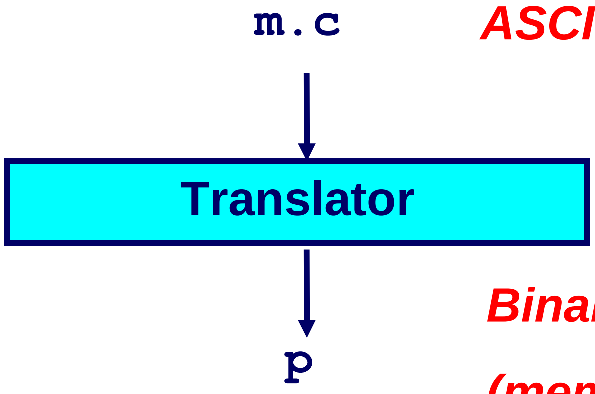
Linking

Topics

- Static linking
- Object files
- Static libraries
- Loading
- Dynamic linking of shared libraries

A Simplistic Program Translation Scheme



`m.c`

ASCII source file

Translator

`p`

Binary executable object file

(memory image on disk)

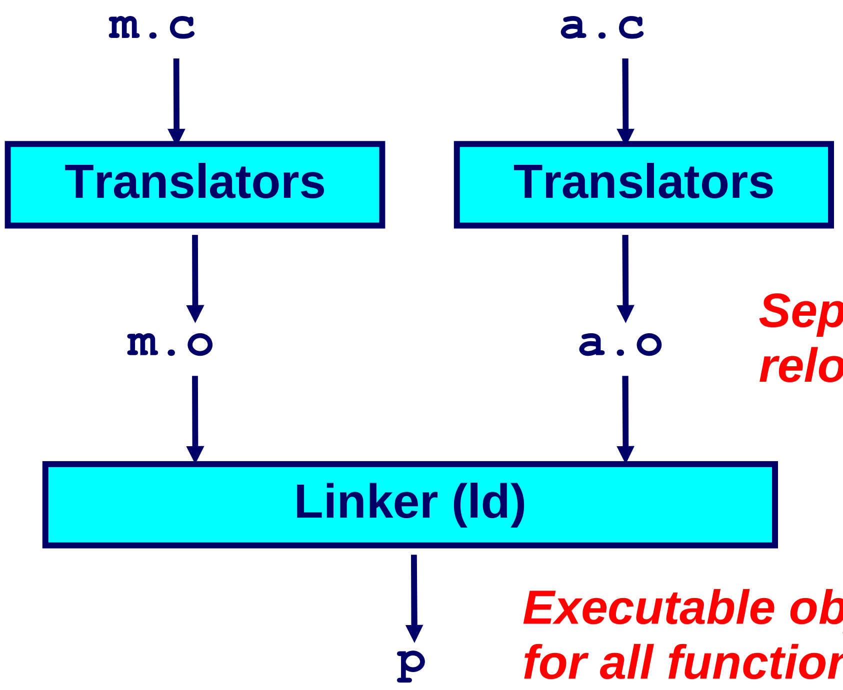
Problems:

- **Efficiency: small change requires complete recompilation**
- **Modularity: hard to share common functions (e.g. `printf`)**

Solution:

- ***Static linker (or linker)***

A Better Scheme Using a Linker



`m.c` `a.c`

Translators **Translators**

`m.o` `a.o`

Separately compiled relocatable object files

Linker (ld)

`p`

Executable object file (contains code and data for all functions defined in `m.c` and `a.c`)

Translating the Example Program

Compiler driver coordinates all steps in the translation and linking process.

- Typically included with each compilation system (e.g., **`gcc`**)
- Invokes preprocessor and compiler (**`cc1`**), assembler (**`as`**), and linker (**`ld`**).
- Passes command line arguments to appropriate phases

Example: create executable **`p`** from **`m.c`** and **`a.c`**:

```
bash> gcc -O2 -v m.c a.c -o p
cc1 m.c -O2 [args] -o /tmp/cca07630.s
as [args] -o /tmp/cca076301.o /tmp/cca07630.s
<similar process for a.c>
ld -o p [system obj files] /tmp/cca076301.o /tmp/cca076302.o
bash>
```

Why Linkers?

Modularity

- Program can be written as a collection of smaller source files, rather than one monolithic mass.
- Can build libraries of common functions (more on this later)
  - e.g., Math library, standard C library

Efficiency

- Time:
  - Change one source file, compile, and then relink.
  - No need to recompile other source files.
- Space:
  - Libraries of common functions can be aggregated into a single file...
  - Yet executable files and running memory images contain only code for the functions they actually use.

What Does a Linker Do?

Linker:

- merges multiple relocatable (.o) object files into a single executable object file that can loaded and executed by the loader.
- resolves external references

External reference: reference to a symbol defined in another object file.

References can be in either code or data

code: **`a();`**   **`/* reference to symbol a */`**

data: **`int *xp=&x;`** **`/* reference to symbol x */`**

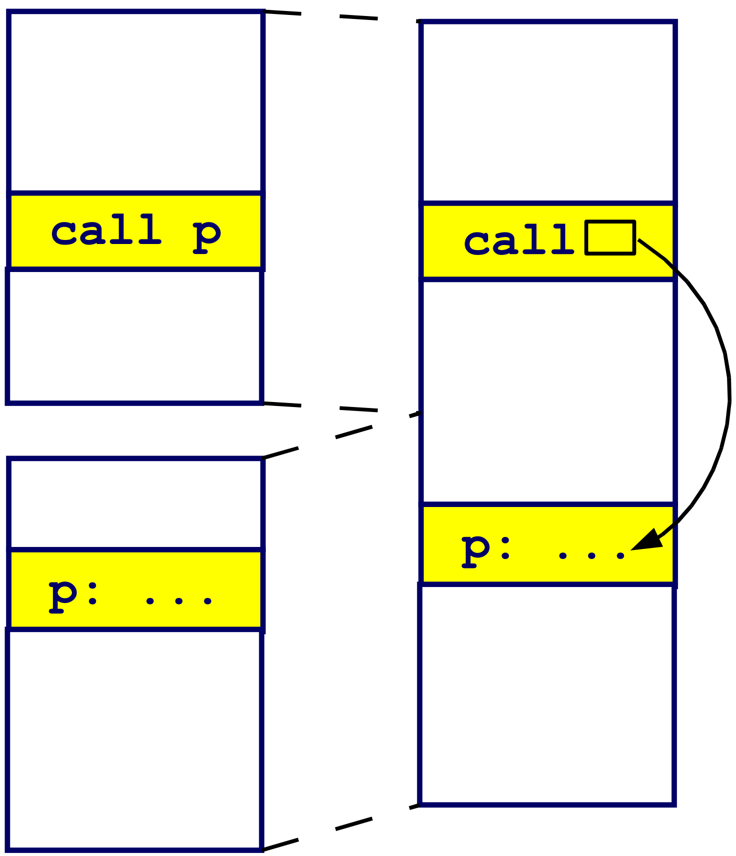
Suppose:

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

The linker must:

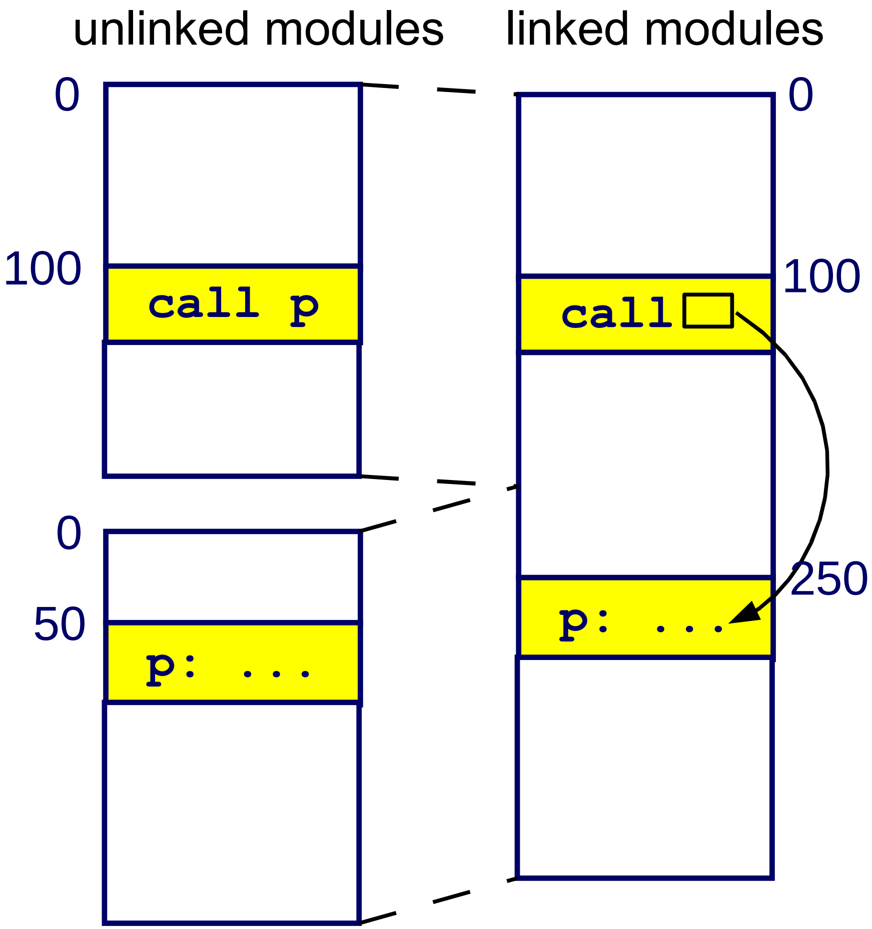
- determine the location of p in the object module obtained from merging *A* and *B*; and
- modify references to p (in both *A* and *B*) to refer to this location.

unlinked modules   linked modules



Fixing Addresses

- Addresses in an object file are usually relative to the start of the code or data segment in that file.
- When different object files are combined:
  - The same kind of sctions (text, data, read-only data, etc.) from the different object files get merged.
  - Addresses have to be “fixed up” to account for this merging.
  - The fixing up is done by the linker, using information embedded in the executable for this purpose (“relocations”).



Information for Symbol Resolution

- Each linkable module contains a *symbol table*, whose contents include:
  - Global symbols defined (maybe referenced) in the module.
  - Global symbols referenced but not defined in the module (these are generally called *externals*).
  - Segment names (e.g., *text*, *data*, *rodata*).
    - *These are usually considered to be global symbols defined to be at the beginning of the segment*.
  - Non-global symbols and line number information (optional), for debuggers.

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.

Linker Actions: Pass 1

- Construct a table of all the object modules and their lengths.
- Based on this table, assign a *load address* to each module.
- For each module:
  - Read in its symbol table into a *global symbol table* in the linker.
  - Determine the address of each symbol defined in the module in the output:

    Use the symbol value together with the module load address.

Linker Actions: Pass 2

Copy the object modules in the order of their load addresses:

- *Address relocation*:
  - find each instruction that contains a memory address;
  - modify the address to point to the correct value
- *External symbol resolution*:
  - For each instruction that references an external object, insert the actual address for that object.

Executable and Linkable Format (ELF)

Standard binary format for object files

Derives from AT&T System V Unix

- Later adopted by BSD Unix variants and Linux

One unified format for

- Relocatable object files (`.o`),
- Executable object files
- Shared object files (`.so`)

Generic name: ELF binaries

Better support for shared libraries than old `a.out` formats.

ELF Object File Format

Elf header

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

Program header table

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

`.text` section

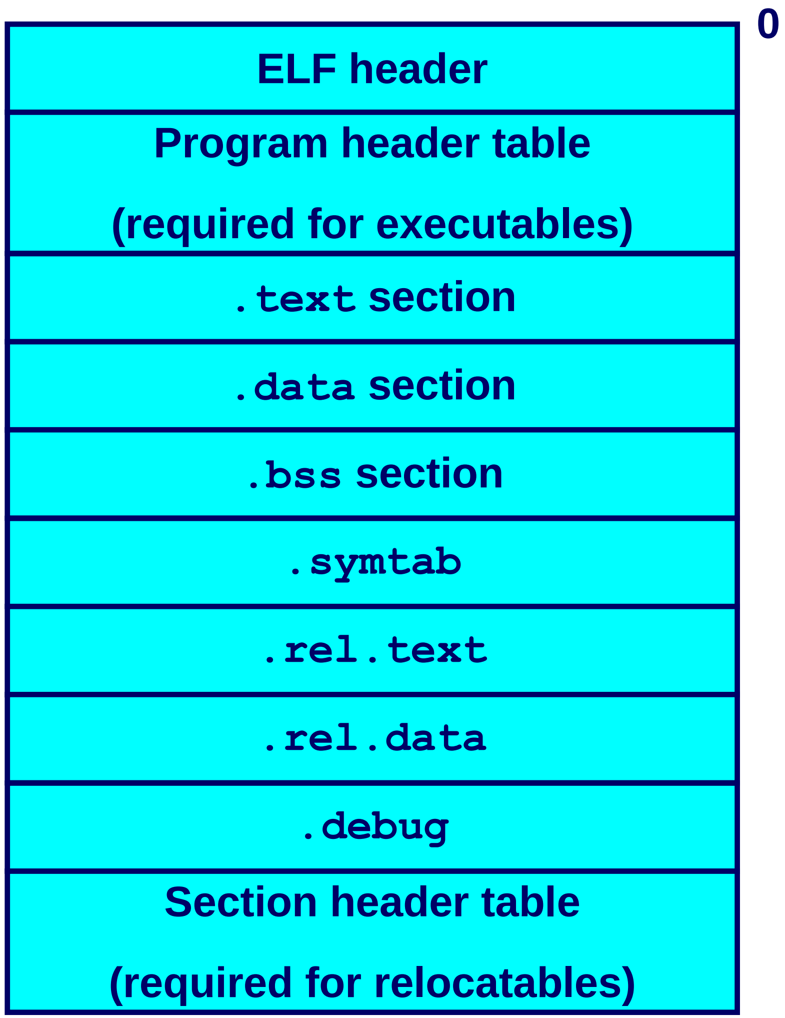
- Code

`.data` section

- Initialized (static) data

`.bss` section

- Uninitialized (static) data
- “Block Started by Symbol”
- “Better Save Space”
- Has section header but occupies no space



ELF Object File Format (cont)

`.symtab` section

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

`.rel.text` section

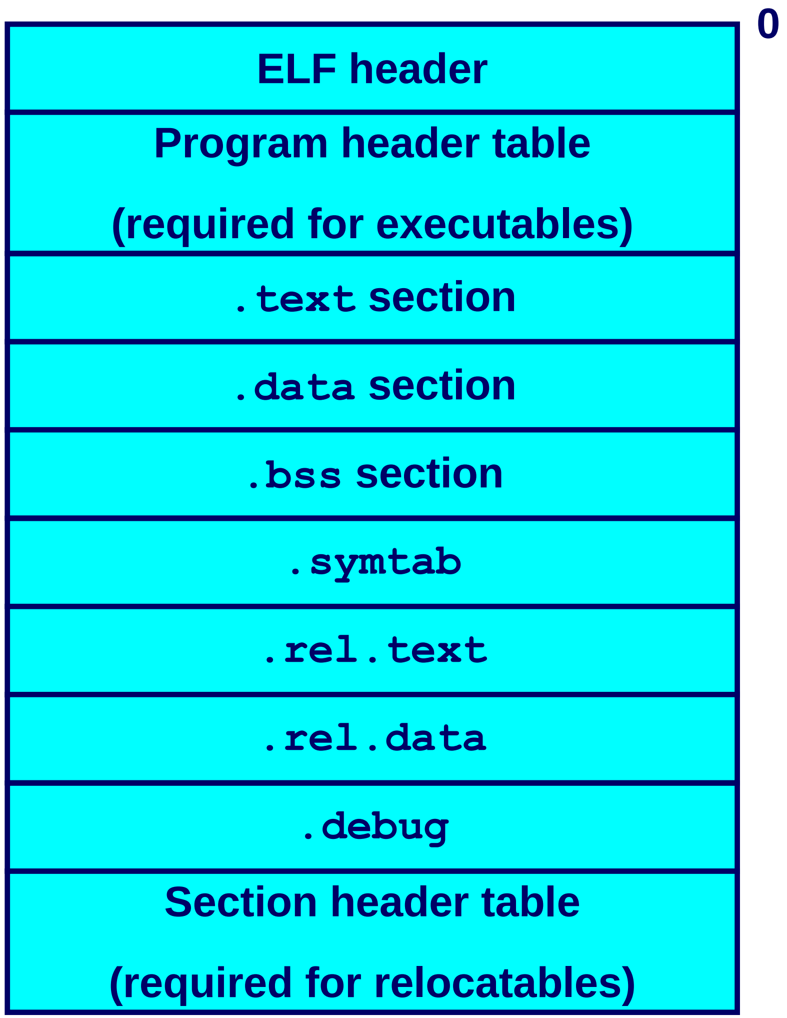
- Relocation info for `.text` section
- Addresses of instructions that will need to be modified in the executable
- Instructions for modifying.

`.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`)



0

ELF header

Program header table

(required for executables)

`.text` section

`.data` section

`.bss` section

`.symtab`

`.rel.text`

`.rel.data`

`.debug`

Section header table

(required for relocatables)

Example C Program

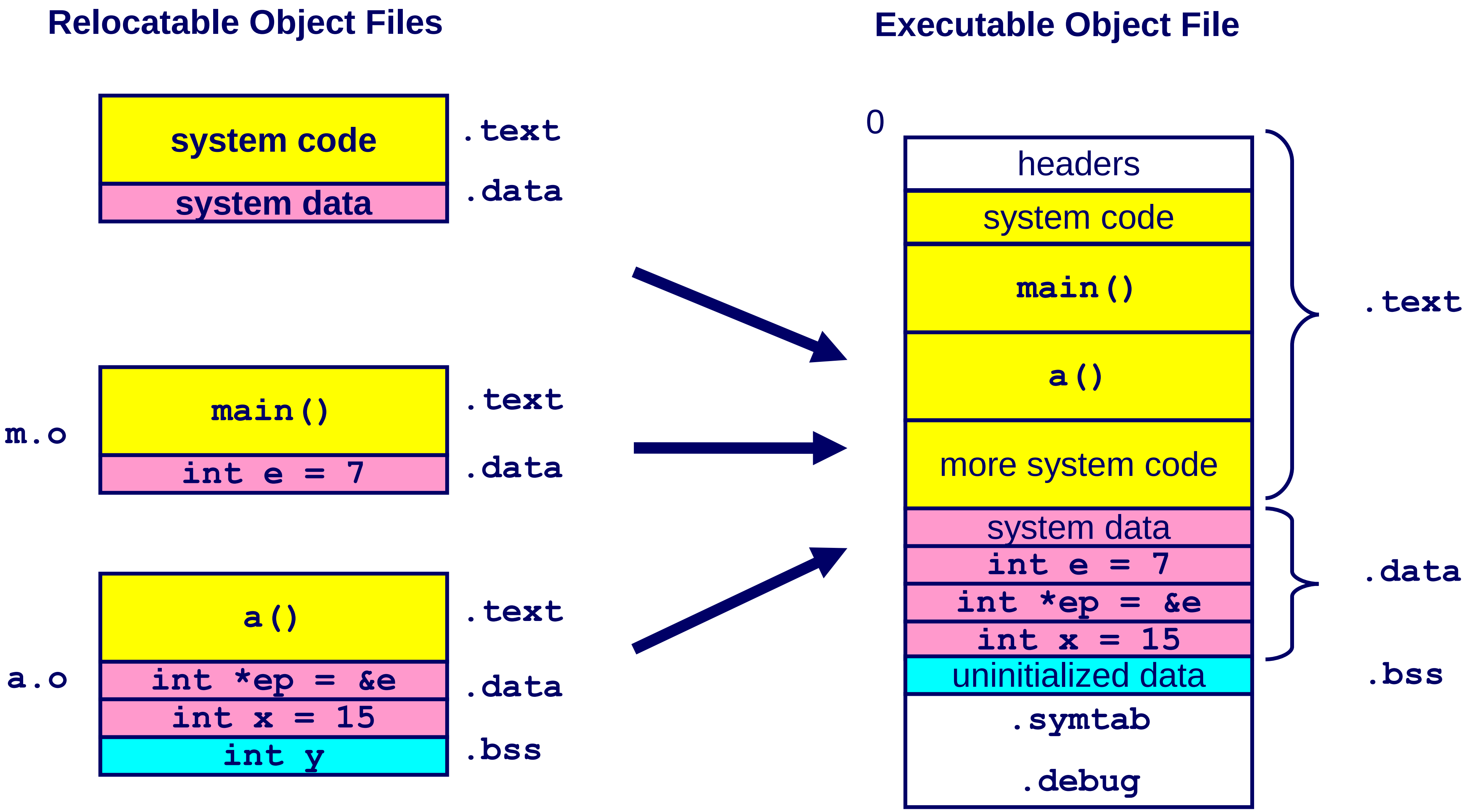
`m.c`

```
int e=7;


int main() {
  int r = a();
  exit(0);
}
```

`a.c`

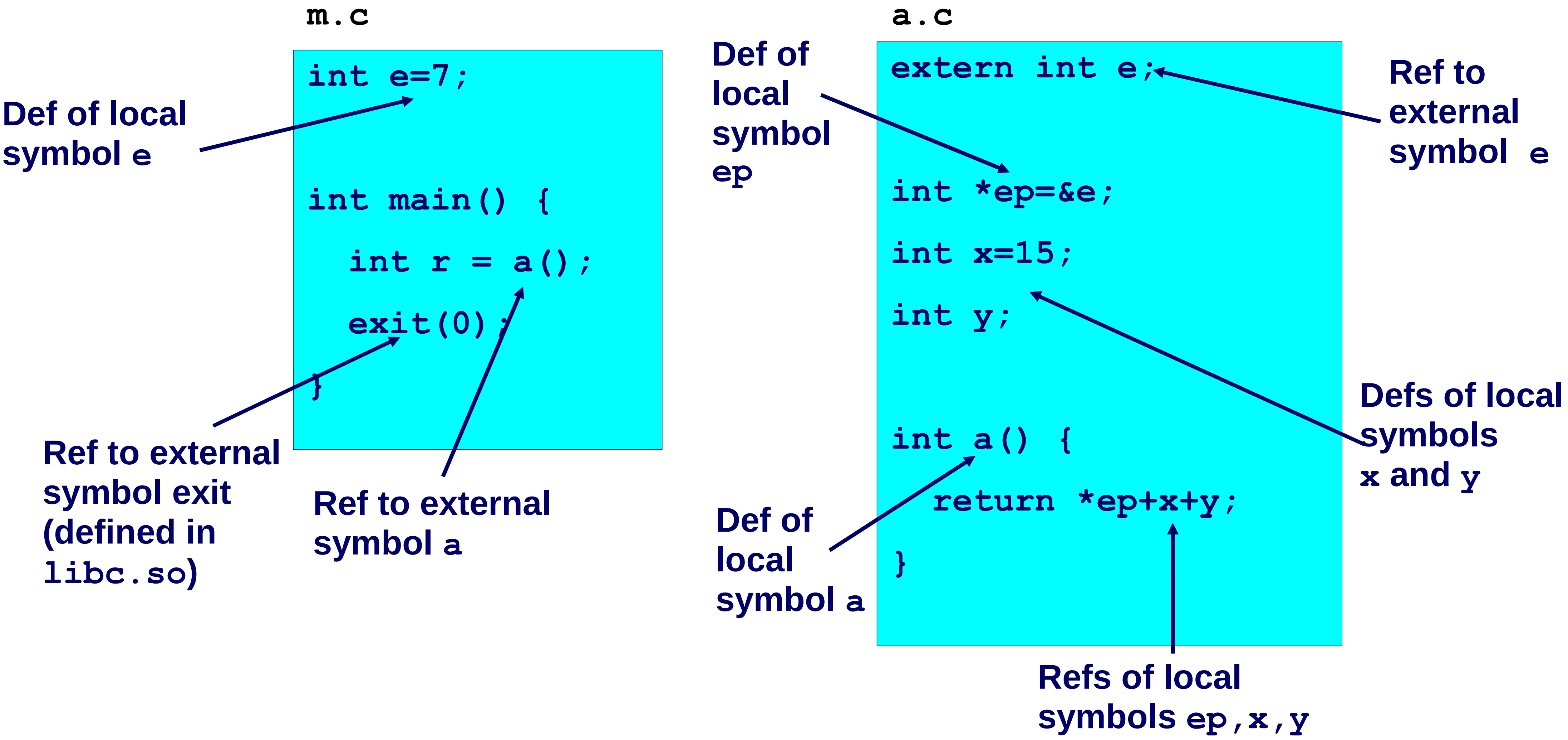
```
extern int e;


int *ep=&e;
int x=15;
int y;

int a() {
  return *ep+x+y;
}
```

Merging Relocatable Object Files into an Executable Object File



Relocating Symbols and Resolving External References

- *Symbols* are lexical entities that name functions and variables.
- Each symbol has a *value* (typically a memory address).
- Code consists of symbol *definitions* and *references*.
- References can be either *local* or *external*.



`m.c`

```
int e=7;

int main() {
   int r = a();
   exit(0);
}
```

`a.c`

```
extern int e;

int *ep=&e;
int x=15;
int y;

int a() {
   return *ep+x+y;
}
```

Def of local symbol `e`

Ref to external symbol exit (defined in `libc.so`)

Ref to external symbol `a`

Def of local symbol `ep`

Ref to external symbol `e`

Defs of local symbols `x` and `y`

Def of local symbol `a`

Refs of local symbols `ep`,`x`,`y`

m.o Relocation Info

m.c

```
int e=7;


int main() {
  int r = a();
  exit(0);
}
```

```
objdump -d -r m.o
objdump -d -r -j.data m.o
```

```
Disassembly of section .text:


00000000 <main>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   83 ec 08                sub    $0x8,%esp
   6:   83 e4 f0                and    $0xfffffff0,%esp
   9:   e8 fc ff ff ff          call   a <main+0xa>
                         a: R_386_PC32    a
   e:   c7 04 24 00 00 00 00    movl   $0x0,(%esp)
  15:   e8 fc ff ff ff          call   16 <main+0x16>
                         16: R_386_PC32   exit
```

```
Disassembly of section .data:
 00000000 <e>:
   0:   07 00 00 00
```

a.o Relocation Info (.text)

a.c

```
extern int e;


int *ep=&e;

int x=15;

int y;


int a() {

  return *ep+x+y;

}
```

```
Disassembly of section .text:


00000000 <a>:

   0:    55                             push    %ebp

   1:    8b 15 00 00 00 00              mov     0x0,%edx

                            3: R_386_32        ep

   7:    a1 00 00 00 00                 mov     0x0,%eax

                            8: R_386_32        x

   c:    89 e5                          mov     %esp,%ebp

   e:    8b 0a                          mov     (%edx),%ecx

  10:    8b 15 00 00 00 00              mov     0x0,%edx

                            12: R_386_32       y

  16:    5d                             pop     %ebp

  17:    01 c8                          add     %ecx,%eax

  19:    01 d0                          add     %edx,%eax

  1b:    c3                             ret
```

`a.o` Relocation Info (`.data`)

a.c

```
extern int e;


int *ep=&e;

int x=15;

int y;


int a() {

  return *ep+x+y;

}
```

```
Disassembly of section .data:

00000000 <ep>:

   0:   00 00 00 00

                                   0: R_386_32     e

00000004 <x>:

   4:   0f 00 00 00
```

Executable After Relocation and External Reference Resolution (.text)

```
08048390 <main>:
 8048390:       55                      push   %ebp
 8048391:       89 e5                   mov    %esp,%ebp
 8048393:       83 ec 08                sub    $0x8,%esp
 8048396:       83 e4 f0                and    $0xfffffff0,%esp
 8048399:       e8 12 00 00 00          call   80483b0 <a>
 804839e:       c7 04 24 00 00 00 00    movl   $0x0,(%esp)
 80483a5:       e8 06 ff ff ff          call   80482b0 <_init+0x38>
 80483aa:       90                      nop

...

080483b0 <a>:
 80483b0:       55                      push   %ebp
 80483b1:       8b 15 f4 94 04 08       mov    0x80494f4,%edx
 80483b7:       a1 f8 94 04 08          mov    0x80494f8,%eax
 80483bc:       89 e5                   mov    %esp,%ebp
 80483be:       8b 0a                   mov    (%edx),%ecx
 80483c0:       8b 15 f8 95 04 08       mov    0x80495f8,%edx
 80483c6:       5d                      pop    %ebp
 80483c7:       01 c8                   add    %ecx,%eax
 80483c9:       01 d0                   add    %edx,%eax
 80483cb:       c3                      ret
 80483cc:       90                      nop
```

Executable After Relocation and External Reference Resolution (`.data`)

`m.c`

```
int e=7;

int main() {
  int r = a();
  exit(0);
}
```

`a.c`

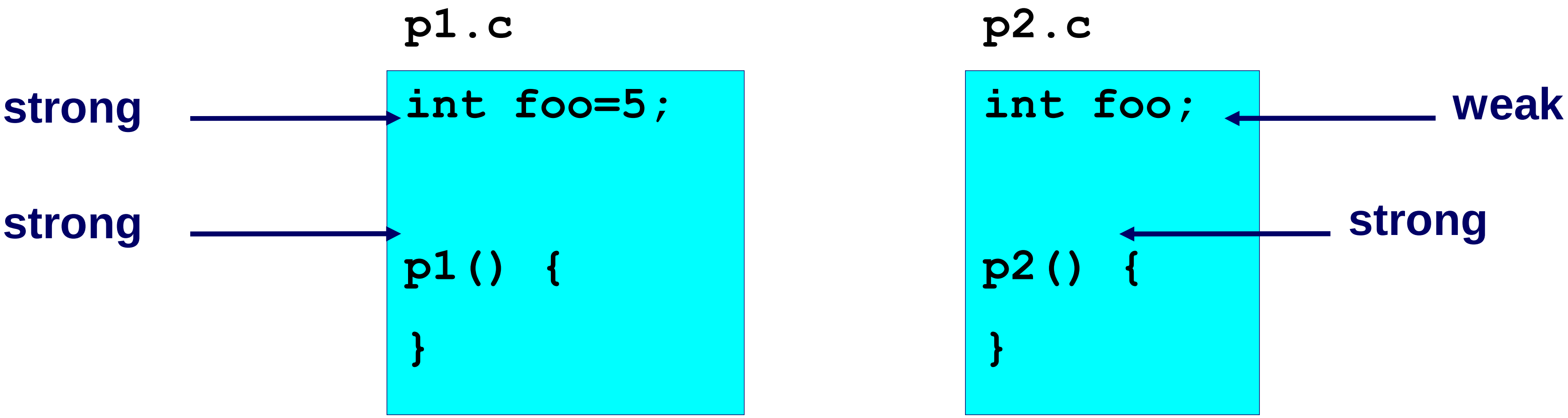
```
extern int e;

int *ep=&e;
int x=15;
int y;

int a() {
  return *ep+x+y;
}
```

```
080494f0 <e>:
 80494f0:       07 00 00 00
080494f4 <ep>:
 80494f4:       f0 94 04 08
080494f8 <x>:
 80494f8:       0f 00 00 00
```

Strong and Weak Symbols

Program symbols are either strong or weak

- *strong*: procedures and initialized globals
- *weak*: uninitialized globals



strong

strong

`p1.c`

```
int foo=5;

p1() {
}
```

`p2.c`

```
int foo;

p2() {
}
```

weak

strong

Linker’s Symbol Rules

Rule 1. A strong symbol can only appear once.

Rule 2. A weak symbol can be overridden by a strong symbol of the same name.

- references to the weak symbol resolve to the strong symbol.

Rule 3. If there are multiple weak symbols, the linker can pick an arbitrary one.

Linker Puzzles

```
int x;
p1() {}
```

```
p1() {}
```

Link time error: two strong symbols (`p1`)

```
int x;
p1() {}
```

```
int x;
p2() {}
```

References to `x` will refer to the same uninitialized int. Is this what you really want?

```
int x;
int y;
p1() {}
```

```
double x;
p2() {}
```

Writes to `x` in `p2` might overwrite `y`! Evil!

```
int x=7;
int y=5;
p1() {}
```

```
double x;
p2() {}
```

Writes to `x` in `p2` will overwrite `y`! Nasty!

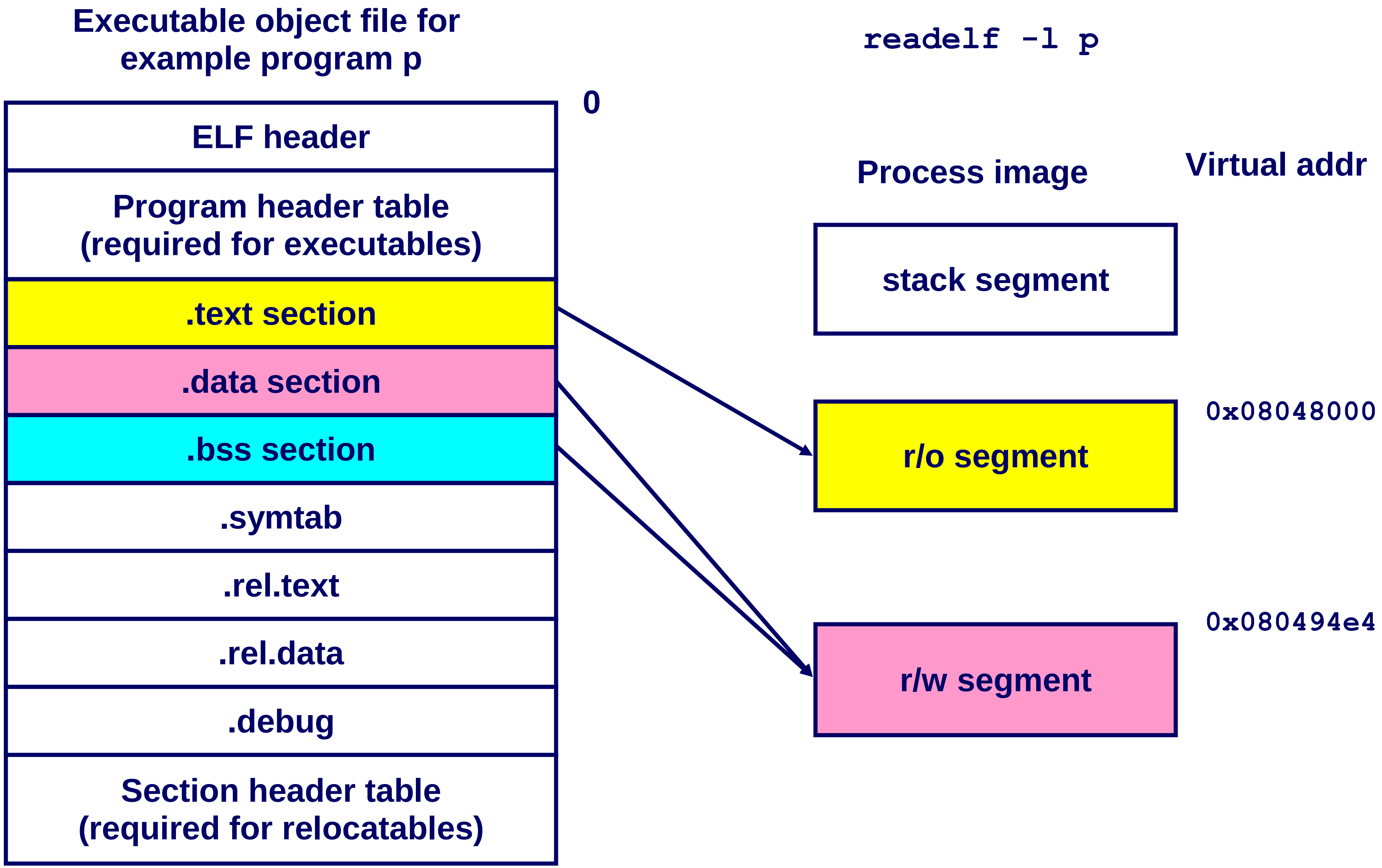
```
int x=7;
p1() {}
```

```
int x;
p2() {}
```

References to `x` will refer to the same initialized variable.

Nightmare scenario: two identical weak structs, compiled by different compilers with different alignment rules.

Loading Executable Binaries



Executable object file for example program p

`readelf -l p`

When programs are loaded to memory, sections are mapped to segments. A segment can contain information from more than one section

Packaging Commonly Used Functions

How to package functions commonly used by programmers?

- Math, I/O, memory management, string manipulation, etc.

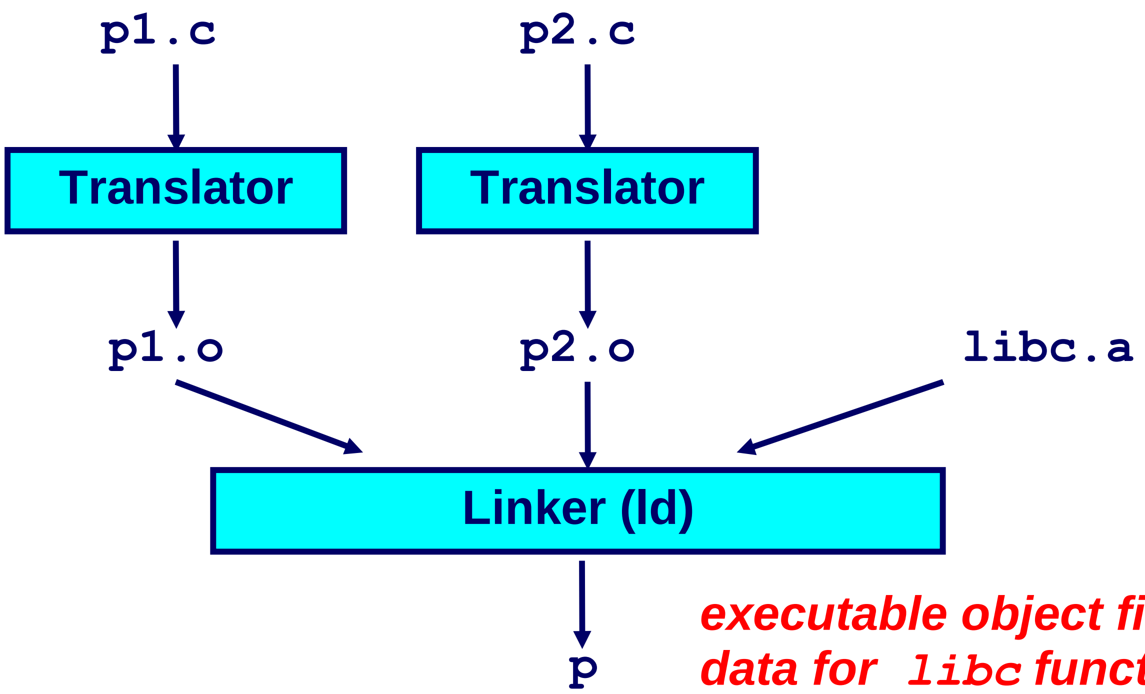
Awkward, given the linker framework so far:

- Option 1: Put all functions in a single source file
  - Programmers link big object file into their programs
  - Space and time inefficient
- Option 2: Put each function in a separate source file
  - Programmers explicitly link appropriate binaries into their programs
  - More efficient, but burdensome on the programmer

Solution: *static libraries* (`.a` archive files)

- Concatenate related relocatable object files into a single file with an index (called an archive).
- Enhance linker so that it tries to resolve unresolved external references by looking for the symbols in one or more archives.
- If an archive member file resolves reference, link into executable.

Static Libraries (archives)



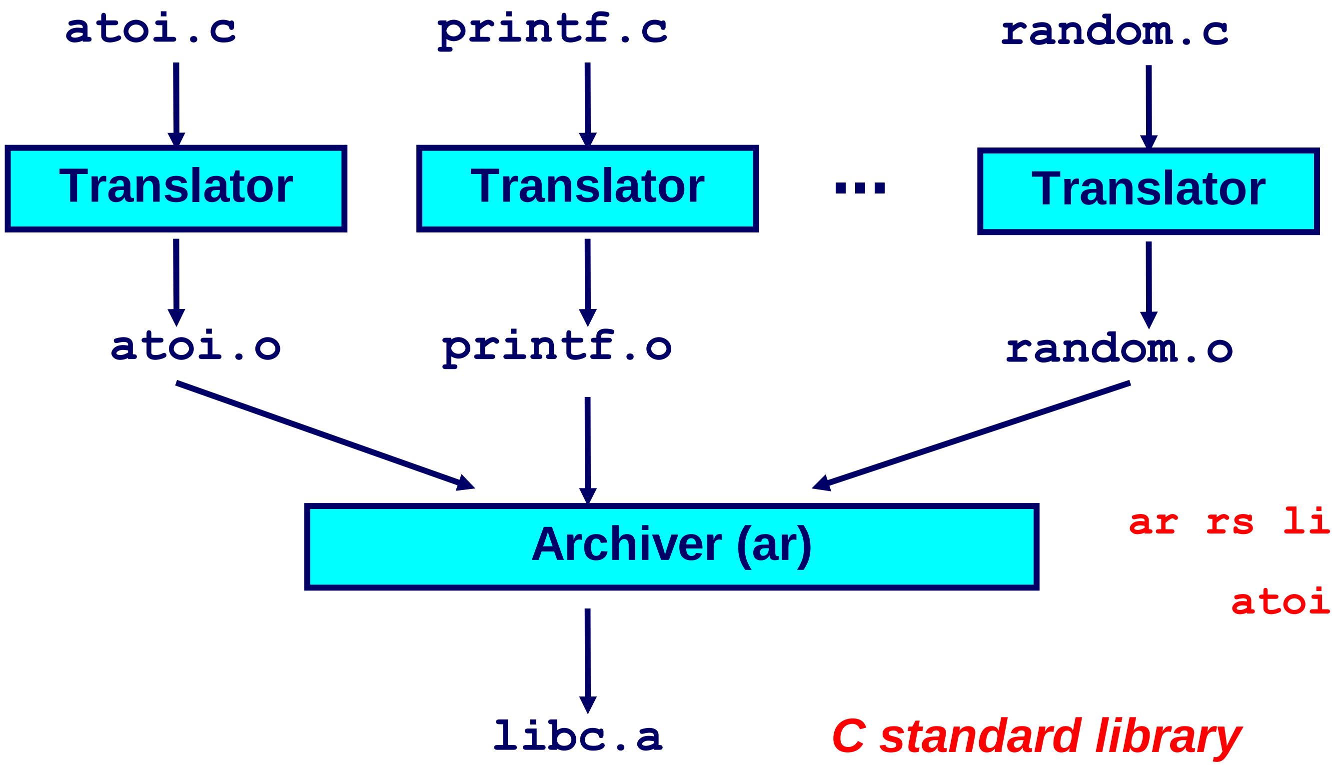
static library (archive) of relocatable object files concatenated into one file.

executable object file (only contains code and data for `libc` functions that are called from `p1.c` and `p2.c`)

Further improves modularity and efficiency by packaging commonly used functions [e.g., C standard library (`libc`), math library (`libm`)]

Linker selectively only the `.o` files in the archive that are actually needed by the program.

Creating Static Libraries



`atoi.c` `printf.c` … `random.c`

Translator Translator … Translator

`atoi.o` `printf.o` `random.o`

Archiver (ar)

`libc.a` *C standard library*

```
ar rs libc.a \
     atoi.o printf.o … random.o
```

Archiver allows incremental updates:

- **Recompile function that changes and replace .o file in archive.**

Commonly Used Libraries

`libc.a` (the C standard library)

- 2 MB archive of 1265 object files.
- I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math

`libm.a` (the C math library)

- 500 kB archive of 401 object files.
- floating point math (sin, cos, tan, log, exp, sqrt, …)

```
% ar -t /usr/lib/libc.a | sort
…
fork.o
…
fprintf.o
fpu_control.o
fputc.o
freopen.o
fscanf.o
fseek.o
fstab.o
…
```

```
% ar -t /usr/lib/libm.a | sort
…
e_acos.o
e_acosf.o
e_acosh.o
e_acoshf.o
e_acoshl.o
e_acosl.o
e_asin.o
e_asinf.o
e_asinl.o
…
```

Using Static Libraries

Linker’s algorithm for resolving external references:

- Scan `.o` files and `.a` files in the command line order.
- During the scan, keep a list of the current unresolved references.
- As each new `.o` or `.a` file obj is encountered, try to resolve each unresolved reference in the list against the symbols in obj.
- If any entries in the unresolved list at end of scan, then error.

Problem:

- Command line order matters!
- Moral: put libraries at the end of the command line.

```
bass> gcc -L. libtest.o -lmine

bass> gcc -L. -lmine libtest.o

libtest.o: In function `main':

libtest.o(.text+0x4): undefined reference to `libfun'
```

Shared Libraries

Static libraries have the following disadvantages:

- Potential for duplicating lots of common code in the executable files on a filesystem.
  - e.g., every C program needs the standard C library
- Potential for duplicating lots of code in the virtual memory space of many processes.
- Minor bug fixes of system libraries require each application to explicitly relink

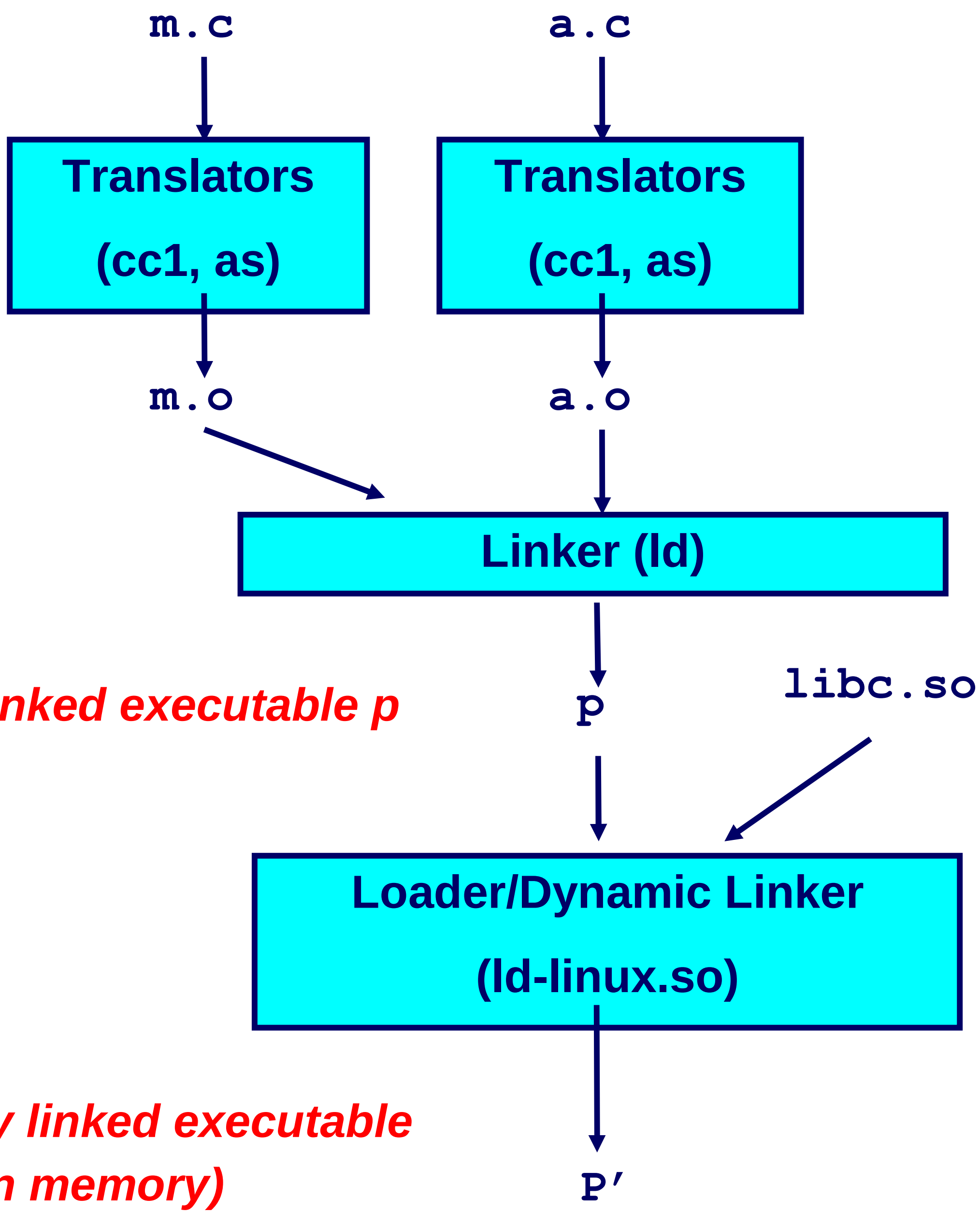
Solution:

- *Shared libraries* (dynamic link libraries, DLLs) whose members are dynamically loaded into memory and linked into an application at run-time.
  - Dynamic linking can occur when executable is first loaded and run.
    - Common case for Linux, handled automatically by **`ld-linux.so`**.
  - Dynamic linking can also occur after program has begun.
    - In Linux, this is done explicitly by user with **`dlopen()`**.
    - Basis for High-Performance Web Servers.
  - Shared library routines can be shared by multiple processes.

Dynamic Linking

- Defers much of the linking process until the program starts running.
- Easier to create, update than statically linked shared libraries.
- Has higher runtime performance cost than statically linked libraries:
    - 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.

Dynamically Linked Shared Libraries



`m.c`

`a.c`

Translators

(cc1, as)

Translators

(cc1, as)

`m.o`

`a.o`

Linker (ld)

Partially linked executable p

(on disk)

`p`

`libc.so`

Shared library of dynamically relocatable object files

Loader/Dynamic Linker

(ld-linux.so)

`libc.so` functions called by `m.c`

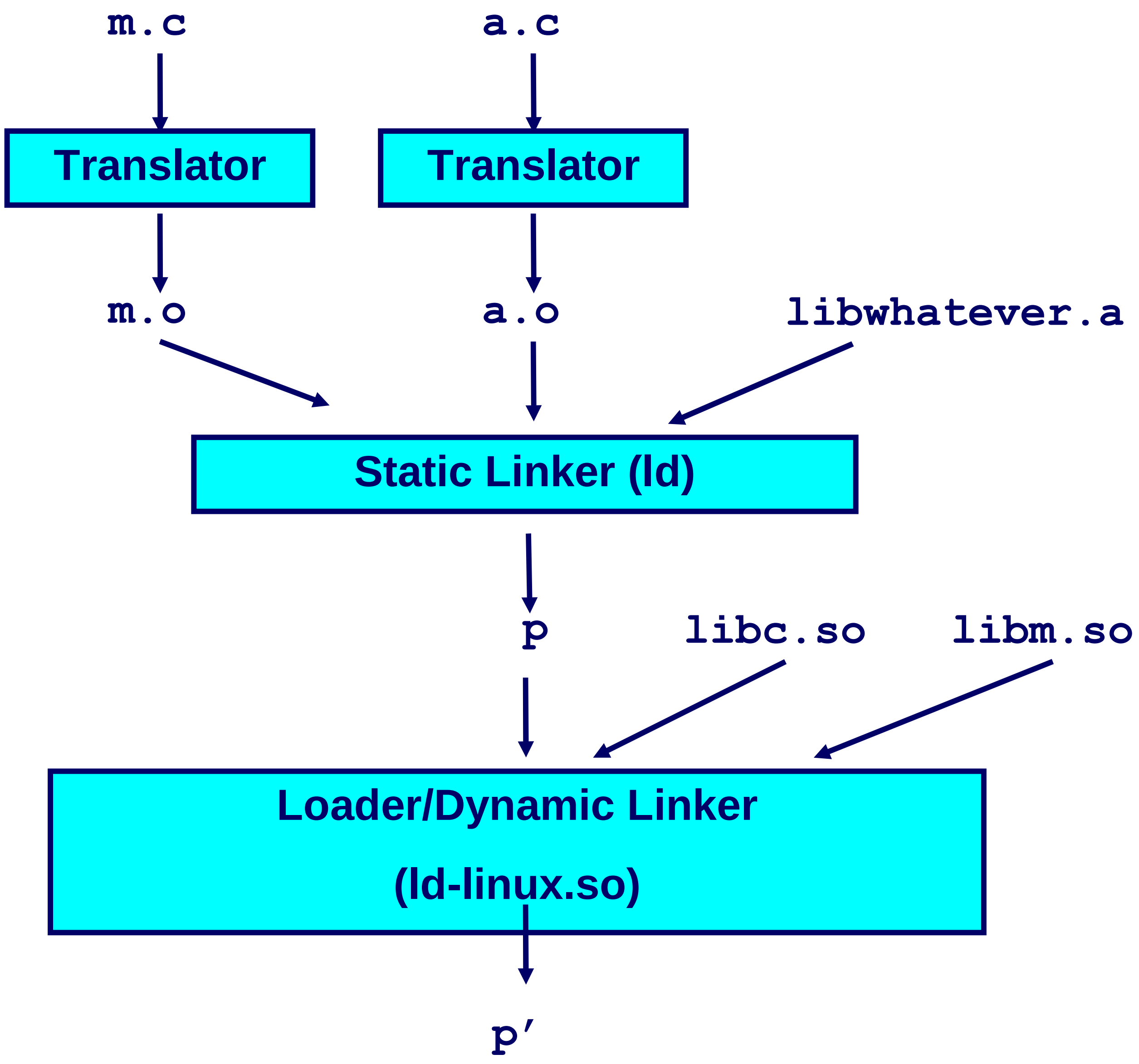
and `a.c` are loaded, linked, and (potentially) shared among processes.

Fully linked executable

p’ (in memory)

`P’`

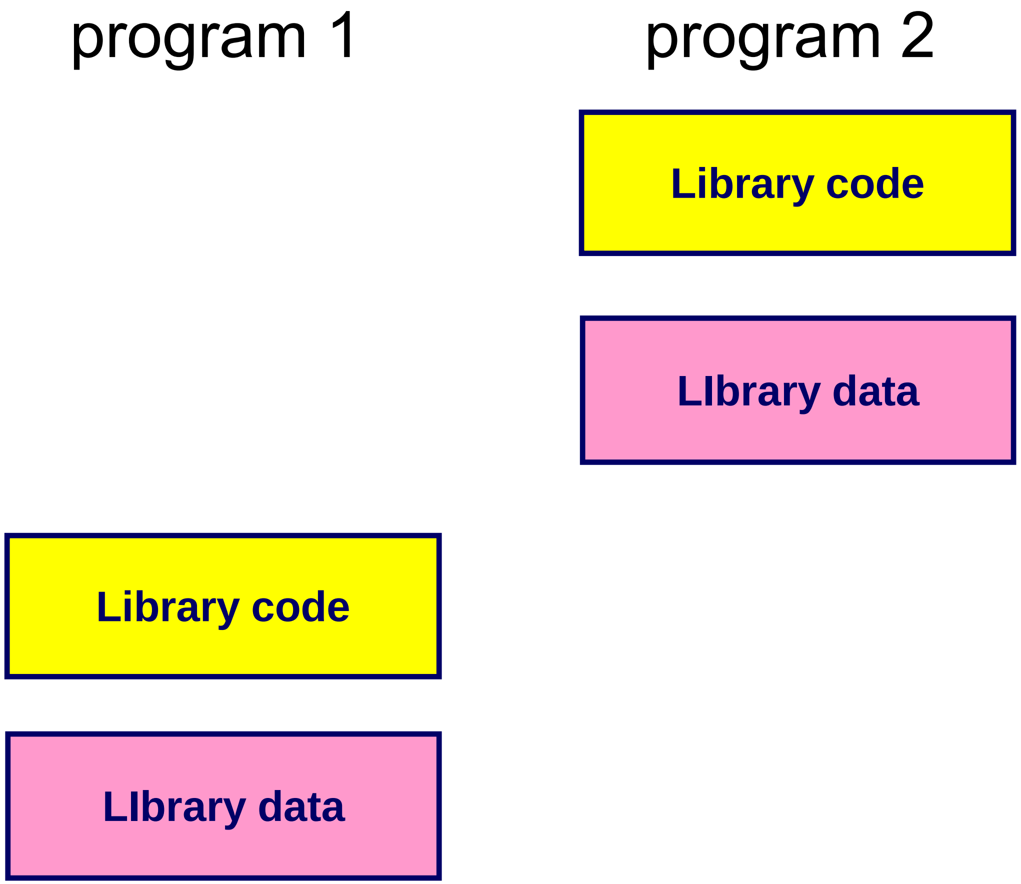
The Complete Picture



Position-Independent Code (PIC)

- If the load address for a program is not fixed (e.g., shared libraries), we use *position independent code*.
- 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.

  This may not be enough, e.g.: data references, instruction peculiarities (e.g., **call** *instruction in Intel x86) may not permit the use of PC-relative addressing.*



PIC (cont'd): ELF Files

- 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 a program can load its own address into a register, it can then use a fixed offset to access the GOT, and thence the data.

PIC code on ELF: cont'd

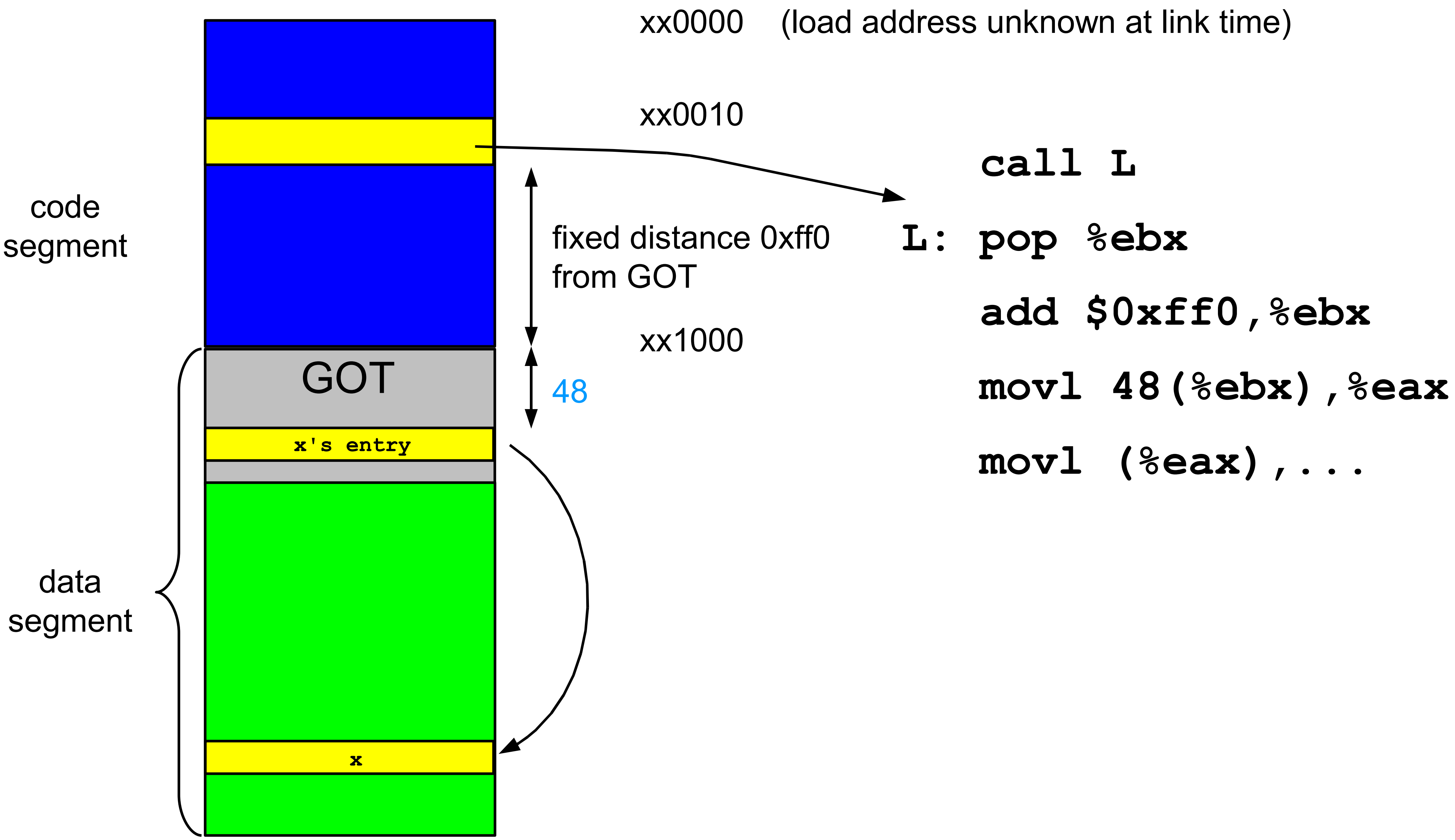
Code to figure out its own address (x86):

```
    call  L    /* push address of next instruction on stack */
 L: pop %ebx   /* pop address of this instruction into %ebx */
```

Accessing a global variable x in PIC:

1) GOT has an entry, say at position k, for x. The dynamic linker fills in the address of x into this entry at load time.
2) Compute "my address" into a register, say %ebx (above);
3) %ebx += offset_to_GOT; /* fixed for a given program */
4) %eax = contents of location k(%ebx) /* %eax = addr. of x */
5) access memory location pointed at by %eax;

PIC on ELF: Example



```
    call L
L:  pop %ebx
    add $0xff0,%ebx
    movl 48(%ebx),%eax
    movl (%eax),...
```

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 the memory pages of code, even if they don’t have the same address space allocated.

Disadvantages:

- GOT needs to be relocated at load time.
  In big libraries, GOT can be very large, so this may be slow.
- PIC code is bigger and slower than non-PIC code.
  The slowdown is architecture dependent (in an architecture with few registers, using one to hold GOT address can affect code quality significantly.)

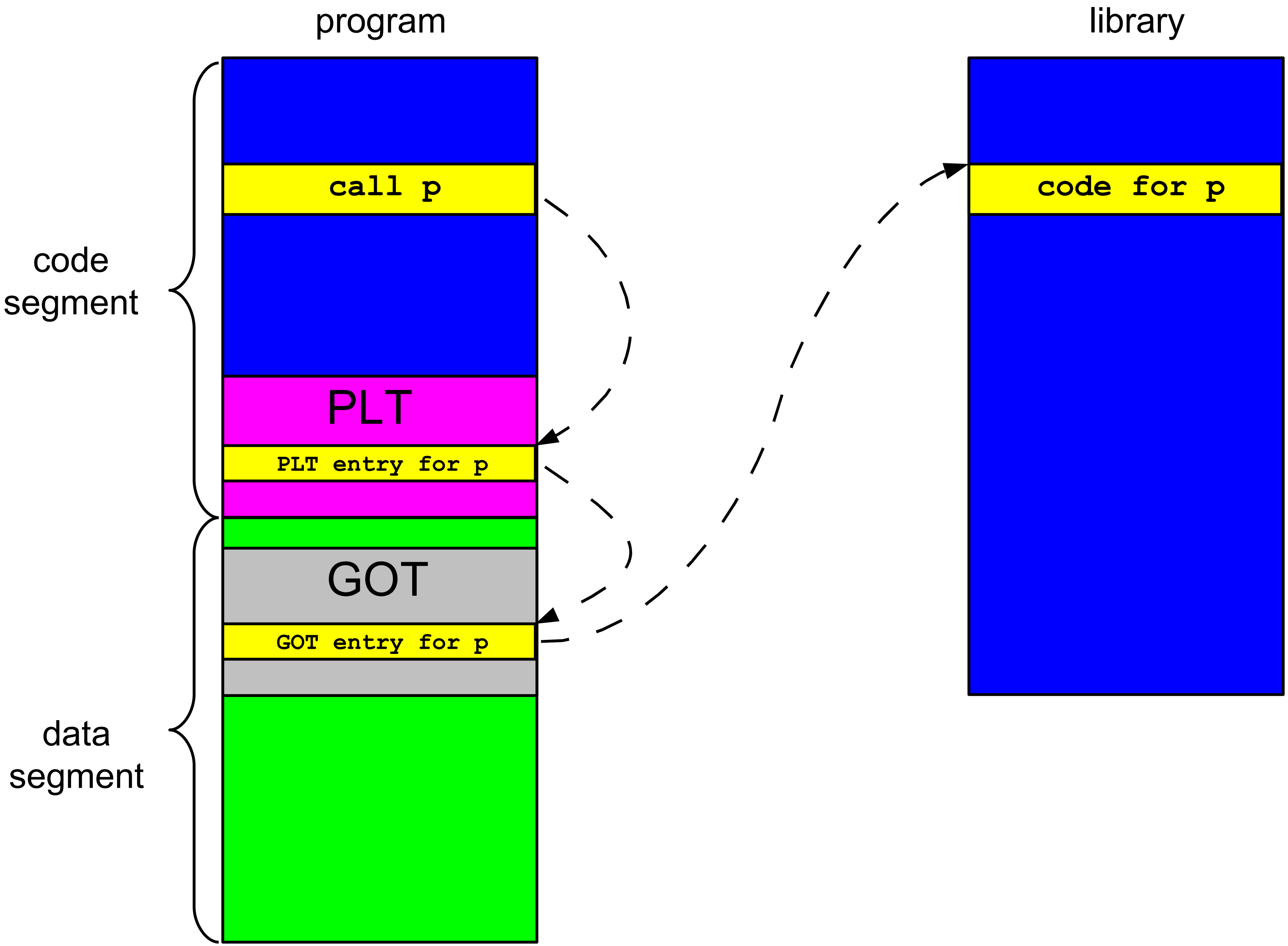
Dynamic Linking: Basic Mechanism

- A reference to a dynamically linked procedure p is mapped to code that invokes a *handler*.
- At runtime, when p is called, the handler gets executed:
  - The handler checks to see whether p has been loaded already (due to some other reference);
  - if so, the current reference is linked in, and execution continues normally.
  - otherwise, the code for p is loaded and linked in.

Dynamic Linking: ELF Files

- ELF shared libraries use PIC (position independent code), so text sections do not need relocation.
- Data references use a 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 (analogous to the GOT for data references).

ELF Dynamic Linking: PLT and GOT



ELF Dynamic Linking: Lazy Linkage

- 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.
- In PIC code on x86, %ebx must contain the address of GOT when PLT entry is called.