

# 143A: Principles of Operating Systems

## Lecture 4: Linking and Loading (Basic architecture of a program)

Anton Burtsev  
October, 2019

# What is a program?

- What parts do we need to run code?

# Parts needed to run a program

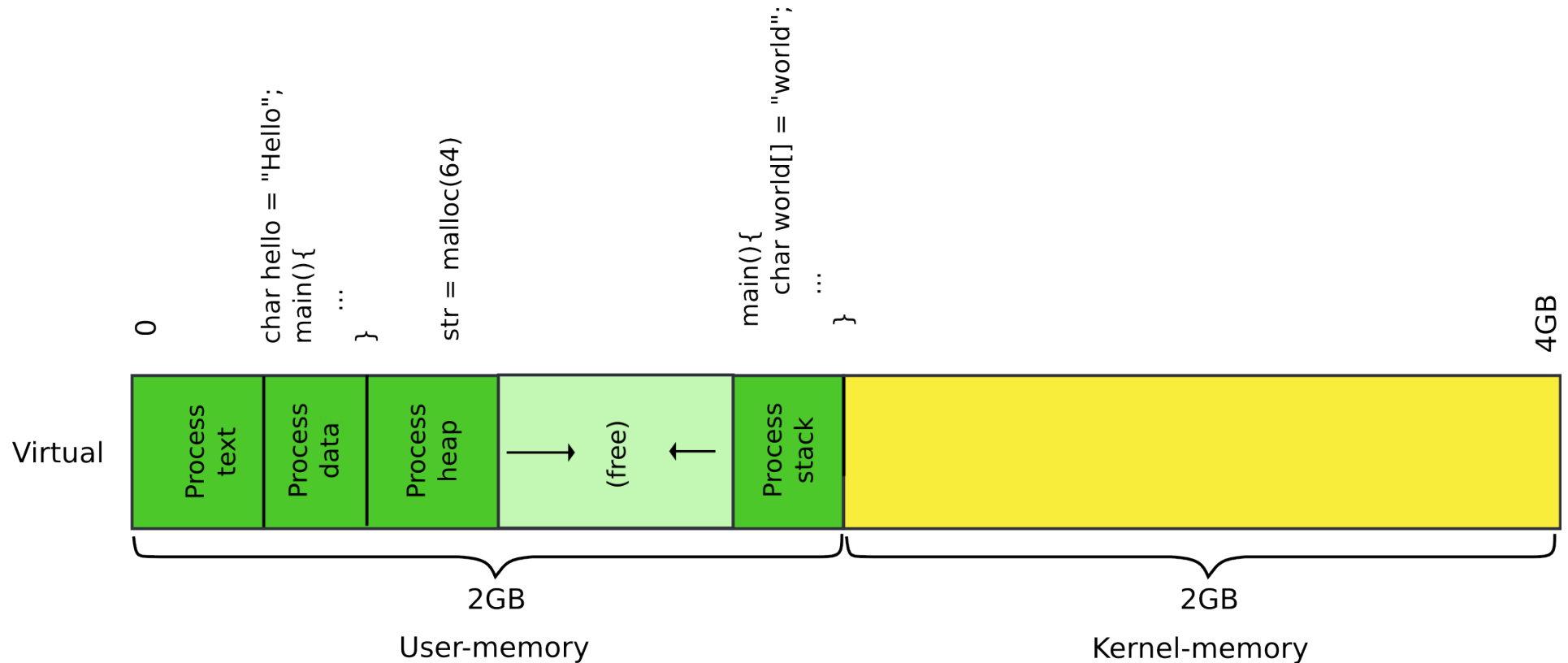
- Code itself
  - By convention it's called text
- Stack
  - To call functions
- Space for variables

What types of variables do you know?

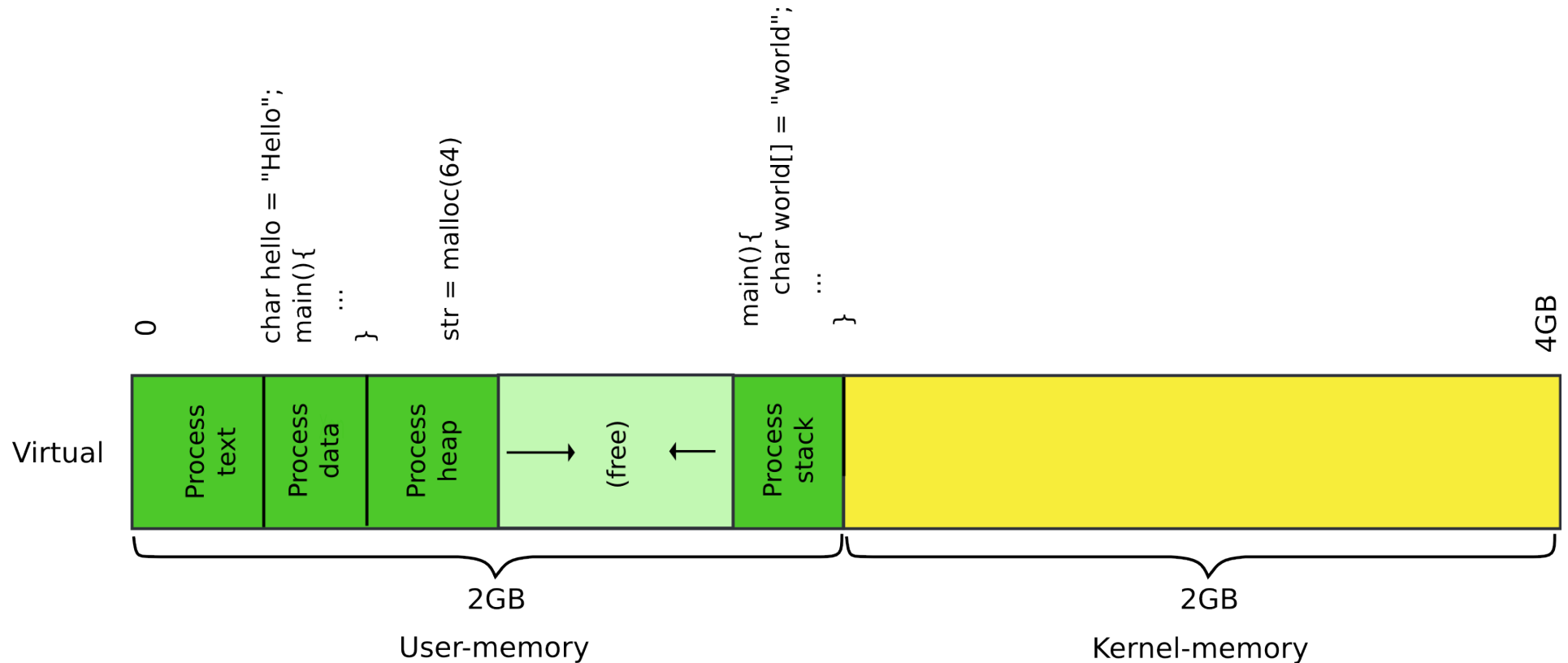
# What types of variables do you know?

- Global variables
  - Initialized → data section
  - Uninitialized → BSS
- Local variables
  - Stack
- Dynamic variables
  - Heap

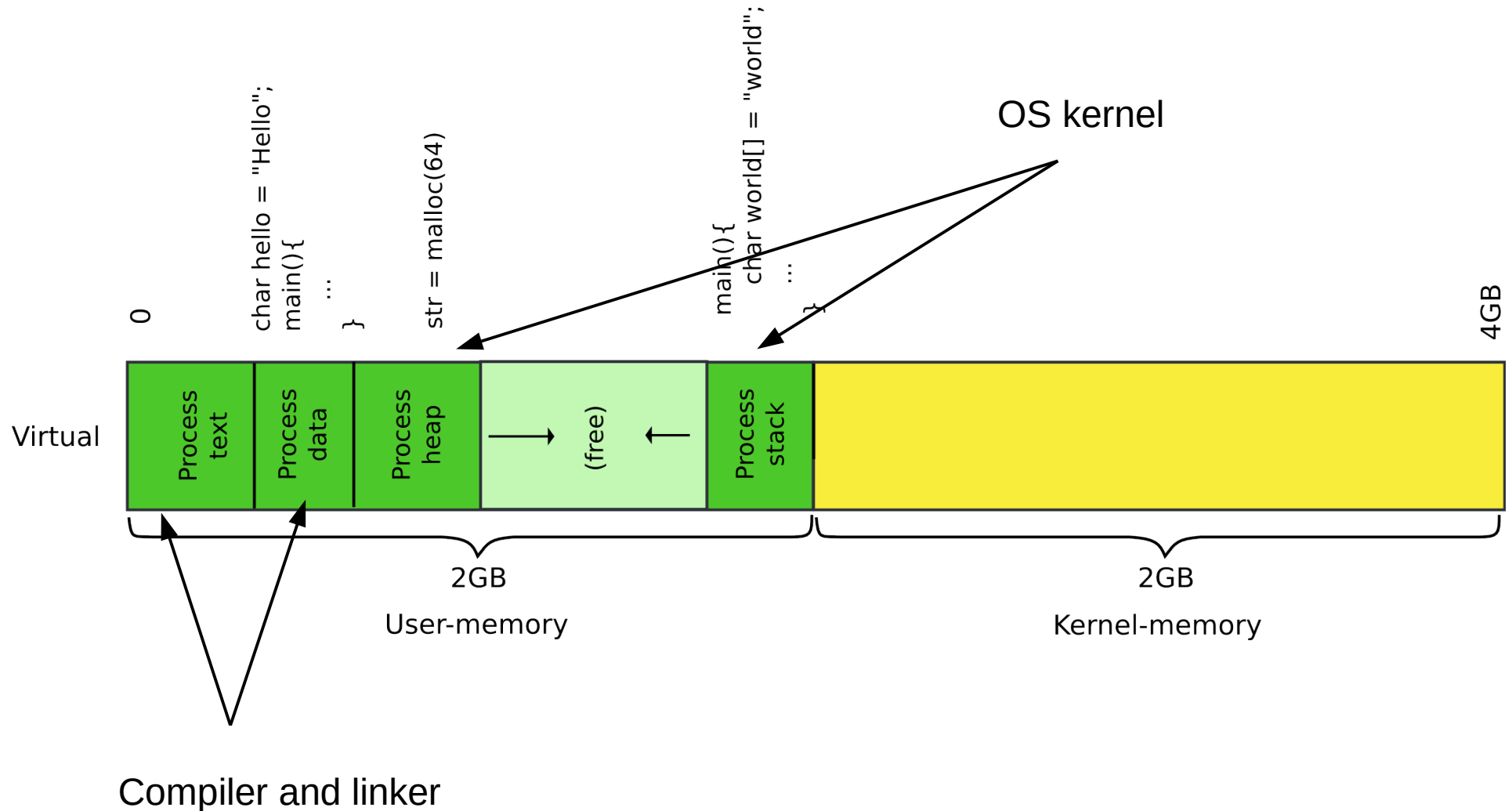
# Memory layout of a process



# Where do these areas come from?



# Memory layout of a process





# Example program

- Compute  $5 + 6$

```
#include <stdio.h>
```

```
int main(int ac, char **av)
{
    int a = 5, b = 6;
    return a + b;
}
```

- We build it like
  - I'm on 64 bit system, but want 32bit code, hence -m32

```
gcc -m32 hello-int.c
```

a.out: file format elf32-i386

**Contents of section .text:**

```
80483e0 d0c9e979 ffffffff90 e973ffff ff5589e5 ...y.....s...U..
80483f0 83ec10c7 45f80500 0000c745 fc060000 ....E.....E....
8048400 008b45fc 8b55f801 d0c9c366 90669090 ..E..U.....f.f..
8048410 555731ff 5653e805 ffffffff81 c3e51b00 UW1.VS.....
8048420 0083ec1c 8b6c2430 8db30cff ffffe861 .....l$0.....a
8048430 feffff8d 8308ffff ff29c6c1 fe0285f6 .....).....
```

**Contents of section .rodata:**

```
8048498 03000000 01000200 .....
```

**Contents of section .data:**

```
804a014 00000000 00000000 .....
```

**Disassembly of section .text:**

```
...
080483ed <main>:
80483ed:    55                push    %ebp
80483ee:    89 e5            mov     %esp,%ebp
80483f0:    83 ec 10         sub     $0x10,%esp
80483f3:    c7 45 f8 05 00 00 00 movl    $0x5,-0x8(%ebp)
80483fa:    c7 45 fc 06 00 00 00 movl    $0x6,-0x4(%ebp)
8048401:    8b 45 fc         mov     -0x4(%ebp),%eax
8048404:    8b 55 f8         mov     -0x8(%ebp),%edx
8048407:    01 d0           add     %edx,%eax
8048409:    c9              leave
804840a:    c3              ret
804840b:    66 90           xchg    %ax,%ax
804840d:    66 90           xchg    %ax,%ax
804840f:    90              nop
```

# objdump -sd a.out

- GCC syntax, i.e.  
`mov %esp, %ebp`  
`// EBP = ESP`

a.out: file format elf32-i386

#### Contents of section .text:

```
80483e0 d0c9e979 ffffffff e973ffff ff5589e5 ...y.....s...U..
80483f0 83ec10c7 45f80500 0000c745 fc060000 ....E.....E....
8048400 008b45fc 8b55f801 d0c9c366 90669090 ..E..U.....f.f..
8048410 555731ff 5653e805 ffffffff81 c3e51b00 UW1.VS.....
8048420 0083ec1c 8b6c2430 8db30cff ffffe861 .....l$0.....a
8048430 feffff8d 8308ffff ff29c6c1 fe0285f6 .....). ....
```

#### Contents of section .rodata:

```
8048498 03000000 01000200 .....
```

#### Contents of section .data:

```
804a014 00000000 00000000 .....
```

#### Disassembly of section .text:

```
...
080483ed <main>:
80483ed: 55          push    %ebp
80483ee: 89 e5      mov     %esp,%ebp
80483f0: 83 ec 10   sub     $0x10,%esp
80483f3: c7 45 f8 05 00 00 00 movl    $0x5,-0x8(%ebp)
80483fa: c7 45 fc 06 00 00 00 movl    $0x6,-0x4(%ebp)
8048401: 8b 45 fc   mov     -0x4(%ebp),%eax
8048404: 8b 55 f8   mov     -0x8(%ebp),%edx
8048407: 01 d0     add     %edx,%eax
8048409: c9        leave
804840a: c3        ret
804840b: 66 90     xchg    %ax,%ax
804840d: 66 90     xchg    %ax,%ax
804840f: 90        nop
```

# objdump -sd a.out

- GCC syntax, i.e.  
`mov %esp, %ebp`  
`// EBP = ESP`

a.out: file format elf32-i386

**Contents of section .text:**

```
80483e0 d0c9e979 ffffffff90 e973ffff ff5589e5 ...y.....s...U..
80483f0 83ec10c7 45f80500 0000c745 fc060000 ....E.....E....
8048400 008b45fc 8b55f801 d0c9c366 90669090 ..E..U.....f.f..
8048410 555731ff 5653e805 ffffffff81 c3e51b00 UW1.VS.....
8048420 0083ec1c 8b6c2430 8db30cff ffffe861 .....l$0.....a
8048430 feffff8d 8308ffff ff29c6c1 fe0285f6 .....).....
```

**Contents of section .rodata:**

```
8048498 03000000 01000200 .....
```

**Contents of section .data:**

```
804a014 00000000 00000000 .....
```

**Disassembly of section .text:**

...

080483ed <main>:

80483ed:	55	push	%ebp	# Maintain the stack frame
80483ee:	89 e5	mov	%esp,%ebp	
80483f0:	83 ec 10	sub	\$0x10,%esp	
80483f3:	c7 45 f8 05 00 00 00	movl	\$0x5,-0x8(%ebp)	
80483fa:	c7 45 fc 06 00 00 00	movl	\$0x6,-0x4(%ebp)	
8048401:	8b 45 fc	mov	-0x4(%ebp),%eax	
8048404:	8b 55 f8	mov	-0x8(%ebp),%edx	
8048407:	01 d0	add	%edx,%eax	
8048409:	c9	leave		
804840a:	c3	ret		
804840b:	66 90	xchg	%ax,%ax	
804840d:	66 90	xchg	%ax,%ax	
804840f:	90	nop		

# objdump -sd a.out

- GCC syntax, i.e.  
`mov %esp, %ebp`  
`// EBP = ESP`

a.out: file format elf32-i386

**Contents of section .text:**

```
80483e0 d0c9e979 ffffffff90 e973ffff ff5589e5 ...y.....s...U..
80483f0 83ec10c7 45f80500 0000c745 fc060000 ....E.....E....
8048400 008b45fc 8b55f801 d0c9c366 90669090 ..E..U.....f.f..
8048410 555731ff 5653e805 ffffffff81 c3e51b00 UW1.VS.....
8048420 0083ec1c 8b6c2430 8db30cff ffffe861 .....l$0.....a
8048430 feffff8d 8308ffff ff29c6c1 fe0285f6 .....).....
```

**Contents of section .rodata:**

```
8048498 03000000 01000200 .....
```

**Contents of section .data:**

```
804a014 00000000 00000000 .....
```

**Disassembly of section .text:**

...

080483ed <main>:

```
80483ed:      55                push    %ebp
80483ee:      89 e5            mov     %esp,%ebp
80483f0:      83 ec 10         sub     $0x10,%esp      # Allocate space for a and b
80483f3:      c7 45 f8 05 00 00 00 movl    $0x5,-0x8(%ebp)
80483fa:      c7 45 fc 06 00 00 00 movl    $0x6,-0x4(%ebp)
8048401:      8b 45 fc         mov     -0x4(%ebp),%eax
8048404:      8b 55 f8         mov     -0x8(%ebp),%edx
8048407:      01 d0           add     %edx,%eax
8048409:      c9              leave   %eax
804840a:      c3              ret
804840b:      66 90           xchg    %ax,%ax
804840d:      66 90           xchg    %ax,%ax
804840f:      90              nop
```

# objdump -sd a.out

- GCC syntax, i.e.  
`mov %esp, %ebp`  
`// EBP = ESP`

a.out: file format elf32-i386

#### Contents of section .text:

```
80483e0 d0c9e979 ffffffff90 e973ffff ff5589e5 ...y.....s...U..
80483f0 83ec10c7 45f80500 0000c745 fc060000 ....E.....E....
8048400 008b45fc 8b55f801 d0c9c366 90669090 ..E..U.....f.f..
8048410 555731ff 5653e805 ffffffff81 c3e51b00 UW1.VS.....
8048420 0083ec1c 8b6c2430 8db30cff ffffe861 .....l$0.....a
8048430 feffff8d 8308ffff ff29c6c1 fe0285f6 .....).....
```

#### Contents of section .rodata:

```
8048498 03000000 01000200 .....
```

#### Contents of section .data:

```
804a014 00000000 00000000 .....
```

#### Disassembly of section .text:

...

080483ed <main>:

```
80483ed: 55          push    %ebp
80483ee: 89 e5      mov     %esp,%ebp
80483f0: 83 ec 10   sub     $0x10,%esp    # Allocate space for a and b
80483f3: c7 45 18 05 00 00 00 movl    $0x5,-0x8(%ebp)
80483fa: c7 45 fc 06 00 00 00 movl    $0x6,-0x4(%ebp)
8048401: 8b 45 fc   mov     -0x4(%ebp),%eax
8048404: 8b 55 f8   mov     -0x8(%ebp),%edx
8048407: 01 d0     add     %edx,%eax
8048409: c9        leave
804840a: c3        ret
804840b: 66 90     xchg    %ax,%ax
804840d: 66 90     xchg    %ax,%ax
804840f: 90        nop
```

# objdump -sd a.out

- GCC syntax, i.e.  
`mov %esp, %ebp`  
`// EBP = ESP`

a.out: file format elf32-i386

**Contents of section .text:**

```
80483e0 d0c9e979 ffffffff90 e973ffff ff5589e5 ...y.....s...U..
80483f0 83ec10c7 45f80500 0000c745 fc060000 ....E.....E....
8048400 008b45fc 8b55f801 d0c9c366 90669090 ..E..U.....f.f..
8048410 555731ff 5653e805 ffffffff81 c3e51b00 UW1.VS.....
8048420 0083ec1c 8b6c2430 8db30cff ffffe861 .....l$0.....a
8048430 feffff8d 8308ffff ff29c6c1 fe0285f6 .....).....
```

**Contents of section .rodata:**

```
8048498 03000000 01000200 .....
```

**Contents of section .data:**

```
804a014 00000000 00000000 .....
```

**Disassembly of section .text:**

...

080483ed <main>:

```
80483ed:      55                push    %ebp
80483ee:      89 e5            mov     %esp,%ebp
80483f0:      83 ec 10         sub     $0x10,%esp
80483f3:      c7 45 f8 05 00 00 00 movl    $0x5,-0x8(%ebp) # Initialize a = 5
80483fa:      c7 45 fc 06 00 00 00 movl    $0x6,-0x4(%ebp) # Initialize b = 6
8048401:      8b 45 fc         mov     -0x4(%ebp),%eax
8048404:      8b 55 f8         mov     -0x8(%ebp),%edx
8048407:      01 d0           add     %edx,%eax
8048409:      c9              leave   %eax
804840a:      c3              ret
804840b:      66 90           xchg    %ax,%ax
804840d:      66 90           xchg    %ax,%ax
804840f:      90              nop
```

# objdump -sd a.out

- GCC syntax, i.e.  
`mov %esp, %ebp`  
`// EBP = ESP`

a.out: file format elf32-i386

#### Contents of section .text:

```
80483e0 d0c9e979 ffffffff90 e973ffff ff5589e5 ...y.....s...U..
80483f0 83ec10c7 45f80500 0000c745 fc060000 ....E.....E....
8048400 008b45fc 8b55f801 d0c9c366 90669090 ..E..U.....f.f..
8048410 555731ff 5653e805 ffffffff81 c3e51b00 UW1.VS.....
8048420 0083ec1c 8b6c2430 8db30cff ffffe861 .....l$0.....a
8048430 feffff8d 8308ffff ff29c6c1 fe0285f6 .....).....
```

#### Contents of section .rodata:

```
8048498 03000000 01000200 .....
```

#### Contents of section .data:

```
804a014 00000000 00000000 .....
```

#### Disassembly of section .text:

...

080483ed <main>:

80483ed:	55		push	%ebp
80483ee:	89 e5		mov	%esp,%ebp
80483f0:	83 ec 10		sub	\$0x10,%esp
80483f3:	c7 45 f8 05 00 00 00		movl	\$0x5,-0x8(%ebp) # Initialize a = 5
80483fa:	c7 45 fc 06 00 00 00		movl	\$0x6,-0x4(%ebp) # Initialize b = 6
8048401:	8b 45 fc		mov	-0x4(%ebp),%eax
8048404:	8b 55 f8		mov	-0x8(%ebp),%edx
8048407:	01 d0		add	%edx,%eax
8048409:	c9		leave	
804840a:	c3		ret	
804840b:	66 90		xchg	%ax,%ax
804840d:	66 90		xchg	%ax,%ax
804840f:	90		nop	

# objdump -sd a.out

- GCC syntax, i.e.  
`mov %esp, %ebp`  
`// EBP = ESP`



a.out: file format elf32-i386

#### Contents of section .text:

```
80483e0 d0c9e979 ffffffff90 e973ffff ff5589e5 ...y.....s...U..
80483f0 83ec10c7 45f80500 0000c745 fc060000 ....E.....E....
8048400 008b45fc 8b55f801 d0c9c366 90669090 ..E..U.....f.f..
8048410 555731ff 5653e805 ffffffff81 c3e51b00 UW1.VS.....
8048420 0083ec1c 8b6c2430 8db30cff ffffe861 .....l$0.....a
8048430 feffff8d 8308ffff ff29c6c1 fe0285f6 .....).....
```

#### Contents of section .rodata:

```
8048498 03000000 01000200 .....
```

#### Contents of section .data:

```
804a014 00000000 00000000 .....
```

#### Disassembly of section .text:

...

080483ed <main>:

80483ed:	55	push	%ebp
80483ee:	89 e5	mov	%esp,%ebp
80483f0:	83 ec 10	sub	\$0x10,%esp
80483f3:	c7 45 f8 05 00 00 00	movl	\$0x5,-0x8(%ebp) # Initialize a = 5
80483fa:	c7 45 fc 06 00 00 00	movl	\$0x6,-0x4(%ebp) # Initialize b = 6
8048401:	8b 45 1c	mov	-0x4(%ebp),%eax
8048404:	8b 55 f8	mov	-0x8(%ebp),%edx
8048407:	01 d0	add	%edx,%eax
8048409:	c9	leave	
804840a:	c3	ret	
804840b:	66 90	xchg	%ax,%ax
804840d:	66 90	xchg	%ax,%ax
804840f:	90	nop	

# objdump -sd a.out

- GCC syntax, i.e.  
`mov %esp, %ebp`  
`// EBP = ESP`

a.out: file format elf32-i386

#### Contents of section .text:

```
80483e0 d0c9e979 ffffffff90 e973ffff ff5589e5 ...y.....s...U..
80483f0 83ec10c7 45f80500 0000c745 fc060000 ....E.....E....
8048400 008b45fc 8b55f801 d0c9c366 90669090 ..E..U.....f.f..
8048410 555731ff 5653e805 ffffffff81 c3e51b00 UW1.VS.....
8048420 0083ec1c 8b6c2430 8db30cff ffffe861 .....l$0.....a
8048430 feffff8d 8308ffff ff29c6c1 fe0285f6 .....)......
```

#### Contents of section .rodata:

```
8048498 03000000 01000200 .....
```

#### Contents of section .data:

```
804a014 00000000 00000000 .....
```

#### Disassembly of section .text:

...

080483ed <main>:

80483ed:	55	push	%ebp
80483ee:	89 e5	mov	%esp,%ebp
80483f0:	83 ec 10	sub	\$0x10,%esp
80483f3:	c7 45 f8 05 00 00 00	movl	\$0x5,-0x8(%ebp)
80483fa:	c7 45 fc 06 00 00 00	movl	\$0x6,-0x4(%ebp)
8048401:	8b 45 <b>fc</b>	mov	<b>-0x4(%ebp)</b> ,%eax # Move b into %eax
8048404:	8b 55 <b>f8</b>	mov	<b>-0x8(%ebp)</b> ,%edx # Move a into %edx
8048407:	01 d0	add	%edx,%eax
8048409:	c9	leave	
804840a:	c3	ret	
804840b:	66 90	xchg	%ax,%ax
804840d:	66 90	xchg	%ax,%ax
804840f:	90	nop	

# objdump -sd a.out

- GCC syntax, i.e.  
`mov %esp, %ebp`  
`// EBP = ESP`

a.out: file format elf32-i386

**Contents of section .text:**

```
80483e0 d0c9e979 ffffffff90 e973ffff ff5589e5 ...y.....s...U..
80483f0 83ec10c7 45f80500 0000c745 fc060000 ....E.....E....
8048400 008b45fc 8b55f801 d0c9c366 90669090 ..E..U.....f.f..
8048410 555731ff 5653e805 ffffffff81 c3e51b00 UW1.VS.....
8048420 0083ec1c 8b6c2430 8db30cff ffffe861 .....l$0.....a
8048430 feffff8d 8308ffff ff29c6c1 fe0285f6 .....).....
```

**Contents of section .rodata:**

```
8048498 03000000 01000200 .....
```

**Contents of section .data:**

```
804a014 00000000 00000000 .....
```

**Disassembly of section .text:**

...

080483ed <main>:

```
80483ed:      55                push    %ebp
80483ee:      89 e5            mov     %esp,%ebp
80483f0:      83 ec 10         sub     $0x10,%esp
80483f3:      c7 45 f8 05 00 00 00 movl    $0x5,-0x8(%ebp)
80483fa:      c7 45 fc 06 00 00 00 movl    $0x6,-0x4(%ebp)
8048401:      8b 45 fc         mov     -0x4(%ebp),%eax
8048404:      8b 55 f8         mov     -0x8(%ebp),%edx
8048407:      01 d0            add     %edx,%eax      # a + b
8048409:      c9              leave   %eax
804840a:      c3              ret
804840b:      66 90            xchg    %ax,%ax
804840d:      66 90            xchg    %ax,%ax
804840f:      90              nop
```

# objdump -sd a.out

- GCC syntax, i.e.  
`mov %esp, %ebp`  
`// EBP = ESP`

a.out: file format elf32-i386

**Contents of section .text:**

```
80483e0 d0c9e979 ffffffff90 e973ffff ff5589e5 ...y.....s...U..
80483f0 83ec10c7 45f80500 0000c745 fc060000 ....E.....E....
8048400 008b45fc 8b55f801 d0c9c366 90669090 ..E..U.....f.f..
8048410 555731ff 5653e805 ffffffff81 c3e51b00 UW1.VS.....
8048420 0083ec1c 8b6c2430 8db30cff ffffe861 .....l$0.....a
8048430 feffff8d 8308ffff ff29c6c1 fe0285f6 .....).....
```

**Contents of section .rodata:**

```
8048498 03000000 01000200 .....
```

**Contents of section .data:**

```
804a014 00000000 00000000 .....
```

**Disassembly of section .text:**

...

080483ed <main>:

```
80483ed:      55                push    %ebp
80483ee:      89 e5            mov     %esp,%ebp
80483f0:      83 ec 10         sub     $0x10,%esp
80483f3:      c7 45 f8 05 00 00 00 movl    $0x5,-0x8(%ebp)
80483fa:      c7 45 fc 06 00 00 00 movl    $0x6,-0x4(%ebp)
8048401:      8b 45 fc         mov     -0x4(%ebp),%eax
8048404:      8b 55 f8         mov     -0x8(%ebp),%edx
8048407:      01 d0           add     %edx,%eax
8048409:      c9             leave   ,%eax          # Pop the frame ESP = EBP
804840a:      c3             ret     ,%eax          # return
804840b:      66 90           xchg    %ax,%ax
804840d:      66 90           xchg    %ax,%ax
804840f:      90             nop
```

# objdump -sd a.out

- GCC syntax, i.e.  
`mov %esp, %ebp`  
`// EBP = ESP`

a.out: file format elf32-i386

**Contents of section .text:**

```
80483e0 d0c9e979 ffffffff90 e973ffff ff5589e5 ...y.....s...U..
80483f0 83ec10c7 45f80500 0000c745 fc060000 ....E.....E....
8048400 008b45fc 8b55f801 d0c9c366 90669090 ..E..U.....f.f..
8048410 555731ff 5653e805 ffffffff81 c3e51b00 UW1.VS.....
8048420 0083ec1c 8b6c2430 8db30cff ffffe861 .....l$0.....a
8048430 feffff8d 8308ffff ff29c6c1 fe0285f6 .....).....
```

**Contents of section .rodata:**

```
8048498 03000000 01000200 .....
```

**Contents of section .data:**

```
804a014 00000000 00000000 .....
```

**Disassembly of section .text:**

...

080483ed <main>:

```
80483ed:      55                push    %ebp
80483ee:      89 e5            mov     %esp,%ebp
80483f0:      83 ec 10         sub     $0x10,%esp
80483f3:      c7 45 f8 05 00 00 00 movl    $0x5,-0x8(%ebp)
80483fa:      c7 45 fc 06 00 00 00 movl    $0x6,-0x4(%ebp)
8048401:      8b 45 fc         mov     -0x4(%ebp),%eax
8048404:      8b 55 f8         mov     -0x8(%ebp),%edx
8048407:      01 d0           add     %edx,%eax
8048409:      c9              leave   %eax
804840a:      c3              ret
804840b:      66 90           xchg    %ax,%ax      # Code alignment
804840d:      66 90           xchg    %ax,%ax      # 2 byte no op
804840f:      90              nop                 # 1 byte no op
```

# objdump -sd a.out

- GCC syntax, i.e.  
`mov %esp, %ebp`  
`// EBP = ESP`

# “Optimizing subroutines in assembly language” by Agner Fog:

[https://www.agner.org/optimize/optimizing\\_assembly.pdf](https://www.agner.org/optimize/optimizing_assembly.pdf)

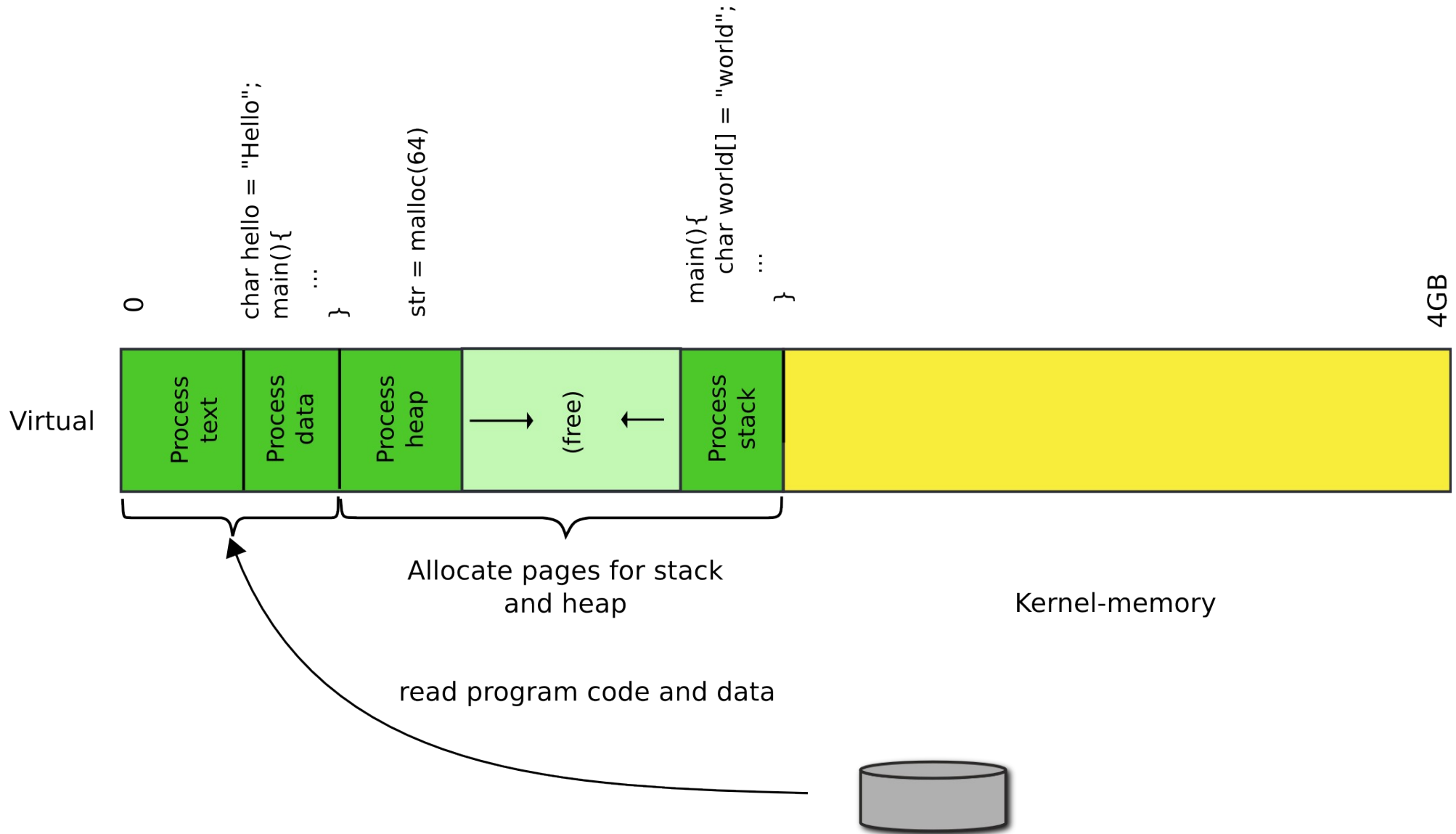
## 11.5 Alignment of code

**Most microprocessors fetch code in aligned 16-byte or 32-byte blocks.** If an important subroutine entry or jump label happens to be near the end of a 16-byte block then the microprocessor will only get a few useful bytes of code when fetching that block of code. It may have to fetch the next 16 bytes too before it can decode the first instructions after the label. This can be avoided by aligning important subroutine entries and loop entries by 16.

...

Aligning a subroutine entry is as simple as putting as many NOP 's as needed before the subroutine entry to make the address divisible by 8, 16, 32 or 64, as desired.

# Load program in memory



# We however build programs from multiple files

## Part of the xv6 Makefile

```
bootblock: bootasm.S bootmain.c
```

```
$(CC) $(CFLAGS) -fno-pic -O -nostdinc -I. -c bootmain.c
```

```
$(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c bootasm.S
```

```
$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o bootblock.o bootasm.o bootmain.o
```

```
$(OBJDUMP) -S bootblock.o > bootblock.asm
```

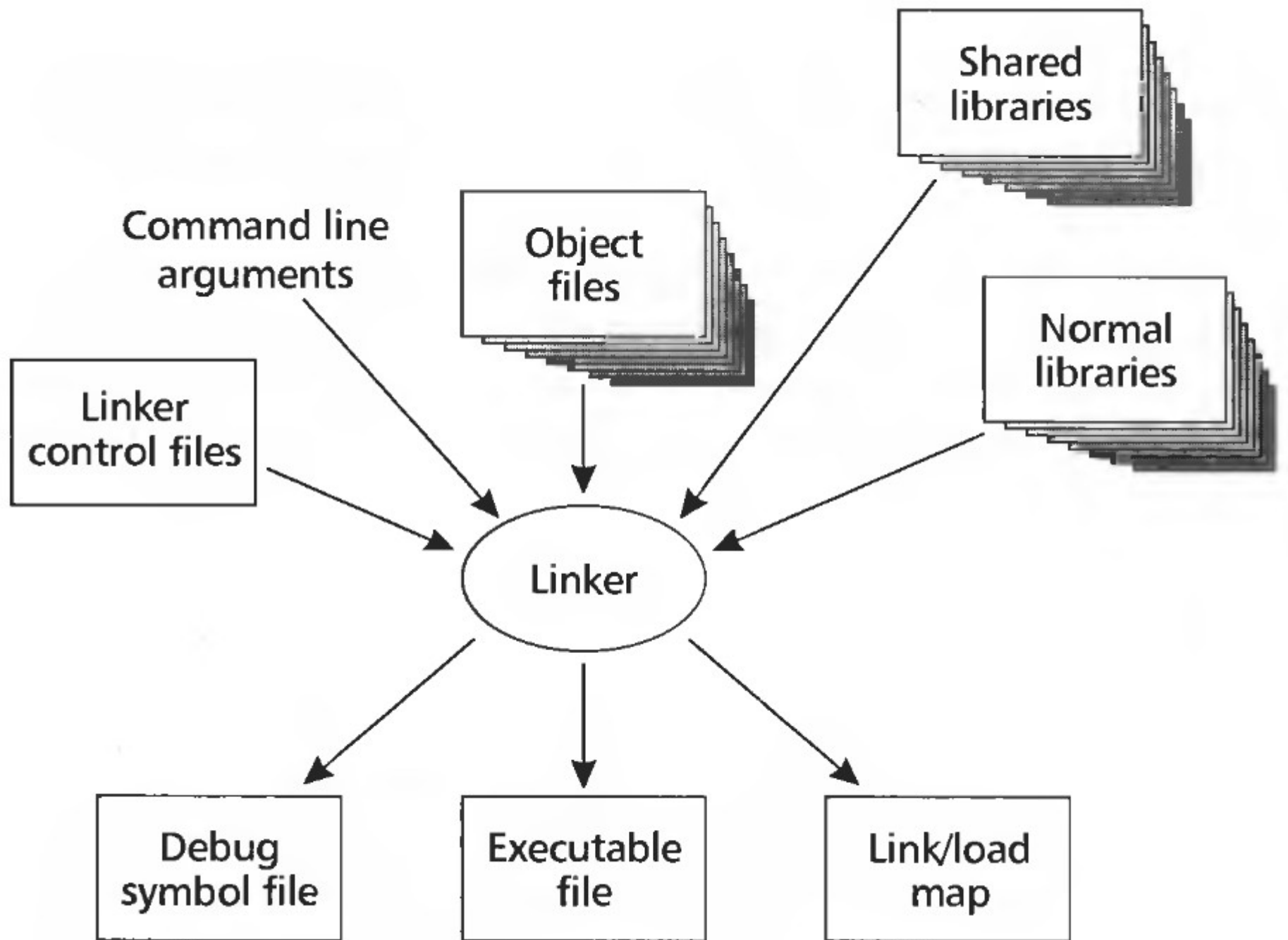
```
$(OBJCOPY) -S -O binary -j .text bootblock.o bootblock
```

```
./sign.pl bootblock
```

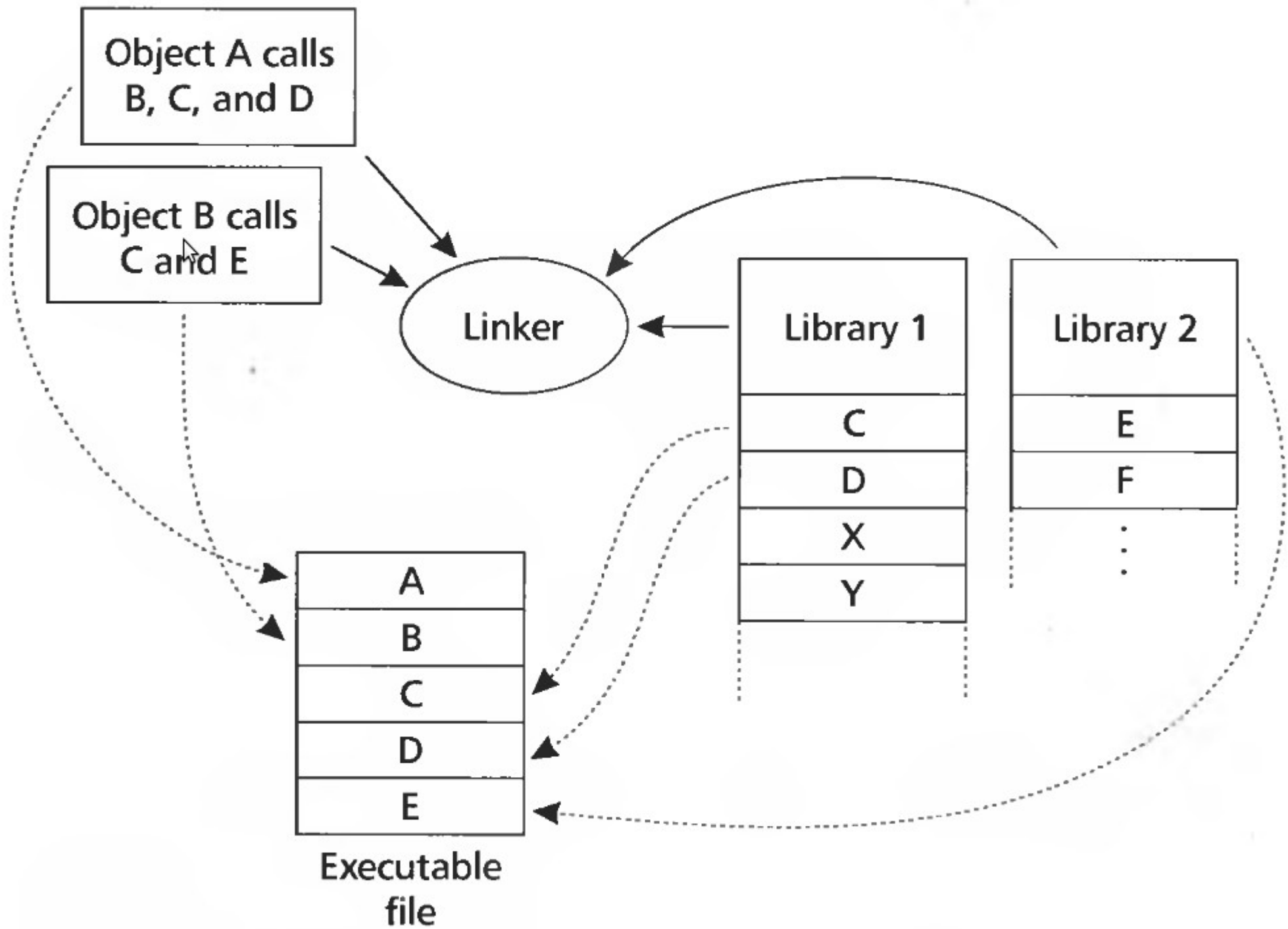


# Linking and loading

- Linking
  - Combining multiple code modules into a single executable
  - E.g., use standard libraries in your own code
- Loading
  - Process of getting an executable running on the machine



- Input: object files (code modules)
- Each object file contains
  - A set of segments
    - Code
    - Data
  - A symbol table
    - Imported & exported symbols
- Output: executable file, library, etc.



# Why linking?

# Why linking?

- Modularity
  - Program can be written as a collection of modules
  - Can build libraries of common functions
- Efficiency
  - Code compilation
    - Change one source file, recompile it, and re-link the executable
  - Space efficiency
    - Share common code across executables
    - On disk and in memory

# Two path process

- Path 1: scan input files
  - Identify boundaries of each segment
  - Collect all defined and undefined symbol information
  - Determine sizes and locations of each segment
- Path 2
  - Adjust memory addresses in code and data to reflect relocated segment addresses

# Example

- Save a into b, e.g.,  $b = a$

```
mov a, %eax
```

```
mov %eax, b
```

- Generated code
  - a is defined in the same file at 0x1234, **b is imported**
  - Each instruction is 1 byte opcode + 4 bytes address

```
A1 34 12 00 00 mov a, %eax
```

```
A3 00 00 00 00 mov %eax, b
```



# Example

- Save a into b, e.g.,  $b = a$

`mov a, %eax`

- 1 byte opcode

generated code

a is defined in the same file at 0x1234, **b is imported**

Each instruction is 1 byte opcode + 4 bytes address

**A1** 34 12 00 00 `mov a, %eax`

A3 00 00 00 00 `mov %eax, b`

# Example

- Save a into b, e.g.,  $b = a$

`mov a, %eax`

- 4 byte address

Generate code

- a is defined in the same file at 0x1234, **b is imported**
- Each instruction is 1 byte opcode + 4 bytes address

A1 34 12 00 00 `mov a, %eax`

A3 `00 00 00 00` `mov %eax, b`

# Example

- Save a into b, e.g.,  $b = a$

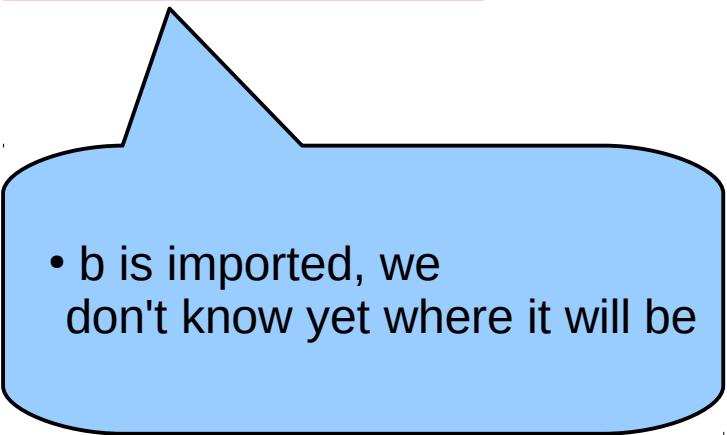
```
mov a, %eax
```

```
mov %eax, b
```

- Generated code
  - a is defined in the same file at 0x1234, **b is imported**
  - Each instruction is 1 byte opcode + 4 bytes address

```
A1 34 12 00 00 mov a, %eax
```

```
A3 00 00 00 00 mov %eax, b
```

- 
- b is imported, we don't know yet where it will be

# Example

- Save a into b, e.g.,  $b = a$

```
mov a, %eax
```

```
mov %eax, b
```

- Generated code

- a is defined in the same file at 0x1234, **b is imported**

- Each instruction is 1 byte opcode + 4 bytes address

```
A1 34 12 00 00 mov a, %eax
```

```
A3 00 00 00 00 mov %eax, b
```

- Assume that a is relocated by 0x10000 bytes, and b is found at 0x9a12

```
A1 34 12 01 00 mov a, %eax
```

```
A3 12 9A 00 00 mov %eax, b
```

# Example

- Save a into b, e.g.,  $b = a$

```
mov a, %eax
```

```
mov %eax, b
```

- Generated code

- a is defined in the same file at 0x1234, **b is imported**

- Each instruction is 1 byte opcode + 4 bytes address

```
A1 34 12 00 00 mov a, %eax
```

```
A3 00 00 00 00 mov %eax, b
```

- Assume that a is relocated by 0x10000 bytes, and b is found at 0x9a12

```
A1 34 12 01 00 mov a, %eax
```

```
A3 12 9A 00 00 mov %eax, b
```

# More realistic example

- Source file m.c

```
1  extern void a(char *);
2  int main(int ac, char **av)
3  {
4      static char string[] = "Hello, world!\n";
5      a(string);
6  }
```

- Source file a.c

```
1  #include <unistd.h>
2  #include <string.h>
3  void a(char *s)
4  {
5      write(1, s, strlen(s));
6  }
```

# More realistic example

- Source file m.c

```
1  extern void a(char *);
2  int main(int ac, char **av)
3  {
4      static char string[] = "Hello, world!\n";
5      a(string);
6  }
```

- Source file a.c

```
1  #include <unistd.h>
2  #include <string.h>
3  void a(char *s)
4  {
5      write(1, s, strlen(s));
6  }
```

# More realistic example

- Source file m.c

```
1  extern void a(char *);
2  int main(int ac, char **av)
3  {
4      static char string[] = "Hello, world!\n";
5      a(string);
6  }
```

- Source file a.c

```
1  #include <unistd.h>
2  #include <string.h>
3  void a(char *s)
4  {
5      write(1, s, strlen(s));
6  }
```



# More realistic example

Sections:

Idx	Name	Size	VMA	LMA	File off	Algn
0	.text	00000010	00000000	00000000	00000020	2**3
1	.data	00000010	00000010	00000010	00000030	2**3

Disassembly of section .text:

00000000 <\_main>:

```
0: 55                pushl %ebp
1: 89 e5            movl %esp,%ebp
3: 68 10 00 00 00  pushl $0x10
4: 32 .data
8: e8 f3 ff ff ff  call 0
9: DISP32 _a
d: c9                leave
e: c3                ret
...
```

# More realistic example

- Two sections:
  - Text (0x10 – 16 bytes)
  - Data ( 16 bytes)

Sections:

Idx	Name	Size	VMA	LMA	File off	Algn
0	.text	00000010	00000000	00000000	00000020	2**3
1	.data	00000010	00000010	00000010	00000030	2**3

Disassembly of section .text:

00000000 <\_main>:

```
0: 55                pushl %ebp
1: 89 e5             movl %esp,%ebp
3: 68 10 00 00 00    pushl $0x10
4: 32 .data
8: e8 f3 ff ff ff    call 0
9: DISP32 _a
d: c9                leave
e: c3                ret
...
```

# More realistic example

- Two sections:
  - Text starts at 0x0
  - Data starts at 0x10

Sections:

Idx	Name	Size	VMA	LMA	File off	Algn
0	.text	00000010	00000000	00000000	00000020	2**3
1	.data	00000010	00000010	00000010	00000030	2**3

Disassembly of section .text:

00000000 <\_main>:

```
0: 55                pushl %ebp
1: 89 e5             movl %esp,%ebp
3: 68 10 00 00 00    pushl $0x10
4: 32 .data
8: e8 f3 ff ff ff    call 0
9: DISP32 _a
d: c9                leave
e: c3                ret
...
```

# More realistic example

Sections:

Idx	Name	Size	VMA	LMA	File off	Algn
0	.text		00000000	00000000	00000020	2**3
1	.data		00000010	00000010	00000030	2**3

• Code starts at 0x0

Disassembly of section .text:

00000000 <\_main>:

```
0: 55                pushl %ebp
1: 89 e5             movl %esp,%ebp
3: 68 10 00 00 00    pushl $0x10
4: 32 .data
8: e8 f3 ff ff ff    call 0
9: DISP32 _a
d: c9              leave
e: c3              ret
...
```

# More realistic example

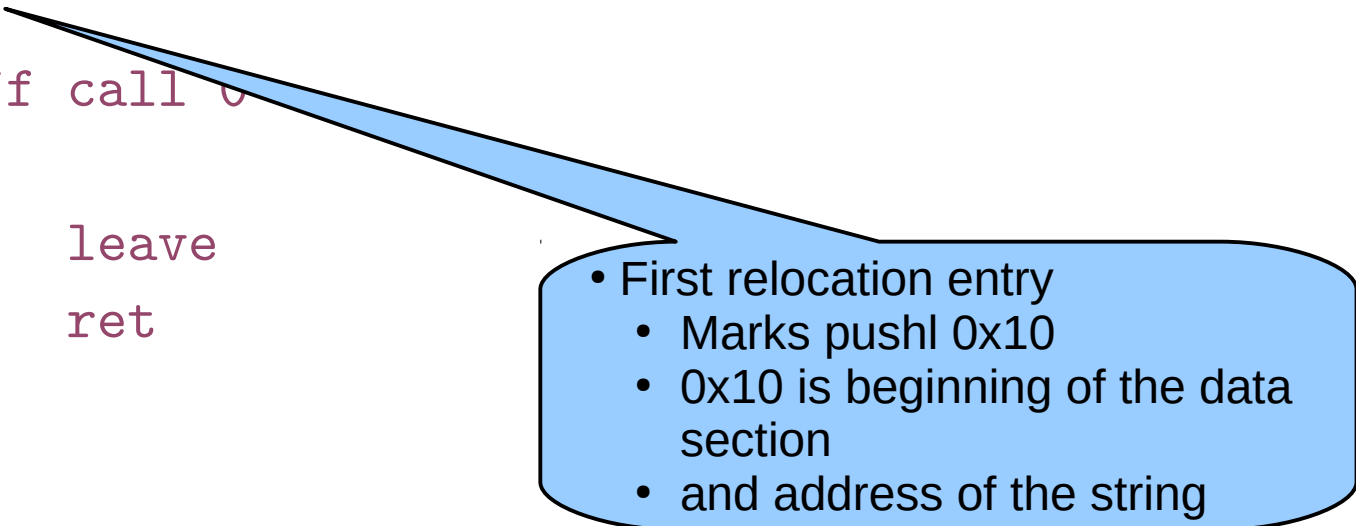
Sections:

Idx	Name	Size	VMA	LMA	File off	Algn
0	.text	00000010	00000000	00000000	00000020	2**3
1	.data	00000010	00000010	00000010	00000030	2**3

Disassembly of section .text:

00000000 <\_main>:

```
0: 55                pushl %ebp
1: 89 e5             movl %esp,%ebp
3: 68 10 00 00 00    pushl $0x10 # push string on the stack
4: 32 .data
8: e8 f3 ff ff ff    call 0
9: DISP32 _a
d: c9              leave
e: c3              ret
...
```

- 
- First relocation entry
  - Marks pushl 0x10
  - 0x10 is beginning of the data section
  - and address of the string

# More realistic example

- Source file m.c

```
1  extern void a(char *);
2  int main(int ac, char **av)
3  {
4      static char string[] = "Hello, world!\n";
5      a(string);
6  }
```

- Source file a.c

```
1  #include <unistd.h>
2  #include <string.h>
3  void a(char *s)
4  {
5      write(1, s, strlen(s));
6  }
```

# More realistic example

Sections:

Idx	Name	Size	VMA	LMA	File off	Algn
0	.text	00000010	00000000	00000000	00000020	2**3
1	.data	00000010	00000010	00000010	00000030	2**3

Disassembly of section .text:

00000000 <\_main>:

```
0: 55                pushl %ebp
1: 89 e5             movl %esp,%ebp
3: 68 10 00 00 00    pushl $0x10
4: 32 .data
8: e8 f3 ff ff ff    call 0
9: DISP32 _a
d: c9                leave
e: c3                ret
...
```

- Second relocation entry
  - Marks call
  - 0x0 – address is unknown

# More realistic example

- Source file m.c

```
1  extern void a(char *);
2  int main(int ac, char **av)
3  {
4      static char string[] = "Hello, world!\n";
5      a(string);
6  }
```

- Source file a.c

```
1  #include <unistd.h>
2  #include <string.h>
3  void a(char *s)
4  {
5      write(1, s, strlen(s));
6  }
```



# More realistic example

Sections:

Idx	Name	Size	VMA	LMA	File off	Algn
0	.text	0000001c	00000000	00000000	00000020	2**2
	CONTENTS, ALLOC, LOAD, RELOC, CODE					
1	.data	00000000	0000001c	0000001c	0000003c	2**2
	CONTENTS, ALLOC, LOAD, DATA					

Disassembly of section .text:

00000000 <\_a>:

```
0: 55                pushl %ebp
1: 89 e5             movl %esp,%ebp
3: 53                pushl %ebx
4: 8b 5d 08          movl 0x8(%ebp),%ebx
7: 53                pushl %ebx
8: e8 f3 ff ff ff    call 0
9: DISP32 _strlen
d: 50                pushl %eax
e: 53                pushl %ebx
f: 6a 01            pushl $0x1
11: e8 ea ff ff ff    call 0
12: DISP32 _write
16: 8d 65 fc          leal -4(%ebp),%esp
19: 5b                popl %ebx
1a: c9                leave
1b: c3                ret
```

- Two sections:
  - Text (0 bytes)
  - Data (28 bytes)

# More realistic example

- Source file m.c

```
1  extern void a(char *);
2  int main(int ac, char **av)
3  {
4      static char string[] = "Hello, world!\n";
5      a(string);
6  }
```

- Source file a.c

```
1  #include <unistd.h>
2  #include <string.h>
3  void a(char *s)
4  {
5      write(1, s, strlen(s));
6  }
```

# More realistic example

Sections:

Idx	Name	Size	VMA	LMA	File off	Algn
0	.text	0000001c	00000000	00000000	00000020	2**2
	CONTENTS, ALLOC, LOAD, RELOC, CODE					
1	.data	00000000	0000001c	0000001c	0000003c	2**2
	CONTENTS, ALLOC, LOAD, DATA					

Disassembly of section .text:

00000000 <\_a>:

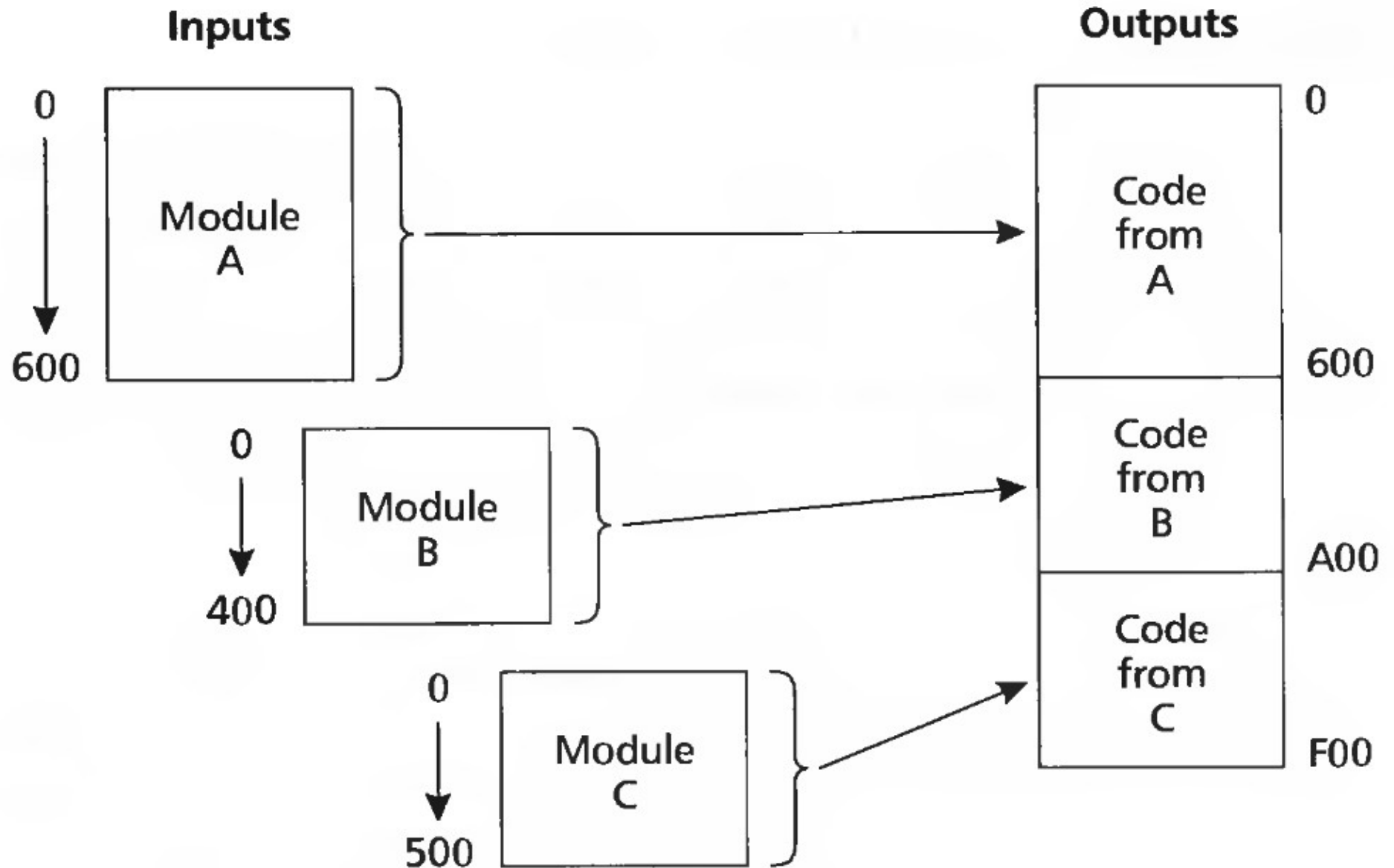
```
0: 55                pushl %ebp
1: 89 e5             movl %esp,%ebp
3: 53                pushl %ebx
4: 8b 5d 08          movl 0x8(%ebp),%ebx
7: 53                pushl %ebx
8: e8 f3 ff ff ff    call 0
  9: DISP32 _strlen
d: 50                pushl %eax
e: 53                pushl %ebx
f: 6a 01            pushl $0x1
11: e8 ea ff ff ff    call 0
 12: DISP32 _write
16: 8d 65 fc          leal -4(%ebp),%esp
19: 5b                popl %ebx
1a: c9                leave
1b: c3                ret
```

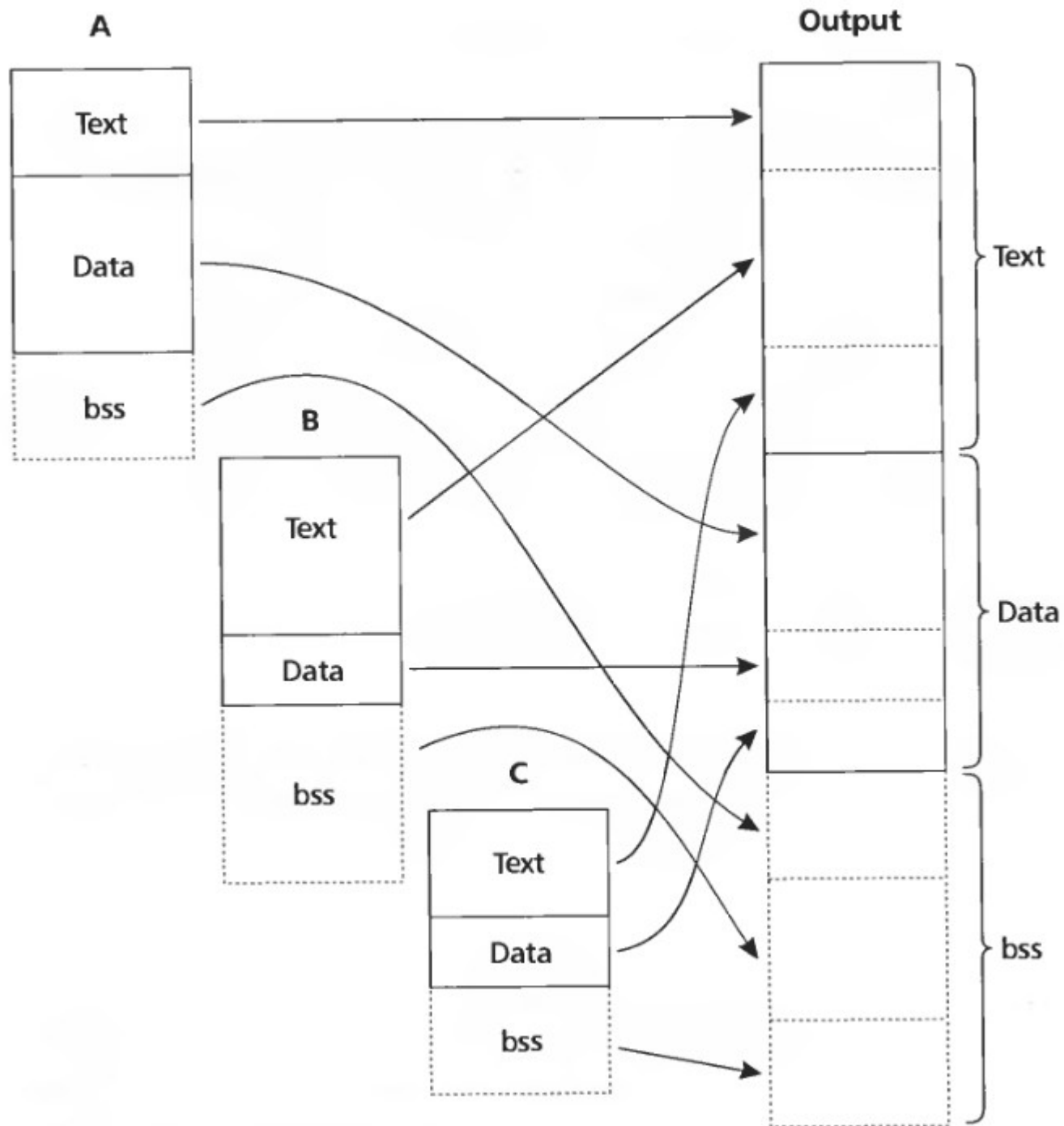
- Two relocation entries:
  - strlen()
  - write()

# Producing an executable

- Combine corresponding segments from each object file
  - Combined text segment
  - Combined data segment
- Pad each segment to 4KB to match the page size

# Multiple object files





# Merging segments

## Sections:

Idx	Name	Size	VMA	LMA	File off	Algn
0	.text	00000fe0	00001020	00001020	00000020	2**3
1	.data	00001000	00002000	00002000	00001000	2**3
2	.bss	00000000	00003000	00003000	00000000	2**3

## Disassembly of section .text:

00001020 <start-c>:

...

1092: e8 0d 00 00 00 call 10a4 <\_main>

...

000010a4 <\_main>:

10a7: 68 24 20 00 00 pushl \$0x2024

10ac: e8 03 00 00 00 call 10b4 <\_a>

...

000010b4 <\_a>:

10bc: e8 37 00 00 00 call 10f8 <\_strlen>

...

10c3: 6a 01 pushl \$0x1

10c5: e8 a2 00 00 00 call 116c <\_write>

...

000010f8 <\_strlen>:

...

0000116c <\_write>:

...

# Linked executable

## Sections:

Idx	Name	Size	VMA	LMA	File off	Algn
0	.text	00000fe0	00001020	00001020	00000020	2**3
1	.data	00001000	00002000	00002000	00001000	2**3
2	.bss	00000000	00003000	00003000	00000000	2**3

## Disassembly of section .text:

00001020 <start-c>:

...

1092: e8 0d 00 00 00 call 10a4 <\_main>

...

000010a4 <\_main>:

10a7: 68 24 20 00 00 pushl \$0x2024

10ac: e8 03 00 00 00 call 10b1 <\_a>

...

000010b4 <\_a>:

10bc: e8 37 00 00 00 call 10f8 <\_strlen>

...

10c3: 6a 01 pushl \$0x1

10c5: e8 a2 00 00 00 call 116c <\_write>

...

000010f8 <\_strlen>:

...

0000116c <\_write>:

...

- Relative to EIP address
- Hence 3

# Linked executable



# x86 Call instruction

## x86 Instruction Set Reference

### CALL

#### Call Procedure

Opcode	Mnemonic	Description
E8 cw	CALL rel16	Call near, relative, displacement relative to next instruction
E8 cd	CALL rel32	Call near, relative, displacement relative to next instruction
FF /2	CALL r/m16	Call near, absolute indirect, address given in r/m16
FF /2	CALL r/m32	Call near, absolute indirect, address given in r/m32
9A cd	CALL ptr16:16	Call far, absolute, address given in operand
9A cp	CALL ptr16:32	Call far, absolute, address given in operand
FF /3	CALL m16:16	Call far, absolute indirect, address given in m16:16
FF /3	CALL m16:32	Call far, absolute indirect, address given in m16:32

Description
Saves procedure linking information on the stack and branches to the procedure (called procedure) specified with the destination (target) operand. The target operand specifies the address of the first instruction in the called procedure. This operand can be an immediate value, a generalpurpose register, or a memory location.

# Tasks involved

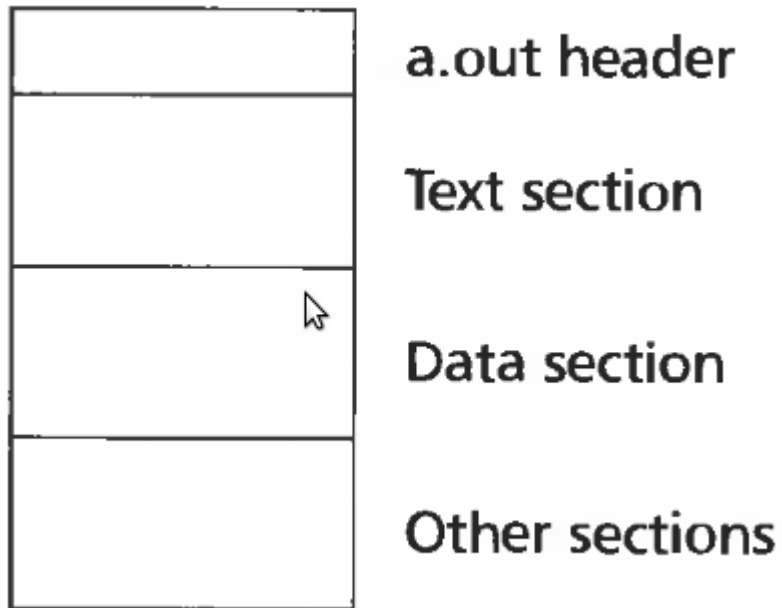
- Program loading
  - Copy a program from disk to memory so it is ready to run
    - Allocation of memory
    - Setting protection bits (e.g. read only)
- Relocation
  - Assign load address to each object file
  - Adjust the code
- Symbol resolution
  - Resolve symbols imported from other object files

Object files

# Object files

- Conceptually: five kinds of information
  - Header: code size, name of the source file, creation date
  - Object code: binary instruction and data generated by the compiler
  - Relocation information: list of places in the object code that need to be patched
  - Symbols: global symbols defined by this module
    - Symbols to be imported from other modules
  - Debugging information: source file and file number information, local symbols, data structure description

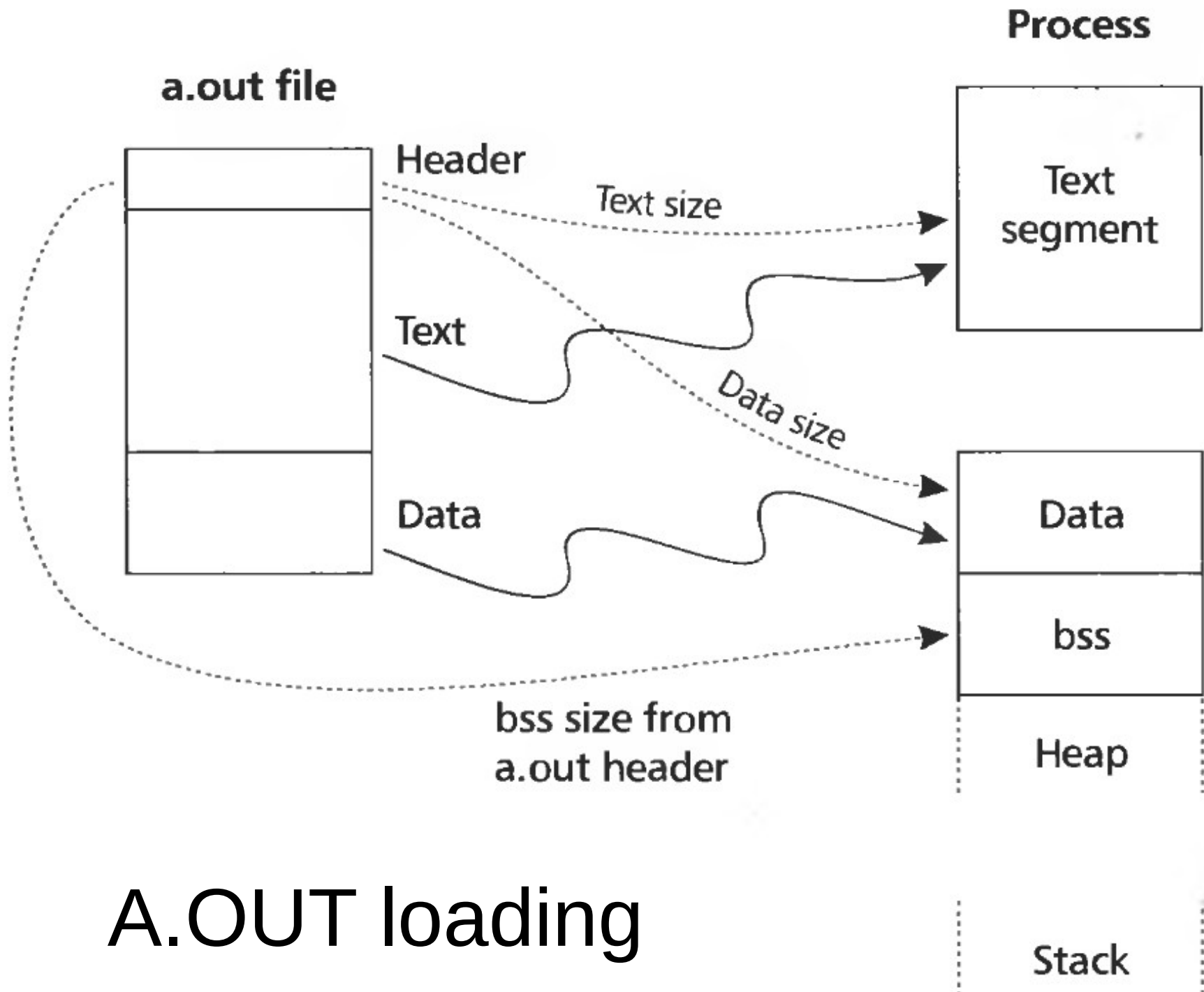
# Example: UNIX A.OUT



- Small header
- Text section
  - Executable code
- Data section
  - Initial values for static data

- A.OUT header

```
int a_magic;    // magic number
int a_text;     // text segment size
int a_data;     // initialized data size
int a_bss;      // uninitialized data size
int a_syms;     // symbol table size
int a_entry;    // entry point
int a_trsize;   // text relocation size
int a_drsize;   // data relocation size
```



# A.OUT loading

# A.OUT loading

- Read the header to get segment sizes
- Check if there is a shareable code segment for this file
  - If not, create one,
  - Map into the address space,
  - Read segment from a file into the address space
- Create a private data segment
  - Large enough for data and BSS
  - Read data segment, zero out the BSS segment
- Create and map stack segment
  - Place arguments from the command line on the stack
- Jump to the entry point



# Types of object files

- Relocatable object files (.o)
  - Static libraries (.a)
  - Shared libraries (.so)
  - Executable files
- 
- We looked at A.OUT, but Unix has a general format capable to hold all of these files

# ELF

## Elf header

- Magic number, type (.o, exec, .so), machine, byte ordering, etc.

## Segment header table

- Page size, virtual addresses memory segments (sections), segment sizes.

## .text section

- Code

## .data section

- Initialized global variables

## .bss section

- Uninitialized global variables
- “Block Started by Symbol”
- “Better Save Space”
- Has section header but occupies no space

0	ELF header
	Segment header table (required for executables)
	.text section
	.data section
	.bss section
	.symtab section
	.rel.txt section
	.rel.data section
	.debug section
	Section header table

# ELF (continued)

## `.symtab` section

- Symbol table
- Procedure and static variable names
- Section names and locations

## `.rel.text` section

- Relocation info for `.text` section
- Addresses of instructions that will need to be modified in the executable
- Instructions for modifying.

## `.rel.data` section

- Relocation info for `.data` section
- Addresses of pointer data that will need to be modified in the merged executable

## `.debug` section

- Info for symbolic debugging (`gcc -g`)

## Section header table

- Offsets and sizes of each section

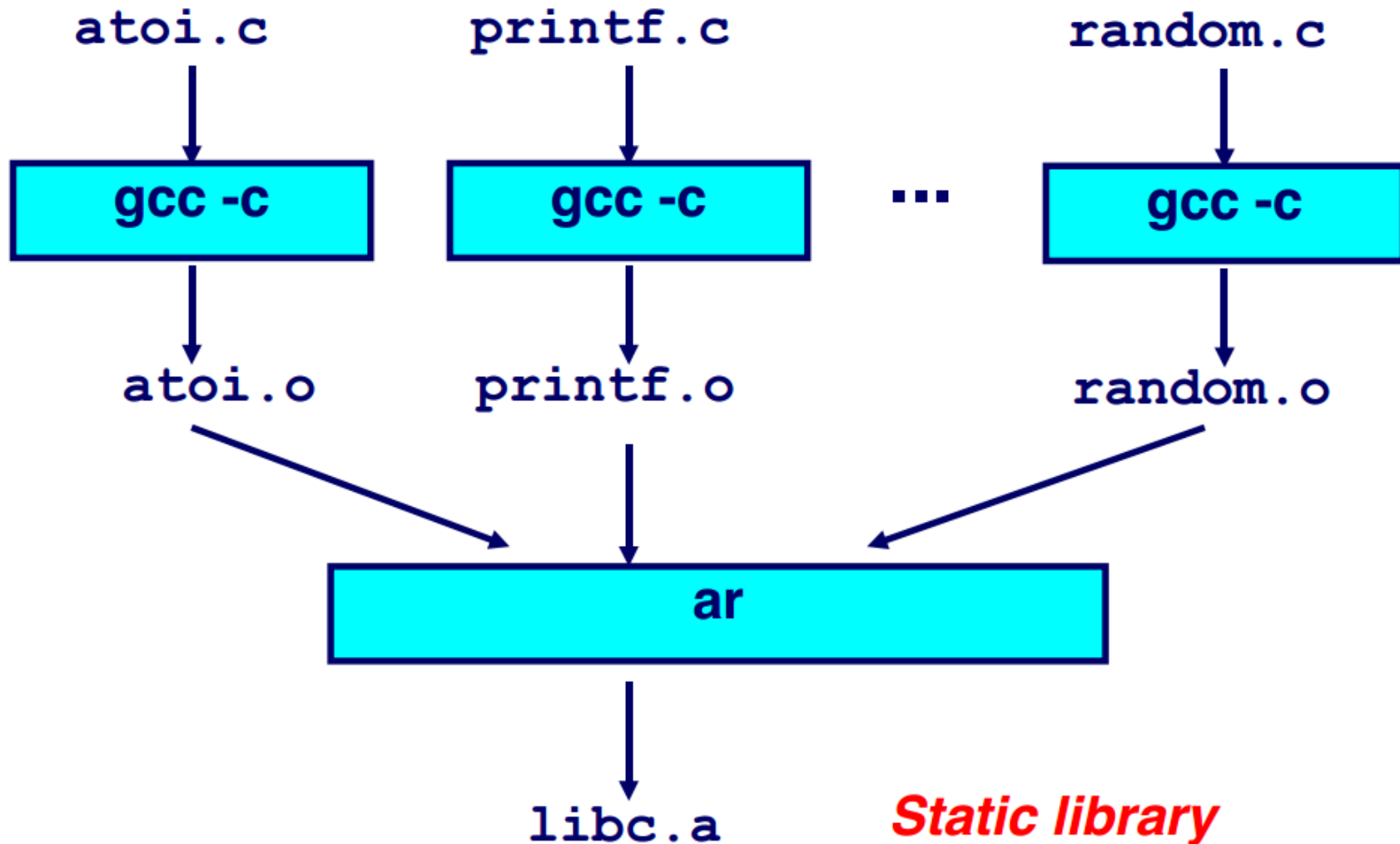
0	ELF header
	Segment header table (required for executables)
	<code>.text</code> section
	<code>.data</code> section
	<code>.bss</code> section
	<code>.symtab</code> section
	<code>.rel.text</code> section
	<code>.rel.data</code> section
	<code>.debug</code> section
	Section header table

# Static libraries

# Libraries

- Conceptually a library is
  - Collection of object files
- UNIX uses an archive format
  - Remember the **ar** tool
  - Can support collections of any objects
  - Rarely used for anything instead of libraries

# Creating a static library



# Example

- Create a library

```
ar rcs libclass.a class1.o class2.o class3.o
```

- Linking (linker can read ar files)
  - C compiler calls linker

```
cc main.c libclass.a
```

- or (if libclass.a is placed in standard library path, like /usr/local/lib)

```
cc main.c -lclass
```

- or (during linking)

```
ld ... main.o -lclass ...
```

- is the same as:

```
cc main.c class1.o class2.o class3.o
```

# Searching libraries

- First linker path needs resolve symbol names into function locations
- To improve the search library formats add an index
  - Map names to member positions



Shared libraries  
(.so or .dll)

# Motivation

- 1000 programs in a typical UNIX system
- 1000 copies of `printf`
- **How big is `printf()` actually?**

# Motivation

- Disk space
  - 2504 programs in /usr/bin on my Linux laptop
    - `ls /usr/bin | wc -l`
  - **printf() is a large function**
  - **Handles conversion of multiple types to strings**
    - **5-10K**
  - This means 10-25MB of disk can be wasted just on `printf()`
- Runtime memory costs are
  - 5-10K times the number of running programs
  - 250 programs running on my Linux laptop
    - `ps -aux | wc -l`
    - 1MB-2.5MB – huge number for most systems 15-20 years ago

# Shared libraries

- Motivation
  - Share code of a library across all processes
    - E.g. libc is linked by all processes in the system
  - Code section should remain identical
    - To be shared read-only
  - What if library is loaded at different addresses?
    - Remember it needs to be relocated

# Example: size of a statically vs dynamically linked program

- On Ubuntu 16.04 (gcc 5.4.0, libc 2.23)
  - Statically linked trivial example
    - `gcc -m32 -static hello-int.c -o test`
    - 725KB
  - Dynamically linked trivial example
    - `gcc -m32 hello-int.c -o test`
    - 7KB

# Motivation for shared libraries

# Position independent code

(Parts adapted from Eli Bendersky)

<https://eli.thegreenplace.net/2011/11/03/position-independent-code-pic-in-shared-libraries/>

# Position independent code

- Motivation
  - Share code of a library across all processes
    - E.g. libc is linked by all processes in the system
  - Code section should remain identical
    - To be shared read-only
  - What if library is loaded at different addresses?
    - Remember it needs to be relocated



# Position independent code (PIC)

- Main idea:
  - Generate code in such a way that it can work no matter where it is located in the address space
  - Share code across all address spaces

# What needs to be changed?

- Can stay untouched
  - Local jumps and calls are relative
  - Stack data is relative to the stack
- Needs to be modified
  - Global variables
  - Imported functions

# Example

```
000010a4 <_main>:
  10a4: 55                pushl %ebp
  10a5: 89 e5            movl %esp,%ebp
  10a7: 68 10 00 00 00   pushl $0x10
  10a8: 32 .data
  10ac: e8 03 00 00 00   call 10b4 <_a>
...
000010b4 <_a>:
  10bc: e8 37 00 00 00   call 10f8 <_strlen>
...
  10c3: 6a 01           pushl $0x1
  10c5: e8 a2 00 00 00   call 116c <_write>
...
```

- Reference to a data section
  - Code and data sections can be moved around

# Example

```
000010a4 <_main>:
  10a4: 55                pushl %ebp
  10a5: 89 e5            movl %esp,%ebp
  10a7: 68 10 00 00 00   pushl $0x10
  10a8: 32 .data
  10ac: e8 03 00 00 00   call 10b4 <_a>
  ...
000010b4 <_a>:
  10bc: e8 37 00 00 00   call 10f8 <_strlen>
  ...
  10c3: 6a 01           pushl $0x1
  10c5: e8 a2 00 00 00   call 116c <_write>
  ...
```

- Local function invocations use relative addresses
- No need to relocate

# Position independent code

- How would you build it?

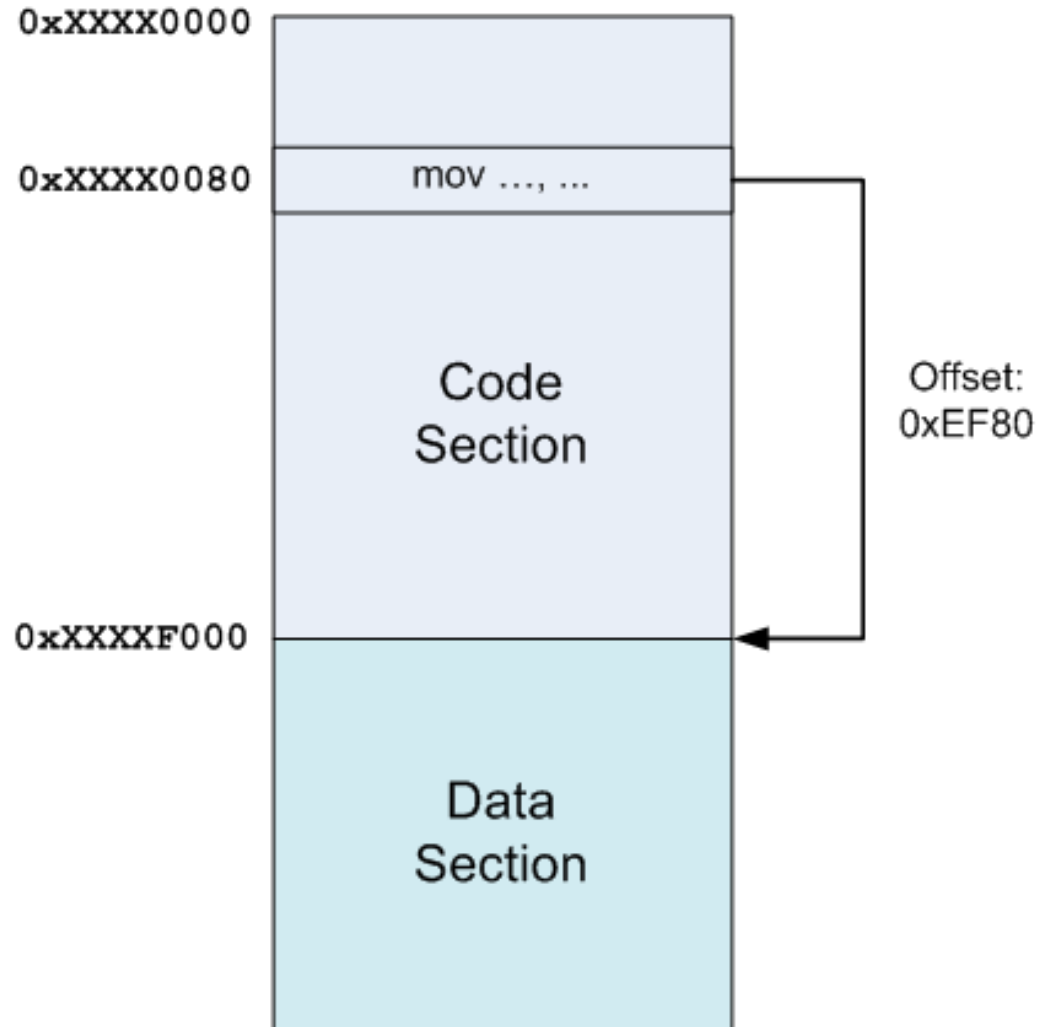
# Position independent code

- How would you build it?
- Main idea:
  - Add additional layer of indirection to all
    - Global data
    - Function
    - ...references in the code

# Position independent code

- Main insight
  - Code sections are followed by data sections
  - The distance between code and data **remains constant even if code is relocated**
    - Linker knows the distance
    - Even if it combines multiple code sections together

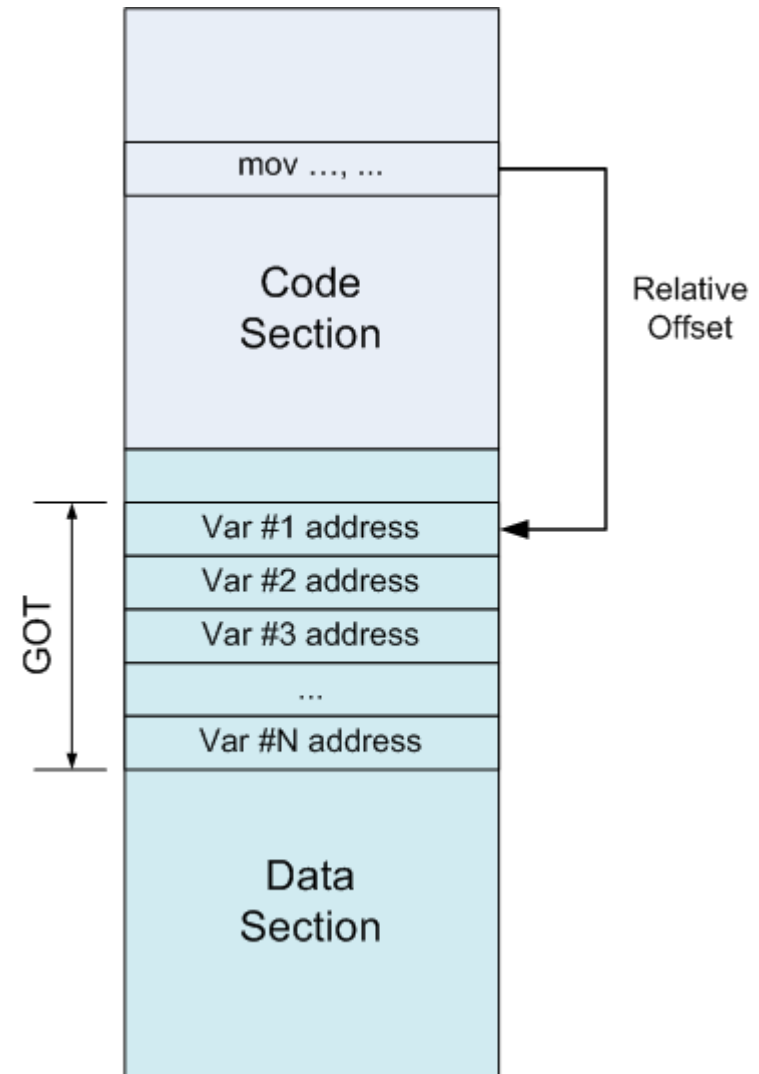
# Insight 1: Constant offset between text and data sections





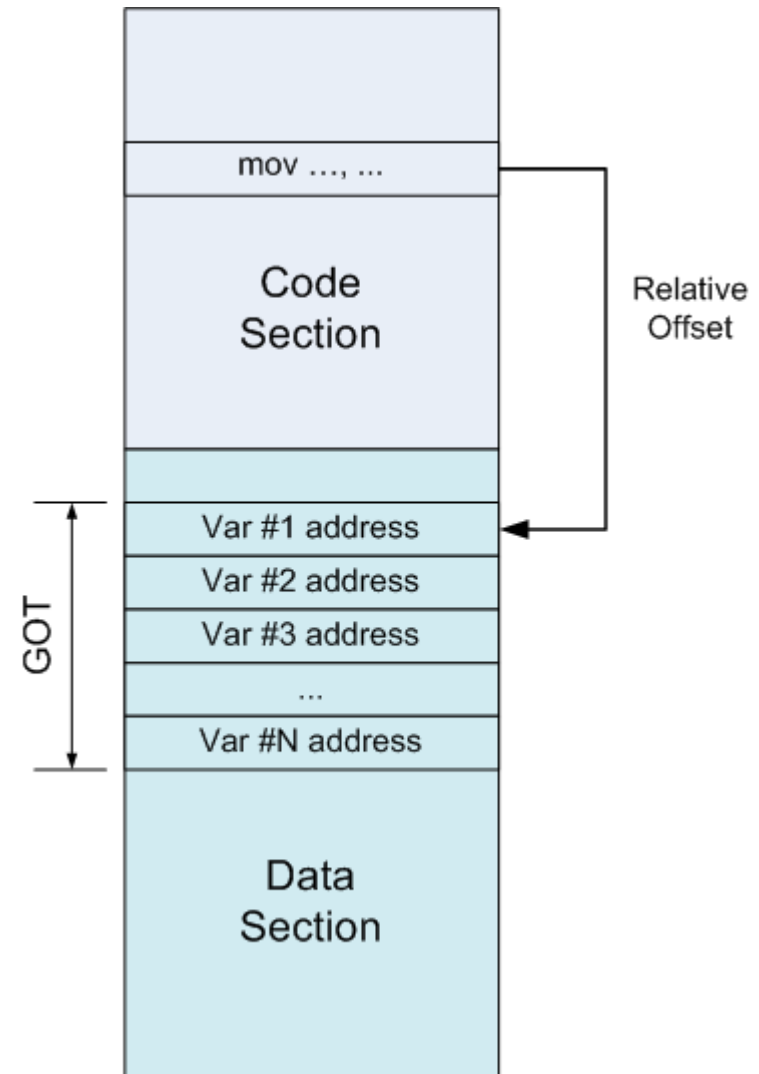
# Global offset table (GOT)

- Insight #2:
  - Instead of referring to a variable by its absolute address
    - Which would require a relocation
  - Refer through GOT



# Global offset table (GOT)

- GOT
  - Table of addresses
  - Each entry contains absolute address of a variable
  - GOT is patched by the linker at relocation time



How to find position of the code in memory at run time?

# How to find position of the code in memory at run time?

- Is there an x86 instruction that does this?
  - i.e., give me my current code address
- x86 32bit architecture requires absolute addresses for `mov` instructions
  - No relative addresses allowed
- There is no instruction to learn the value of EIP
  - Instruction pointer

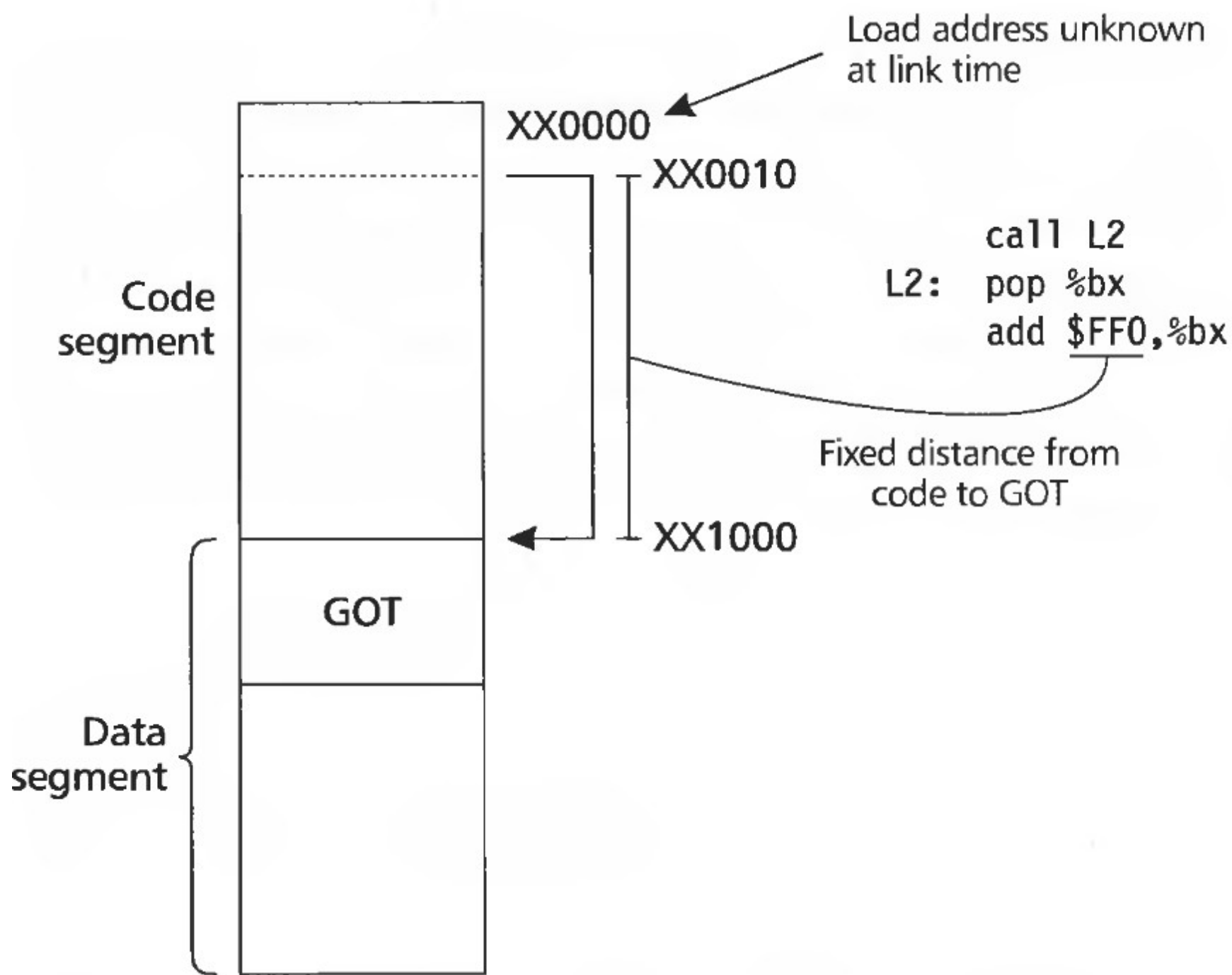
# How to find position of the code in memory at run time?

- Simple trick

```
call L2
```

```
L2: popl %ebx
```

- Call next instruction
  - Saves EIP on the stack
  - EIP holds current position of the code
  - Use popl to fetch EIP into a register



# Examples of position independent code

```
int myglob = 42;
```

```
int ml_func(int a, int b)
{
    return myglob + a + b;
}
```

```
0000043c <ml_func>:
```

43c:	55	push	ebp
43d:	89 e5	mov	ebp,esp
43f:	e8 16 00 00 00	call	45a <__i686.get_pc_thunk.cx>
444:	81 c1 b0 1b 00 00	add	ecx,0x1bb0
44a:	8b 81 f0 ff ff ff	mov	eax,DWORD PTR [ecx-0x10]
450:	8b 00	mov	eax,DWORD PTR [eax]
452:	03 45 08	add	eax,DWORD PTR [ebp+0x8]
455:	03 45 0c	add	eax,DWORD PTR [ebp+0xc]
458:	5d	pop	ebp
459:	c3	ret	

```
0000045a <__i686.get_pc_thunk.cx>:
```

45a:	8b 0c 24	mov	ecx,DWORD PTR [esp]
45d:	c3	ret	

# PIC example



```
int myglob = 42;
```

```
int ml_func(int a, int b)
{
    return myglob + a + b;
}
```

# PIC example

- Access a global variable **myglob**

```
0000043c <ml_func>:
```

43c:	55	push	ebp
43d:	89 e5	mov	ebp,esp
43f:	e8 16 00 00 00	call	45a <__i686.get_pc_thunk.cx>
444:	81 c1 b0 1b 00 00	add	ecx,0x1bb0
44a:	8b 81 f0 ff ff ff	mov	eax,DWORD PTR [ecx-0x10]
450:	8b 00	mov	eax,DWORD PTR [eax]
452:	03 45 08	add	eax,DWORD PTR [ebp+0x8]
455:	03 45 0c	add	eax,DWORD PTR [ebp+0xc]
458:	5d	pop	ebp
459:	c3	ret	

```
0000045a <__i686.get_pc_thunk.cx>:
```

45a:	8b 0c 24	mov	ecx,DWORD PTR [esp]
45d:	c3	ret	

```
int myglob = 42;
```

```
int ml_func(int a, int b)
{
    return myglob + a + b;
}
```

# PIC example

- Save EIP into ECX

0000043c <ml\_func>:

43c:	55	push	ebp
43d:	89 e5	mov	ebp,esp
43f:	e8 16 00 00 00	call	45a <__i686.get_pc_thunk.cx>
444:	81 c1 b0 1b 00 00	add	ecx,0x1bb0
44a:	8b 81 f0 ff ff ff	mov	eax,DWORD PTR [ecx-0x10]
450:	8b 00	mov	eax,DWORD PTR [eax]
452:	03 45 08	add	eax,DWORD PTR [ebp+0x8]
455:	03 45 0c	add	eax,DWORD PTR [ebp+0xc]
458:	5d	pop	ebp
459:	c3	ret	

0000045a <\_\_i686.get\_pc\_thunk.cx>:

45a:	8b 0c 24	mov	ecx,DWORD PTR [esp]
45d:	c3	ret	

```
int myglob = 42;
```

```
int ml_func(int a, int b)
{
    return myglob + a + b;
}
```

```
0000043c <ml_func>:
```

43c:	55	push	ebp
43d:	89 e5	mov	ebp,esp
43f:	e8 16 00 00 00	call	45a <__i686.get_pc_thunk.cx>
444:	81 c1 b0 1b 00 00	add	ecx,0x1bb0
44a:	8b 81 f0 ff ff ff	mov	eax,DWORD PTR [ecx-0x10]
450:	8b 00	mov	eax,DWORD PTR [eax]
452:	03 45 08	add	eax,DWORD PTR [ebp+0x8]
455:	03 45 0c	add	eax,DWORD PTR [ebp+0xc]
458:	5d	pop	ebp
459:	c3	ret	

```
0000045a <__i686.get_pc_thunk.cx>:
```

45a:	8b 0c 24	mov	ecx,DWORD PTR [esp]
45d:	c3	ret	

# PIC example

- Add offset to GOT
  - 0x1bb0

```
int myglob = 42;
```

```
int ml_func(int a, int b)
{
    return myglob + a + b;
}
```

# PIC example

- Access address of a specific GOT entry (address of `myglob`)
- Save it in EAX

```
0000043c <ml_func>:
```

43c:	55	push	ebp
43d:	89 e5	mov	ebp,esp
43f:	e8 16 00 00 00	call	45a <__i686.get_pc_thunk.cx>
444:	81 c1 b0 1b 00 00	add	ecx,0x1bb0
44a:	8b 81 f0 ff ff ff	mov	eax,DWORD PTR [ecx-0x10]
450:	8b 00	mov	eax,DWORD PTR [eax]
452:	03 45 08	add	eax,DWORD PTR [ebp+0x8]
455:	03 45 0c	add	eax,DWORD PTR [ebp+0xc]
458:	5d	pop	ebp
459:	c3	ret	

```
0000045a <__i686.get_pc_thunk.cx>:
```

45a:	8b 0c 24	mov	ecx,DWORD PTR [esp]
45d:	c3	ret	

```
int myglob = 42;
```

```
int ml_func(int a, int b)
{
    return myglob + a + b;
}
```

```
0000043c <ml_func>:
```

43c:	55	push	ebp
43d:	89 e5	mov	ebp,esp
43f:	e8 16 00 00 00	call	45a <__i686.get_pc_thunk.cx>
444:	81 c1 b0 1b 00 00	add	ecx,0x1bb0
44a:	8b 81 f0 ff ff ff	mov	eax,DWORD PTR [ecx-0x10]
450:	8b 00	mov	eax,DWORD PTR [eax]
452:	03 45 08	add	eax,DWORD PTR [ebp+0x8]
455:	03 45 0c	add	eax,DWORD PTR [ebp+0xc]
458:	5d	pop	ebp
459:	c3	ret	

```
0000045a <__i686.get_pc_thunk.cx>:
```

45a:	8b 0c 24	mov	ecx,DWORD PTR [esp]
45d:	c3	ret	

# PIC example

- Load the value of the variable at the address pointed by EAX
  - i.e., load `myglob` into EAX

# What about function calls?

# What about function calls?

- Same approach can work
- But this is not how it is done

# Late binding

- When a shared library refers to some function, the real address of that function is not known until load time
  - Resolving this address is called binding
- We can use GOT
  - Same as for variables



# Lazy procedure binding

- In large libraries many routines are never called
  - Libc has over 600
    - **The number of functions is much larger than the number of global variables**
  - It's ok to bind all routines when the program is **statically linked**
    - Binding is done offline, no runtime cost
  - But with **dynamic linking** run-time overhead is too high
    - **Lazy approach, i.e., linking only when used, works better**

# Procedure linkage table (PLT)

Code:

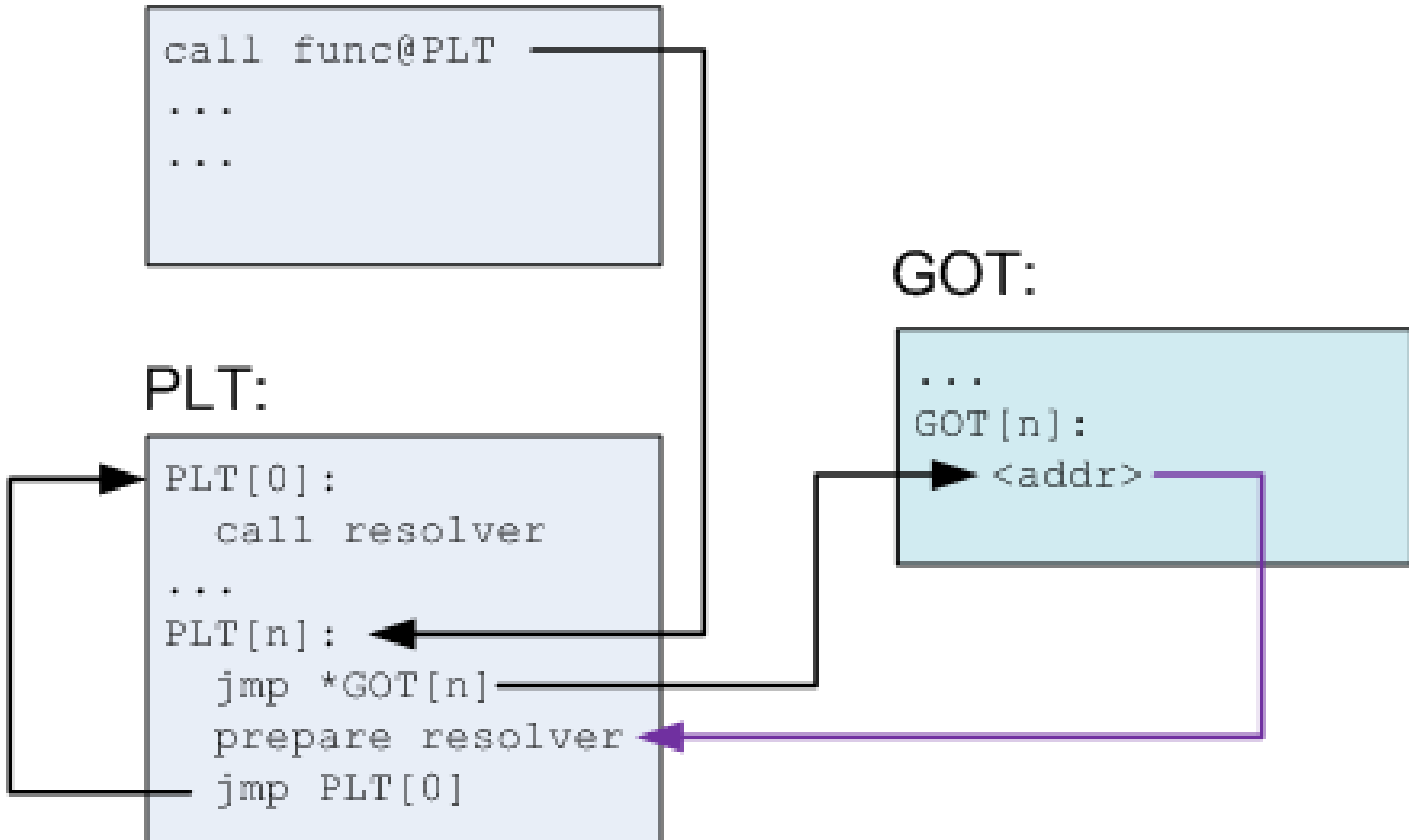
```
call func@PLT  
...  
...
```

PLT:

```
PLT[0]:  
    call resolver  
...  
PLT[n]:  
    jmp *GOT[n]  
    prepare resolver  
    jmp PLT[0]
```

GOT:

```
...  
GOT[n]:  
    <addr>
```



# Procedure linkage table (PLT)

- PLT is part of the executable text section
  - A set of entries
    - A special first entry
    - One for each external function
- Each PLT entry
  - Is a short chunk of executable code
  - Has a corresponding entry in the GOT
    - Contains an actual offset to the function
    - Only after it is resolved by the dynamic loader

- Each PLT entry but the first consists of these parts:
  - A jump to a location which is specified in a corresponding GOT entry
  - Preparation of arguments for a "resolver" routine
  - Call to the resolver routine, which resides in the first entry of the PLT

# Before function is resolved

- Nth GOT entry points to after the jump

Code:

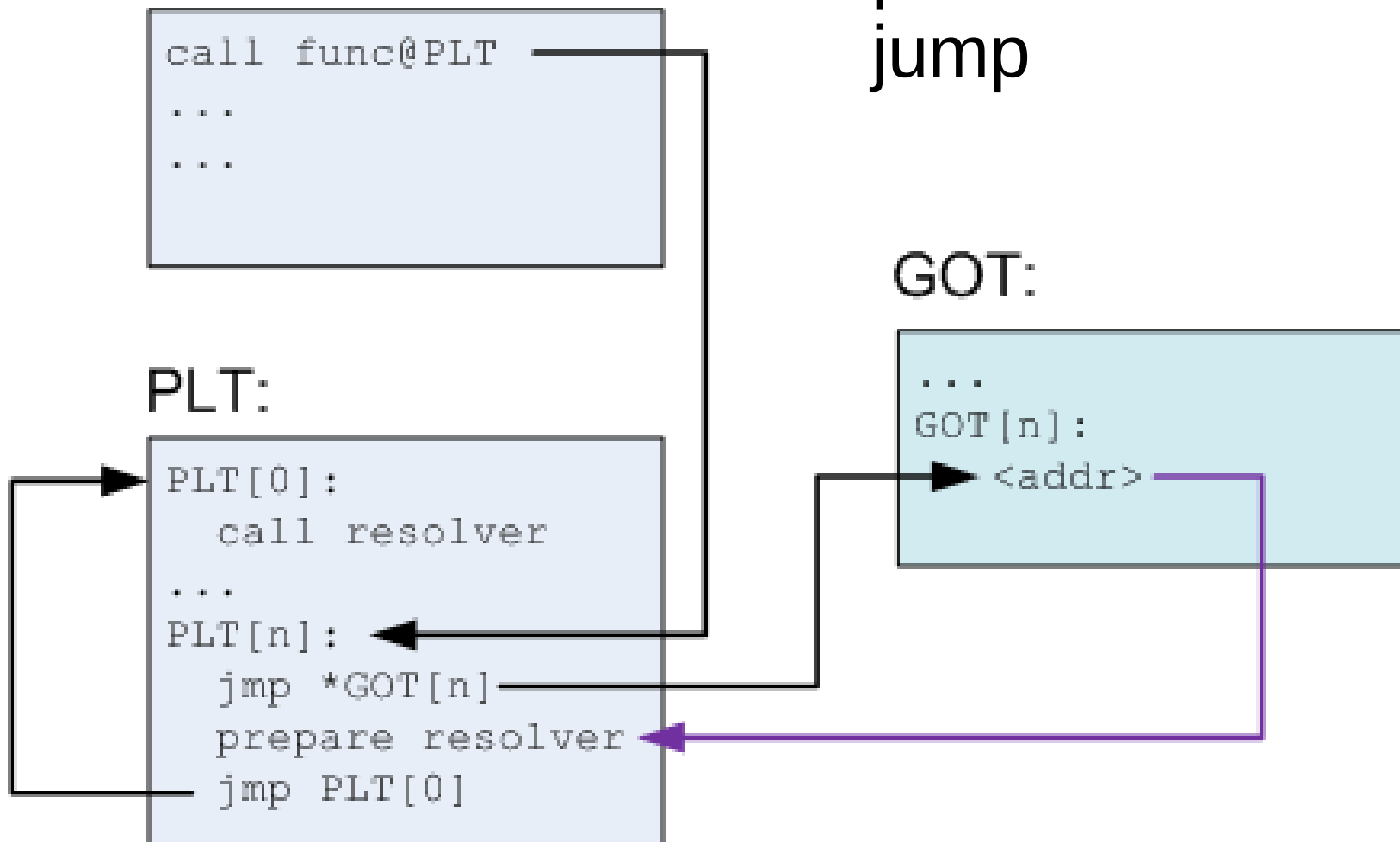
```
call func@PLT  
...  
...
```

PLT:

```
PLT[0]:  
    call resolver  
...  
PLT[n]:  
    jmp *GOT[n]  
    prepare resolver  
    jmp PLT[0]
```

GOT:

```
...  
GOT[n]:  
    <addr>
```



# PLT after the function is resolved

- Nth GOT entry points to the actual function

Code:

```
call func@PLT
...
...
```

PLT:

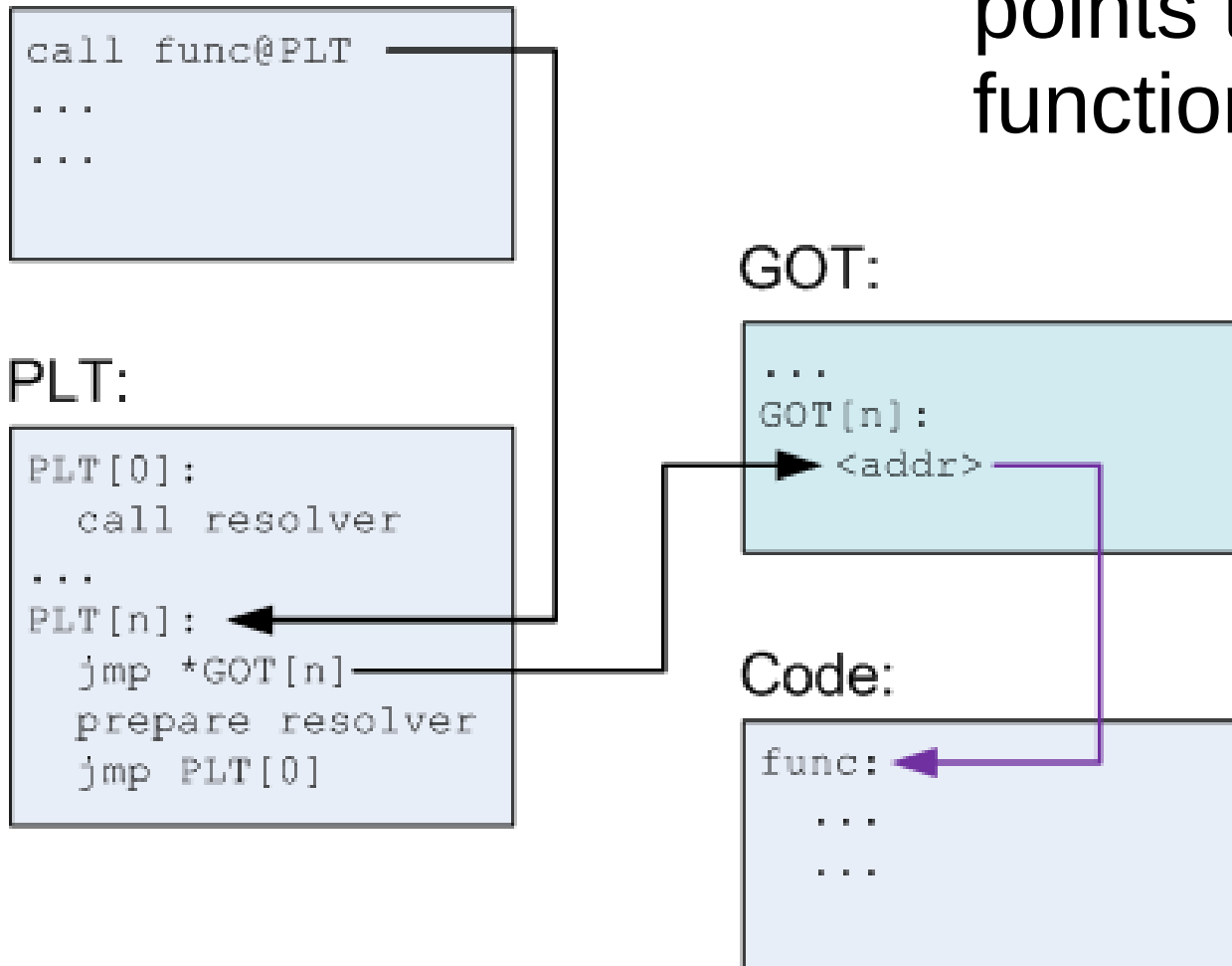
```
PLT[0]:
  call resolver
...
PLT[n]: ←
  jmp *GOT[n]
  prepare resolver
  jmp PLT[0]
```

GOT:

```
...
GOT[n]:
  → <addr>
```

Code:

```
func: ←
...
...
```



# PIC example (functions)

```
int ml_util_func(int a)
{
    return a + 1;
}

int ml_func(int a, int b)
{
    int c = b + ml_util_func(a);
    myglob += c;
    return b + myglob;
}
```

00000477 <ml\_func>:

477:	55	push	ebp
478:	89 e5	mov	ebp,esp
47a:	53	push	ebx
47b:	83 ec 24	sub	esp,0x24
47e:	e8 e4 ff ff ff	call	467 <__i686.get_pc_thunk.bx>
483:	81 c3 71 1b 00 00	add	ebx,0x1b71
489:	8b 45 08	mov	eax,DWORD PTR [ebp+0x8]
48c:	89 04 24	mov	DWORD PTR [esp],eax
48f:	e8 0c ff ff ff	call	3a0 <ml_util_func@plt>
...			

000003a0 <ml\_util\_func@plt>:

3a0:	ff a3 14 00 00 00	jmp	DWORD PTR [ebx+0x14]
3a6:	68 10 00 00 00	push	0x10
3ab:	e9 c0 ff ff ff	jmp	370 <_init+0x30>

# PIC example (functions)

```
int ml_util_func(int a)
{
    return a + 1;
}

int ml_func(int a, int b)
{
    int c = b + ml_util_func(a);
    myglob += c;
    return b + myglob;
}
```

- Resolve the address of GOT
- First learn EIP
  - Saved in EBX
- Then add offset to EBX

00000477 <ml\_func>:

477:	55	push	ebp
478:	89 e5	mov	ebp,esp
47a:	53	push	ebx
47b:	83 ec 24	sub	esp,0x24
47e:	e8 e4 ff ff ff	call	467 <__i686.get_pc_thunk.bx>
483:	81 c3 71 1b 00 00	add	ebx,0x1b71
489:	8b 45 08	mov	eax,DWORD PTR [ebp+0x8]
48c:	89 04 24	mov	DWORD PTR [esp],eax
48f:	e8 0c ff ff ff	call	3a0 <ml_util_func@plt>
...			

000003a0 <ml\_util\_func@plt>:

3a0:	ff a3 14 00 00 00	jmp	DWORD PTR [ebx+0x14]
3a6:	68 10 00 00 00	push	0x10
3ab:	e9 c0 ff ff ff	jmp	370 <_init+0x30>



```

int ml_util_func(int a)
{
    return a + 1;
}
int ml_func(int a, int b)
{
    int c = b + ml_util_func(a);
    myglob += c;
    return b + myglob;
}

```

# PIC example (functions)

- Push the argument a on the stack

00000477 <ml\_func>:

477:	55	push	ebp
478:	89 e5	mov	ebp,esp
47a:	53	push	ebx
47b:	83 ec 24	sub	esp,0x24
47e:	e8 e4 ff ff ff	call	467 <__i686.get_pc_thunk.bx>
483:	81 c3 71 1b 00 00	add	ebx,0x1b71
489:	8b 45 08	mov	eax,DWORD PTR [ebp+0x8]
48c:	89 04 24	mov	DWORD PTR [esp],eax
48f:	e8 0c ff ff ff	call	3a0 <ml_util_func@plt>
...			

000003a0 <ml\_util\_func@plt>:

3a0:	ff a3 14 00 00 00	jmp	DWORD PTR [ebx+0x14]
3a6:	68 10 00 00 00	push	0x10
3ab:	e9 c0 ff ff ff	jmp	370 <_init+0x30>

```

int ml_util_func(int a)
{
    return a + 1;
}

int ml_func(int a, int b)
{
    int c = b + ml_util_func(a);
    myglob += c;
    return b + myglob;
}

```

# PIC example (functions)

- Call the PLT entry for the `ml_util_func()`

00000477 <ml\_func>:

477:	55	push	ebp
478:	89 e5	mov	ebp,esp
47a:	53	push	ebx
47b:	83 ec 24	sub	esp,0x24
47e:	e8 e4 ff ff ff	call	467 <__i686.get_pc_thunk.bx>
483:	81 c3 71 1b 00 00	add	ebx,0x1b71
489:	8b 45 08	mov	eax,DWORD PTR [ebp+0x8]
48c:	89 04 24	mov	DWORD PTR [esp],eax
48f:	e8 0c ff ff ff	call	3a0 <ml_util_func@plt>
...			

000003a0 <ml\_util\_func@plt>:

3a0:	ff a3 14 00 00 00	jmp	DWORD PTR [ebx+0x14]
3a6:	68 10 00 00 00	push	0x10
3ab:	e9 c0 ff ff ff	jmp	370 <_init+0x30>

# PIC example (functions)

```
int ml_util_func(int a)
{
    return a + 1;
}

int ml_func(int a, int b)
{
    int c = b + ml_util_func(a);
    myglob += c;
    return b + myglob;
}
```

00000477 <ml\_func>:

```
477:  55                push    ebp
478:  89 e5             mov     ebp,esp
47a:  53                push    ebx
47b:  83 ec 24          sub     esp,0x24
47e:  e8 e4 ff ff ff    call   467 <__i686.get_pc_thunk.bx>
483:  81 c3 71 1b 00 00 add     ebx,0x1b71
489:  8b 45 08           mov     eax,DWORD PTR [ebp+0x8]
48c:  89 04 24           mov     DWORD PTR [esp],eax
48f:  e8 0c ff ff ff    call   3a0 <ml_util_func@plt>
...
```

000003a0 <ml\_util\_func@plt>:

```
3a0:  ff a3 14 00 00 00 jmp     DWORD PTR [ebx+0x14]
3a6:  68 10 00 00 00    push   0x10
3ab:  e9 c0 ff ff ff    jmp     370 <_init+0x30>
```

- Jump to an address specified in GOT
  - [ebx+0x14] contains address 0x3a6
  - i.e., effectively we jump to the next instruction via GOT

```

int ml_util_func(int a)
{
    return a + 1;
}

int ml_func(int a, int b)
{
    int c = b + ml_util_func(a);
    myglob += c;
    return b + myglob;
}

```

# PIC example (functions)

- Prepare arguments for the resolver

```

00000477 <ml_func>:
477:  55                push    ebp
478:  89 e5             mov     ebp,esp
47a:  53                push    ebx
47b:  83 ec 24          sub     esp,0x24
47e:  e8 e4 ff ff ff    call    467 <__i686.get_pc_thunk.bx>
483:  81 c3 71 1b 00 00 add     ebx,0x1b71
489:  8b 45 08           mov     eax,DWORD PTR [ebp+0x8]
48c:  89 04 24           mov     DWORD PTR [esp],eax
48f:  e8 0c ff ff ff    call    3a0 <ml_util_func@plt>
...

```

```

000003a0 <ml_util_func@plt>:
3a0:  ff a3 14 00 00 00 jmp     DWORD PTR [ebx+0x14]
3a6:  68 10 00 00 00 00 push    0x10
3ab:  e9 c0 ff ff ff    jmp     370 <_init+0x30>

```

# PIC example (functions)

- Call resolver

```
int ml_util_func(int a)
{
    return a + 1;
}
int ml_func(int a, int b)
{
    int c = b + ml_util_func(a);
    myglob += c;
    return b + myglob;
}
```

00000477 <ml\_func>:

477:	55	push	ebp
478:	89 e5	mov	ebp,esp
47a:	53	push	ebx
47b:	83 ec 24	sub	esp,0x24
47e:	e8 e4 ff ff ff	call	467 <__i686.get_pc_thunk.bx>
483:	81 c3 71 1b 00 00	add	ebx,0x1b71
489:	8b 45 08	mov	eax,DWORD PTR [ebp+0x8]
48c:	89 04 24	mov	DWORD PTR [esp],eax
48f:	e8 0c ff ff ff	call	3a0 <ml_util_func@plt>
...			

000003a0 <ml\_util\_func@plt>:

3a0:	ff a3 14 00 00 00	jmp	DWORD PTR [ebx+0x14]
3a6:	68 10 00 00 00	push	0x10
3ab:	e9 c0 ff ff ff	jmp	370 <_init+0x30>

# PIC example (functions)

```
int ml_util_func(int a)
{
    return a + 1;
}

int ml_func(int a, int b)
{
    int c = b + ml_util_func(a);
    myglob += c;
    return b + myglob;
}
```

00000477 <ml\_func>:

477:	55	push	ebp
478:	89 e5	mov	ebp,esp
47a:	53	push	ebx
47b:	83 ec 24	sub	esp,0x24
47e:	e8 e4 ff ff ff	call	467 <__i686.get_pc_thunk.bx>
483:	81 c3 71 1b 00 00	add	ebx,0x1b71
489:	8b 45 08	mov	eax,DWORD PTR [ebp+0x8]
48c:	89 04 24	mov	DWORD PTR [esp],eax
48f:	e8 0c ff ff ff	call	3a0 <ml_util_func@plt>
...			

000003a0 <ml\_util\_func@plt>:

3a0:	ff a3 14 00 00 00	jmp	DWORD PTR [ebx+0x14]
3a6:	68 10 00 00 00	push	0x10
3ab:	e9 c0 ff ff ff	jmp	370 <_init+0x30>

- After the address of `ml_util_func()` is resolved
  - i.e., on the next invocation
- This jump goes to the function entry

# What did we gain?

- Processes can share code
- Each have private GOT
- Why is it better?
  - GOT is in the data section,  
private to each process anyway
    - We saved memory
  - We saved some linking time too
    - GOT is patched per variable, not  
per variable reference
    - There are many references to the  
same variable in the code
    - It takes some time to relocate
    - We saved this time

# PIC: Advantages and disadvantages

- Any ideas?



# PIC: Advantages and disadvantages

- Bad
  - Code gets slower
    - One register is wasted to keep GOT pointer
      - x86 has 7 registers (plus maybe EBP is used to maintain the frame, so maybe 6)
      - Losing one of them is bad
    - One more memory dereference
      - GOT can be large (lots of global variables)
      - Extra memory dereferences can have a high cost due to cache misses
    - One more call to find GOT
- Good
  - Share memory of common libraries
  - Address space randomization

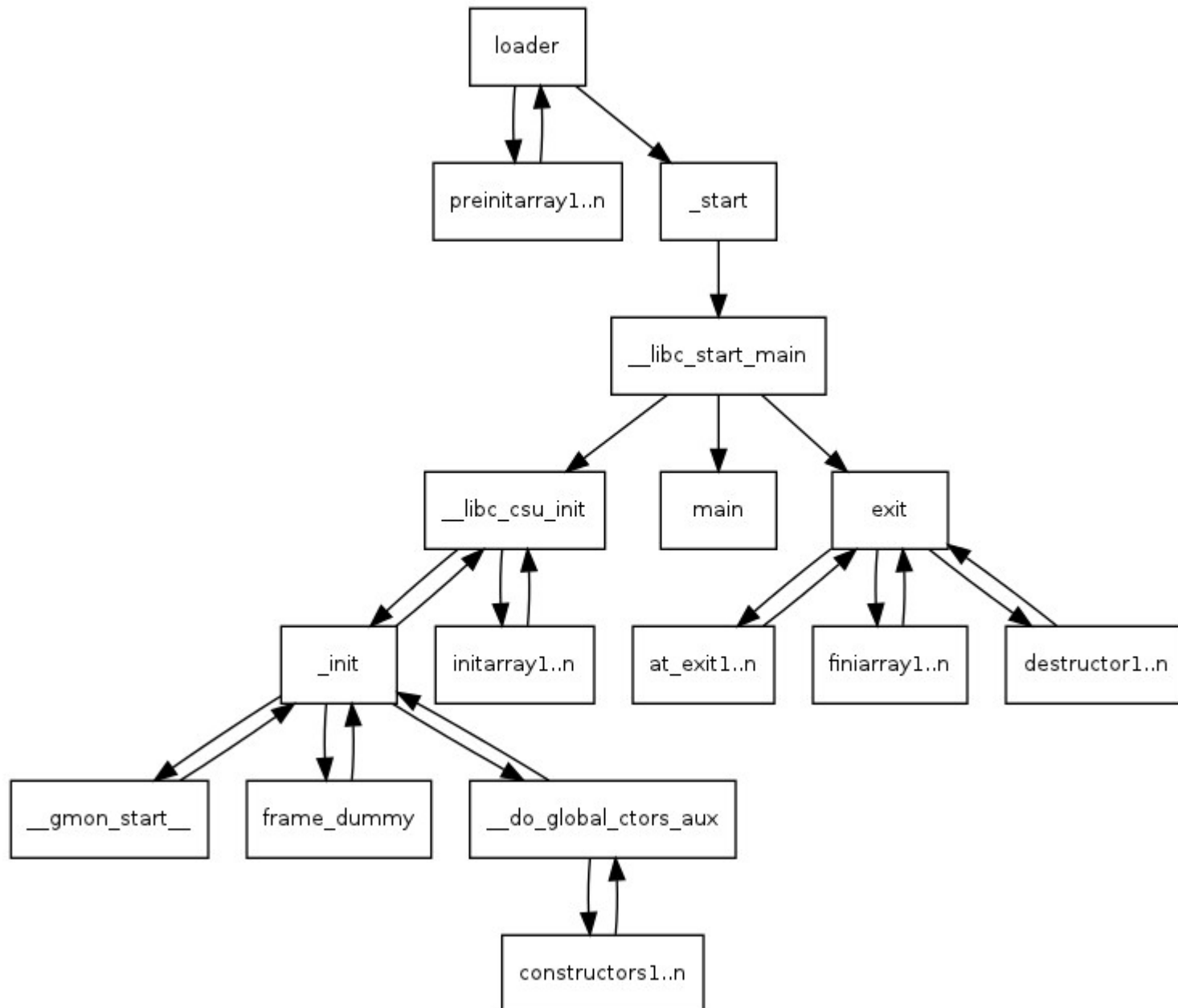
Loading programs in memory

What runs before main()?

This material is based on

<http://dbp-consulting.com/tutorials/debugging/linuxProgramStartup.html>

# Starting main()



# Loading a dynamically linked ELF program

- Map ELF sections into memory
- Note the interpreter section
  - Usually ld.so
- Map ld.so into memory
  - Start ld.so instead of the program
- Linker (ld.so) initializes itself
- Finds the names of shared libraries required by the program
  - DT\_NEEDED entries

# Finding libraries in the file system

- DT\_RPATH symbol
  - Can be linked into a file by a normal linker at link time
- LD\_LIBRARY\_PATH
- Library cache file
  - /etc/ld.so.conf
  - This is the most normal way to resolve library paths
- Default library path
  - /usr/lib

# Loading more libraries

- When the library is found it is loaded into memory
  - Linker adds its symbol table to the linked list of symbol tables
  - Recursively searches if the library depends on other libraries
    - Loads them if needed

# Shared library initialization

- Remember PIC needs relocation in the data segment and GOT
  - ld.so linker performs this relocation



# To see the backtrace before `main()`

- Execute this command in GDB

```
(gdb) set backtrace past-main on
```

- Example for our homework

```
Breakpoint 1, main () at main.c:26
```

```
26      s = sum(100);
```

```
Missing separate debuginfos, use: debuginfo-install glibc-2.17-292.el7.i686
```

```
(gdb) set backtrace past-main on
```

```
(gdb) bt
```

```
#0  main () at main.c:26
```

```
#1  0xf7dfb2a3 in __libc_start_main () from /lib/libc.so.6
```

```
#2  0x08048331 in _start ()
```

```
(gdb)
```

# Alternatively set a breakpoint on \_\_start

- Use readelf

```
readelf -a hello
```

- Example for our homework

ELF Header:

```
Magic:  7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
Class:                               ELF32
Data:                                2's complement, little endian
Version:                             1 (current)
OS/ABI:                               UNIX - System V
ABI Version:                          0
Type:                                 EXEC (Executable file)
Machine:                              Intel 80386
Version:                              0x1
Entry point address:                0x8048310
Start of program headers:              52 (bytes into file)
Start of section headers:              6880 (bytes into file)
Flags:                                 0x0
Size of this header:                    52 (bytes)
Size of program headers:                32 (bytes)
Number of program headers:               9
Size of section headers:                40 (bytes)
Number of section headers:               36
Section header string table index: 35
```

# Initializers and finalizers

- C++ needs a segment for invoking constructors for static variables
  - List of pointers to startup routines
    - Startup code in every module is put into an anonymous startup routine
    - Put into a segment called .init
- Problem
  - Order matters
  - Ideally you should track dependencies
    - This is not done
  - Simple hack
    - System libraries go first (.init), then user (.ctor)

# Example of a constructor

```
#include <stdio.h>

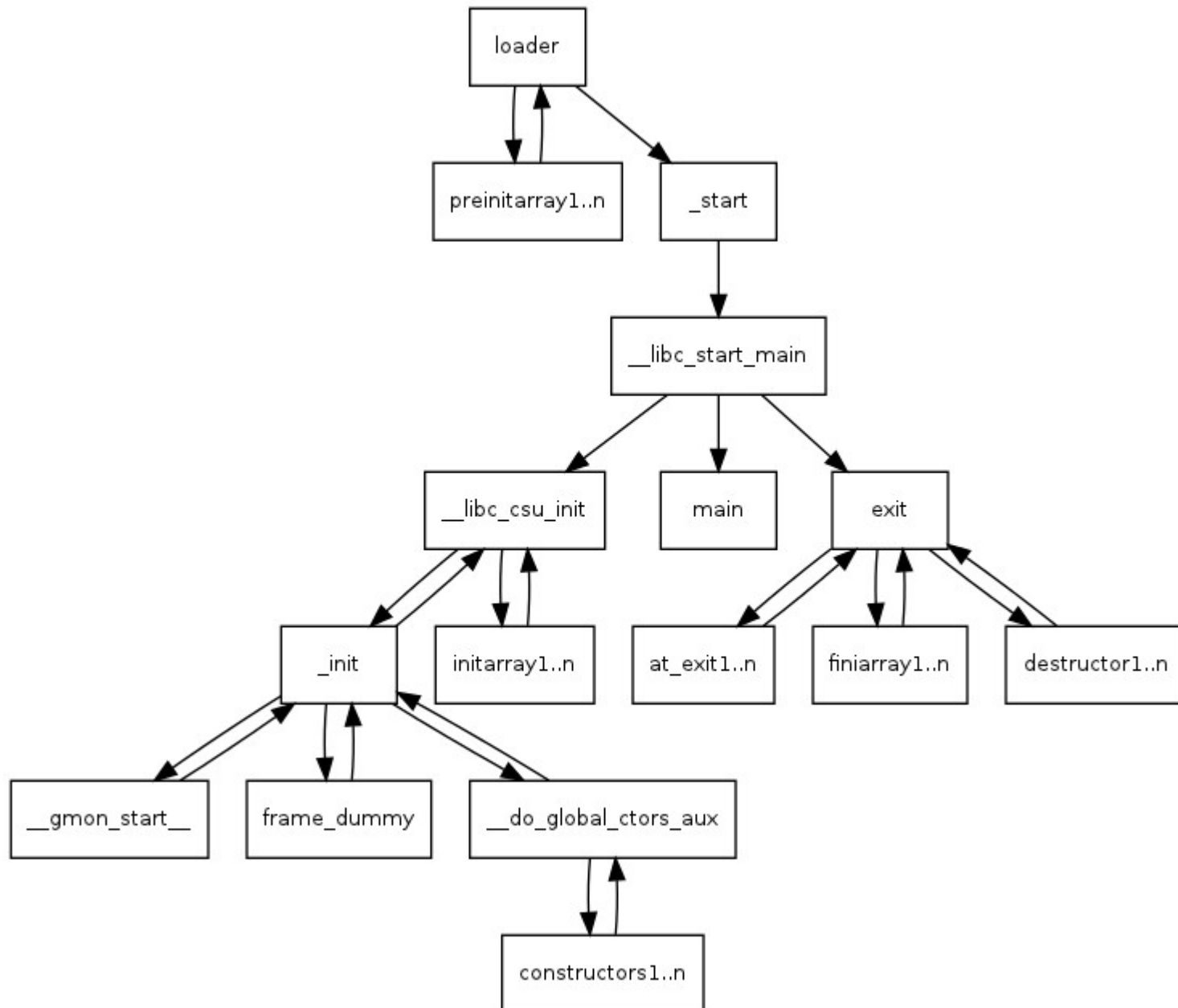
void __attribute__((constructor)) a_constructor() {
    printf("%s\n", __FUNCTION__);
}

int
main()
{
    printf("%s\n", __FUNCTION__);
}
```

- Run it

```
$ ./hello
a_constructor
main
$
```

# Starting main()



# Conclusion

- Program loading
  - Storage allocation
- Relocation
  - Assign load address to each object file
  - Patch the code
- Symbol resolution
  - Resolve symbols imported from other object files

Thank you!

# Weak vs strong symbols

- Virtually every program uses printf
  - Printf can convert floating-point numbers to strings
    - Printf uses fcvt()
  - Does this mean that every program needs to link against floating-point libraries?
- Weak symbols allow symbols to be undefined
  - If program uses floating numbers, it links against the floating-point libraries
    - fcvt() is defined and everything is fine
  - If program doesn't use floating-point libraries
    - fcvt() remains NULL but is never called



# nm a.out

```
0804a01c B __bss_start
0804a01c b completed.6591
0804a014 D __data_start
0804a014 W data_start
...
0804a01c D _edata
0804a020 B _end
08048484 T _fini
...
08048294 T _init
...
080483ed T main
...
080482f0 T _start
...
```