages:

- faster program loading
- no overhead for unused code (often a lot of stuff)

## disadvantages:

- can move errors (missing functions, etc.) to runtime
- possibly more total overhead

# 8086 evolution

Intel 8086 — 1979, 16-bit registers

Intel (80)386 — 1986, 32-bit registers

AMD K8 — 2003, 64-bit registers

# x86 modes

x86 has multiple **modes**

maintains compatibility

e.g.: modern x86 processor can work like 8086  
called “real mode”

different mode for 32-bit/64-bit

same basic encoding; some sizes change