

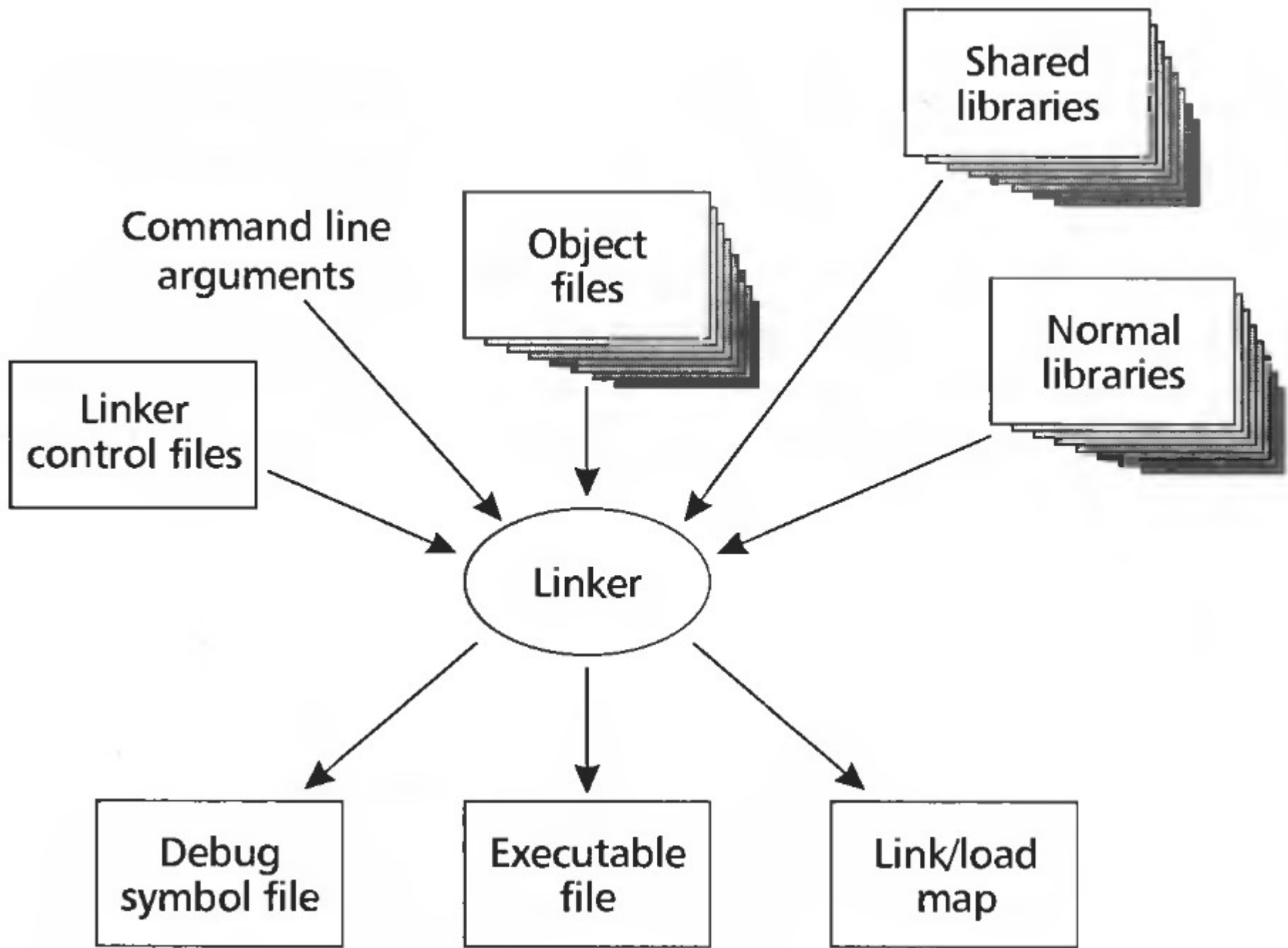
ICS143A: Principles of Operating Systems

Lecture 21: Program linking and loading

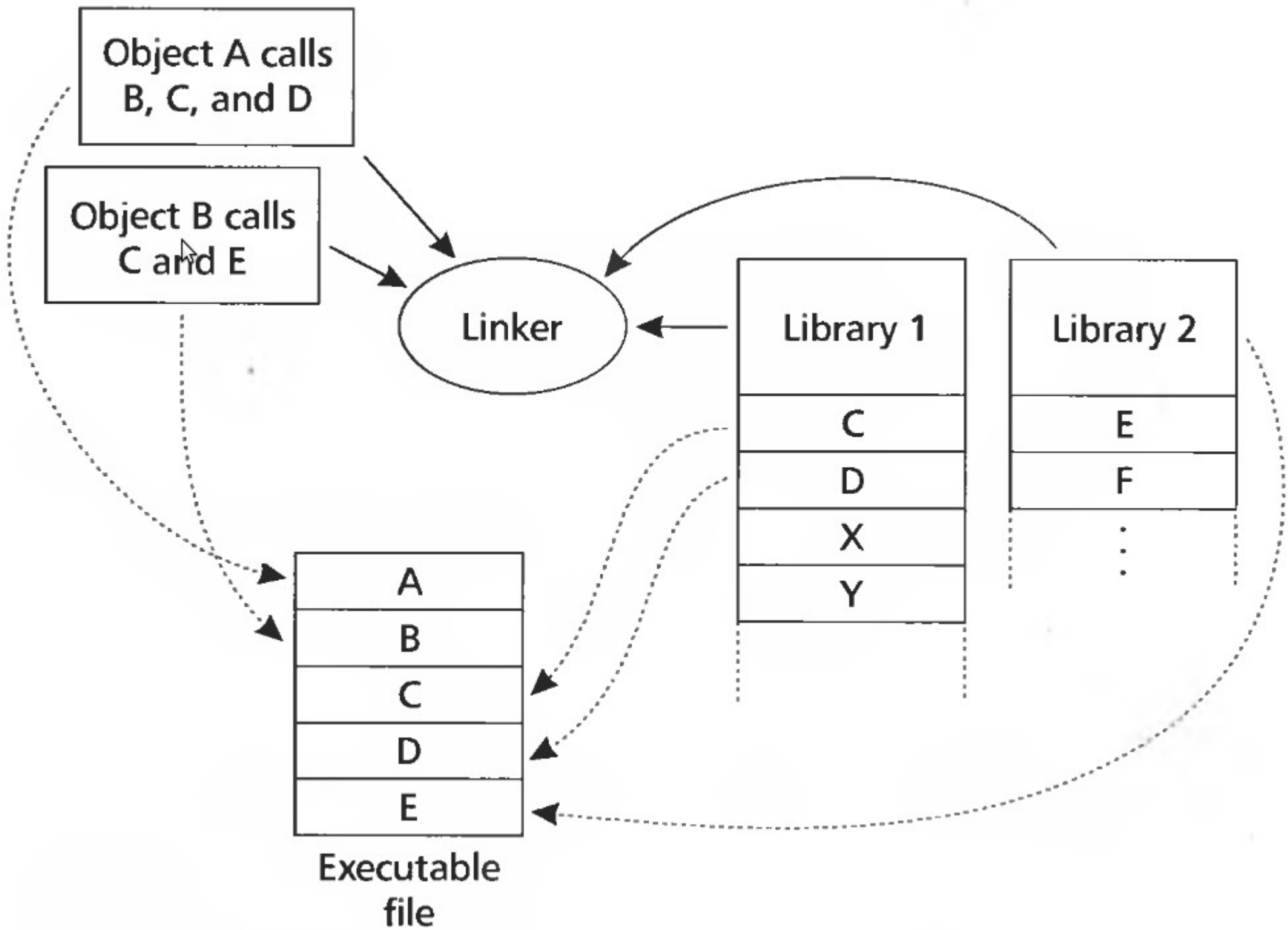
Anton Burtsev
March, 2017

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?

- 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
```

- 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

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

- Source file a.c

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

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

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
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

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

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

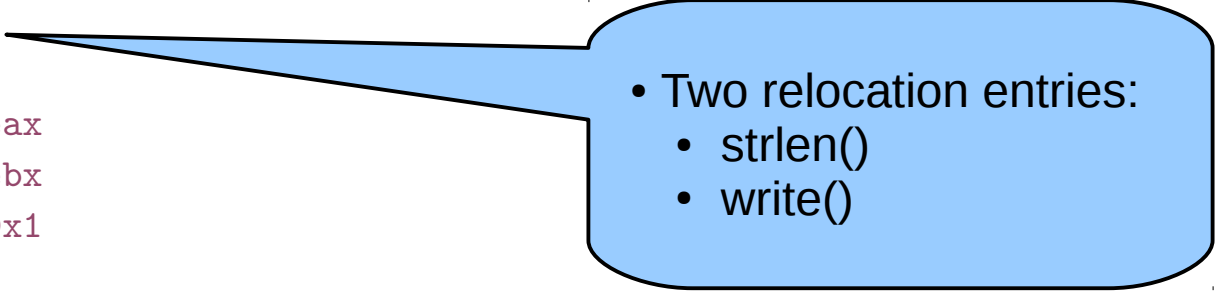
Sections:

```
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>:

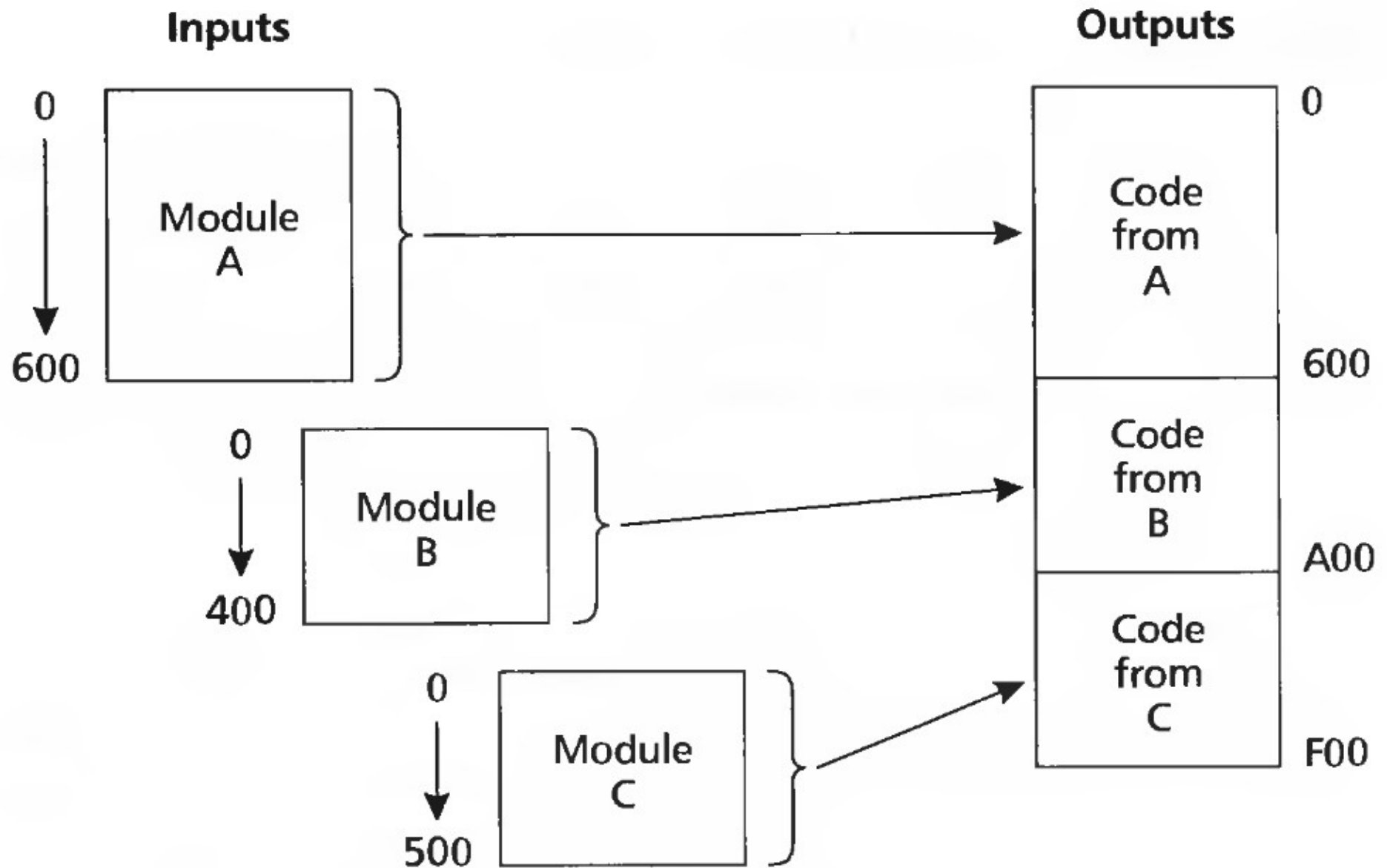
```
0: 55                pushl %ebp
1: 89 e5            movl %esp,%ebp
3: 53              pushl %ebx
4: 8b 5d 08        movl 0x8(%ebp),%ebx
7: 53              pushl %ebx
8: e8 f3 ff ff ff  call 0
 9: DISP32 _strlen
d: 50              pushl %eax
e: 53              pushl %ebx
f: 6a 01          pushl $0x1
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
```

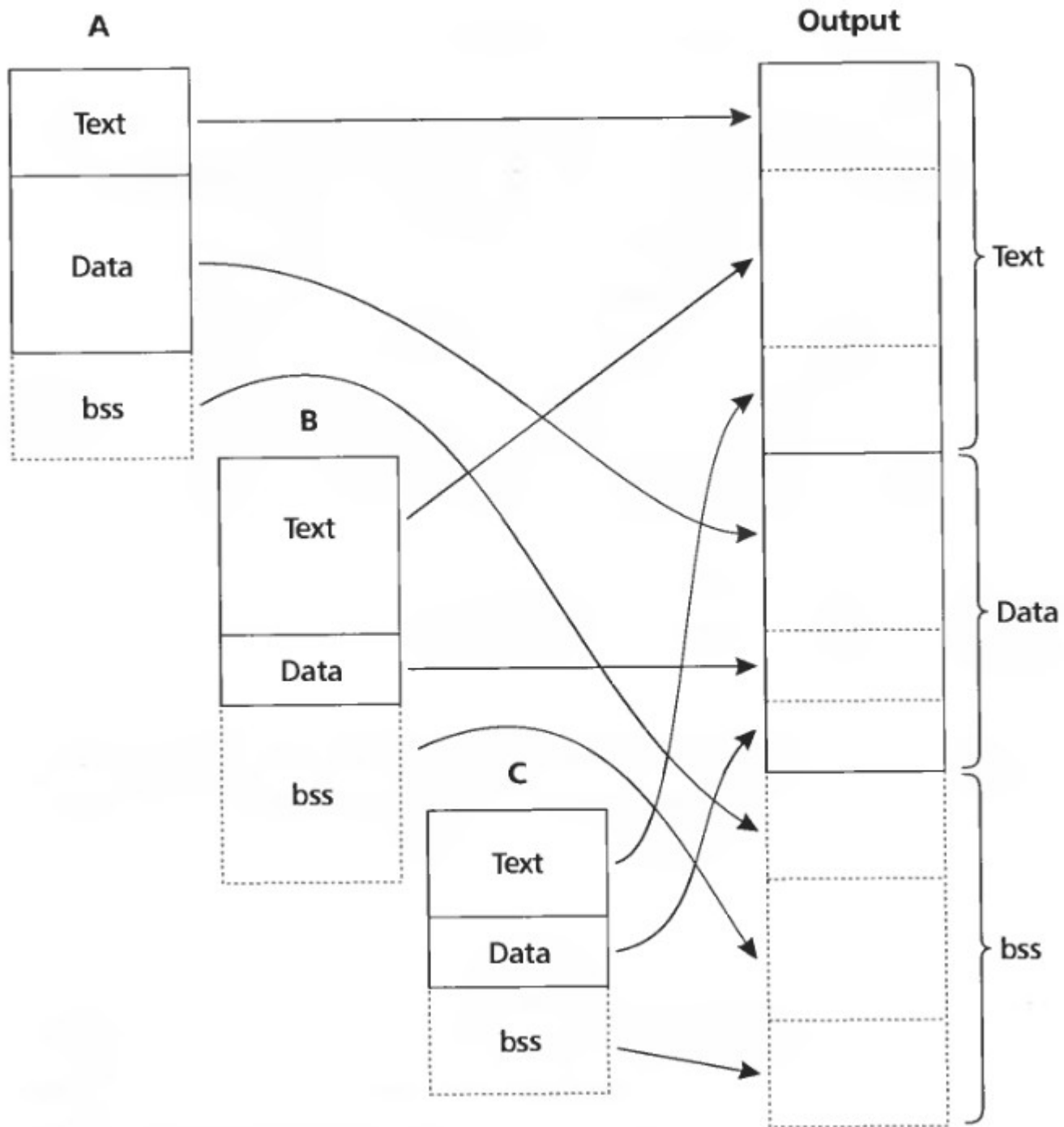
- 
- Two relocation entries:
 - strlen()
 - write()

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

Tasks involved

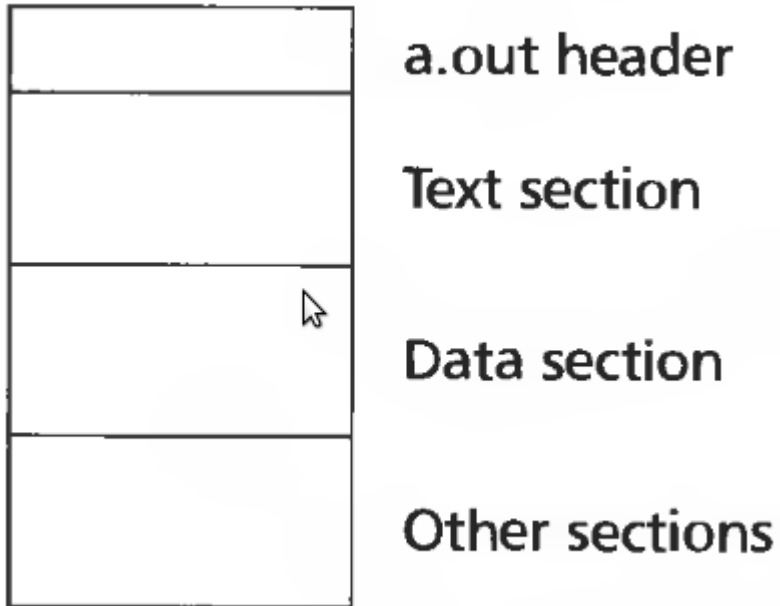
- Program loading
 - Copy a program from disk to memory so it is ready to run
 - Allocation of memory
 - Setting protection bits (e.g. read only)
- Relocation
 - Assign load address to each object file
 - Adjust the code
- Symbol resolution
 - Resolve symbols imported from other object files

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

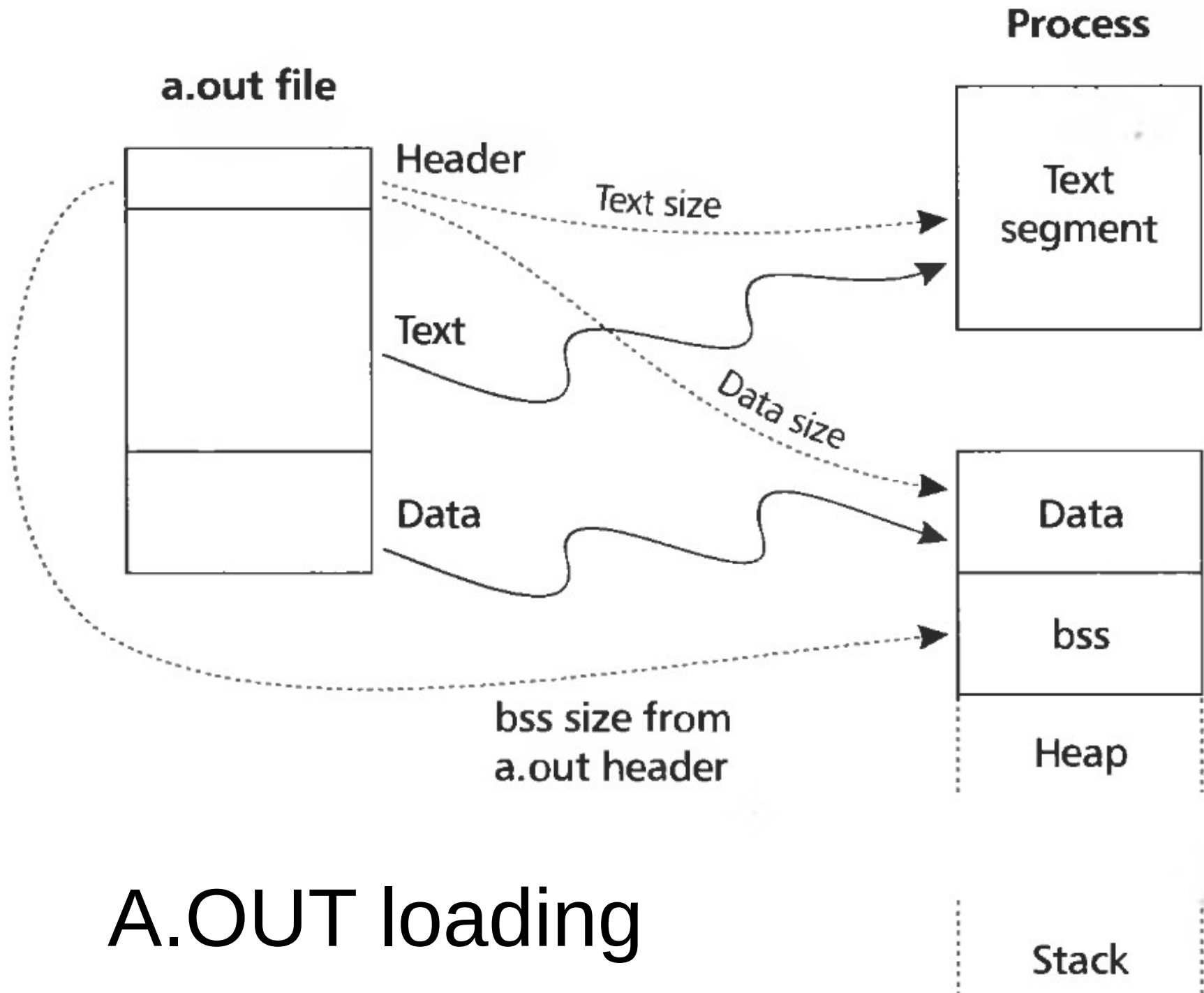
Example: UNIX A.OUT



- 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
```



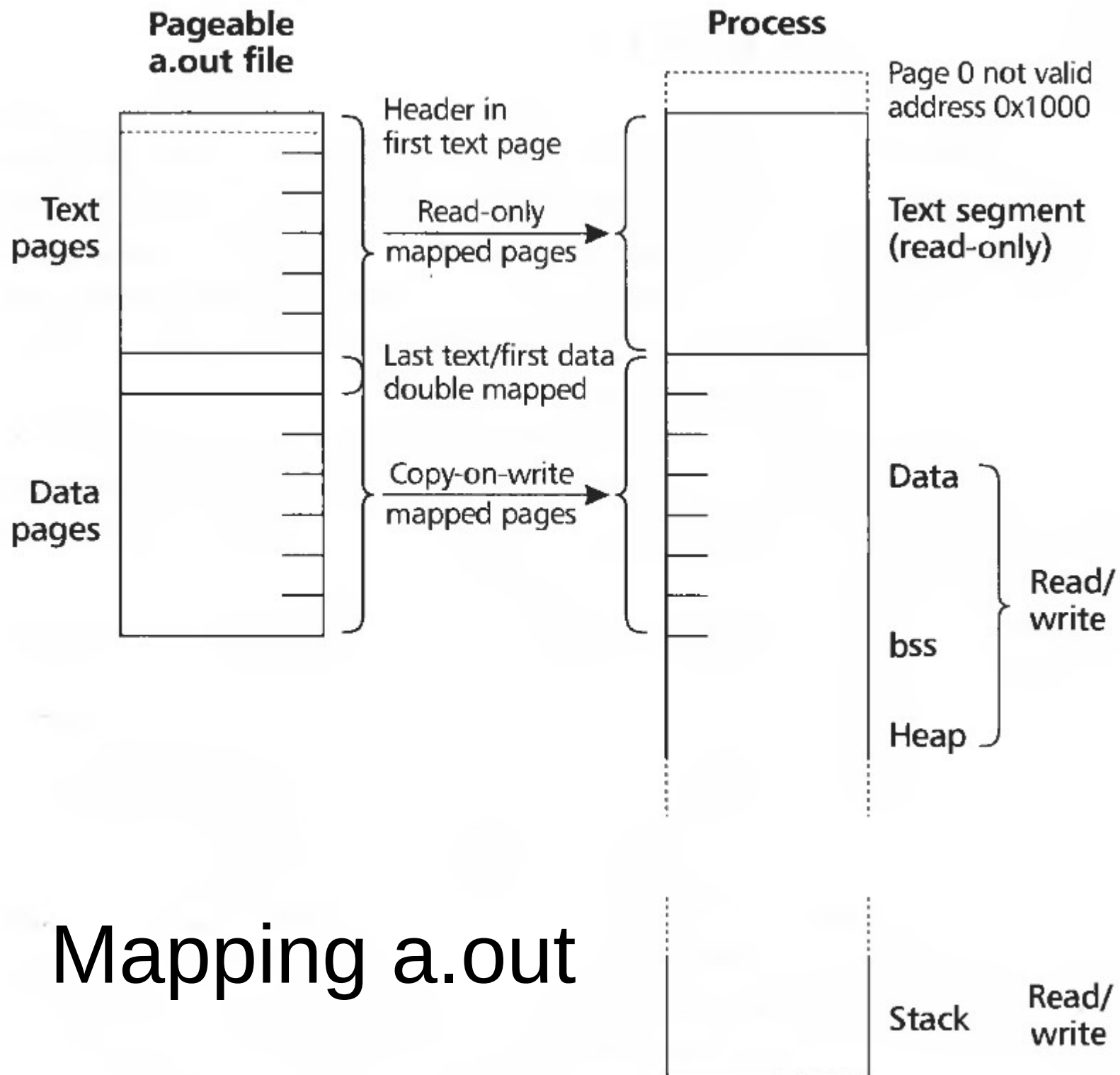
A.OUT loading

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

Interaction with virtual memory

- Virtual memory unifies paging and file I/O
 - Memory mapped files
 - Memory pages are backed up by files
 - Loading a segment is just mapping it into memory
- Linker must provide some support
 - Sections are page aligned



Mapping a.out

Relocation

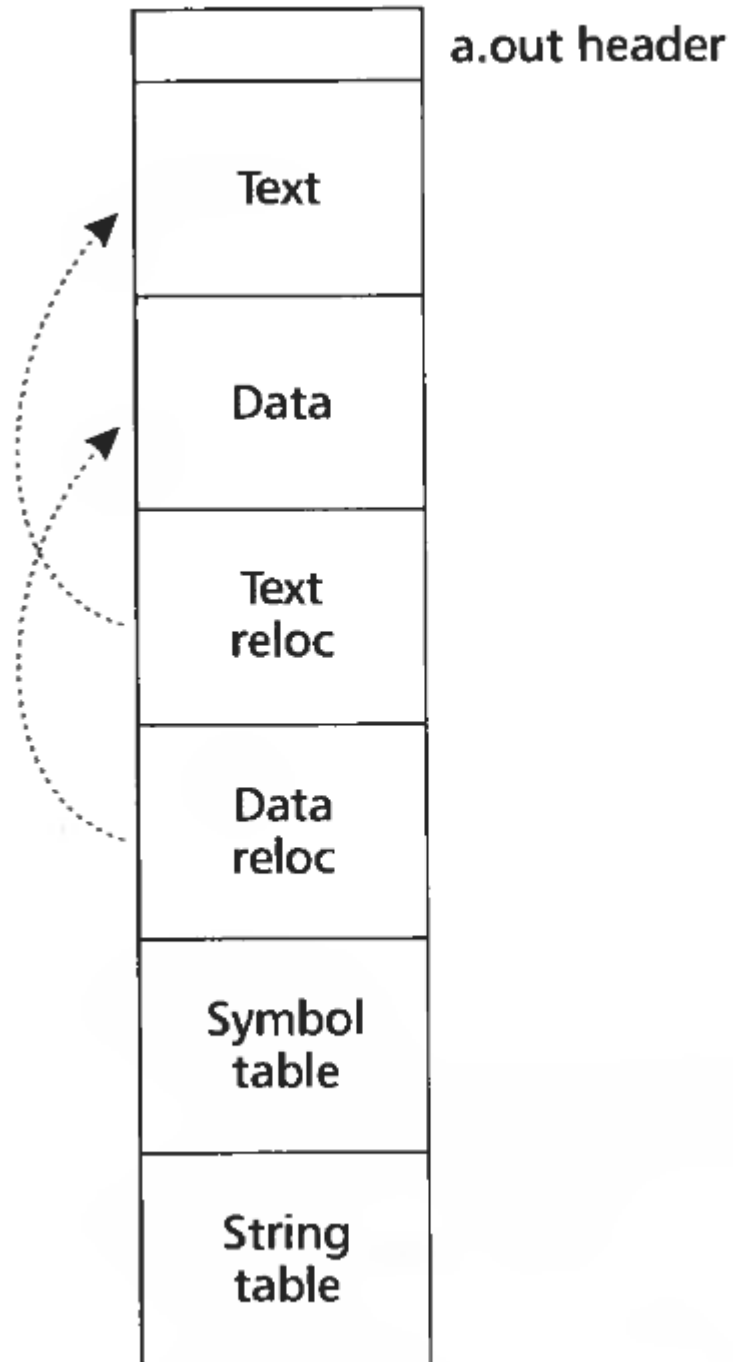
Relocation, why?

- Each program gets its own private space, why relocate?
 - Linkers combine multiple libraries into a single executable
 - Each library assumes private address space
 - E.g., starts at 0x0
- Is it possible to go away with segments?
 - Each library gets a private segment (starts at 0x0)
 - All cross-library references are patched to use segment numbers
- Possible!
 - But slow.
 - Segment lookups are slow

Relocation

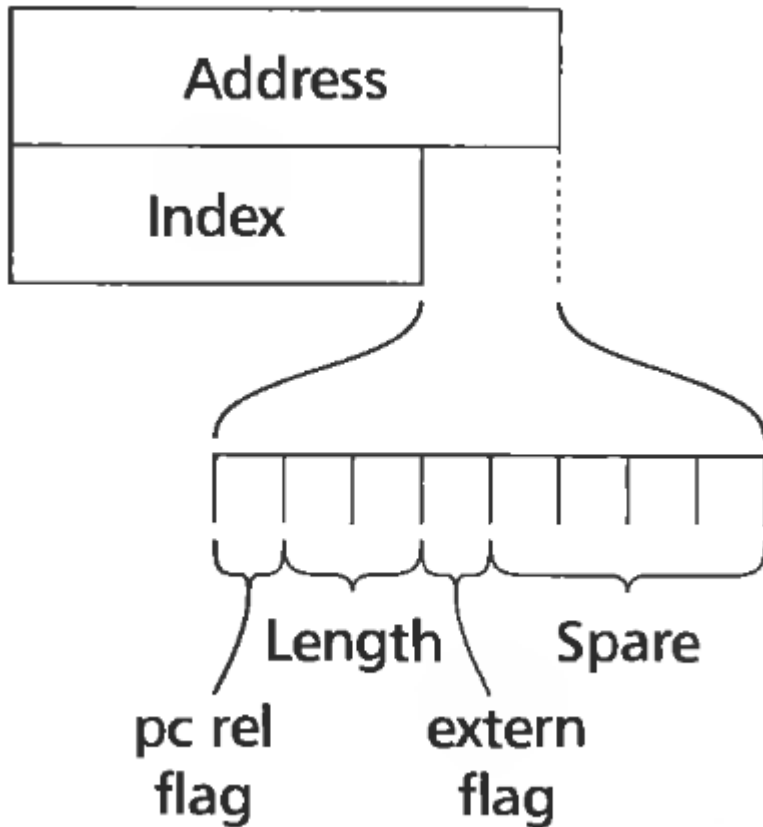
- Each relocatable object file contains a relocation table
 - List of places in each segment that need to be relocated
 - Example:
 - Pointer in the text segment points to offset 200 in the data segment
 - Input file: text starts at 0, data starts at 2000, stored pointer has value 2200
 - Output file: Data segment starts at 15000
 - Linker adds relocated base of the data segment 13000 (DR)
 - Output file: will have pointer value of 15200
 - All jumps are relative on x86
 - No need to relocate
 - Unless its a cross-segment jump, e.g. text segment to data segment

Relocatable A.OUT



- Add relocation information for each section

Relocation entries



- Address relative to the segment
- Length
 - 1, 2, 4, 8 bytes
- Extern
 - Local or extern symbol
- Index
 - Segment number if local
 - Index in the symbol table

Symbol table

- Name offset
 - Offset into the string table
 - UNIX supports symbols of any length
 - Null terminated strings
- Type
 - Whether it is visible to other modules

Name offset		
Type	Spare	Debug info
Value		

We ran out of time here

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 any 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

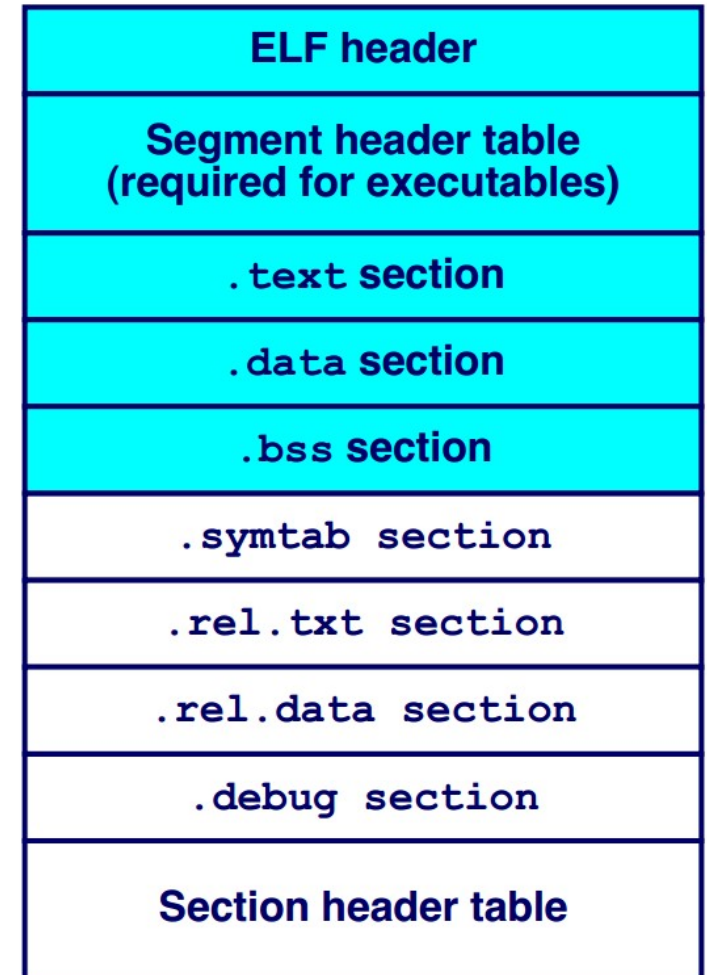
- Code

.data section

- Initialized global variables

.bss section

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



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

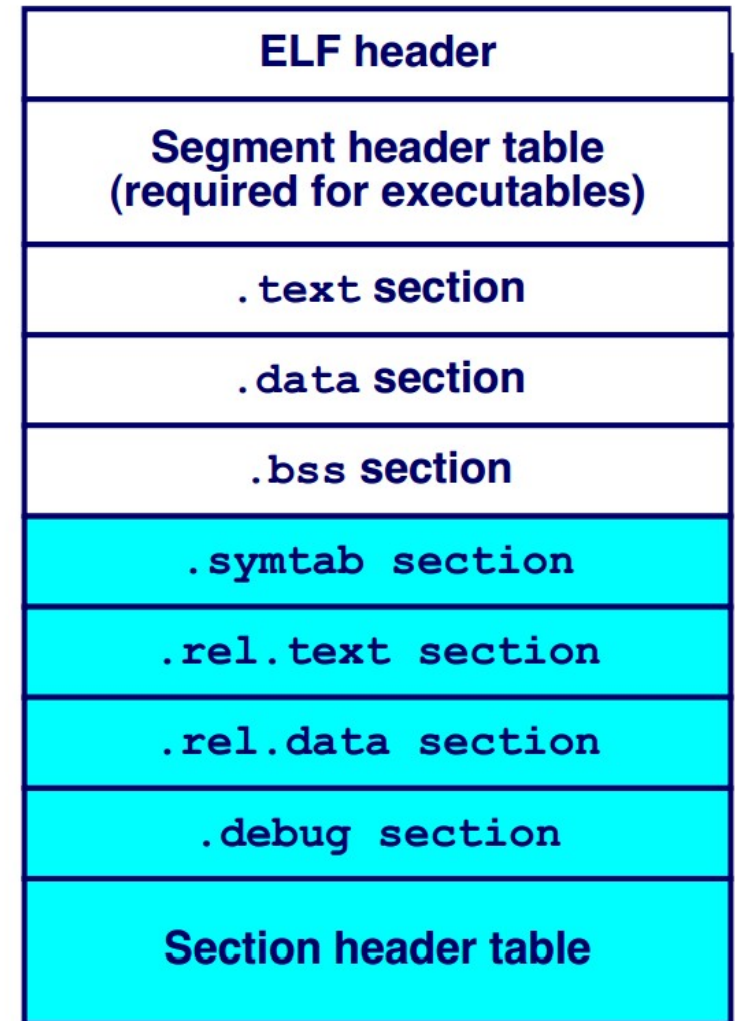
- Relocation info for `.data` section
- Addresses of pointer data that will need to be modified in the merged executable

`.debug` section

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

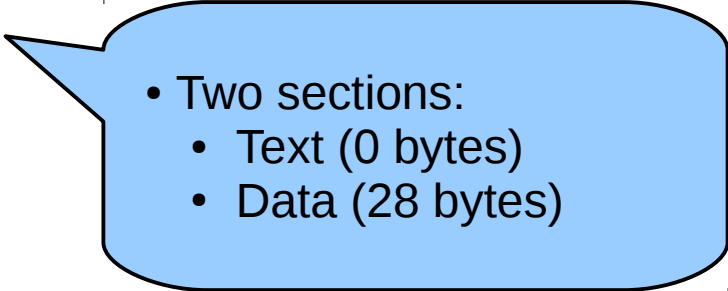
Section header table

- Offsets and sizes of each section



Xv6: exec()

```
6310 exec(char *path, char **argv)
6311 {
...
6320     begin_op();
6321     if((ip = namei(path)) == 0){
6322         end_op();
6323         return -1;
6324     }
6325     ilock(ip);
6326     pgdir = 0;
6327
6328     // Check ELF header
6329     if(readi(ip, (char*)&elf, 0, sizeof(elf)) < sizeof(elf))
6330         goto bad;
6331     if(elf.magic != ELF_MAGIC)
6332         goto bad;
6333
...
6337     // Load program into memory.
6338     sz = 0;
6339     for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
6340         if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
6341             goto bad;
6342         ...
```

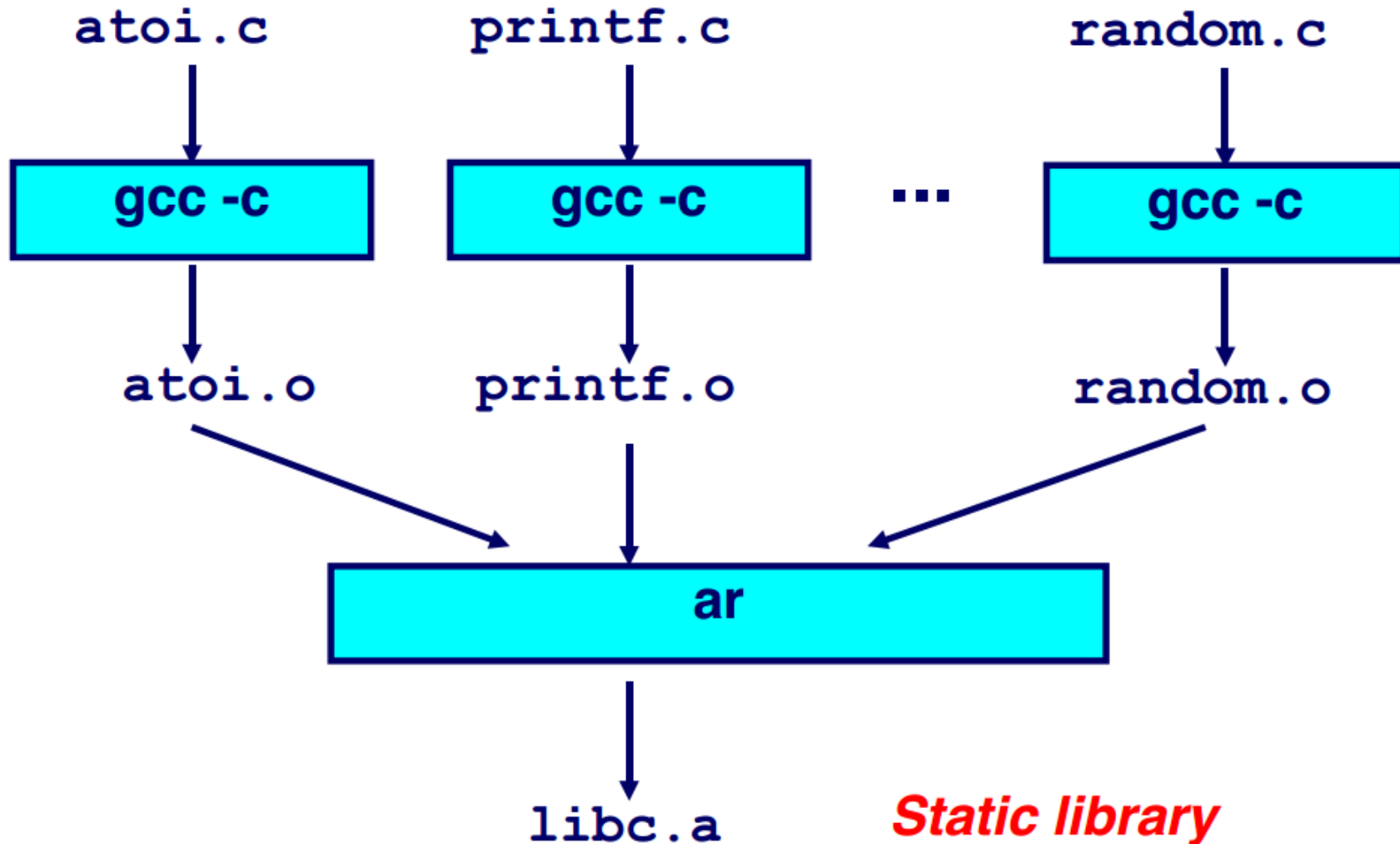
- 
- Two sections:
 - Text (0 bytes)
 - Data (28 bytes)

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



Searching libraries

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

Shared libraries

Motivation

- 1000 programs in a typical UNIX system
- 1000 copies of printf

- **How big is printf actually?**

Motivation

- 1000 programs in a typical UNIX system
- 1000 copies of printf
 - **Printf is a large function**
 - **Handles conversion of multiple types to strings**
 - **5-10K**
- This means 5-10MB of disk is wasted on printf
- Runtime memory costs are
 - 10K x number of running programs

Position independent code

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

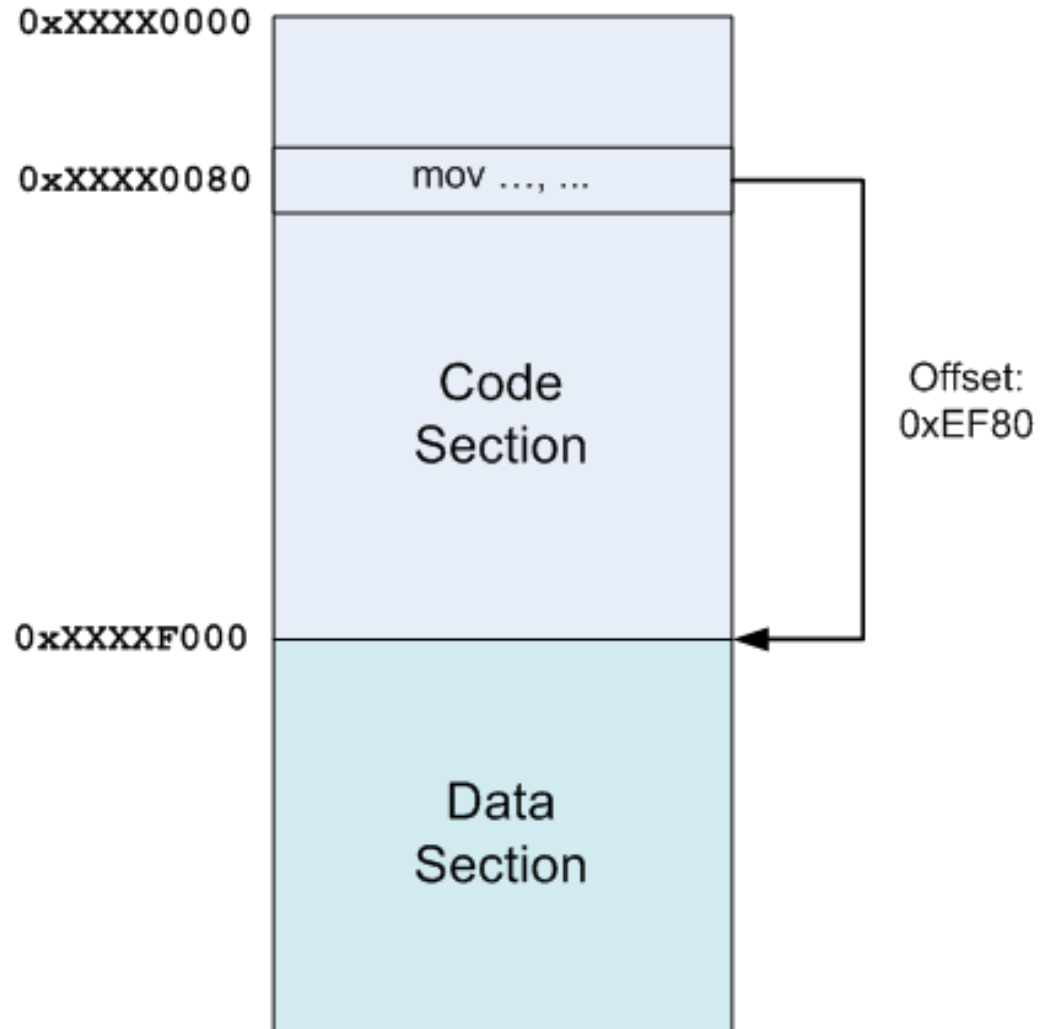
Position independent code

- How would you build it?

Position independent code

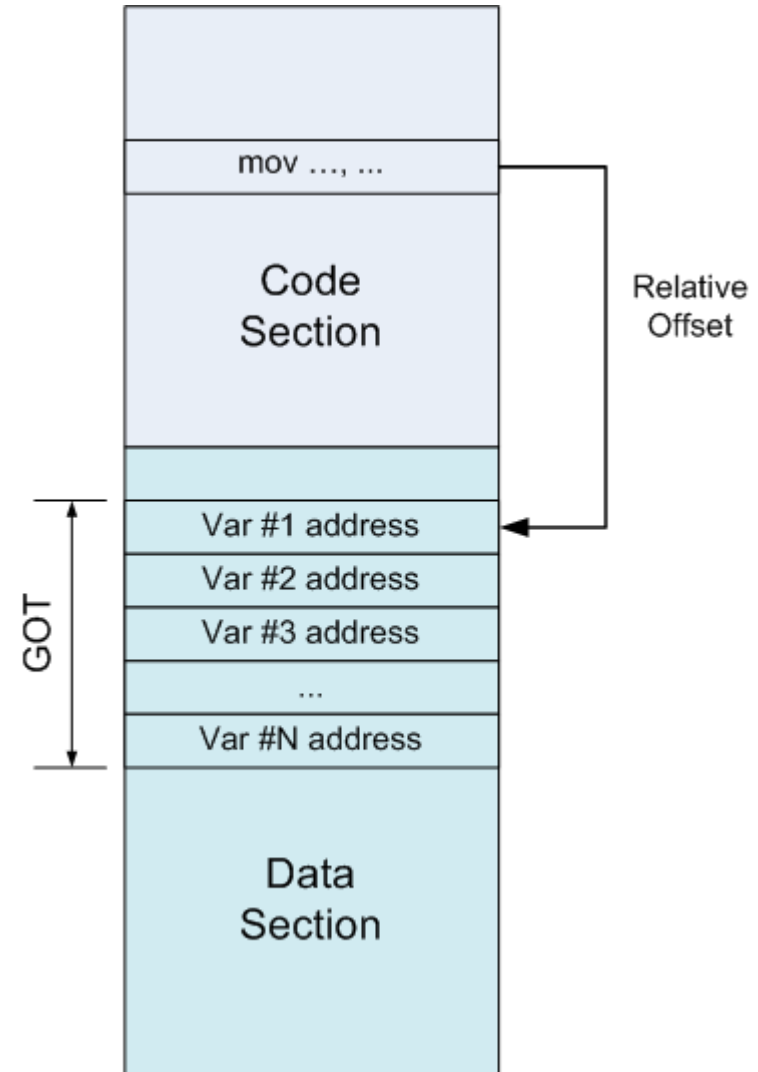
- Main insight
 - Code sections are followed by data sections
 - The distance between code and data **remains constant even if code is relocated**
 - Linker knows the distance
 - Even if it combines multiple code sections together

Insight 1: Constant offset between text and data sections



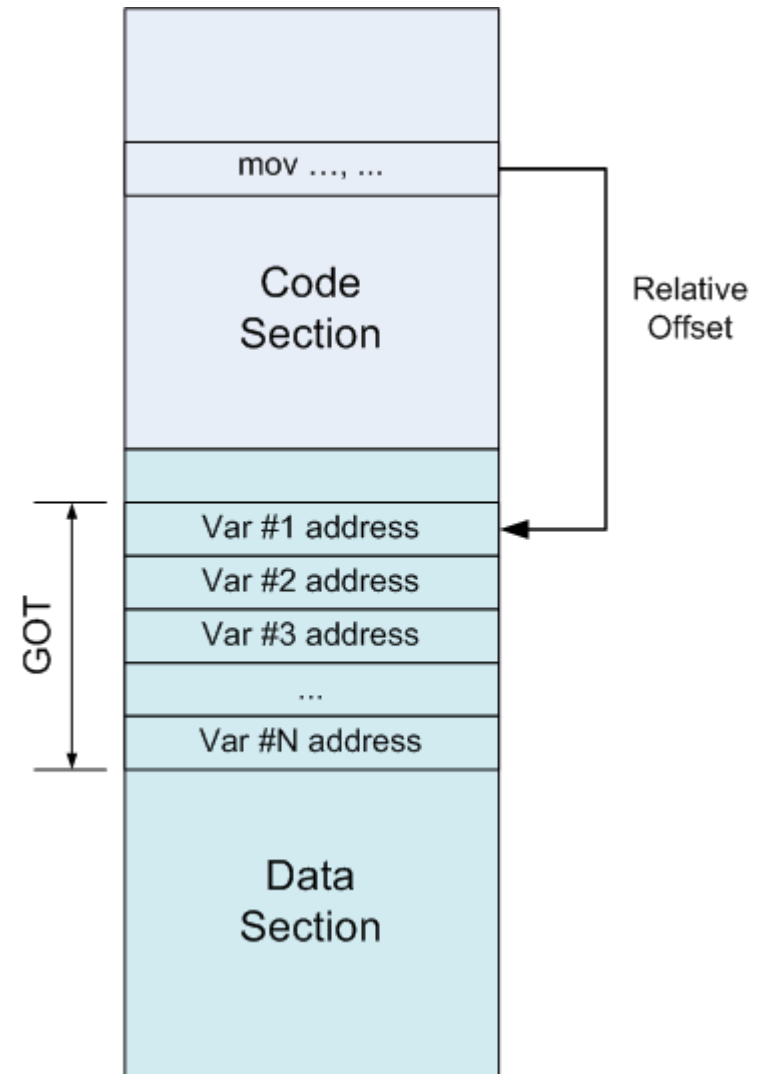
Global offset table (GOT)

- Insight #2:
 - Instead of referring to a variable by its absolute address
 - 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

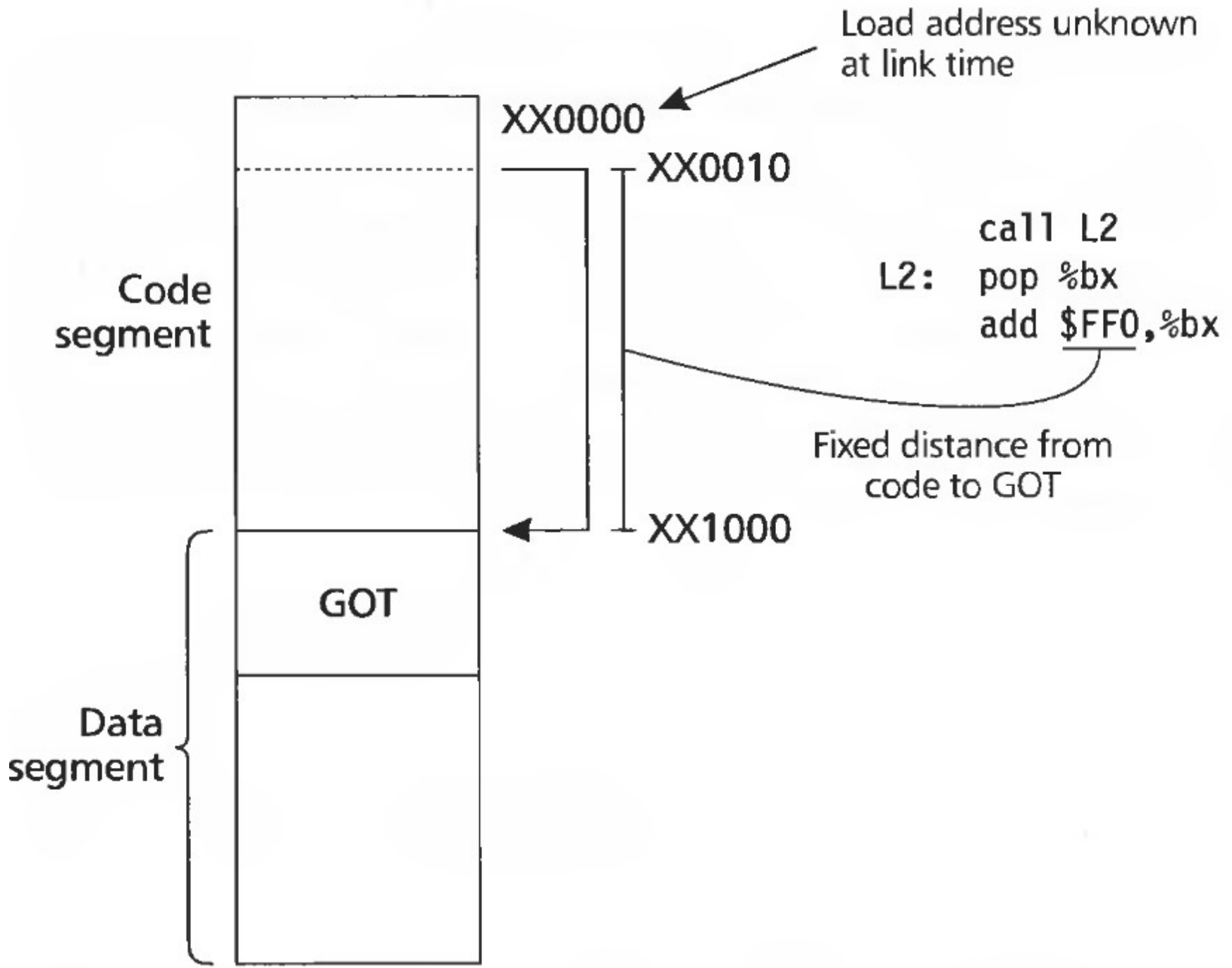
How to find position of the code in memory at run time?

- 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



PIC: Advantages and disadvantages

- Any ideas?

PIC: Advantages and disadvantages

- Bad
 - Code gets slower
 - One register is wasted to keep GOT pointer
 - x86 has 6 registers, 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

Back to shared libraries

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

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
- But really how can we build this?

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
- But really how can we build this?
 - Can we use GOT?

Lazy procedure binding

- GOT will work, but
 - Binding is not trivial
 - Lookup the symbol
 - ELF uses hash tables to optimize symbol lookup
- In large libraries many routines are never called
 - Libc has over 600
 - 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)

- 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

PLT

Code:

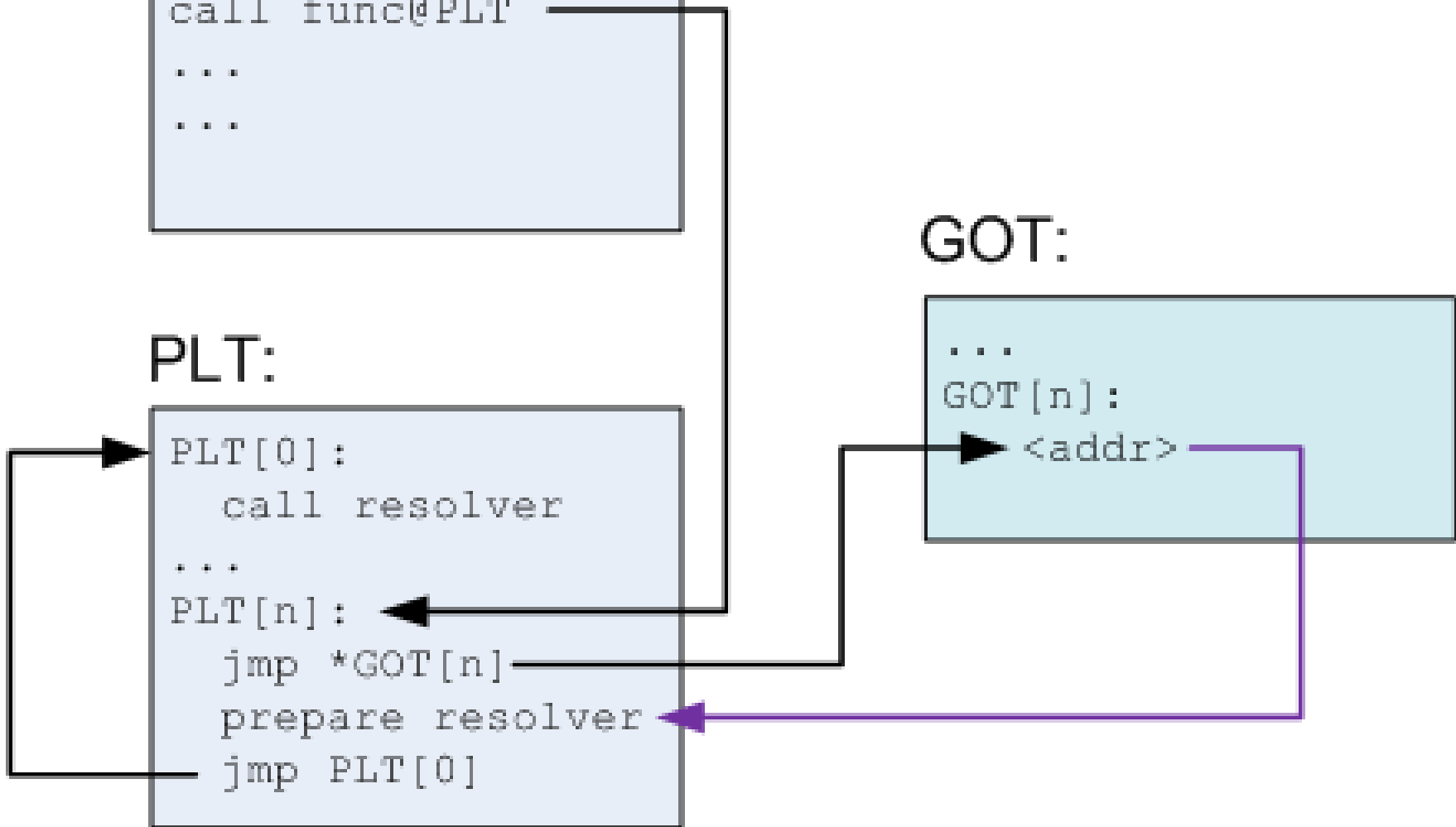
```
call func@PLT
...
...
```

PLT:

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

GOT:

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



Before function is resolved

- Nth GOT entry points to after the jump

Code:

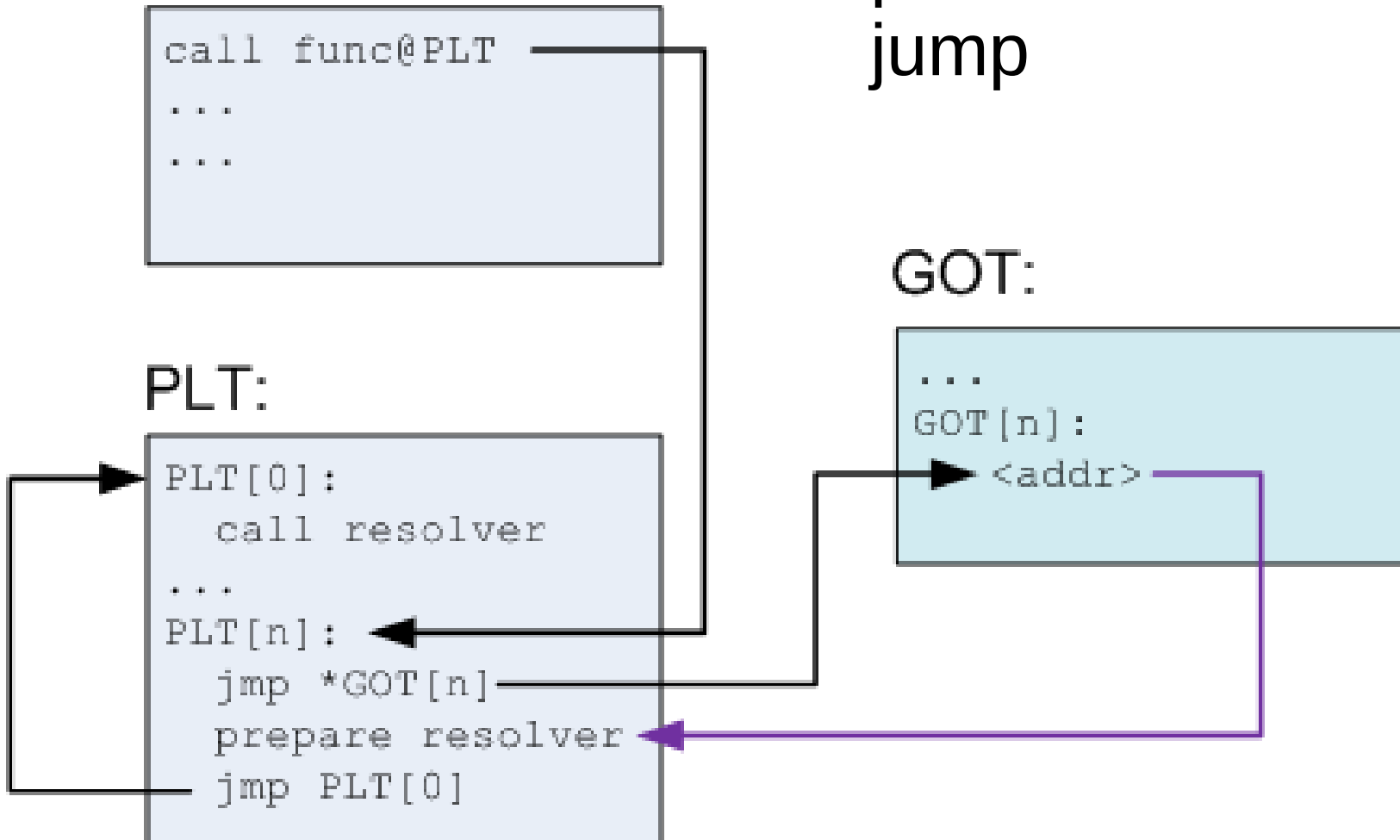
```
call func@PLT
...
...
```

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:

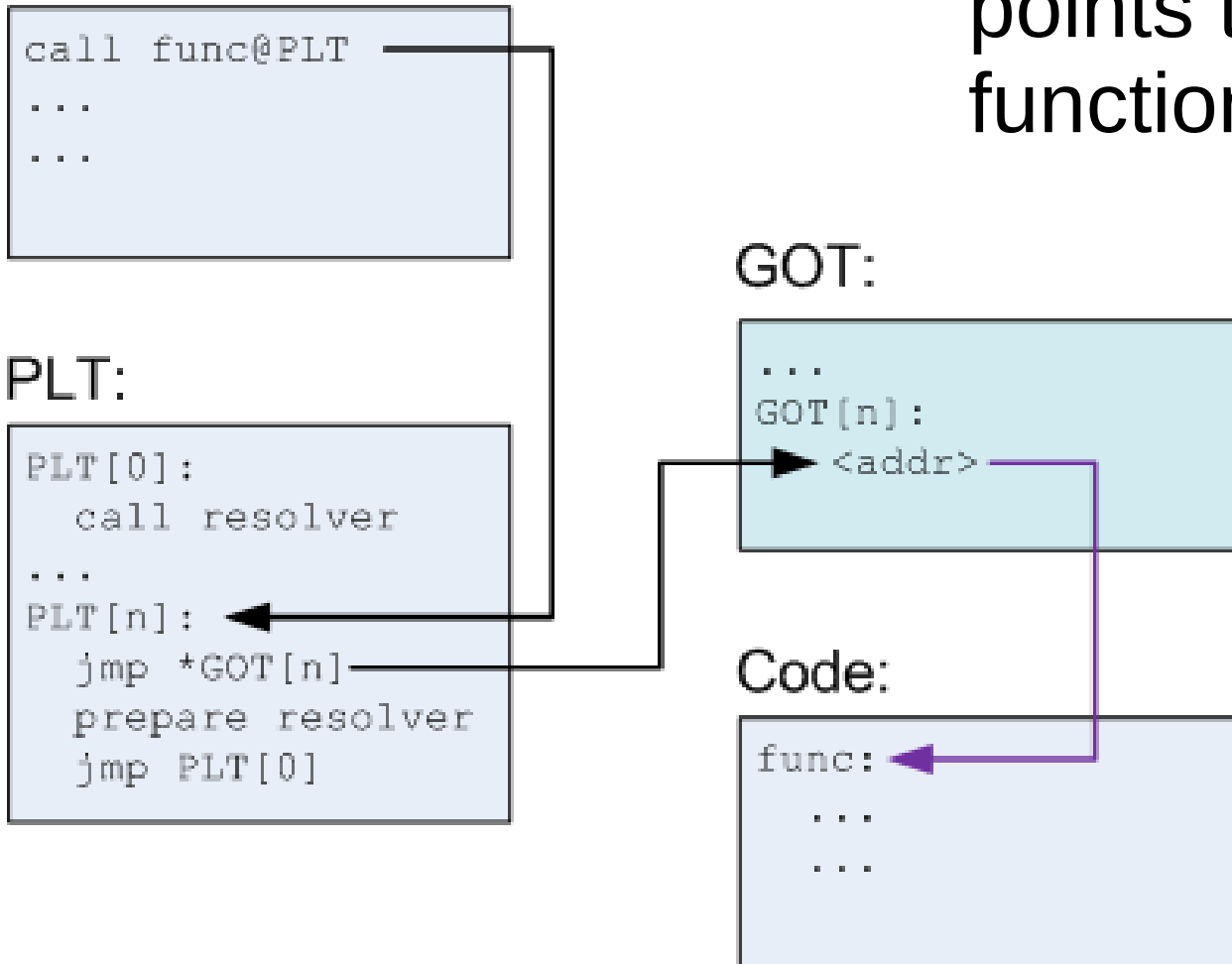
```
PLT[0]:
  call resolver
...
PLT[n]: ←
  jmp *GOT[n]
  prepare resolver
  jmp PLT[0]
```

GOT:

```
...
GOT[n]:
  → <addr>
```

Code:

```
func: ←
...
...
```



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!

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`)
 -

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

Next time

- Static and shared libraries
- Dynamic linking and loading
- Position independent code
- OS management of user space

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

Simplest object file: DOS .com

- Only binary code
 - Loaded at 0x100 offset
 - 0x00 – 0xFF is reserved for program prefix
 - Command line arguments
- Set EIP to 0x100
- Set ESP to the top of the segment
- Run!