CSc 553
Principles of Compilation

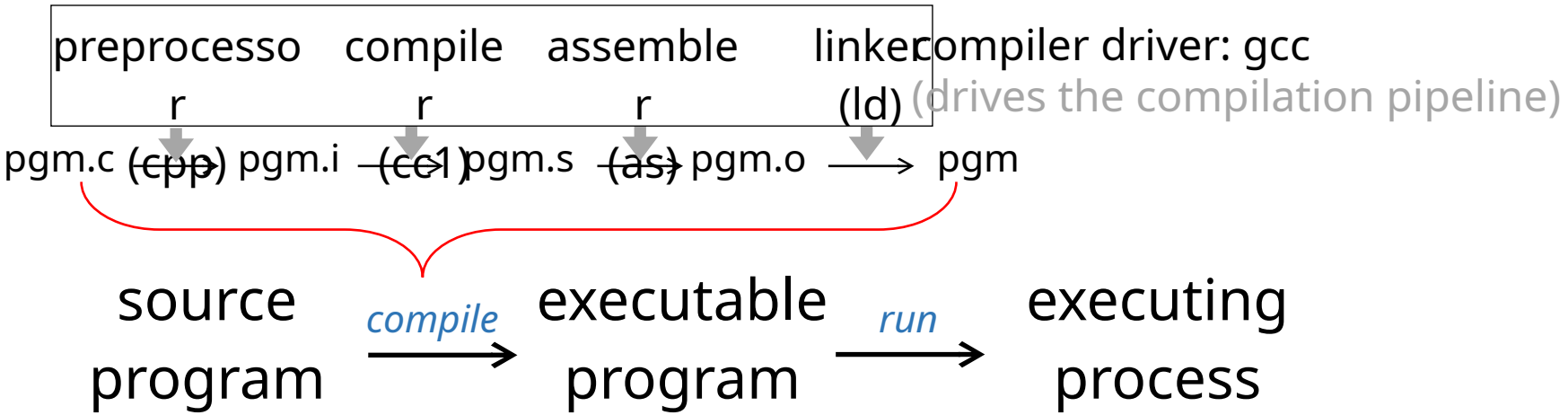
12. From source code to execution

Saumya Debray

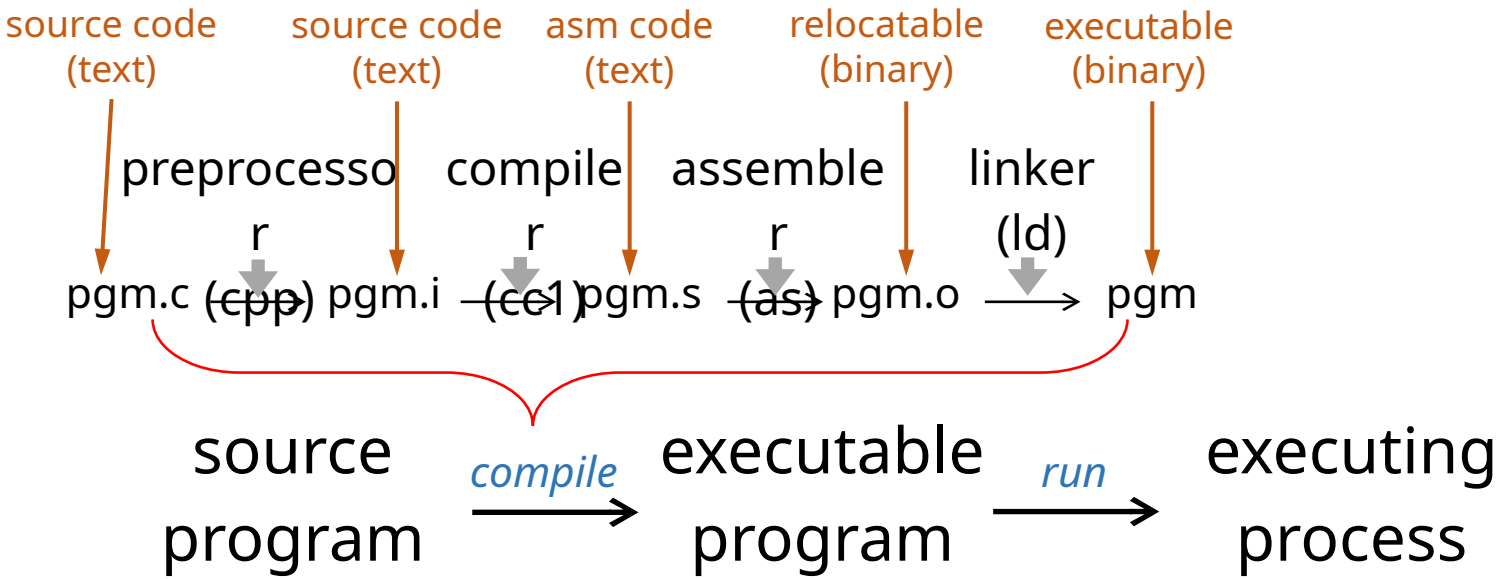
The University of Arizona

Tucson, AZ 85721

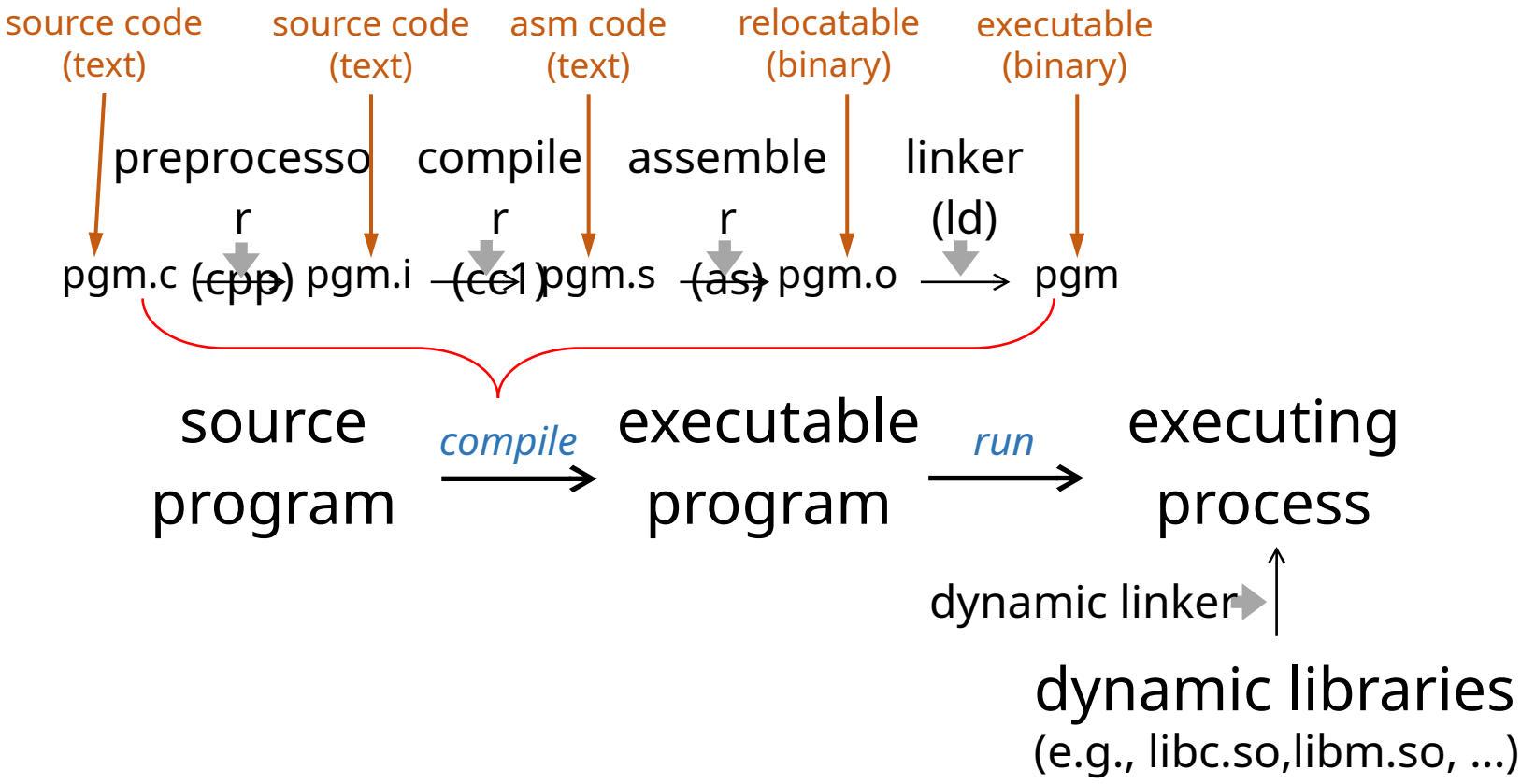
From source code to execution



From source code to execution



From source code to execution



Object file structure

Object files

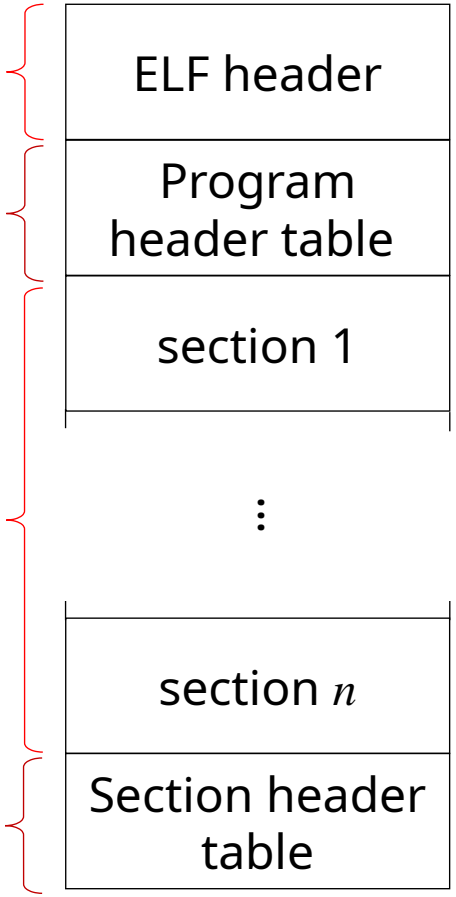
- Binary files containing code and data
- Three kinds:
  - relocatable objects (*.o)
    - output of compiler/assembler
    - intended to be combined with other relocatables
  - shared objects (*.so)
    - dynamically linked libraries
  - executable files (a.out, *.exe)
    - code that can be executed as a process
- They have generally similar structure

Structure of ELF files

- On Linux systems, object files use a file format called "Executable and Linkable Format" (ELF)
- File structure:
  - top-level header that gives general info about the file ("ELF header")
  - two tables that describe the components of the file
  - a number of sections that contain code, data, etc.

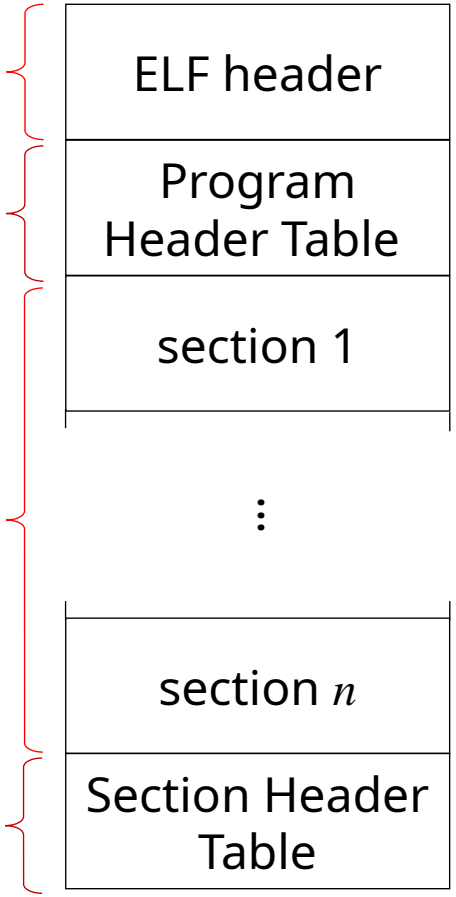
Structure of ELF files

- general info about the file
- a table describing each of section$_1$ … section (execution-time view)
- the actual contents of the file (code, data, etc.)
- no. of sections, their names + sizes, not fixed
- a table describing each of section$_1$ … section$_n$ (pre-execution view)



Structure of ELF files

- general info about the file
- a table describing each of section$_1$ … section (execution-time view)
- code, data, strings, etc.
- no. of sections, their names + sizes, not fixed
- a table describing each of section$_1$ … section$_n$ (pre-execution view)



- type of file
- machine info
- execution start address (executable files)
- Program Header Table info:
  - offset, size, no. of entries
- Section Header Table info:
  - offset, size, no. of entries

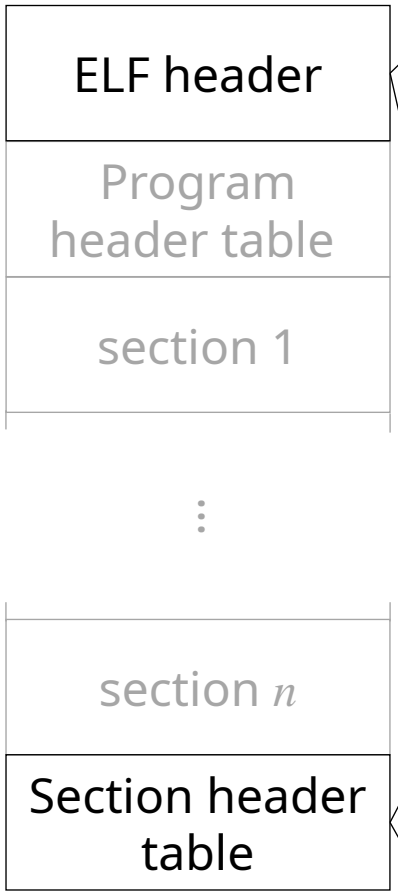
for each section:

- section name, size, offset in the file
- type of contents
- assorted other info

EXERCISE

- Log onto a linux machine (e.g., lectura) and find an object file *f* (e.g., your compiler)
- The command **objdump** will print out information about the file:
  - **objdump –f** : file header info
  - **objdump –h** : section header info
- Use this to get information about the file *f*:
  - how many sections does *f* contain?
  - how many of these sections contain code?
  - what else can you find out about *f* ?

Structure of ELF files




```
typedef struct {
    unsigned char   e_ident[16];   /* ELF identification */
    Elf64_Half   e_type;           /* Object file type */
    Elf64_Half   e_machine;        /* Machine type */
    ...
    Elf64_Addr   e_entry;          /* Entry point address */
    Elf64_Off   e_phoff;           /* Program header offset
*/
    Elf64_Off   e_shoff;           /* Section header offset
*/
    ...
    Elf64_Half   e_shentsize;      /* Size of section header
entry */
    Elf64_Half   e_shnum;          /* No. of section header
entries */
    ...
} Elf64_Ehdr;
```

```
typedef struct {
    uint32_t   sh_name;      /* section name */
    uint32_t   sh_type;      /* section contents and semantics
*/
    ...
    Elf64_Addr   sh_addr;    /* runtime memory addr for this
section */
    Elf64_Off   sh_offset;   /* file offset where the section
starts */
    uint32_t   sh_size;      /* size of the section (bytes) */
    ...
} Elf64_Shdr;
```

Processing ELF files: an example

How **objdump –h** might work:

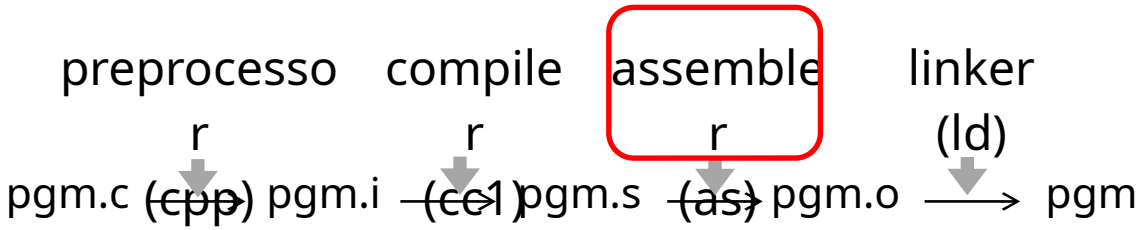
```
ehdr = read ELF header from the ELF file
seek to offset ehdr.e_shoff in the file
for i in range(ehdr.e_shnum):
    s_entry = read ehdr.e_shentsz bytes from file
    extract and print info about the section s_entry
```

offset of section header table

no. of entries in section header table

section header entry size

Assembly


preprocesso
r
compile
r
assemble
r
linker
(ld)
pgm.c
(cpp)
pgm.i
(cc1)
pgm.s
(as)
pgm.o
pgm

Assemblers

- Function: given an assembly code program as input, construct a binary for it (e.g., ELF)
- Data structures:
  - start address for the code
    - use a default address; or
    - provided by the programmer
  - symbol table:
    - maps each global name (including labels) to its address
  - information about each section in the binary

Assemblers [2-pass]: Algorithm

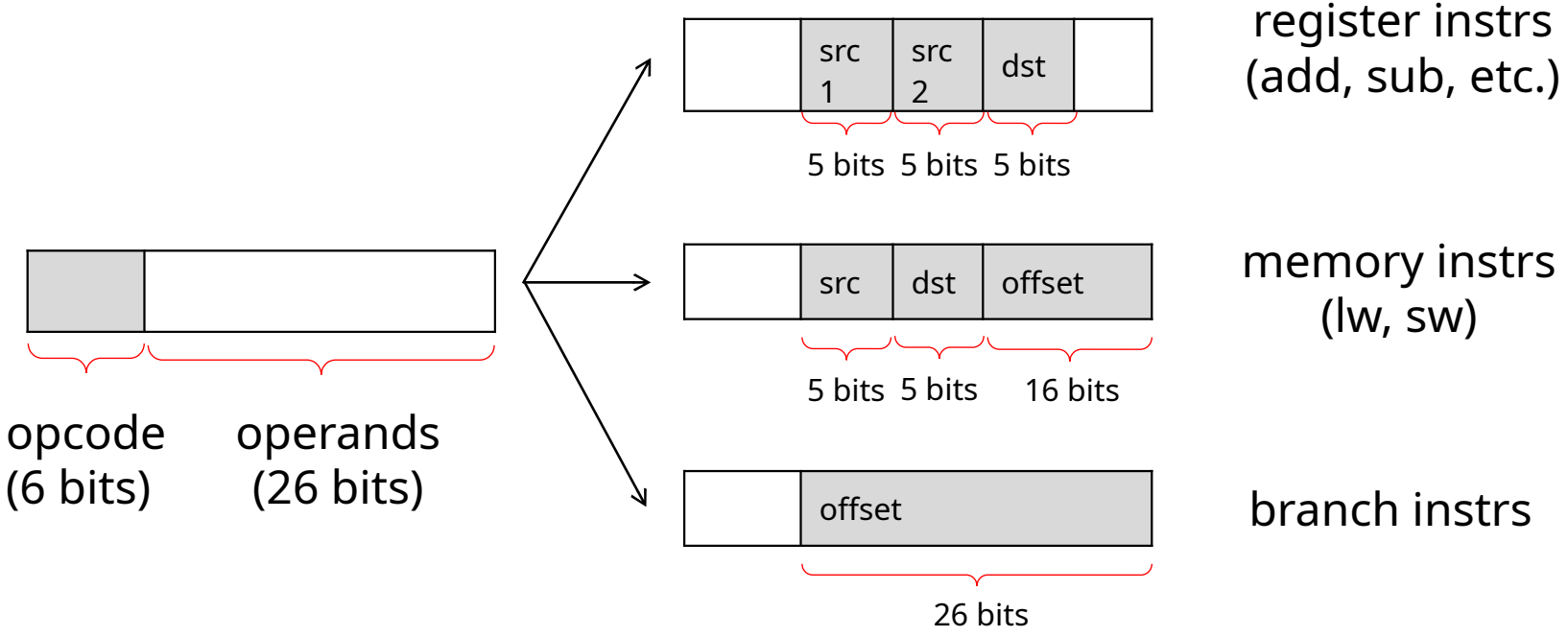
- initialize location counter to start address of code
- Pass 1: *Compute the value for each symbol*
  - process each assembler directive and instruction
  - compute the address of each symbol, save in the symbol table
- Pass 2: *Write out the output (binary) file*
  - write ELF header
  - write out the sections of the output file:
    - process each assembler directive and instruction
    - construct and emit binary representation of instructions
  - write out info about each section into SHT
- patch ELF header with info about SHT

Binary representation of instructions

- An instruction's fields (opcode, source ops, destination ops) usually follow a specific format
  - different kinds of instructions may have different number and types of operands
  - the opcode determines the number and locations of the operands
- For the assembler: constructing the binary representation of an instruction is usually just using table lookups to fill in the fields

Binary representation of instructions

Example: MIPS R2000 (SPIM):



EXERCISE

- Log onto a linux machine (e.g., lectura) and find an object file *f* (e.g., your compiler)
- The command **objdump -d** will disassemble the machine code in the file
  - try this out to see what asm code gcc generates for your program

EXERCISE

Now that you know what **objdump –d** does, how do you think it is implemented?

Linking

Linking

Linking refers to combining multiple binary files so as to allow code and data references across them

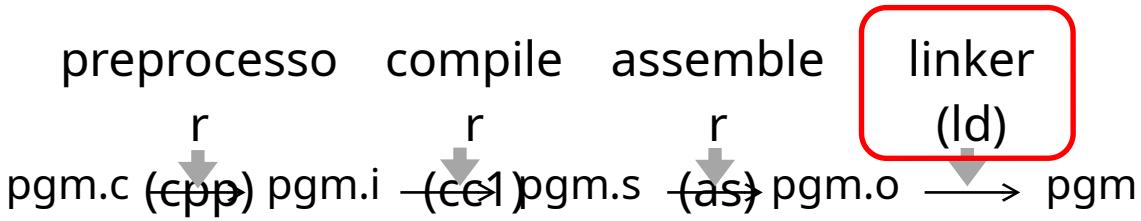
Application code	Application + library code	
"Ordinary" linking	Static linking	Dynamic linking
Combines multiple *.o files for a program into a single executable	Copies all the library code used by a program into the executable file prior to execution	Brings in the library code "as needed" as the program executes

Rarely used
We will not discuss

Issues in linking

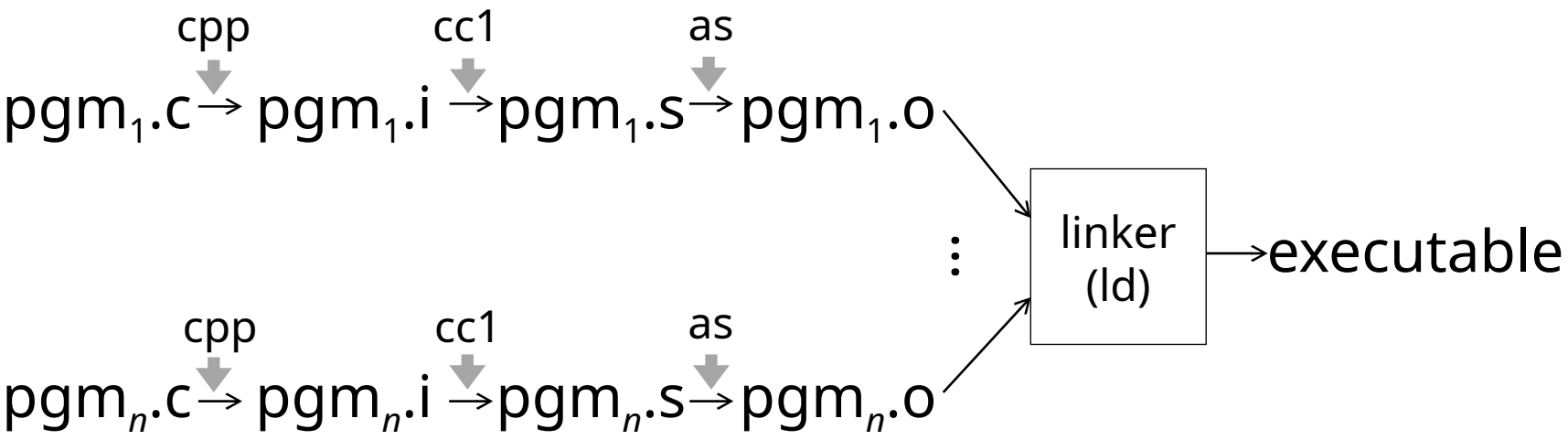
- Address relocation
  - addresses may have to be updated when multiple pieces of code are combined
    - *need a mechanism to identify and update addresses that have to be updated*
- Name resolution
  - the goal of linking is to allow one piece of code *X* to refer to a name *Y* in another piece of code
    - *need a mechanism for X to determine and access the address of Y*

"Ordinary" Linking

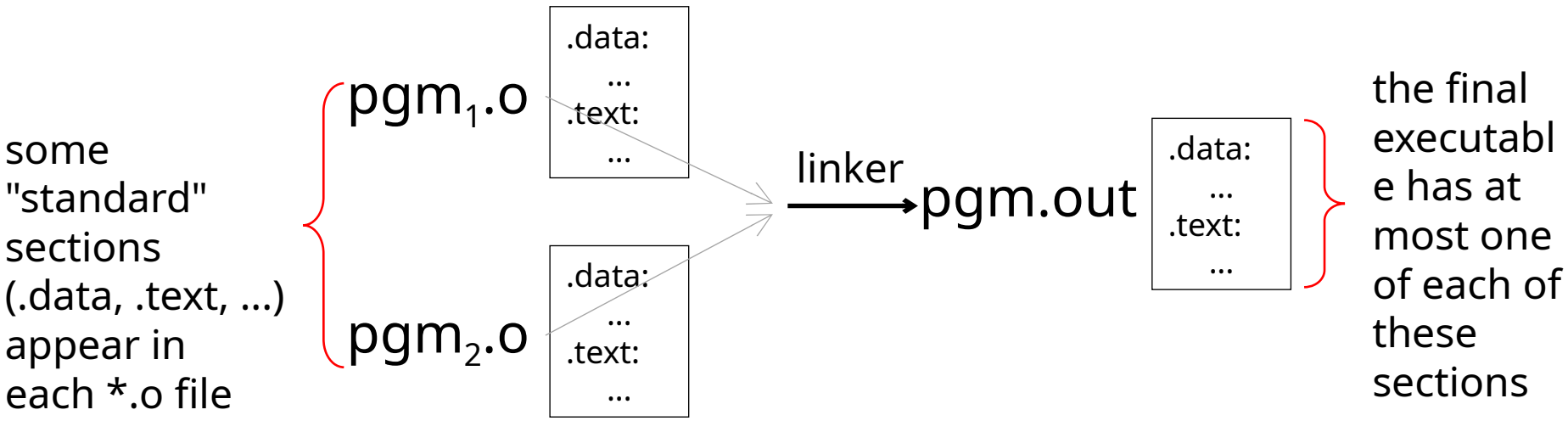


Linker: function

Combine *n* (*n* ≥ 1) relocatable binaries and construct an executable



Linker: issues to address

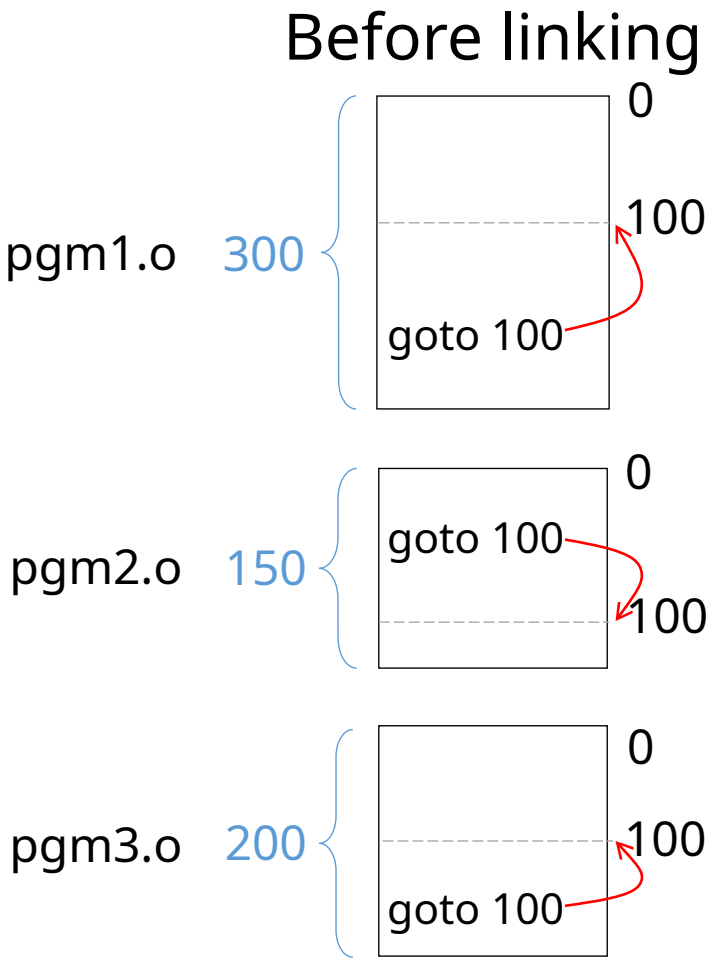


We have to merge corresponding sections from each of the files being linked together
This requires addresses to be fixed up after merging

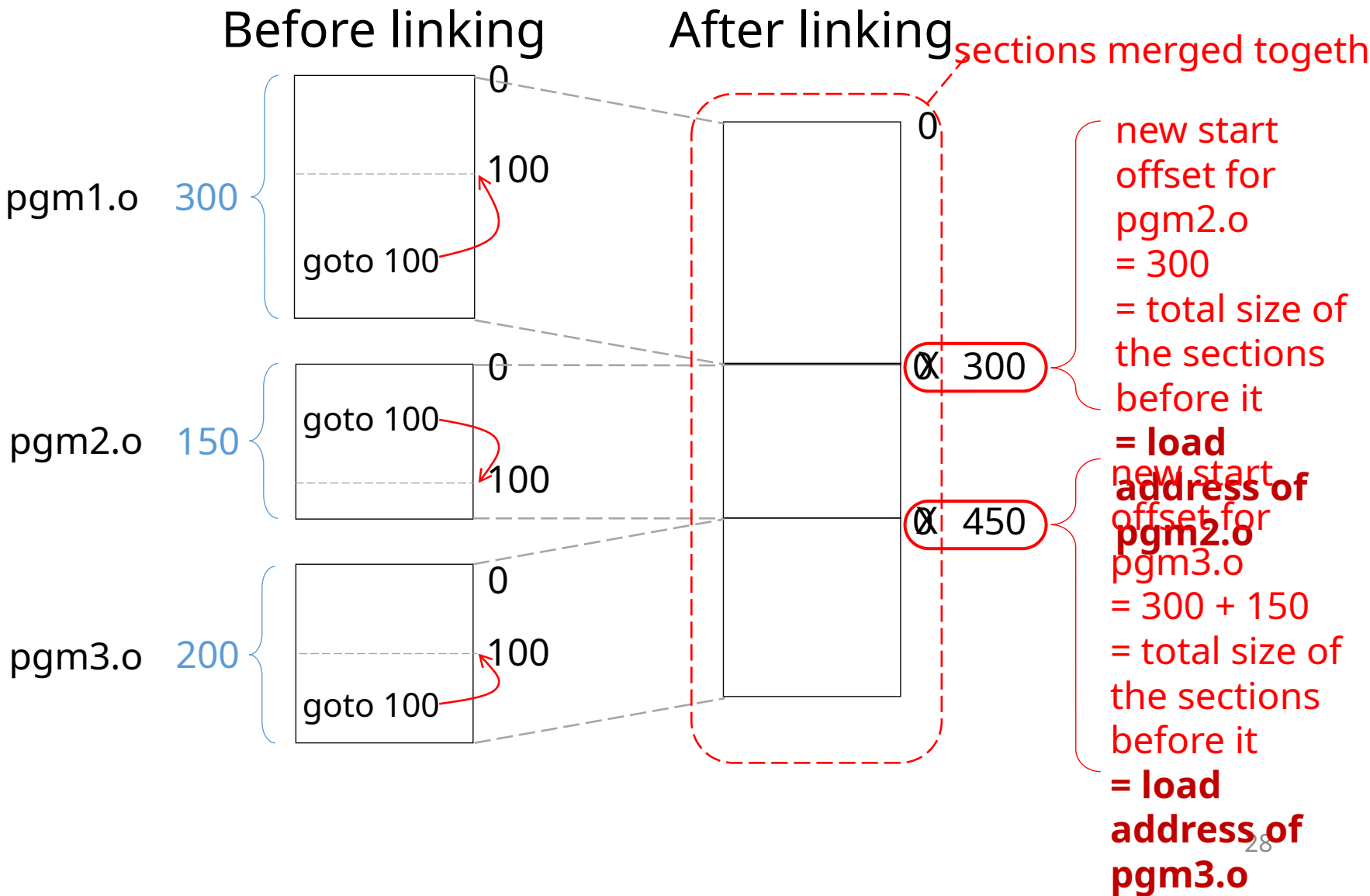
Linker Functions 1: Fixing Addresses

- Addresses in an object file are given relative to the start of the code or data section in the file.
- When different object files are combined:
  - the same kind of sections (text, data, etc.) from the different object files get merged
  - addresses have to be “fixed up” to account for this merging
  - this is done using information embedded in the executable for this purpose (“relocations”).

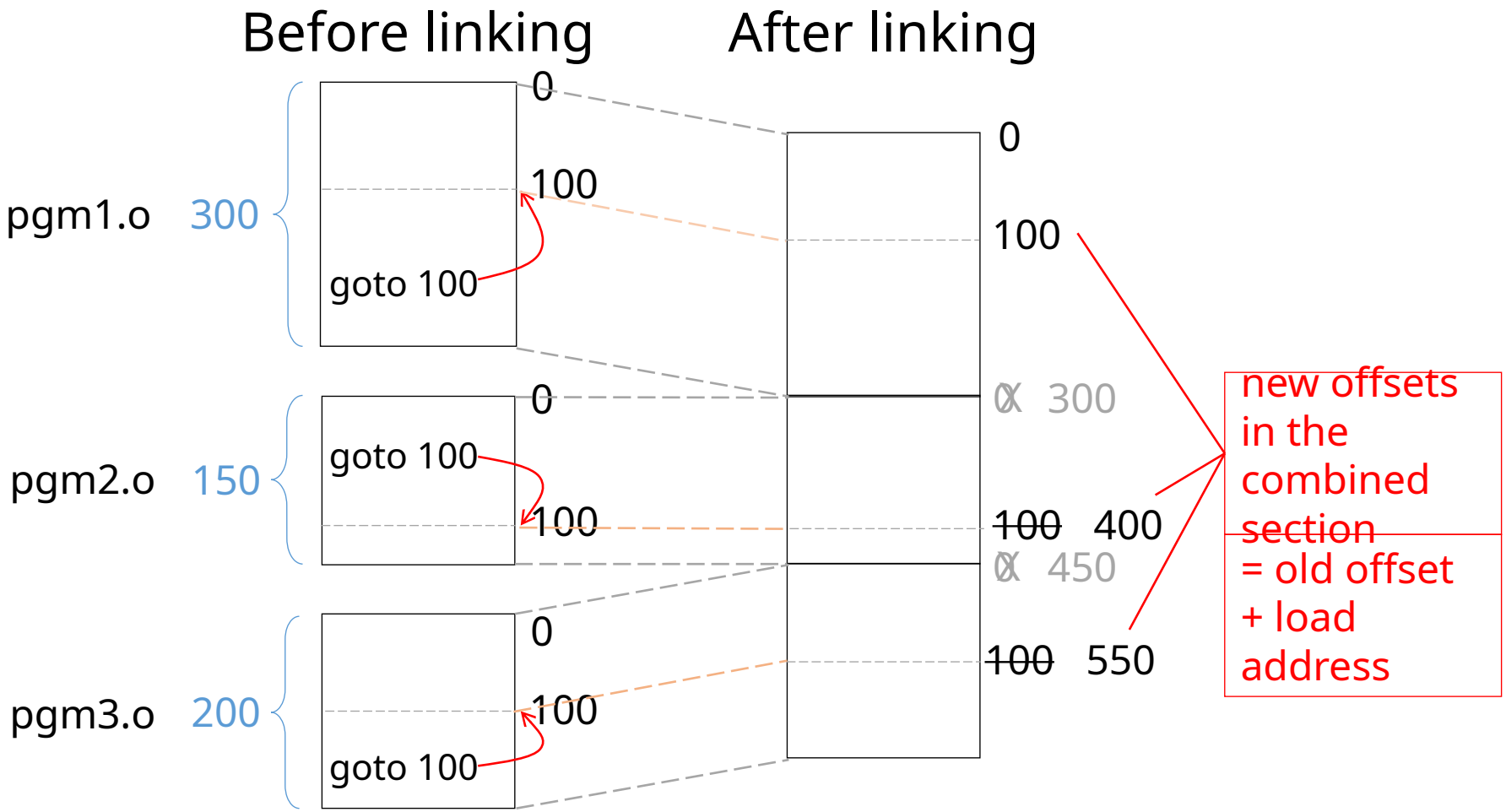
Relocation: Example



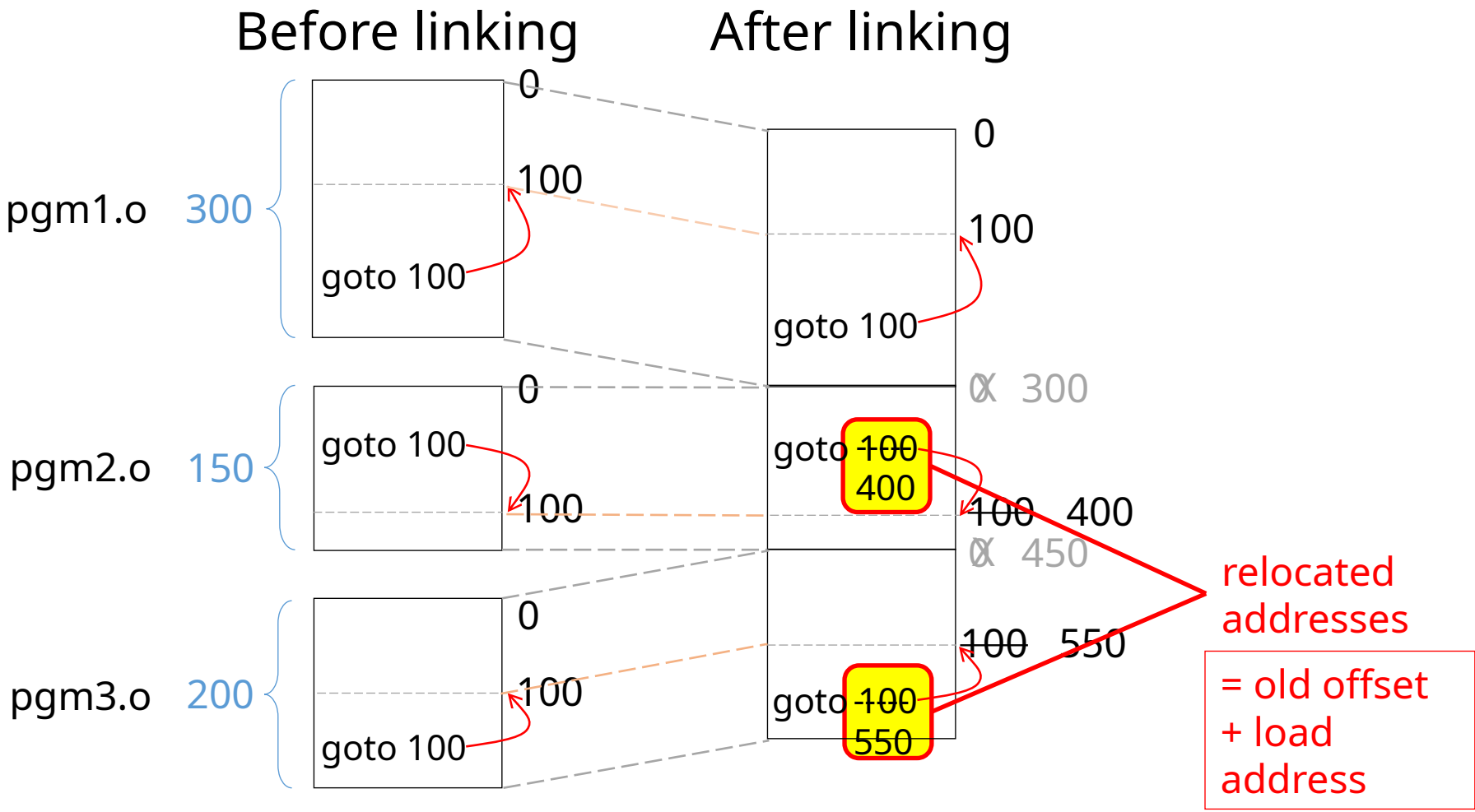
Relocation: Example



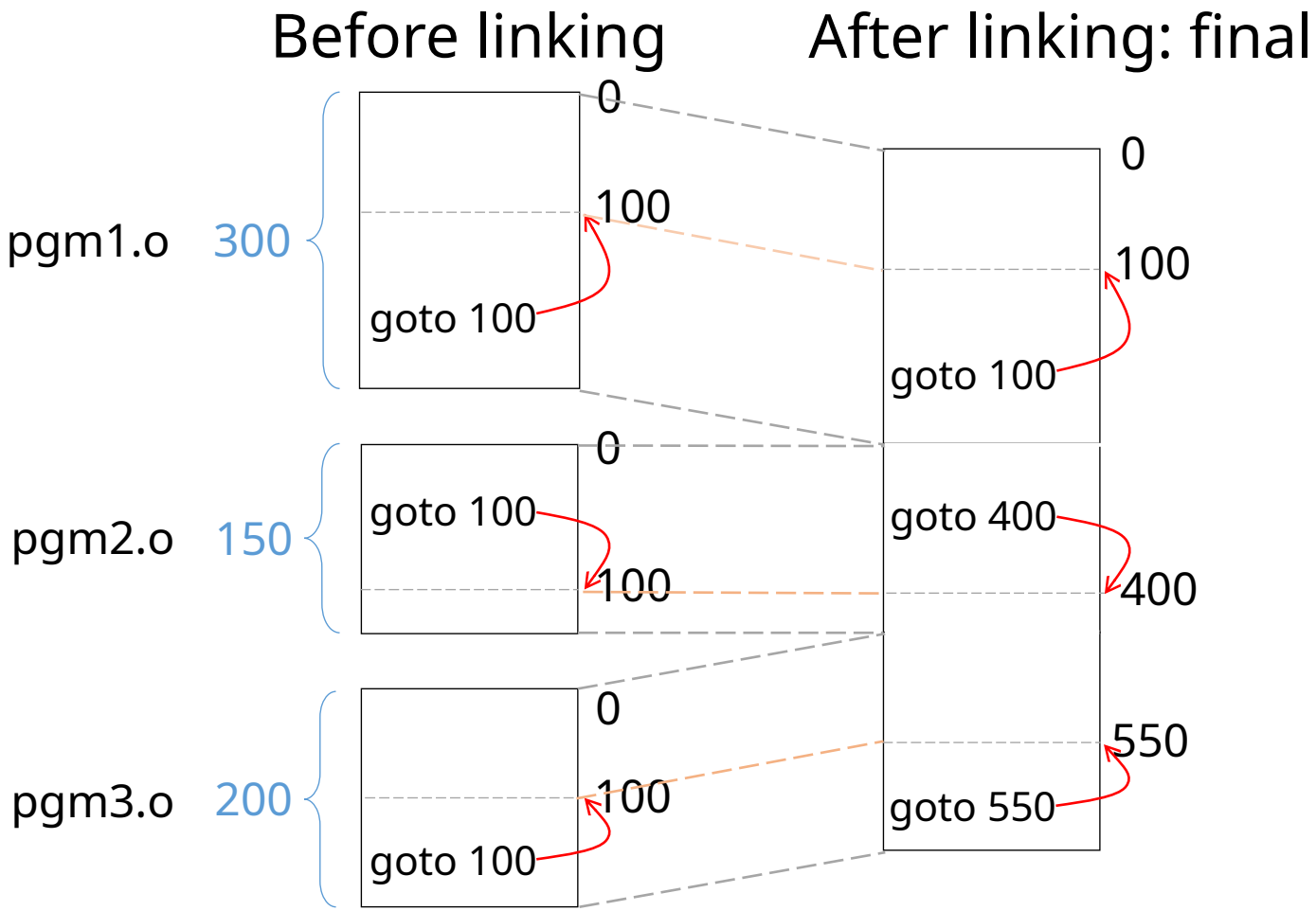
Relocation: Example



Relocation: Example



Relocation: Example



EXERCISE

Suppose we have the following set of relocatable files to link together:

file	text section size
a.o	400
b.o	200
c.o	900
d.o	600

What is the load address of each module if:

1. the order of linking is: a.o, b.o, c.o,d.o
2. the order of linking is: b.o, d.o, a.o, c.o

Identifying addresses to update

- Having the linker try to figure out the update locations is inefficient, can be problematic
- Instead, the compiler puts a list of update locations in a special section in the *.o file
  - each entry specifies a file offset + no. of bytes
  - referred to as a *relocation*
- The linker reads this section and uses it to carry out address relocation
  - the relocation section does not appear in

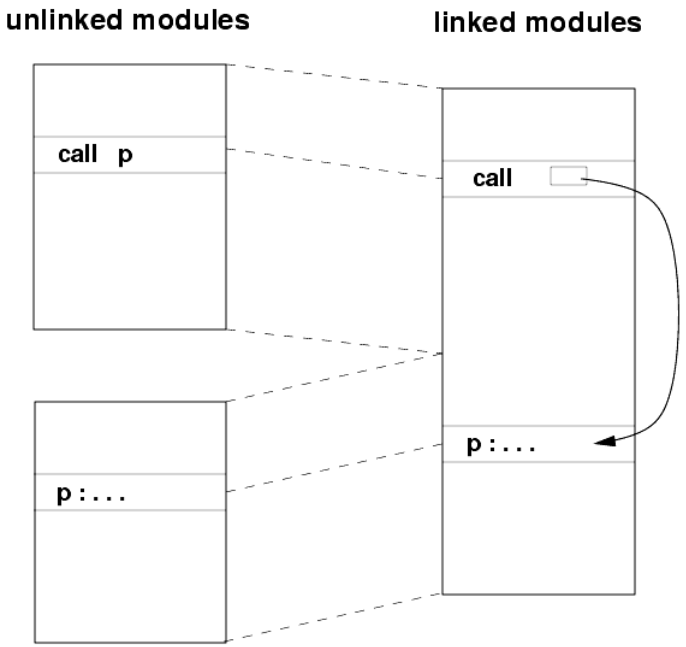
Linker Function 2: Symbol Resolution

Suppose:

- module *B* defines a symbol x;
- module *A* refers to x.

The linker must:

1. determine the location of x in the object module obtained from merging *A* and *B*; and
2. modify references to x (in both *A* and *B*) to refer to this location.



Actions Performed by a Linker

Usually, linkers make two passes:

- *Pass 1*:
  - Collect information about each of the object modules being linked.

- *Pass 2*:
  - Construct the output, carrying out address relocation and symbol resolution using the information collected in Pass 1.

Compile-time vs. link-time errors

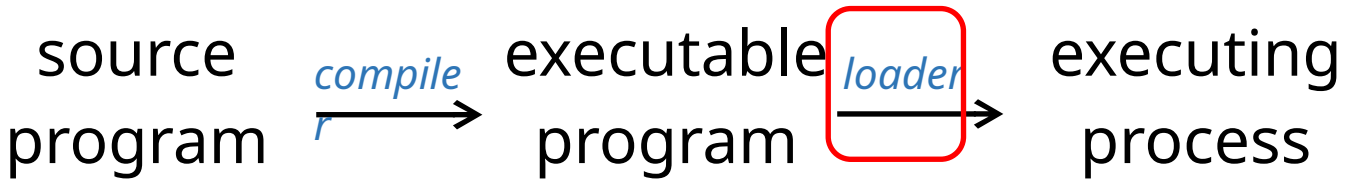
```
#include <stdio.h>
int main() {
    printf("%d\n", x);
    return 0;
}
```

```
gcc pgm.c
pgm.c: In function 'main':
pgm.c: 3:18: error: 'x' undeclared
```

```
#include <stdio.h>
extern int x;
int main() {
    printf("%d\n", x);
    return 0;
}
```

```
gcc pgm.c
/tmp/cc5hNfnk.o: In function 'main':
pgm.c (.text+0x6): undefined reference to x
error: ld returned 1 exit status
```

Loading


source program
compiler
executable program
loader
executing process

Loading

Programs are usually loaded at a fixed address in a fresh address space

Loading involves the following actions:

1. determine how much address space is needed from the object file header; allocate that amount of space
2. read the program into the segments in the address space
3. zero out any uninitialized data (".bss" segment) if not done automatically by the virtual memory system
4. create a stack segment
5. set up any runtime information, e.g., program arguments or environment variables
6. start the program executing

Dynamic linking

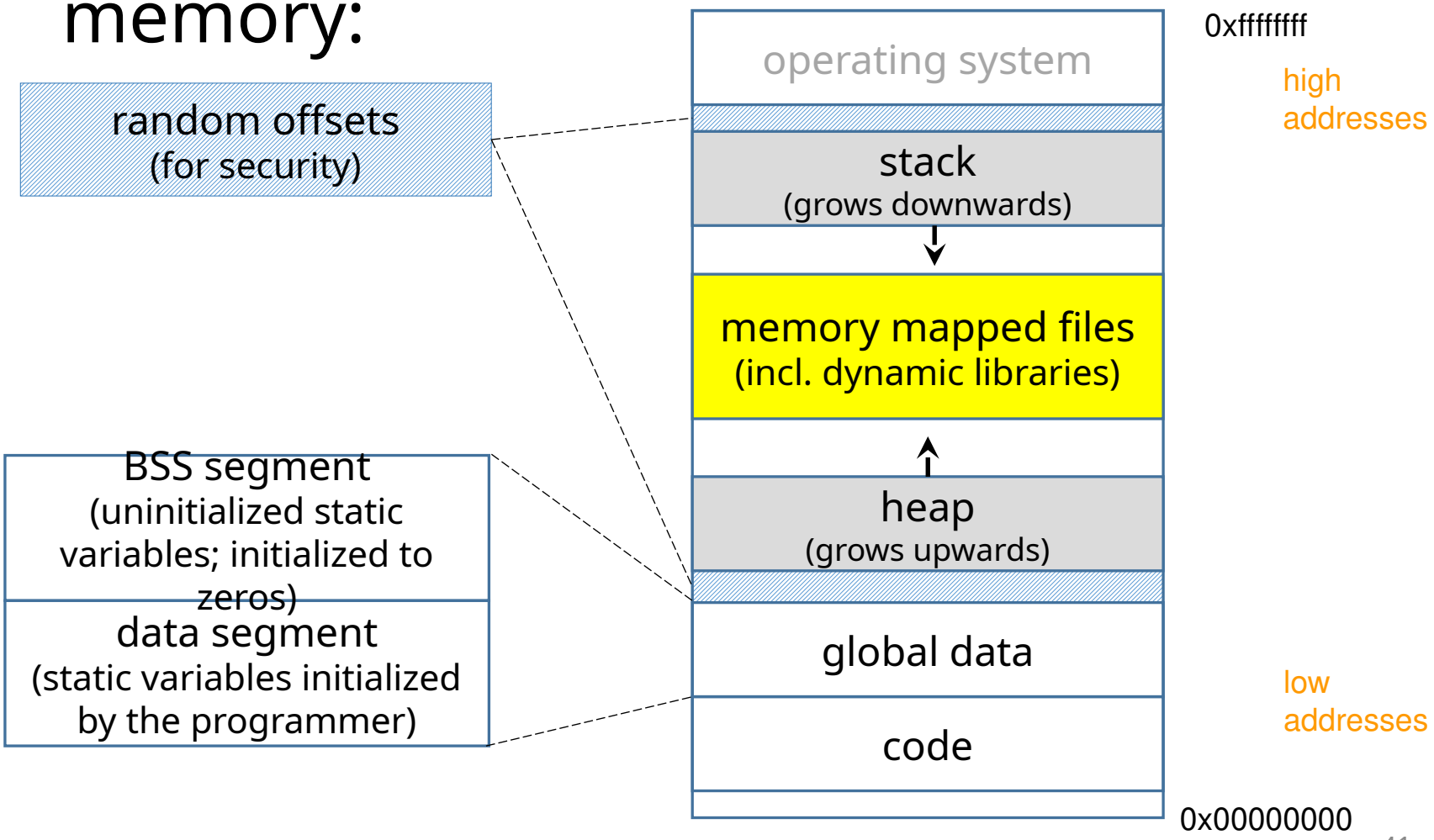
Shared Libraries*

- Have a single copy of the library that is used by all running programs.
- Saves (disk and memory) space by avoiding replication of library code.
- Virtual memory management in the OS allows different processes to share "read-only" pages, e.g., text and read-only data.
  - *This lets us get by with a single physical-memory copy of shared library code.*

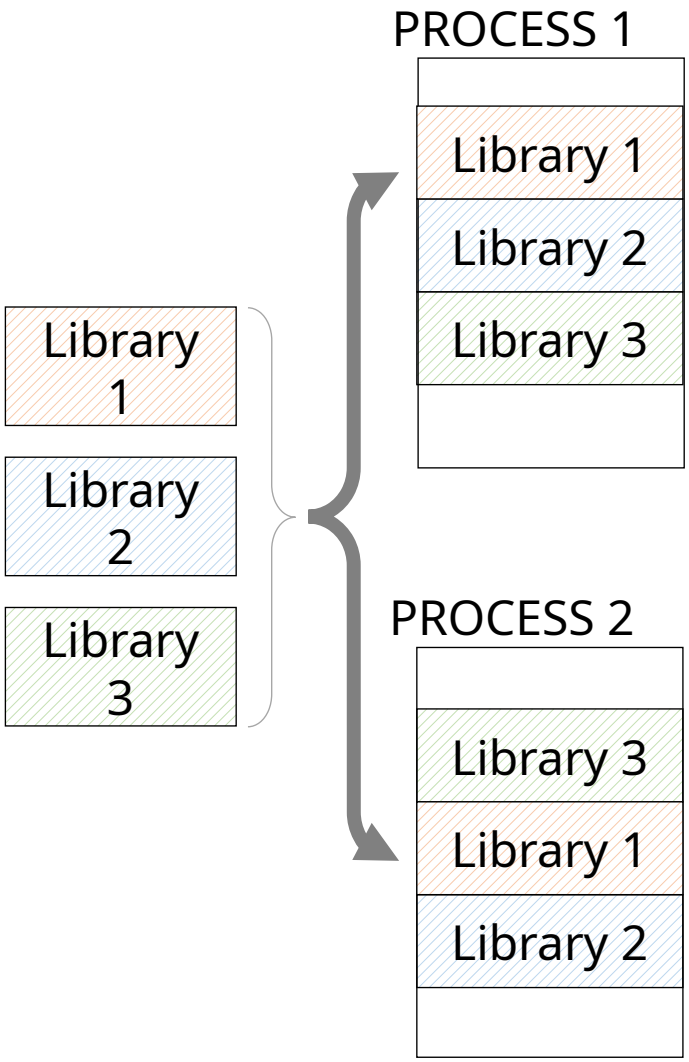
* Microsoft Windows terminology: DLLs

Runtime Memory Organization (Linux)

Layout of an executing process's virtual memory:



Shared libraries: issues



The load order (∴ load address) of libraries may be different for different processes

- address relocation is not possible if we want to share code across
- how to allow loading at different addresses w/o requiring code relocation?

Solution: Position-Independent Code

Position-independent Code

Position-Independent Code (PIC)

- Basic idea:
  - separate code from data
  - generate code that doesn't depend on where it is loaded.
- PC-relative addressing can give position-independent code references.
  - *in some situations (e.g., data references;* **call** *instruction in Intel x86) we may still need absolute addressing*

Position-Independent Code

- ELF executable file characteristics:
  - data pages follow code pages
  - the offset from the code to the data does not depend on where the program is loaded
- The linker creates a *global offset table* (GOT) that contains offsets to all global data used
- If an instruction can load its own address into a register, it can then use a fixed offset to access the GOT, and thence the data

Position-Independent Code

Code to compute the address of instruction I (x86):

```
      call I     /* push address of I on the stack */
I:    pop r1     /* pop address of I into r1 */
```

Accessing a global x from an instruction I in PIC:

1. GOT has an entry for x at position b

   (dynamic linker fills in the address of x into this entry at load time)

2. Compute *addr*(I) into some register r_1 (above)

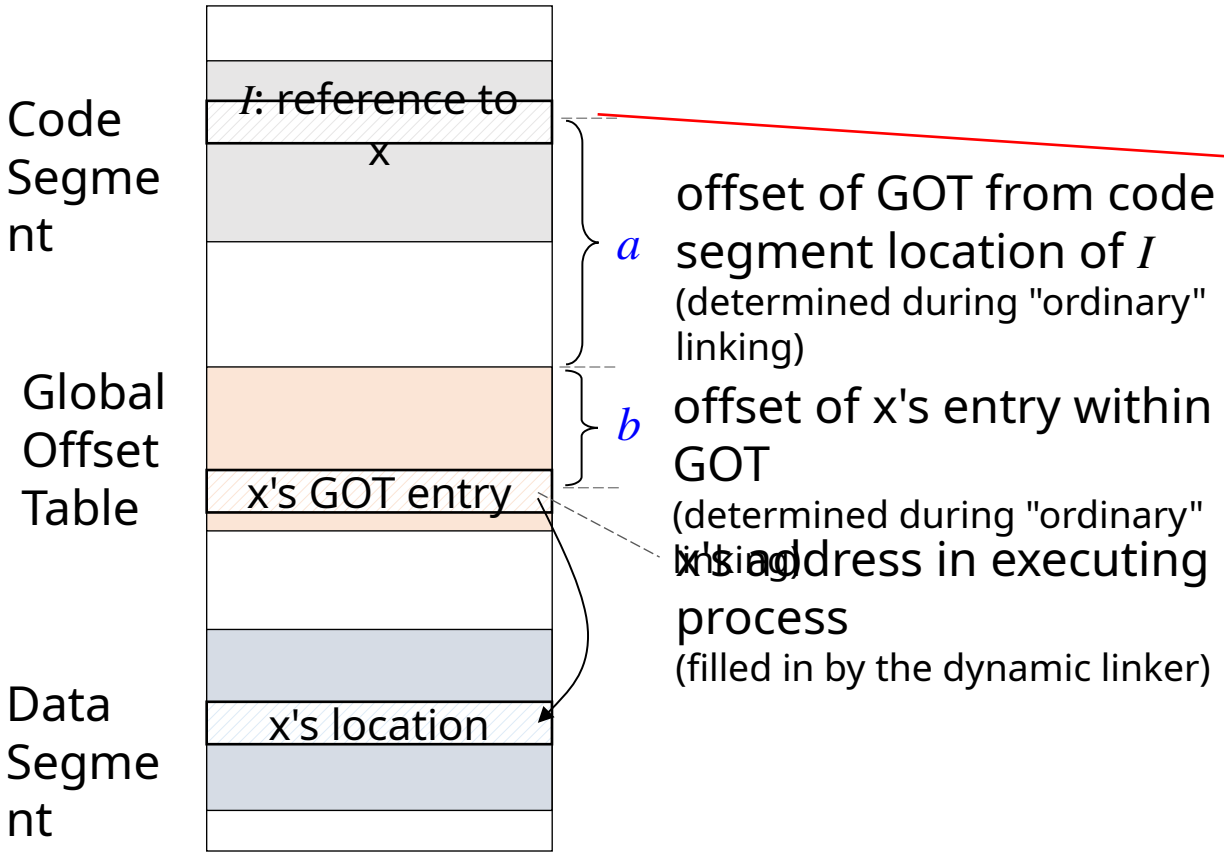
3. r_1 += a

   (a ≡ offset from I to x's GOT entry, fixed for a given program)

4. r_2 = contents of location $b(r_1)$ /* r_2 = *addr*(x) */

5. access memory location pointed at by r_2

PIC: Global data reference



CODE TO ACCESS x

```
    call L
L : pop r1      // addr(I)
    r1 += a     // addr(GOT)
    r2 = mov  b(r1)
    ... access  0(r2) ...
```

Dynamic Linking

- Defers much of the linking process until the program starts running
- Easier to create and update shared libraries
- Incurs a runtime performance cost :
  - Much of the linking process has to be redone each time a program runs
  - Every dynamically linked symbol has to be looked up in the symbol table and resolved at runtime

Dynamic Linking: Basic Mechanism

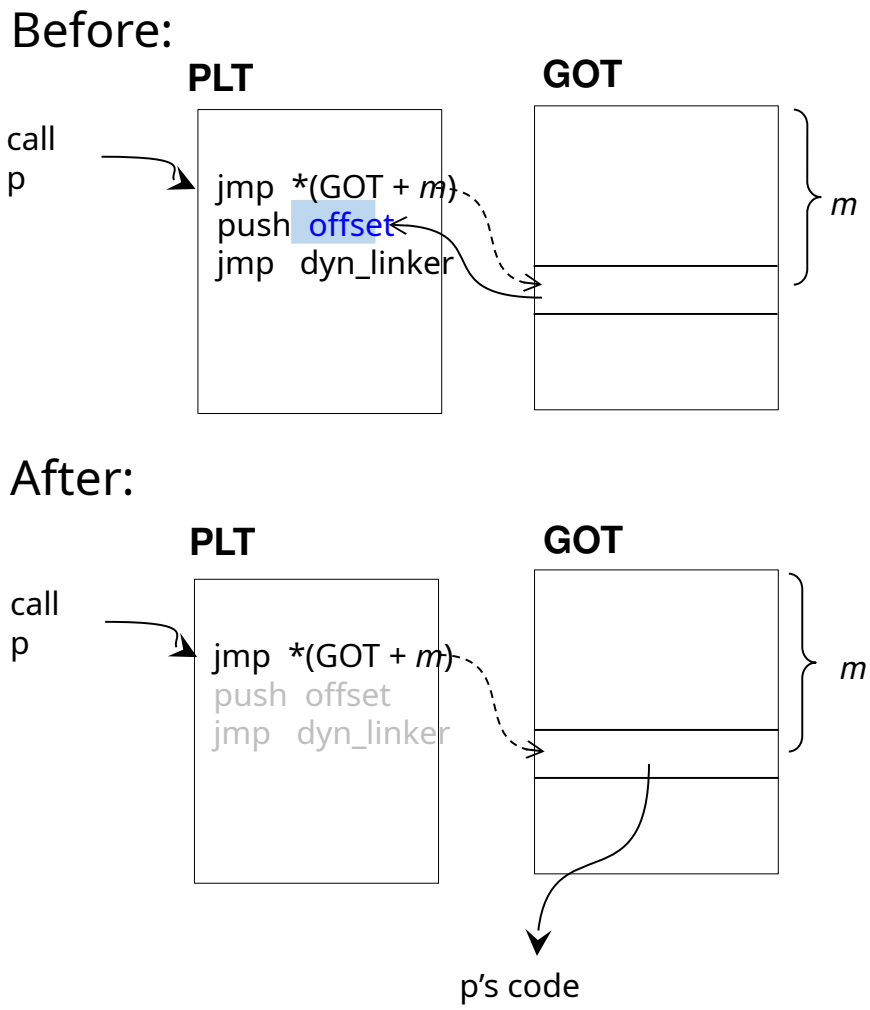
- A reference to a dynamically linked procedure p is mapped to code that invokes a *handler*
- When p is called at runtime:
  - if p's code has not been loaded and linked already (due to some earlier reference):
    - the handler is executed; loads and links p's code
  - otherwise, the code for p is executed normally

Dynamic Linking

- ELF shared libraries use PIC (position independent code), so text sections do not need relocation
- Data references use a Global Offset Table (GOT):
  - each global symbol has a relocatable pointer to it in the GOT
  - the dynamic linker relocates these pointers
- We still need to invoke the dynamic linker on the first reference to a dynamically linked procedure.
  - done using a *procedure linkage table* (PLT)
  - PLT adds a level of indirection for function calls

Dynamic Linking: Code

- Initially, GOT entry points to PLT code that invokes the dynamic linker.
  - offset identifies both the symbol being resolved and the corresponding GOT entry.
- The dynamic linker looks up the symbol value and updates the GOT entry.
- Subsequent calls bypass dynamic linker, go directly to callee.
- This reduces program startup time. Also, routines that are never called are not resolved.



PIC: Advantages and Disadvantages

- Advantages:
  - code does not have to be relocated when loaded (however, data still need to be relocated)
  - different processes can share code memory pages, even if they don't have the same address space allocated
- Disadvantages:
  - PIC code is bigger and slower than non-PIC code
    - amount of slowdown is architecture-dependent
    - depends on availability of a register to point to GOT