

exec-encoding

executable encoding: security issues

malware wants to modify executables

- hide in 'normal' programs (viruses)

- evade program analysis tools by doing weird things

memory vulnerabilities require understanding memory layout

- going from "write memory address X" to "run arbitrary code"

- where are pointers to functions (we could change?)

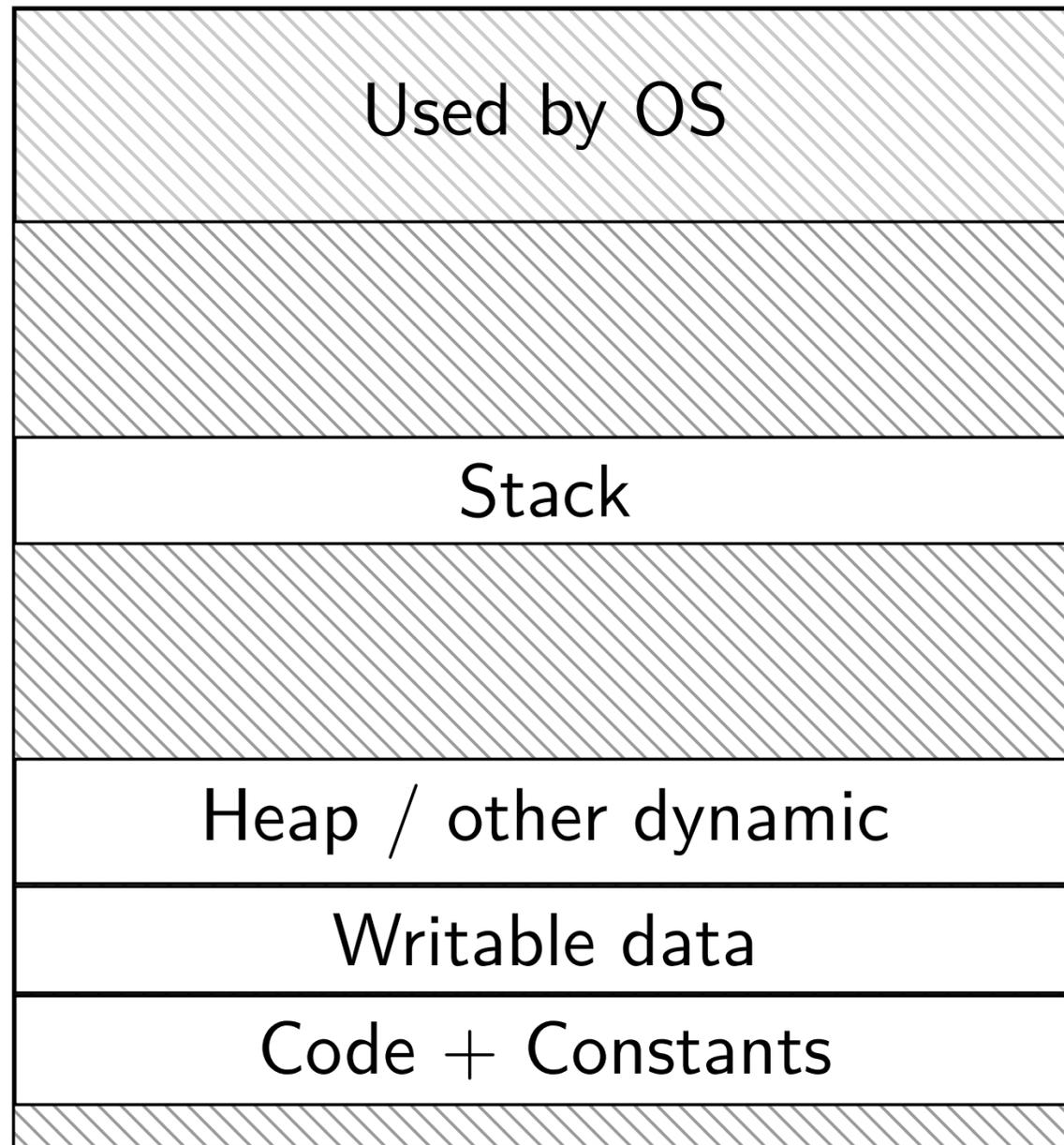
- where is code (vulnerabilities might want to execute)

- where is data (vulnerabilities might want to read/change)

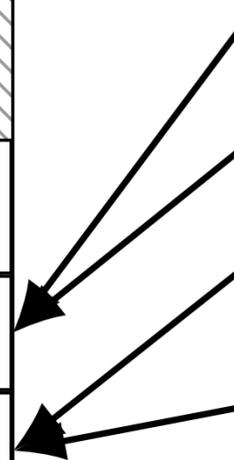
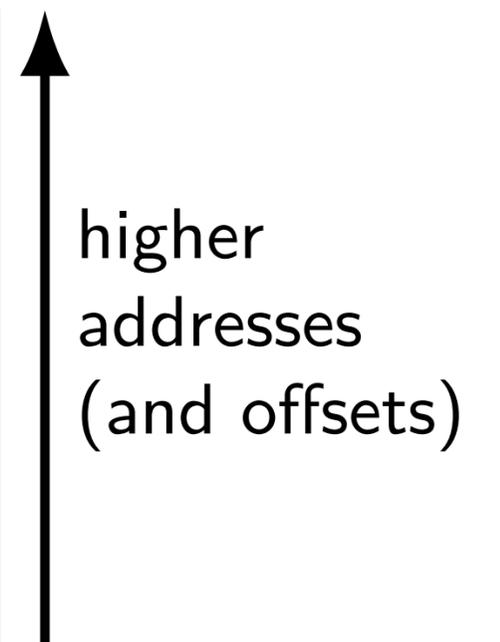
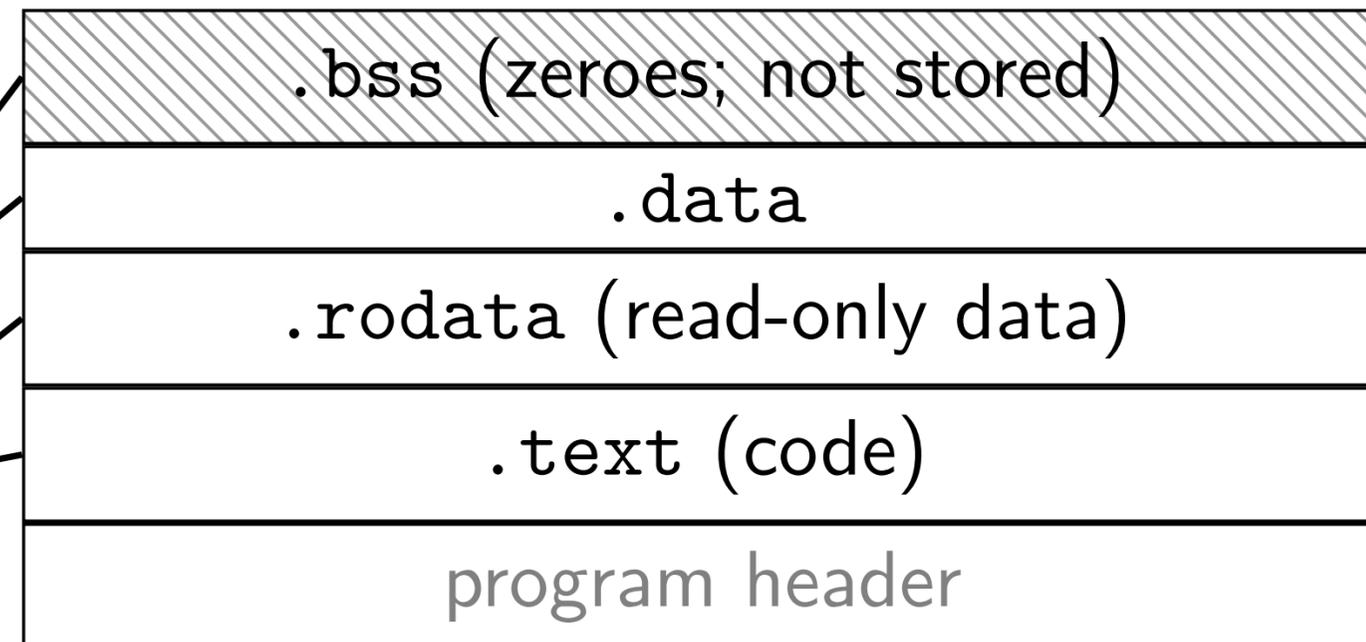
ways we can load+layout programs that are harder to exploit?

memory v. disk

(virtual) memory



program on disk



ELF (executable and linking format)

Linux (and some others) executable/object file format

header: machine type, file type, etc.

program header: “*segments*” to load
(also, some other information)

segment 1 data

segment 2 data

section header:

list of “*sections*” (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

section headers

Sections:

Idx	Name	Size	VMA	LMA	File off	Algn
0	.note.ABI-tag	00000020	00000000000400190	00000000000400190	00000190	2**2
			CONTENTS, ALLOC, LOAD, READONLY, DATA			
1	.note.gnu.build-id	00000024	000000000004001b0	000000000004001b0	000001b0	2**2
			CONTENTS, ALLOC, LOAD, READONLY, DATA			
2	.rela.plt	00000210	000000000004001d8	000000000004001d8	000001d8	2**3
			CONTENTS, ALLOC, LOAD, READONLY, DATA			
3	.init	0000001a	000000000004003e8	000000000004003e8	000003e8	2**2
			CONTENTS, ALLOC, LOAD, READONLY, CODE			
4	.plt	00000160	00000000000400410	00000000000400410	00000410	2**4
			CONTENTS, ALLOC, LOAD, READONLY, CODE			
5	.text	0017ff1d	00000000000400570	00000000000400570	00000570	2**4
			CONTENTS, ALLOC, LOAD, READONLY, CODE			
6	__libc_freeres_fn	00002032	00000000000580490	00000000000580490	00180490	2**4
			CONTENTS, ALLOC, LOAD, READONLY, CODE			
7	__libc_thread_freeres_fn	0000021b	000000000005824d0	000000000005824d0	001824d0	2**4
			CONTENTS, ALLOC, LOAD, READONLY, CODE			
8	.fini	00000009	000000000005826ec	000000000005826ec	001826ec	2**2
			CONTENTS, ALLOC, LOAD, READONLY, CODE			

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

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

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 0x0000000000000402170
```

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 0x0000000000000402170
```

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 0x00000102:
EXEC_P, D_PAGED
start address 0x000000000000402170
```

Program Header:

[...]

Sections:

[...]

a program header (1)

Program Header:

[...]

```
LOAD off      0x0001000 vaddr 0x0401000 paddr 0x0401000 align 2**12
      filesz 0x01b04ed memsz 0x01b04ed flags r-x
```

[...]

```
LOAD off      0x0207950 vaddr 0x0608950 paddr 0x0608950 align 2**12
      filesz 0x0008f40 memsz 0x000c718 flags rw-
```

load 0x1bd04ed bytes (read+executable):

from 0x1000 bytes into the file

to memory at 0x401000

load 0x8f40 bytes (read+write):

from 0x207950 bytes into the file

to memory at 0x608950

plus (0xc718-0x8f40) bytes of zeroes

a program header (1)

Program Header:

```
[...]  
LOAD off      0x0001000 vaddr 0x0401000 paddr 0x0401000 align 2**12  
      filesz 0x01b04ed memsz 0x01b04ed flags r-x  
[...]  
LOAD off      0x0207950 vaddr 0x0608950 paddr 0x0608950 align 2**12  
      filesz 0x0008f40 memsz 0x000c718 flags rw-
```

load *0x1bd04ed* bytes (read+executable):

from 0x1000 bytes into the file
to memory at 0x401000

load 0x8f40 bytes (read+write):

from 0x207950 bytes into the file
to memory at 0x608950
plus (0xc718-0x8f40) bytes of zeroes

a program header (1)

Program Header:

```
[...]  
LOAD off      0x0001000 vaddr 0x0401000 paddr 0x0401000 align 2**12  
      filesz 0x01b04ed memsz 0x01b04ed flags r-x  
[...]  
LOAD off      0x0207950 vaddr 0x0608950 paddr 0x0608950 align 2**12  
      filesz 0x0008f40 memsz 0x000c718 flags rw-
```

load 0x1bd04ed bytes (*read+executable*):

from 0x1000 bytes into the file
to memory at 0x401000

load 0x8f40 bytes (read+write):

from 0x207950 bytes into the file
to memory at 0x608950
plus (0xc718-0x8f40) bytes of zeroes

a program header (1)

Program Header:

[...]

```
LOAD off      0x0001000 vaddr 0x0401000 paddr 0x0401000 align 2**12
      filesz 0x01b04ed memsz 0x01b04ed flags r-x
```

[...]

```
LOAD off      0x0207950 vaddr 0x0608950 paddr 0x0608950 align 2**12
      filesz 0x0008f40 memsz 0x000c718 flags rw-
```

load 0x1bd04ed bytes (read+executable):

from 0x1000 bytes into the file

to memory at 0x401000

load 0x8f40 bytes (read+write):

from 0x207950 bytes into the file

to memory at 0x608950

plus (0xc718-0x8f40) bytes of zeroes

a program header (2)

Program Header:

```
[...]  
    NOTE off      0x0000290 vaddr 0x0400290 paddr 0x0400290 align 2**2  
        filesz 0x0000044 memsz 0x0000044 flags r--  
    TLS off      0x0207950 vaddr 0x0608950 paddr 0x0608950 align 2**3  
        filesz 0x0000030 memsz 0x0000092 flags r--  
0x6474e553 off  0x0000270 vaddr 0x0400270 paddr 0x0400270 align 2**3  
        filesz 0x0000020 memsz 0x0000020 flags r--  
    STACK off    0x0000000 vaddr 0x0000000 paddr 0x0000000 align 2**4  
        filesz 0x0000000 memsz 0x0000000 flags rw-  
    RELRO off    0x0207950 vaddr 0x0608950 paddr 0x0608950 align 2**0  
        filesz 0x00066b0 memsz 0x00066b0 flags r--  
[...]
```

NOTE – comment

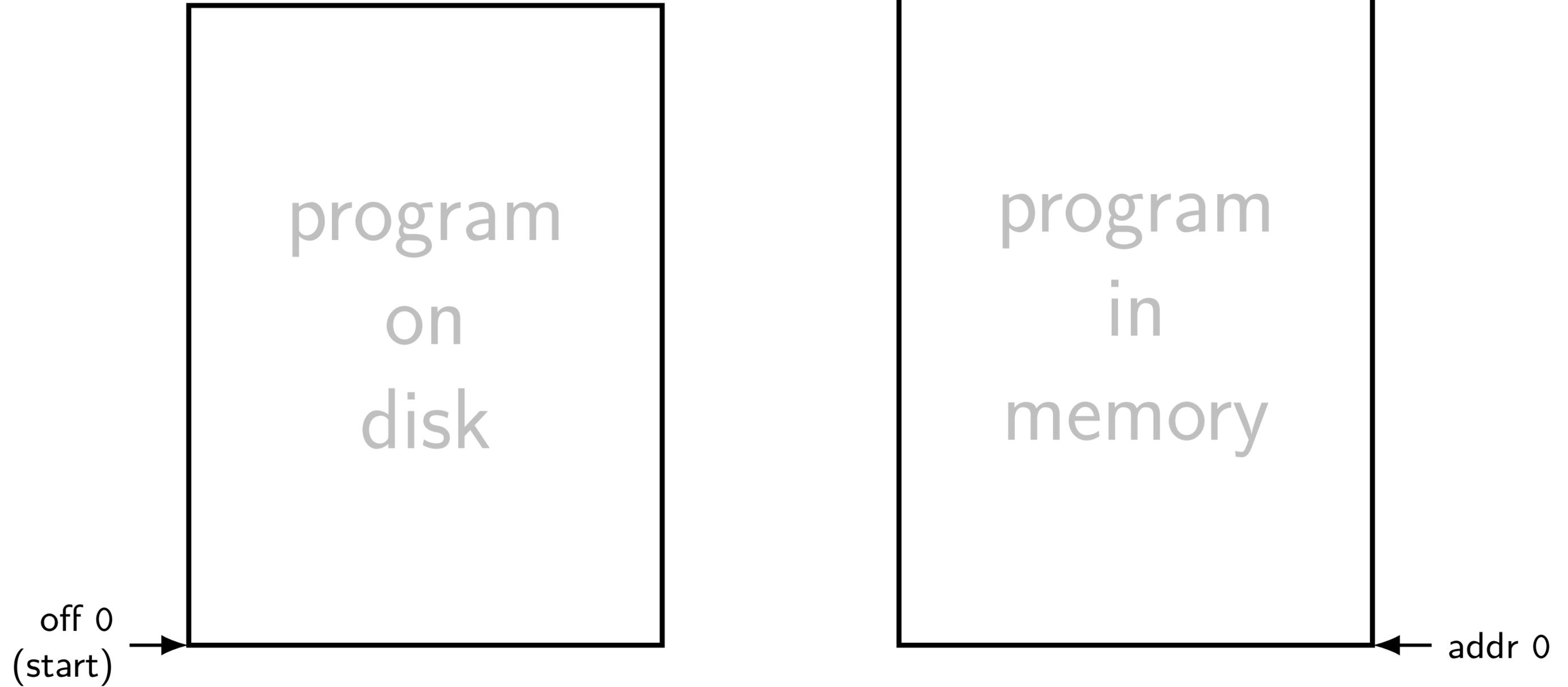
TLS – thread-local storage region (used via %fs)

0x6474e553 – ‘GNU_PROPERTY’ – adtl linker/loader info

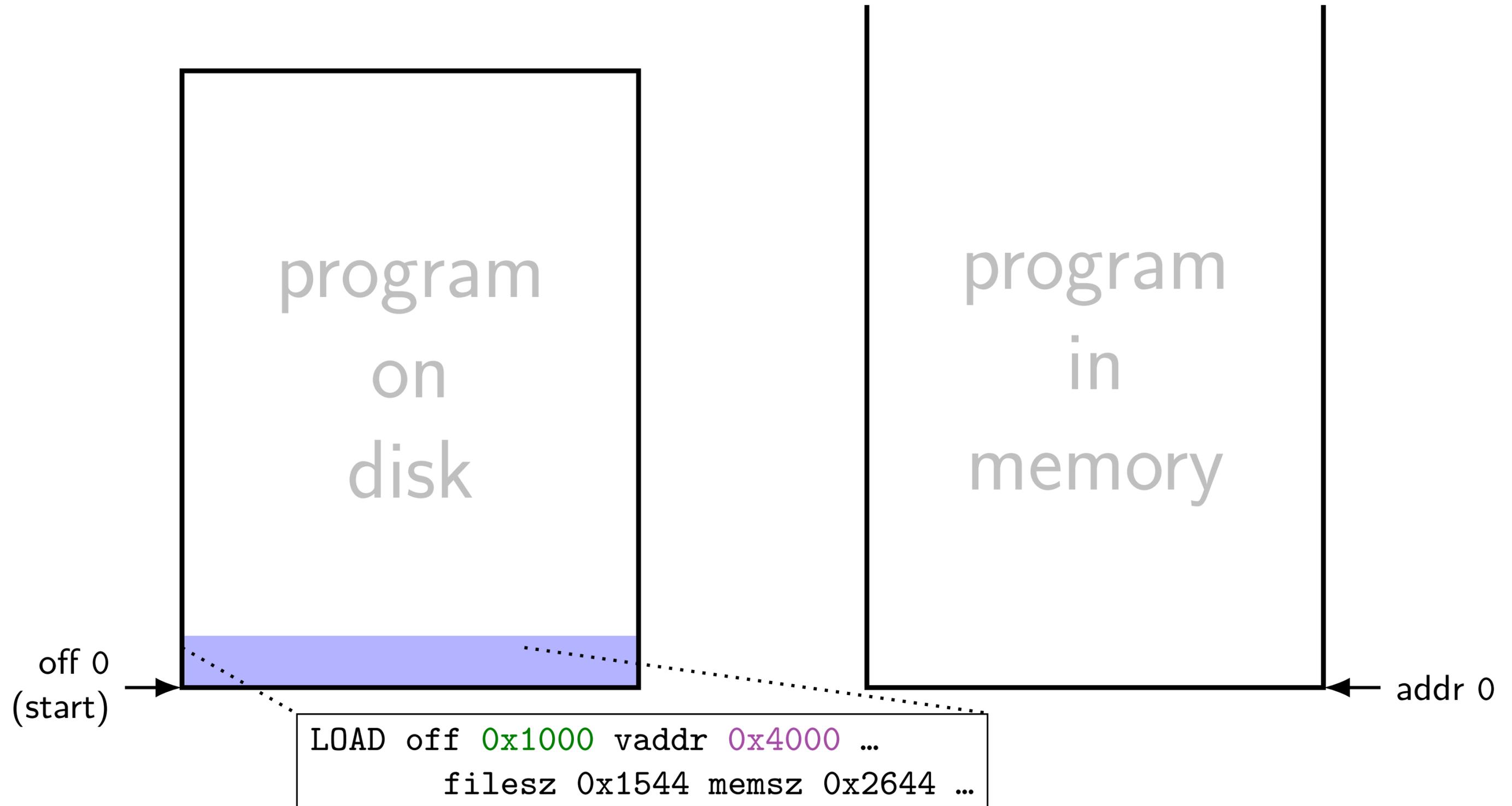
STACK – indicates stack is read/write

RELRO – make this read-only after runtime linking

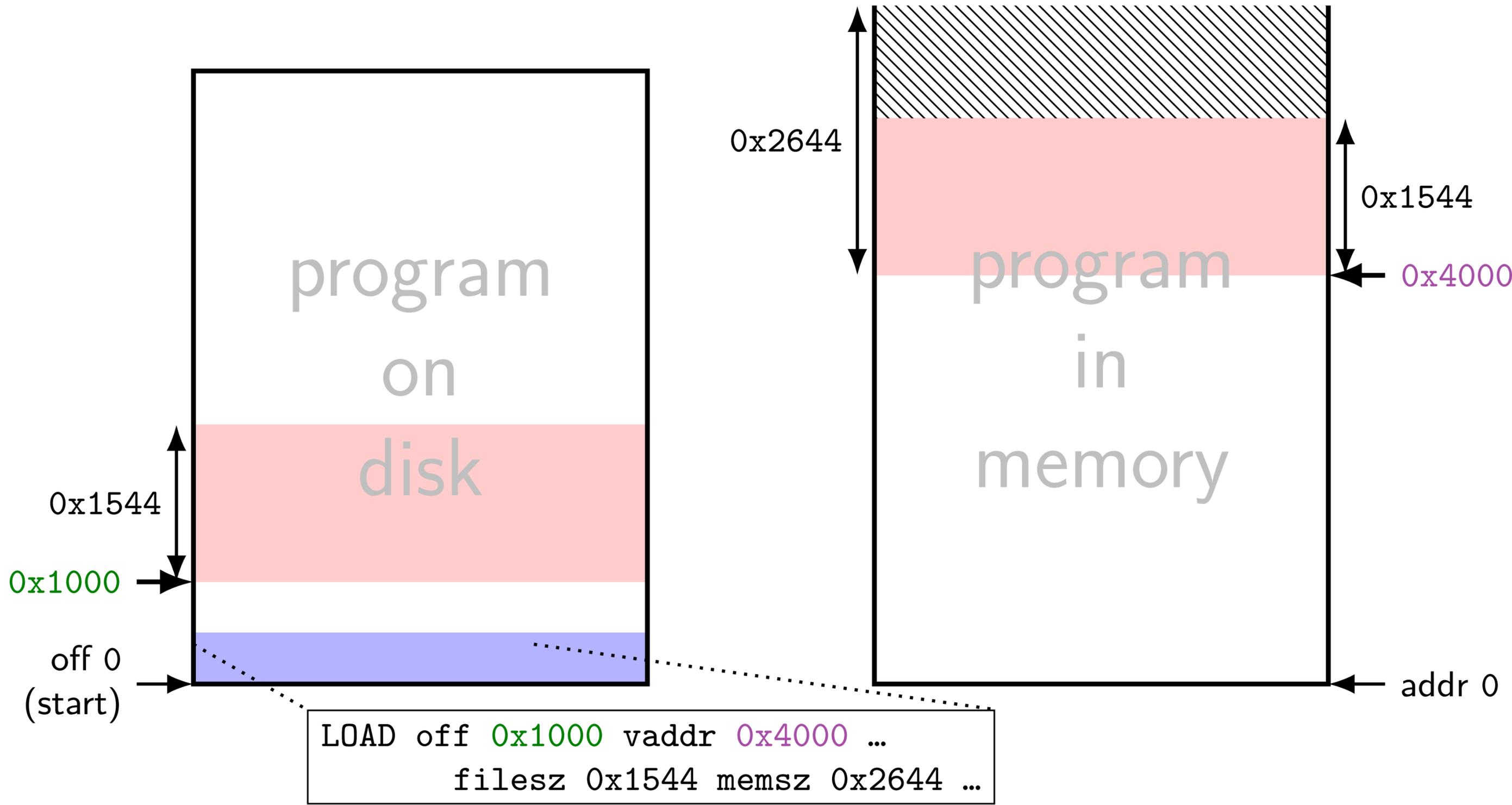
ELF LOAD



ELF LOAD



ELF LOAD



dynamic library headers

```
/lib/x86_64-linux-gnu/libc.so.6:      file format elf64-x86-64
```

```
/lib/x86_64-linux-gnu/libc.so.6
```

```
architecture: i386:x86-64, flags 0x00000150:
```

```
HAS_SYMS, DYNAMIC, D_PAGED
```

```
start address 0x000000000000271f0
```

Program Header:

```
PHDR off      0x0000000000000040 vaddr 0x0000000000000040 paddr 0x0000000000000040 align 2**3
      filesz 0x0000000000000310 memsz 0x0000000000000310 flags r--
INTERP off    0x000000000001c16a0 vaddr 0x000000000001c16a0 paddr 0x000000000001c16a0 align 2**4
      filesz 0x000000000000001c memsz 0x000000000000001c flags r--
LOAD off     0x0000000000000000 vaddr 0x0000000000000000 paddr 0x0000000000000000 align 2**12
      filesz 0x00000000000024940 memsz 0x00000000000024940 flags r--
```

```
...
```

dynamic library headers

```
/lib/x86_64-linux-gnu/libc.so.6:      file format elf64-x86-64
/lib/x86_64-linux-gnu/libc.so.6
architecture: i386:x86-64, flags 0x00000150:
HAS_SYMS, DYNAMIC, D_PAGED
start address 0x000000000000271f0
```

Program Header:

```
PHDR off      0x0000000000000040 vaddr 0x0000000000000040 paddr 0x0000000000000040 align 2**3
      filesz 0x0000000000000310 memsz 0x0000000000000310 flags r--
INTERP off    0x000000000001c16a0 vaddr 0x000000000001c16a0 paddr 0x000000000001c16a0 align 2**4
      filesz 0x000000000000001c memsz 0x000000000000001c flags r--
LOAD off     0x0000000000000000 vaddr 0x0000000000000000 paddr 0x0000000000000000 align 2**12
      filesz 0x00000000000024940 memsz 0x00000000000024940 flags r--
```

...

DYNAMIC — instead of EXEC_P

dynamic library headers

```
/lib/x86_64-linux-gnu/libc.so.6:      file format elf64-x86-64
/lib/x86_64-linux-gnu/libc.so.6
architecture: i386:x86-64, flags 0x00000150:
HAS_SYMS, DYNAMIC, D_PAGED
start address 0x000000000000271f0
```

Program Header:

```
PHDR off      0x0000000000000040 vaddr 0x0000000000000040 paddr 0x0000000000000040 align 2**3
      filesz 0x0000000000000310 memsz 0x0000000000000310 flags r--
INTERP off    0x000000000001c16a0 vaddr 0x000000000001c16a0 paddr 0x000000000001c16a0 align 2**4
      filesz 0x000000000000001c memsz 0x000000000000001c flags r--
LOAD off     0x0000000000000000 vaddr 0x0000000000000000 paddr 0x0000000000000000 align 2**12
      filesz 0x00000000000024940 memsz 0x00000000000024940 flags r--
```

...

specify loading start at addresses zero
when actually run, linker will choose different starting address

position-independent executables

hello.exe: file format elf64-x86-64

hello.exe

architecture: i386:x86-64, flags 0x00000150:

HAS_SYMS, DYNAMIC, D_PAGED

start address 0x00000000000001080

Program Header:

```
PHDR off 0x0000000000000040 vaddr 0x0000000000000040 paddr 0x0000000000000040 align 2**3
      filesz 0x00000000000002d8 memsz 0x00000000000002d8 flags r--
INTERP off 0x0000000000000318 vaddr 0x0000000000000318 paddr 0x0000000000000318 align 2**0
      filesz 0x000000000000001c memsz 0x000000000000001c flags r--
LOAD off 0x0000000000000000 vaddr 0x0000000000000000 paddr 0x0000000000000000 align 2**12
      filesz 0x00000000000005f8 memsz 0x00000000000005f8 flags r--
```

position-independent executables

```
hello.exe:      file format elf64-x86-64
hello.exe
architecture:  i386:x86-64, flags 0x00000150:
HAS_SYMS, DYNAMIC, D_PAGED
start address 0x00000000000001080
```

Program Header:

```
PHDR off      0x0000000000000040 vaddr 0x0000000000000040 paddr 0x0000000000000040 align 2**3
      filesz 0x00000000000002d8 memsz 0x00000000000002d8 flags r--
INTERP off    0x0000000000000318 vaddr 0x0000000000000318 paddr 0x0000000000000318 align 2**0
      filesz 0x000000000000001c memsz 0x000000000000001c flags r--
LOAD  off    0x0000000000000000 vaddr 0x0000000000000000 paddr 0x0000000000000000 align 2**12
      filesz 0x00000000000005f8 memsz 0x00000000000005f8 flags r--
```

executable with headers like dynamic library

“position-independent executable”: can be loaded at any address

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
```



```
callq 0x458F0
```

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 (pre-static or dynamic linking)

SYMBOL TABLE:

```
00000000000000000000 l      d      .text 00000000000000000000 .text
00000000000000000000 l      d      .data 00000000000000000000 .data
00000000000000000000 l      d      .bss  00000000000000000000 .bss
00000000000000000000 l              .data 00000000000000000000 string
00000000000000000000 g              .text 00000000000000000000 main
00000000000000000000          *UND* 00000000000000000000 puts
```

RELOCATION RECORDS FOR [.text]:

OFFSET	TYPE	VALUE
00000000000000000003	R_X86_64_32S	.data
00000000000000000008	R_X86_64_PC32	puts-0x00000000000000000004

hello.o (pre-static or dynamic linking)

SYMBOL TABLE:

```
00000000000000000000 l    d    .text 00000000000000000000 .text
00000000000000000000 l    d    .data 00000000000000000000 .data
00000000000000000000 l    d    .bss 00000000000000000000 .bss
00000000000000000000 l           .data 00000000000000000000 string
00000000000000000000 g           .text 00000000000000000000 main
00000000000000000000           *UND* 00000000000000000000 puts
```

RELOCATION RECORDS FOR [.text]:

OFFSET	TYPE	VALUE
00000000000000000003	R_X86_64_32S	.data
00000000000000000008	R_X86_64_PC32	puts-0x000000000000000004

relocation for puts =
insert address of puts-0x4 at byte 8 of .text (code)
(part of machine code for call puts)

hello.o (pre-static or dynamic linking)

SYMBOL TABLE:

```
00000000000000000000 l    d  .text 00000000000000000000 .text
00000000000000000000 l    d  .data 00000000000000000000 .data
00000000000000000000 l    d  .bss  00000000000000000000 .bss
00000000000000000000 l          .data 00000000000000000000 string
00000000000000000000 g          .text 00000000000000000000 main
00000000000000000000          *UND* 00000000000000000000 puts
```

RELOCATION RECORDS FOR [.text]:

OFFSET	TYPE	VALUE
00000000000000000003	R_X86_64_32S	.data
00000000000000000008	R_X86_64_PC32	puts-0x00000000000000000004

relocation for .data = string location
insert address of string at byte 3 of .text (code)
(part of machine code for `movq $string, %rdi`)

hello.o (pre-static or dynamic linking)

SYMBOL TABLE:

```
00000000000000000000 7 .data.text 00000000000000000000 .text
00000000000000000000 different ways to represent address
00000000000000000000 32S = signed 32-bit value
00000000000000000000 PC32 = 32-bit different from current address
00000000000000000000 (puts-0x4 to make it difference from end of call instead of middle of call)
```

RELOCATION RECORDS FOR [.text]:

OFFSET	TYPE	VALUE
00000000000000000003	R_X86_64_32S	.data
00000000000000000008	R_X86_64_PC32	<i>puts</i> -0x00000000000000000004

hello.o (pre-static or dynamic linking)

SYMBOL TABLE:

```
00000000000000000000 l    d    .text 00000000000000000000 .text
00000000000000000000 l    d    .data 00000000000000000000 .data
00000000000000000000 l    d    .bss 00000000000000000000 .bss
00000000000000000000 l        .data 00000000000000000000 string
00000000000000000000 g        .text 00000000000000000000 main
00000000000000000000      *UND* 00000000000000000000 puts
```

RELOCATION RECORDS FOR [.text]:

OFFSET	TYPE	VALUE
00000000000000000003	R_X86_64_32S	<i>.data</i>
00000000000000000008	R_X86_64_PC32	puts-0x000000000000000004

g = global (used by other files)
l = local

hello.o (pre-static or dynamic linking)

SYMBOL TABLE:

```
00000000000000000000 l      d  .text 00000000000000000000 .text
00000000000000000000 l      d  .data 00000000000000000000 .data
00000000000000000000 l      d  .bss  00000000000000000000 .bss
00000000000000000000 l          .data 00000000000000000000 string
00000000000000000000 g          .text 00000000000000000000 main
00000000000000000000          *UND* 00000000000000000000 puts
```

RELOCATION RECORDS FOR [.text]:

OFFSET	TYPE	VALUE
00000000000000000003	<i>R_X86_64_32S</i>	.data
00000000000000000008	<i>R_X86_64_PC32</i>	puts-0x000000000000000004

main at .text segments beginning + 0 bytes

hello.o (pre-static or dynamic linking)

SYMBOL TABLE:

```
00000000000000000000  l      d      .text  00000000000000000000  .text
00000000000000000000  l      d      .data  00000000000000000000  .data
00000000000000000000  l      d      .bss   00000000000000000000  .bss
00000000000000000000  l              .data  00000000000000000000  string
00000000000000000000  g              .text  00000000000000000000  main
00000000000000000000              *UND*  00000000000000000000  puts
```

RELOCATION RECORDS FOR [.text]:

OFFSET	TYPE	VALUE
00000000000000000003	R_X86_64_32S	.data
00000000000000000008	R_X86_64_PC32	puts-0x00000000000000000004

hello.o (pre-static or dynamic linking)

SYMBOL TABLE:

```
00000000000000000000 l      d      .text  00000000000000000000 .text
00000000000000000000 l      d      .data  00000000000000000000 .data
00000000000000000000 l      d      .bss   00000000000000000000 .bss
00000000000000000000 l              .data  00000000000000000000 string
00000000000000000000 g              .text 00000000000000000000 main
00000000000000000000      *UND*  00000000000000000000 puts
```

RELOCATION RECORDS FOR [.text]:

OFFSET	TYPE	VALUE
00000000000000000003	R_X86_64_32S	.data
00000000000000000008	R_X86_64_PC32	puts-0x00000000000000000004

hello.o / statically linked, no PIE

hello.o:

Disassembly of section .text:

```
000000000000000000 <main>:
  0:  48 c7 c7 00 00 00 00    mov     $0x0,%rdi
                          3: R_X86_64_32S .data
  7:  e8 00 00 00 00         call   c <main+0xc>
                          8: R_X86_64_PLT32 puts-0x4
  c:  c3                    ret
```

hello.exe (gcc -static -no-pie):

Disassembly of section .text:

```
...
0000000000004016c3 <main>:
 4016c3:  48 c7 c7 b5 16 40 00    mov     $0x4016b5,%rdi
 4016ca:  e8 e1 a9 00 00         call   40c0b0 <_IO_puts>
 4016cf:  c3                    ret
...
```

symbols in executable

SYMBOL TABLE:

```
00000000000000000000  l      df *ABS*  00000000000000000000 crt1.o
00000000000004002b4  l      0 .note.ABI-tag  00000000000000000020 __abi_tag
```

...

...

by default, non-dynamic symbol information in executables,
but not used to run it

can be stripped (-s linker option or strip command) instead:

SYMBOL TABLE:

```
no symbols
```

exercise: finding without symbols?

how can I find where functions are without symbols?

ideally: some *automated* way to do this?

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 service requests) used by a program

statically linked hello.exe

```
gcc -no-pie -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) = 62
brk(0x20c71c0) = 0x20c71c0
brk(0x20c8000) = 0x20c8000
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
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 -no-pie -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) = 62
brk(0x20c71c0) = 0x20c71c0
brk(0x20c8000) = 0x20c8000
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
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 +++
```

uname, arch_prctl, access, brk —
part of standard library setup (added by compiler)

statically linked hello.exe

```
gcc -no-pie -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) = 62
brk(0x20c71c0) = 0x20c71c0
brk(0x20c8000) = 0x20c8000
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
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 +++
```

fstat, write – implementation of puts

statically linked hello.exe

```
gcc -no-pie -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) = 62
brk(0x20c71c0) = 0x20c71c0
brk(0x20c8000) = 0x20c8000
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
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 +++
```

exit_group – standard library shutdown

dynamically linked hello.exe

```
gcc -no-pie -o hello.exe hello.s
```

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

```
execve("./hello.exe", ["./hello.exe"], [/* 46 vars */) = 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\3\0>\0\1\0\0\0P\t\2\0\0\0\0\0"... , 832) = 832
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 -no-pie -o hello.exe hello.s
```

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

```
execve("./hello.exe", ["/usr/bin/./hello.exe"], [/* 46 vars */) = 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\3\0>\0\1\0\0\0P\t\2\0\0\0\0\0"... , 832) = 832
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 +++
```

**libc.so.6 – contains puts, etc. code
being opened by executable after it starts running**

dynamically linked hello.exe

```
gcc -no-pie -o hello.exe hello.s
```

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

```
execve("./hello.exe", ["./hello.exe"], [/* 46 vars */) = 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\3\0>\0\1\0\0\0P\t\2\0\0\0\0\0"... , 832) = 832
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)
exit_group(14)
+++ exited with 14 +++
```

code reading from libc.so.6's header
gets contents of all the LOAD directives

dynamically linked hello.exe

```
gcc -no-pie -o hello.exe hello.s
```

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

```
execve("./hello.exe", ["./hello.exe"], [/* 46 vars */]) = 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\3\0>\0\1\0\0\0P\t\2\0\0\0\0\0"... , 832) = 832
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(0) = 0
+++ ex mmap: used to load code/data from library into memory (3 = opened file)
```

dynamically linked hello.exe

```
gcc -no-pie -o hello.exe hello.s
```

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

```
execve("./hello.exe", ["./hello.exe"], [/* 46 vars */) = 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\3\0>\0\1\0\0\0P\t\2\0\0\0\0\0"... , 832) = 832
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
```

mmap: used to allocate zero-initialized data segment

dynamically linked hello.exe

```
gcc -no-pie -o hello.exe hello.s
```

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

```
execve("./hello.exe", ["./hello.exe"], [/* 46 vars */) = 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\3\0>\0\1\0\0\0P\t\2\0\0\0\0\0"... , 832) = 832
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 +++
```

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      0x0000238 vaddr 0x0400318 paddr 0x0400238 align 2**0  
        filesz 0x000001c memsz 0x000001c flags r--
```

...

Contents of section `.interp`:

```
400318 2f6c6962 36342f6c 642d6c69 6e75782d /lib64/ld-linux-  
400328 7838362d 36342e73 6f2e3200 x86-64.so.2.
```

dynamic linking: what to load? (1)

excerpt from `objdump -sx hello.exe`:

Program Header:

```
...  
DYNAMIC off      0x000000000000002e20 vaddr 0x00000000000403e20 paddr 0x00000000000403e20 align 2**3  
          filesz 0x000000000000001d0 memsz 0x000000000000001d0 flags rw-
```

...

Dynamic Section:

```
NEEDED          libc.so.6  
INIT             0x00000000000401000
```

...

```
STRTAB           0x00000000000400420
```

...

program header: identifies where dynamic linking info is

dynamic linking info: array of key-value pairs

- needed libraries

- constructor locations ('INIT')

- string table location

dynamic linking: what to load? (2)

excerpt from `objdump -sx hello.exe`:

Program Header:

...

```
DYNAMIC off      0x000000000000002e20 vaddr 0x00000000000403e20 paddr 0x00000000000403e20 align 2**3
          filesz 0x000000000000001d0 memsz 0x000000000000001d0 flags rw-
```

...

Dynamic Section:

```
NEEDED          libc.so.6
INIT            0x00000000000401000
...
STRTAB          0x00000000000400420
```

...

...

```
403e20 01000000 00000000 01000000 00000000 .....
403e30 0c000000 00000000 00104000 00000000 .....
```

type 0x1 = “DT_NEEDED” (from ELF manual)

value 0x1 = string table entry 1

dynamic linking: what to load? (2)

excerpt from `objdump -sx hello.exe`:

Program Header:

```
...  
DYNAMIC off      0x000000000000002e20 vaddr 0x00000000000403e20 paddr 0x00000000000403e20 align 2**3  
      filesz 0x000000000000001d0 memsz 0x000000000000001d0 flags rw-  
...
```

Dynamic Section:

```
NEEDED          libc.so.6  
INIT           0x00000000000401000  
...  
STRTAB         0x00000000000400420  
...  
...  
403e20 01000000 00000000 01000000 00000000 .....  
403e30 0c000000 00000000 00104000 00000000 .....
```

type `0x1 = "DT_NEEDED"` (from ELF manual)

value `0x1` = string table entry 1

dynamic linking: what to load? (2)

excerpt from `objdump -sx hello.exe`:

Program Header:

```
...  
DYNAMIC off      0x000000000000002e20 vaddr 0x000000000000403e20 paddr 0x000000000000403e20 align 2**3  
      filesz 0x000000000000001d0 memsz 0x000000000000001d0 flags rw-  
...
```

Dynamic Section:

```
NEEDED          libc.so.6  
INIT           0x000000000000401000  
...  
STRTAB        0x000000000000400420  
...  
...  
403e20 01000000 00000000 01000000 00000000 .....  
403e30 0c000000 00000000 00104000 00000000 .....
```

type `0x1` = “DT_NEEDED” (from ELF manual)

value `0x1` = *string table entry 1*

dynamic linking: what to load? (2)

excerpt from `objdump -sx hello.exe`:

Program Header:

...

```
DYNAMIC off      0x000000000000002e20 vaddr 0x00000000000403e20 paddr 0x00000000000403e20 align 2**3
          filesz 0x000000000000001d0 memsz 0x000000000000001d0 flags rw-
```

...

Dynamic Section:

```
NEEDED          libc.so.6
INIT            0x00000000000401000
```

...

```
STRTAB          0x00000000000400420
```

...

...

```
403e20 01000000 00000000 01000000 00000000 .....
403e30 0c000000 00000000 00104000 00000000 .....
```

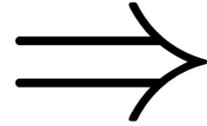
type 0x1 = "DT_NEEDED" (from ELF manual)

value 0x1 = string table entry 1

adding linker stubs

program.s:

```
main:
    ...
    call puts
    ...
```



program.exe (psuedo-assembly):

```
GLOBAL_LOOKUP_TABLE:
    ... // placeholders

puts@plt:
    jmp *GLOBAL_LOOKUP_TABLE[index]
    ...
    ...

main:
    ...
    call puts@plt
    ...
```

+ annotation where to fill puts address in lookup table

dynamic linking information

symbol table in libraries: list of functions/variables to find
with their locations in the library

relocation records in programs: list of functions/variables
with locations (probably in lookup table) to fill in

separate symbol/relocation tables for dynamic linking than
static

dynamically linked puts (non-lazy)

DYNAMIC RELOCATION RECORDS

OFFSET	TYPE	VALUE
0000000000403ff0	R_X86_64_GLOB_DAT	__libc_start_main@GLIBC_2.34
0000000000403ff8	R_X86_64_GLOB_DAT	__gmon_start__@Base
0000000000403fe8	R_X86_64_JUMP_SLOT	puts@GLIBC_2.2.5

...

Text:

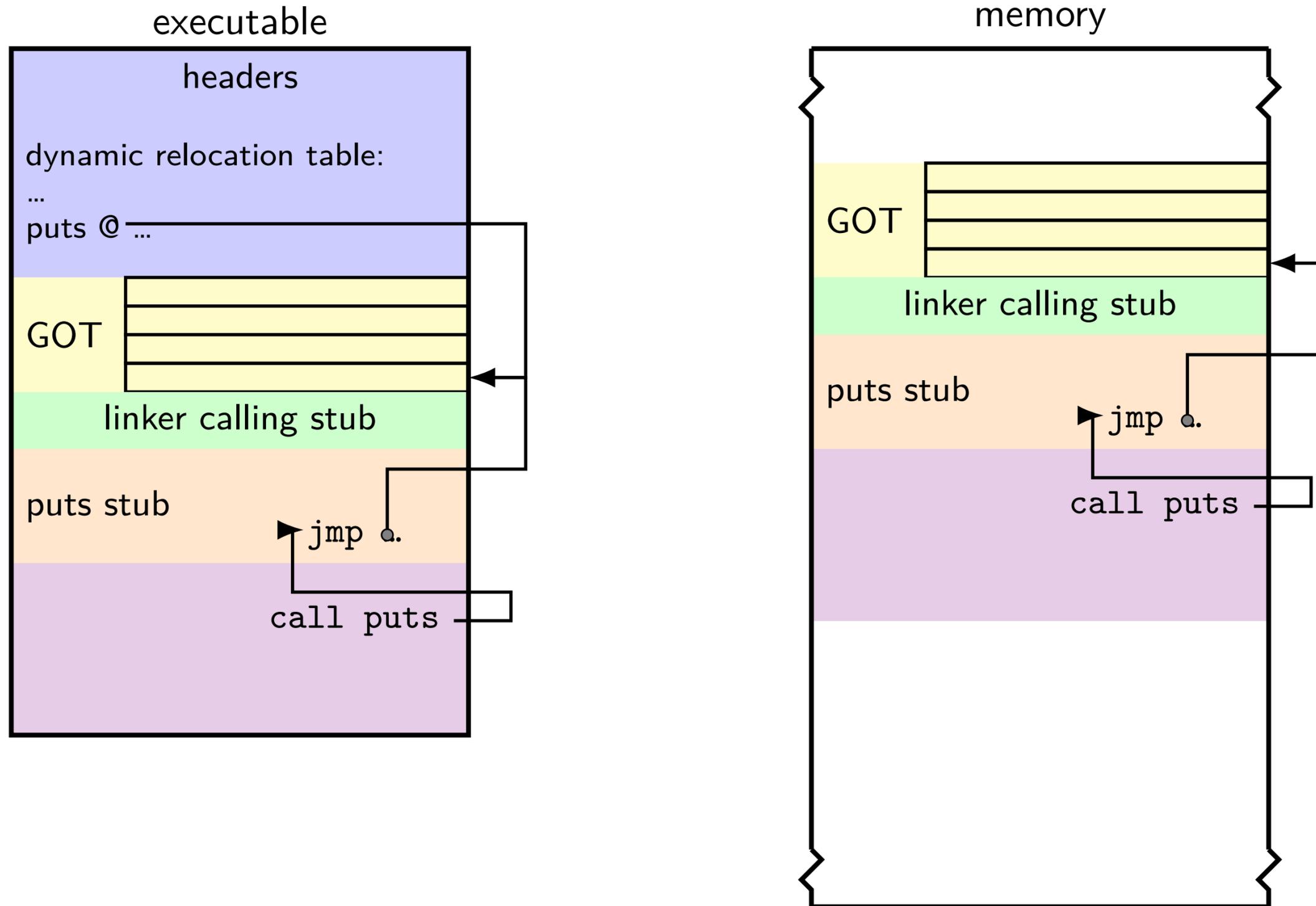
```
0000000000401030 <puts@plt>:  
  401030:      ff 25 b2 2f 00 00      jmp     *0x2fb2(%rip)      # 403fe8 <puts@GLIBC_2.2.5>
```

stub reads pointer from 0x403fe8, jump to location

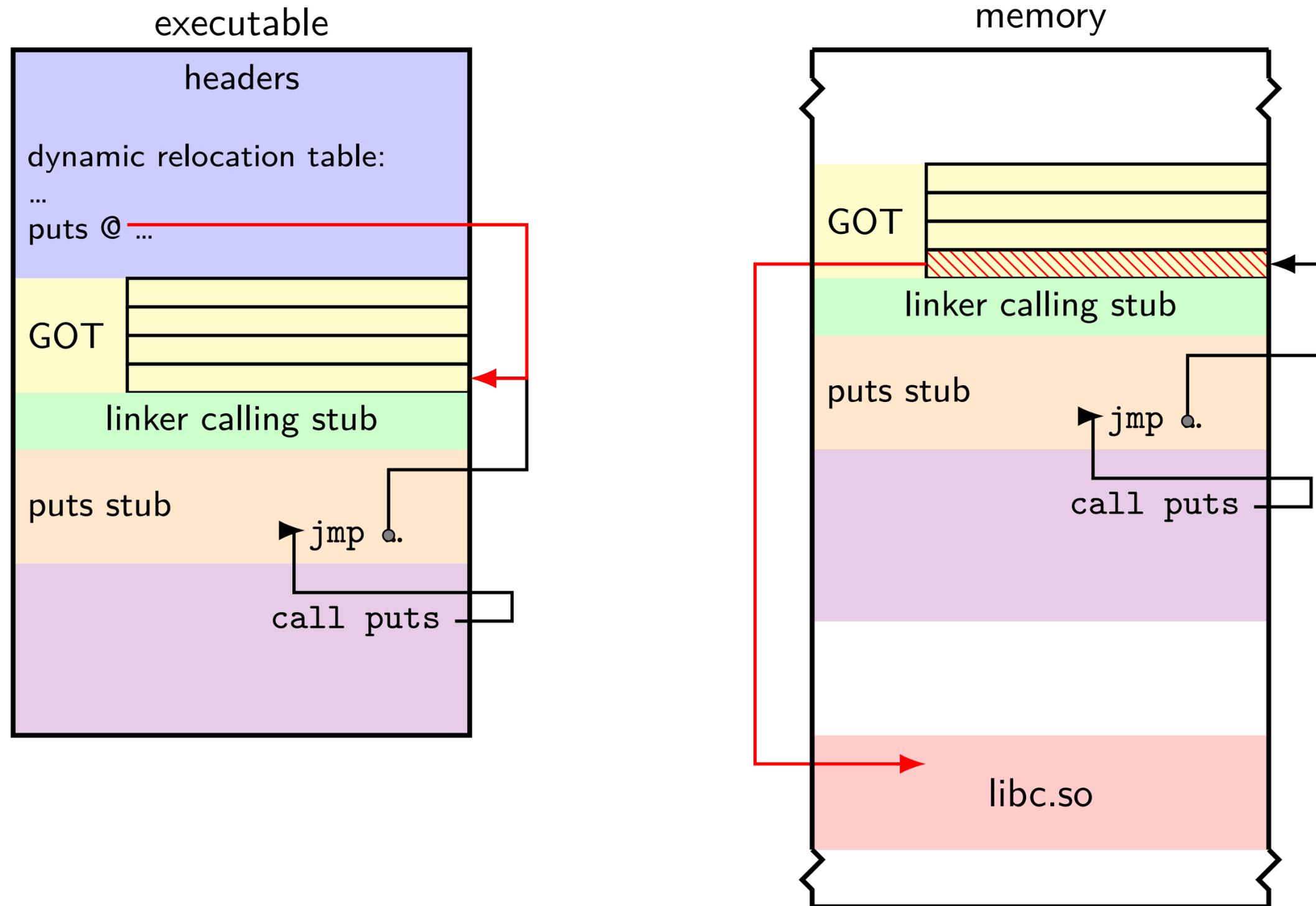
0x403fe8 part of 'global offset table' (GOT)

relocation table entry indicates where puts pointer goes

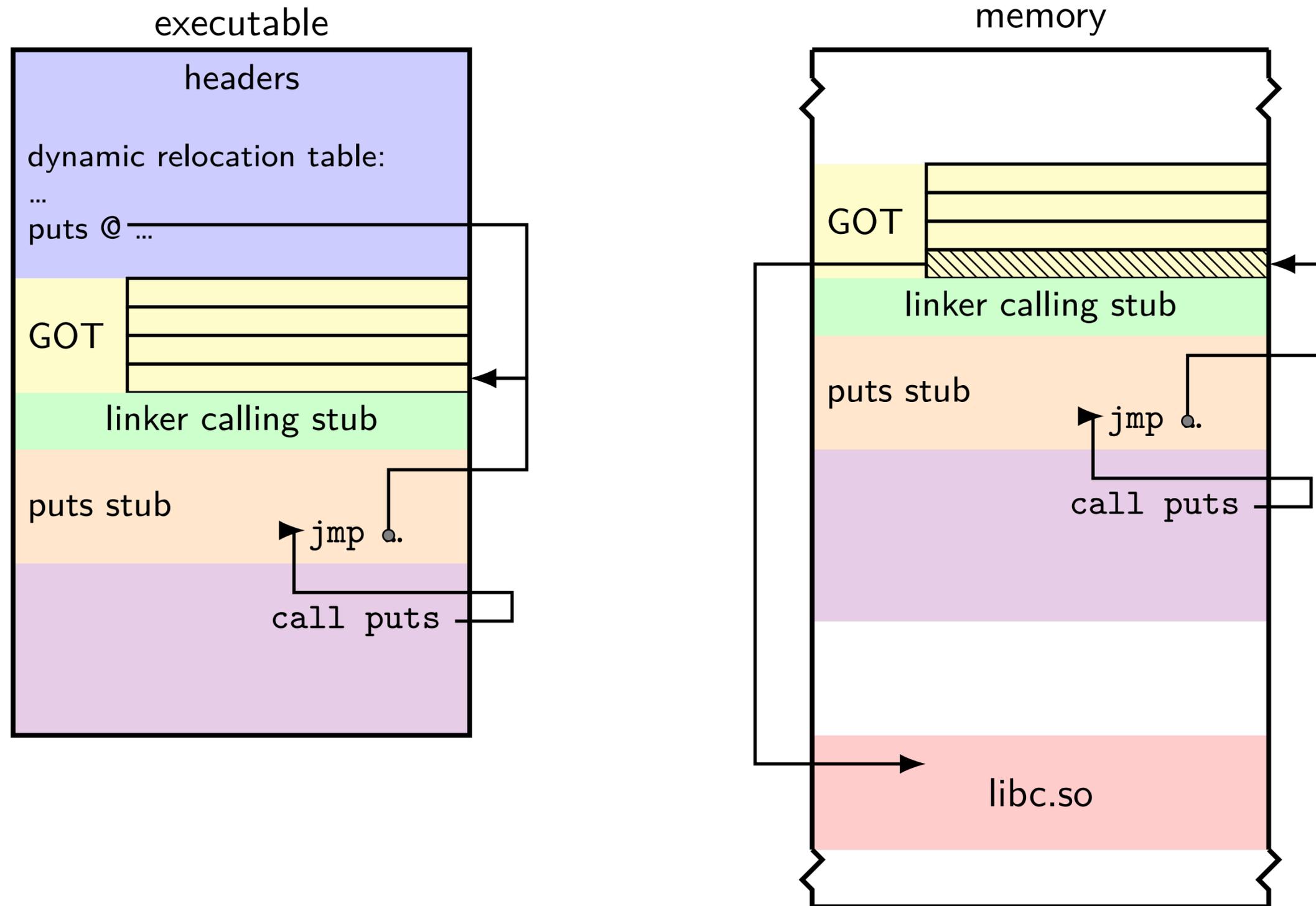
dynamic puts (picture; no lazy)



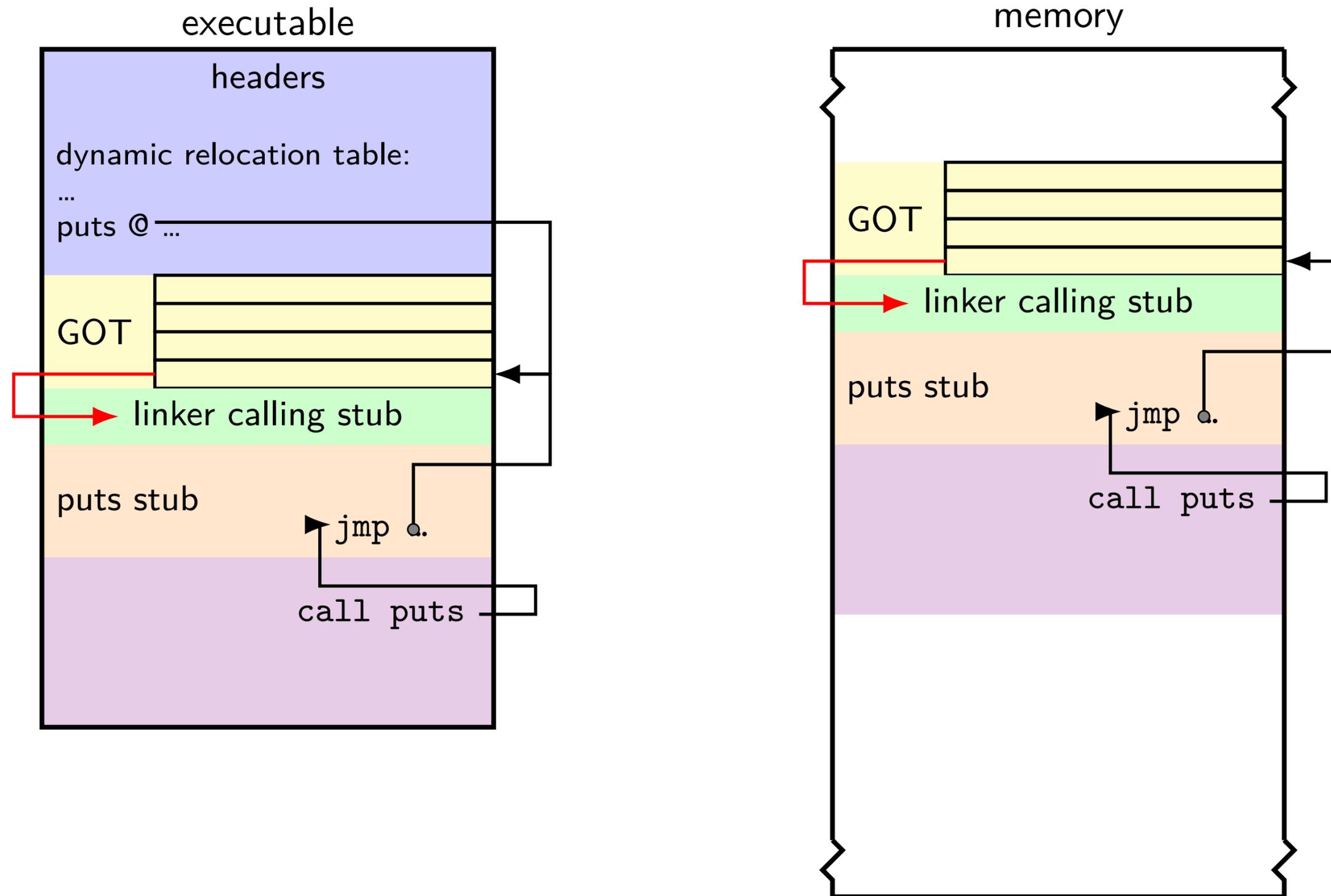
dynamic puts (picture; no lazy)



dynamic puts (picture; no lazy)



dynamic puts (picture; no lazy)



lazy binding

```
0000000000401030 <puts@plt>:
  401030:  ff 25 62 22 00 00  jmp *0x2262(%rip)  # 403298 <puts@GLIBC_2.2.5>
...
Contents of section .got.plt:
 403280 a0304000 00000000 00000000 00000000  .0@.....
 403290 00000000 00000000 36104000 00000000  .....6.@.....
```

initial contents of 0x403298 = 0x401036 (in .got.plt)
not part of standard library

code found at 0x401036 invokes dynamic linker code:

```
401020:  ff 35 62 22 00 00  push 0x2262(%rip)
  # 403288 <_GLOBAL_OFFSET_TABLE_+0x8>
401026:  ff 25 64 22 00 00  jmp *0x2264(%rip)
  # 403290 <_GLOBAL_OFFSET_TABLE_+0x10>
...
401036:  68 00 00 00 00  push $0x0
40103b:  e9 e0 ff ff ff  jmp 401020 <_init+0x20>
```

lazy binding

with lazy binding turned on (not always done)

GOT loaded with address of linker routine hard-coded in executable

first call to puts:

- invoke dynamic linker routine pointed to by GOT

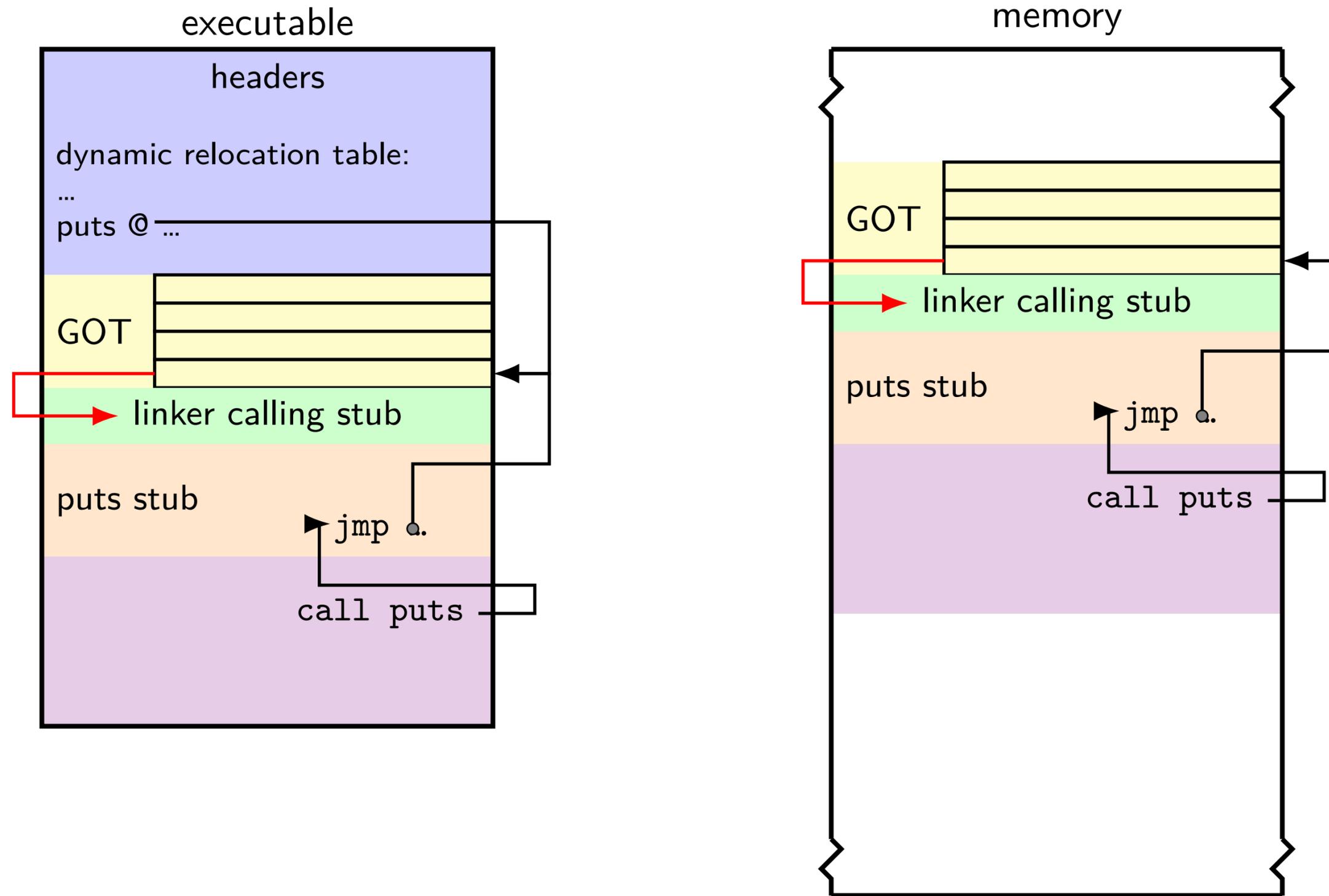
- linker routine fills in puts address in 0x404018

- then jumps to puts

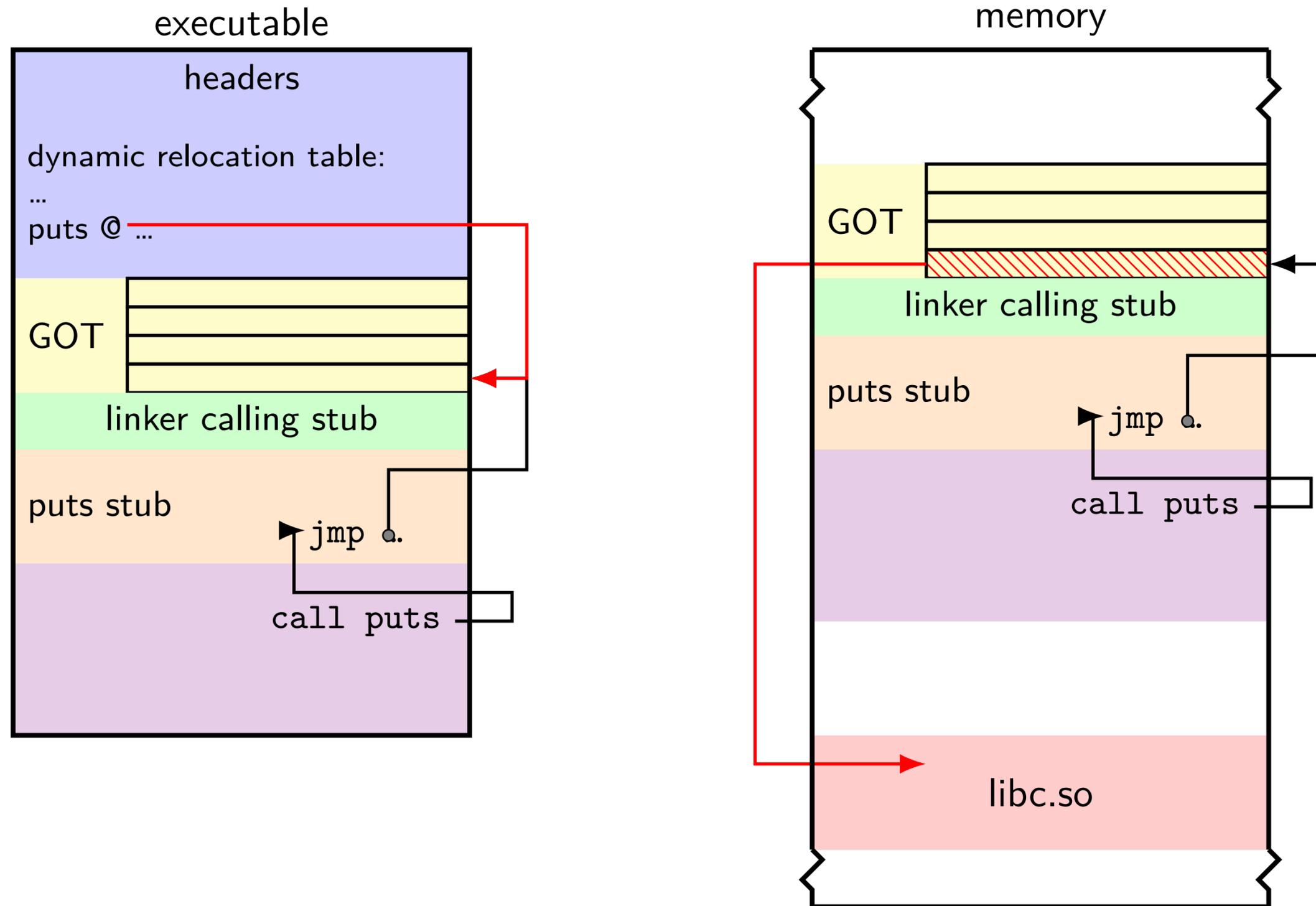
second (and later) call to puts

- 0x404018 contains real address of puts, no indirection

dynamic puts (picture; lazy setup)



dynamic puts (picture; lazy setup)



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

- means global offset table needs to be writable?

preview: exploits and dynamic linking

later we'll talk about memory error exploits

- buffer overflows, etc.

common goal: convert memory overwrite to running code

- bug that "just" allows overwriting memory somewhere
easy to cause crashes...

- but not so easy to do something in particular

global offset table: function pointers in known location

- useful to overwrite in exploits

backup slides

dynamic linking: what to load? (2)

excerpt from `objdump -sx hello.exe`:

Program Header:

```
...  
DYNAMIC off      0x000000000000002e20 vaddr 0x00000000000403e20 paddr 0x00000000000403e20 align 2**3  
          filesz 0x000000000000001d0 memsz 0x000000000000001d0 flags rw-
```

...

Dynamic Section:

```
NEEDED          libc.so.6  
INIT           0x00000000000401000  
...  
STRTAB         0x00000000000400420
```

...

```
...  
403e20 01000000 00000000 01000000 00000000 .....  
403e30 0c000000 00000000 00104000 00000000 .....
```

type 0x1 = "DT_NEEDED" (from ELF manual)

value 0x1 = string table entry 1

dynamic linking: what to load? (2)

excerpt from `objdump -sx hello.exe`:

Program Header:

```
...  
DYNAMIC off      0x000000000000002e20 vaddr 0x00000000000403e20 paddr 0x00000000000403e20 align 2**3  
          filesz 0x000000000000001d0 memsz 0x000000000000001d0 flags rw-
```

...

Dynamic Section:

```
NEEDED          libc.so.6  
INIT           0x00000000000401000  
...  
STRTAB         0x00000000000400420
```

...

```
...  
403e20 01000000 00000000 01000000 00000000 .....  
403e30 0c000000 00000000 00104000 00000000 .....
```

type `0x1 = "DT_NEEDED"` (from ELF manual)

value `0x1` = string table entry 1

dynamic linking: what to load? (2)

excerpt from `objdump -sx hello.exe`:

Program Header:

```
...  
DYNAMIC off      0x000000000000002e20 vaddr 0x000000000000403e20 paddr 0x000000000000403e20 align 2**3  
          filesz 0x000000000000001d0 memsz 0x000000000000001d0 flags rw-
```

Dynamic Section:

```
NEEDED          libc.so.6  
INIT            0x000000000000401000  
...  
STRTAB         0x000000000000400420  
...  
...  
403e20 01000000 00000000 01000000 00000000 .....  
403e30 0c000000 00000000 00104000 00000000 .....
```

type `0x1` = “DT_NEEDED” (from ELF manual)

value `0x1` = *string table entry 1*

dynamic linking: what to load? (2)

excerpt from `objdump -sx hello.exe`:

Program Header:

...

```
DYNAMIC off      0x000000000000002e20 vaddr 0x00000000000403e20 paddr 0x00000000000403e20 align 2**3
          filesz 0x000000000000001d0 memsz 0x000000000000001d0 flags rw-
```

...

Dynamic Section:

```
NEEDED          libc.so.6
INIT           0x00000000000401000
```

...

```
STRTAB         0x00000000000400420
```

...

...

```
403e20 01000000 00000000 01000000 00000000 .....
403e30 0c000000 00000000 00104000 00000000 .....
```

type 0x1 = “DT_NEEDED” (from ELF manual)

value 0x1 = string table entry 1
