CS5460/6460: Operating Systems

Lecture 21: Shared libraries

Anton Burtsev
March, 2014

Recap from last time

- We know what linkers and loaders do

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

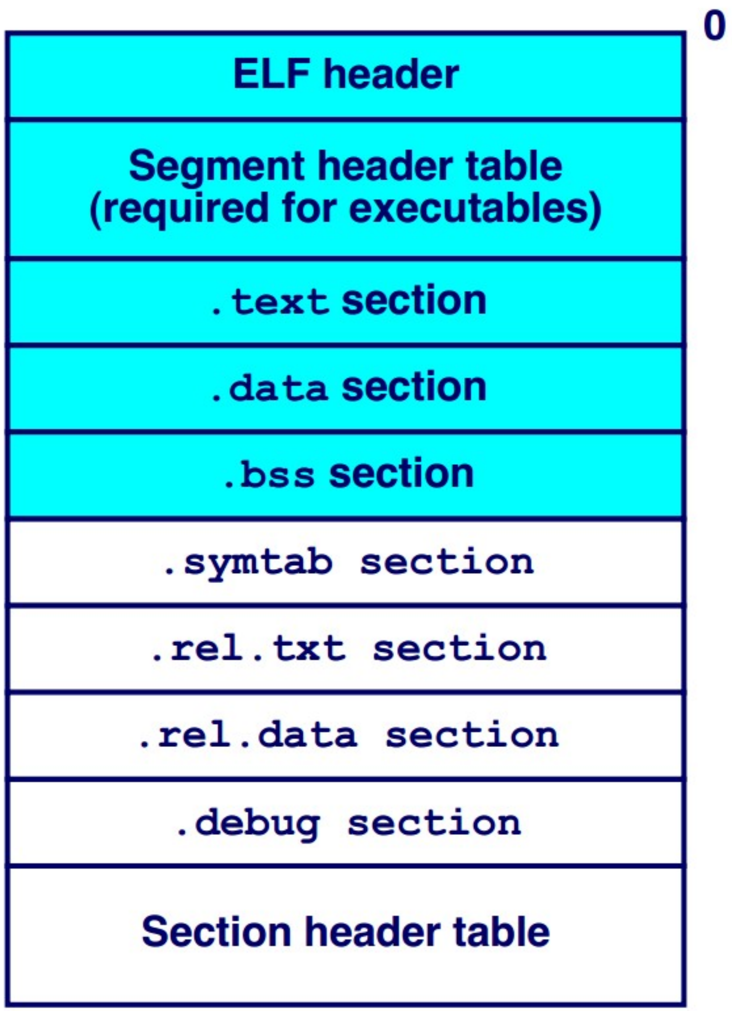
- Code

`.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**

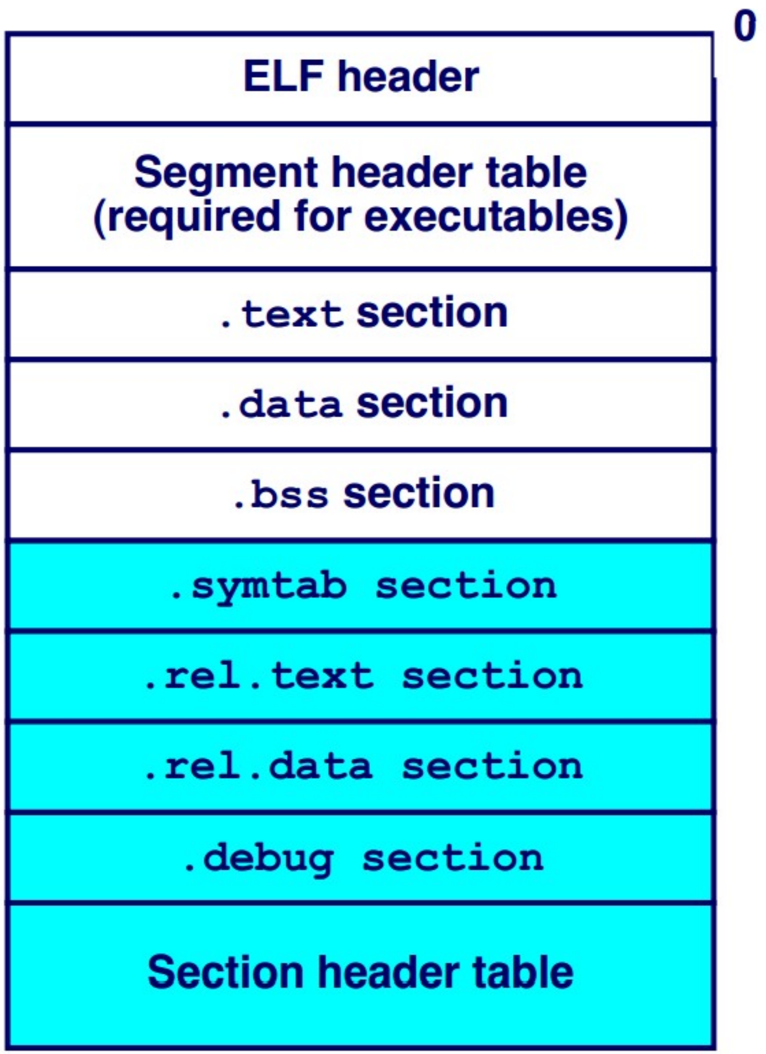
- Relocation info for `.data` section
- Addresses of pointer data that will need to be modified in the merged executable

`.debug` **section**

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

Section header table

- Offsets and sizes of each section

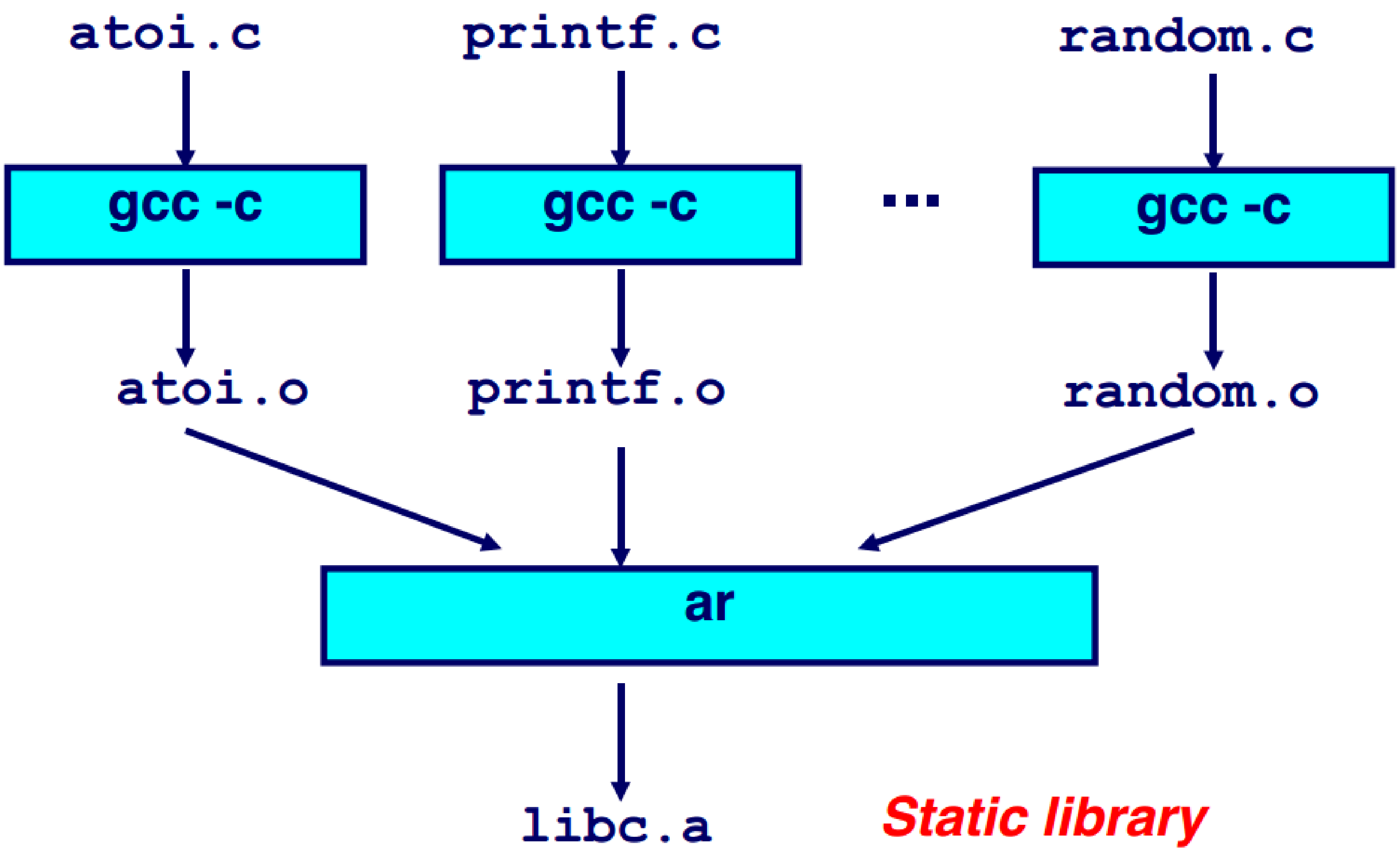


Static libraries

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

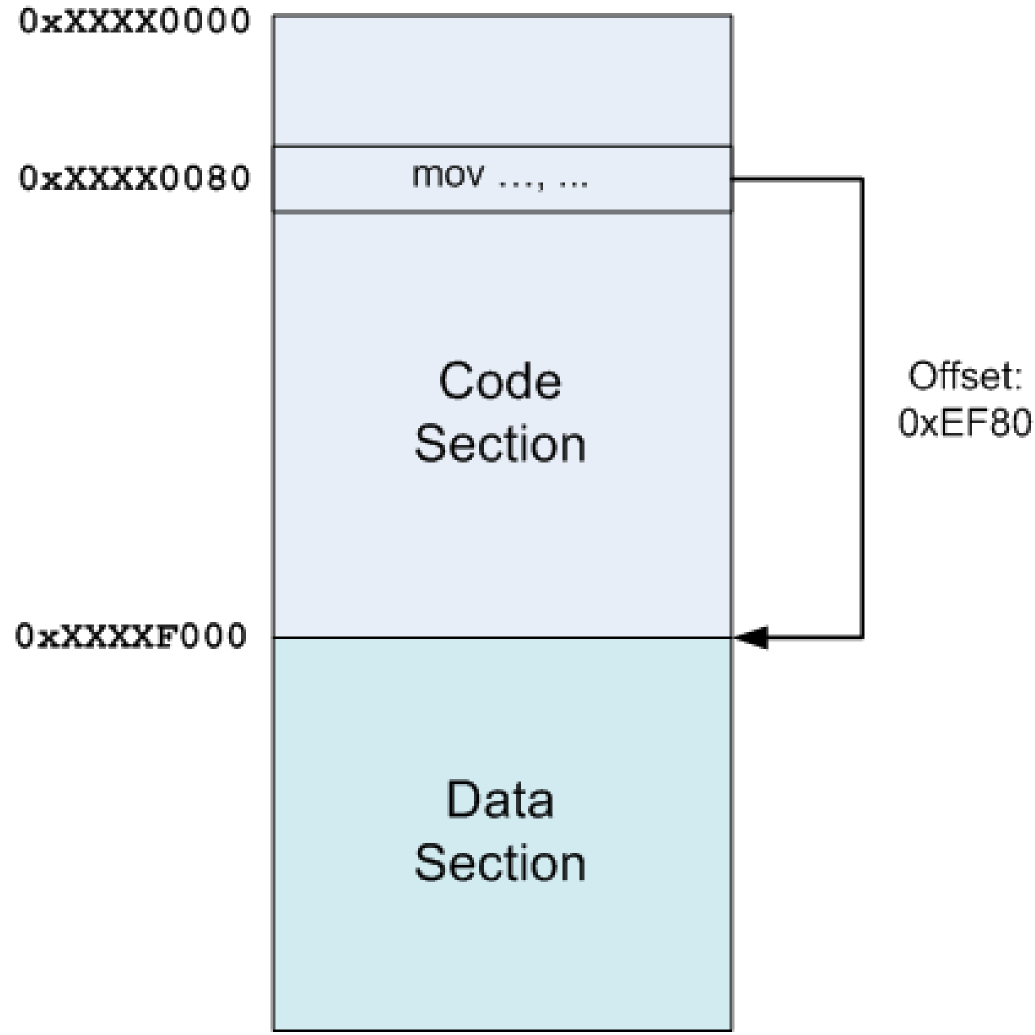
Position independent code

- How would you build it?

Position independent code

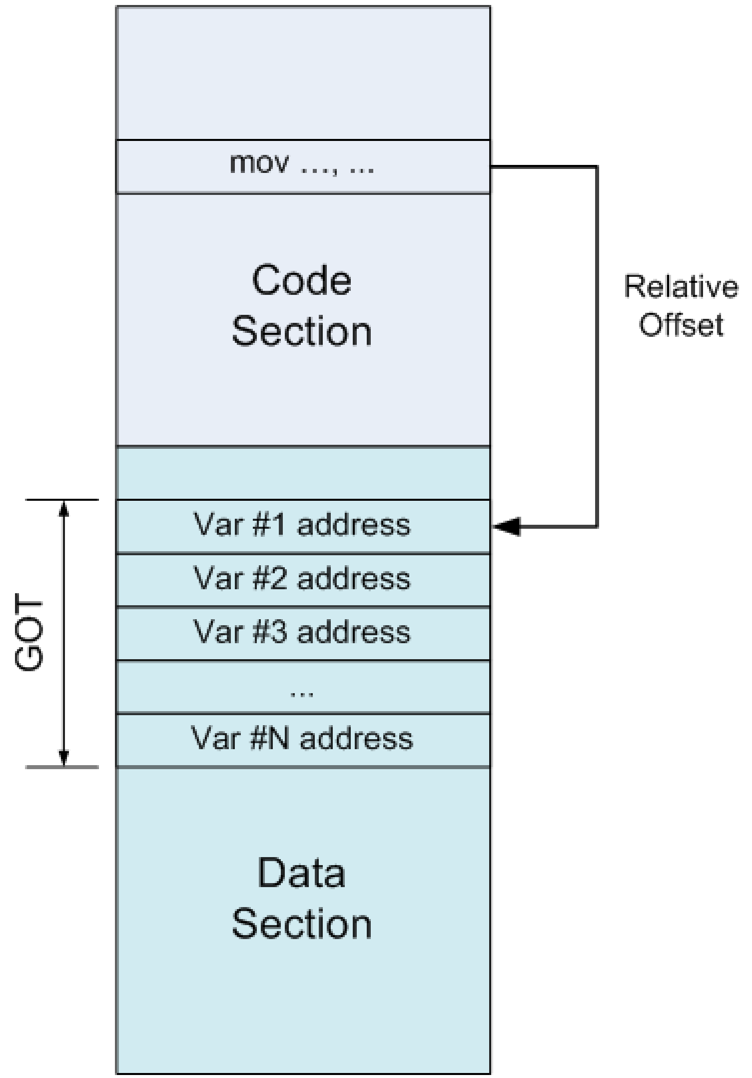
- Main insight
  - Code sections are followed by data sections
  - The distance between code and data **remains constant even if code is relocated**
    - Linker knows the distance
    - Even if it combines multiple code sections together

Insight 1: Constant offset between text and data sections



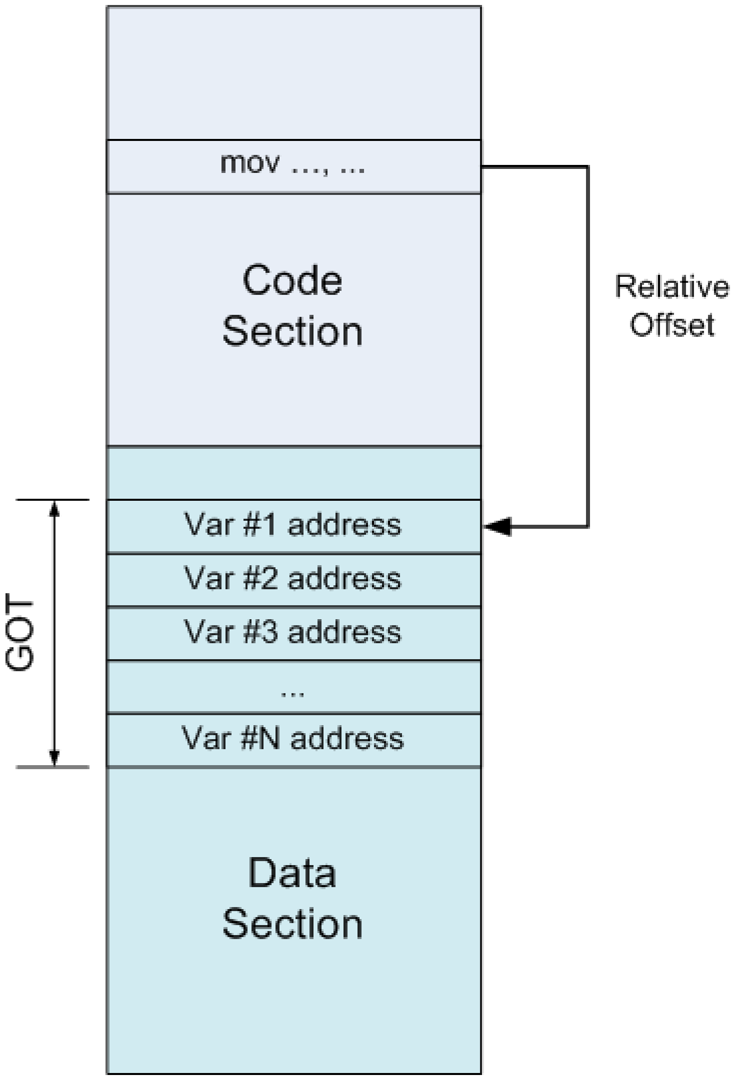
Global offset table (GOT)

- 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?

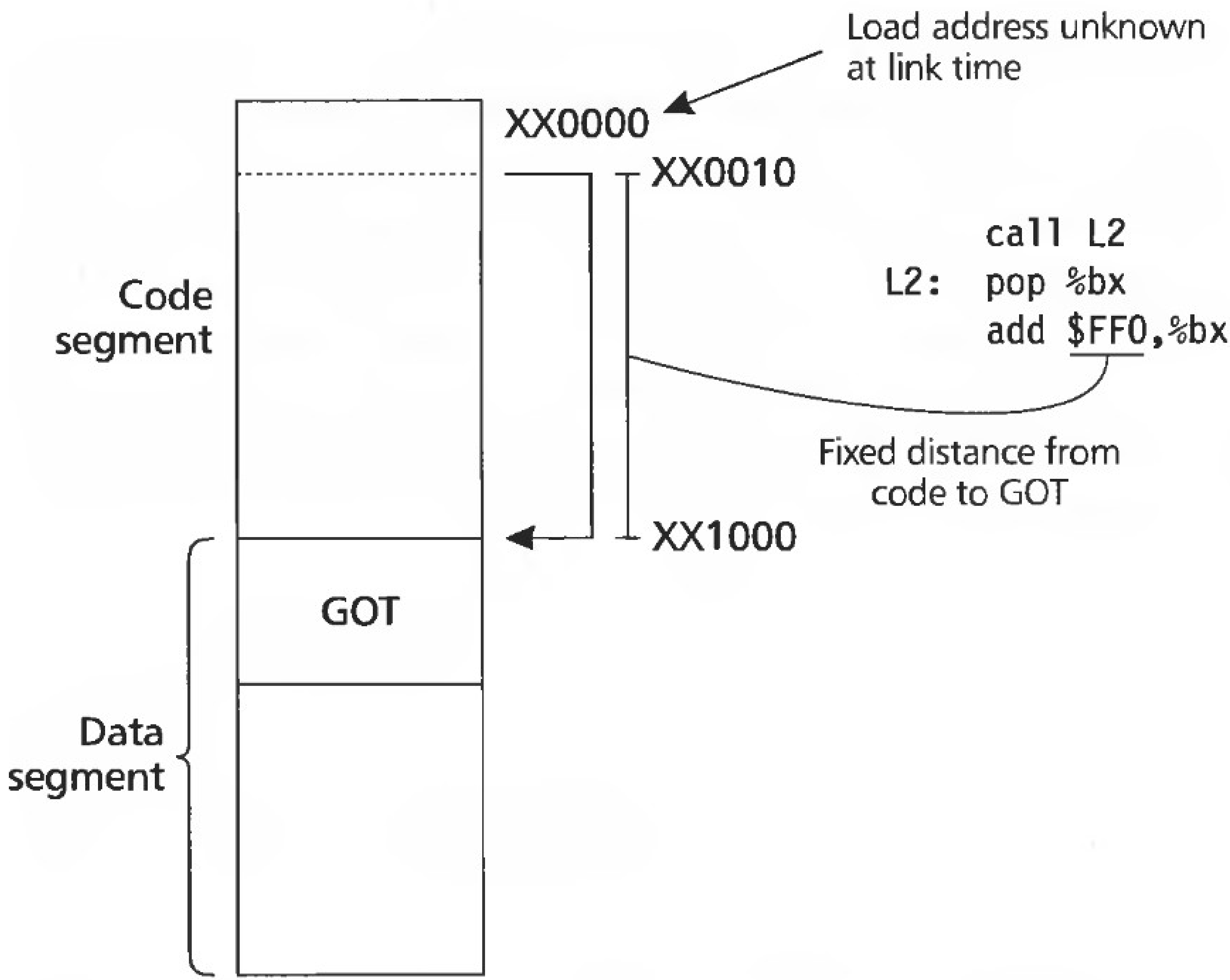
- Is there an x86 instruction that does this?
  - i.e., give me my current code address

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


Load address unknown
at link time
XX0000
XX0010
call L2
L2: pop %bx
add $FF0,%bx
Code
segment
Fixed distance from
code to GOT
XX1000
GOT
Data
segment

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, loosing 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) intializes 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

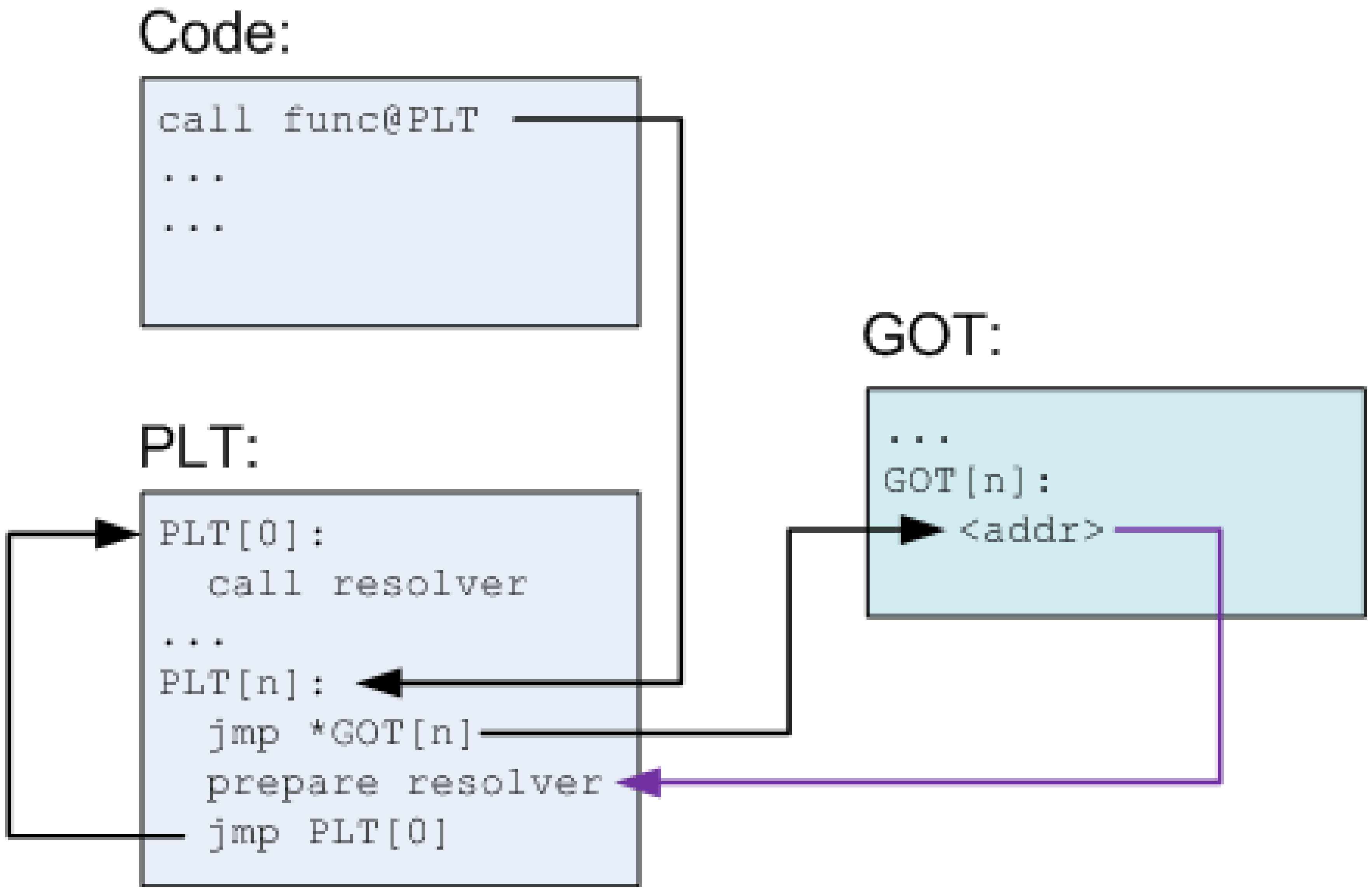
- GOT will work, but
  - Binding is not trivial
    - Lookup the symbol
    - ELF uses hash tables to optimize symbol lookup
- In large libraries many routines are never called
  - Libc has over 600
  - It's ok to bind all routines when the program is statically linked
    - Binding is done offline, no runtime costst
  - 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

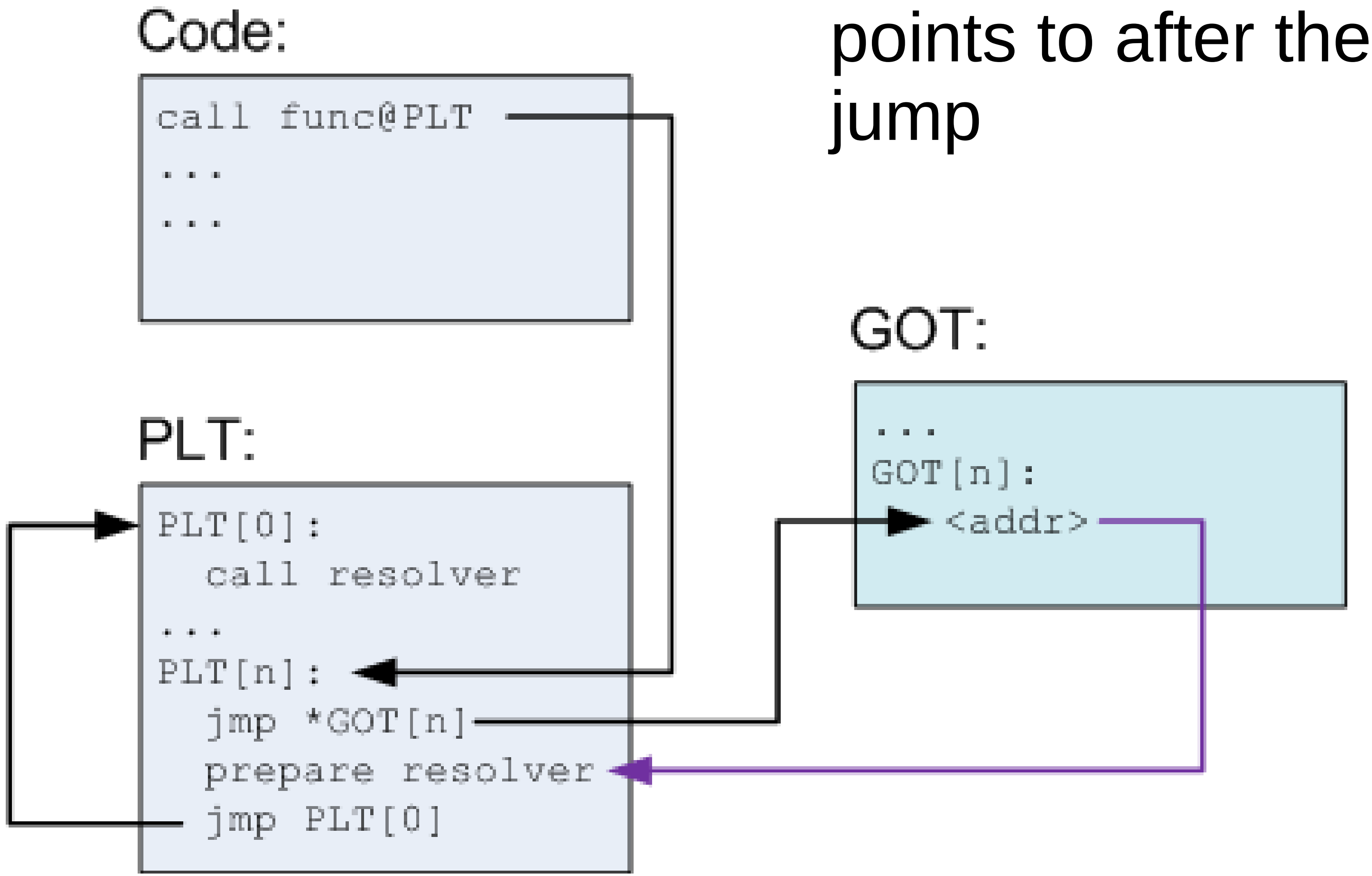
- Each PLT entry but the first consists of these parts:
  - A jump to a location which is specified in a corresponding GOT entry
  - Preparation of arguments for a "resolver" routine
  - Call to the resolver routine, which resides in the first entry of the PLT

PLT


Code:
call func@PLT
...
...
PLT:
PLT[0]:
call resolver
...
PLT[n]:
jmp *GOT[n]
prepare resolver
jmp PLT[0]
GOT:
...
GOT[n]:
<addr>

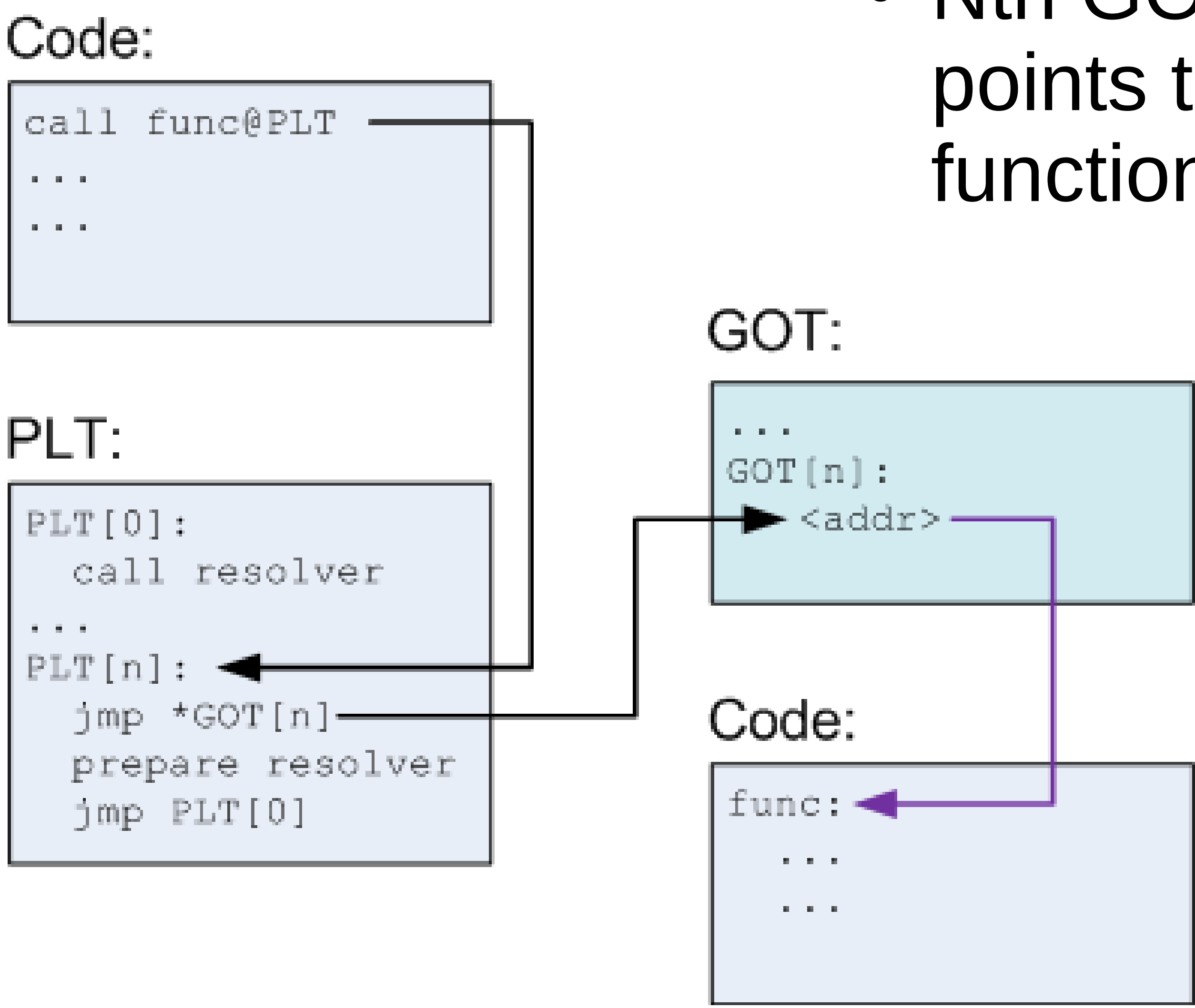
Before function is resolved

- Nth GOT entry points to after the jump



PLT after the function is resolved

- Nth GOT entry points to the actual function



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!