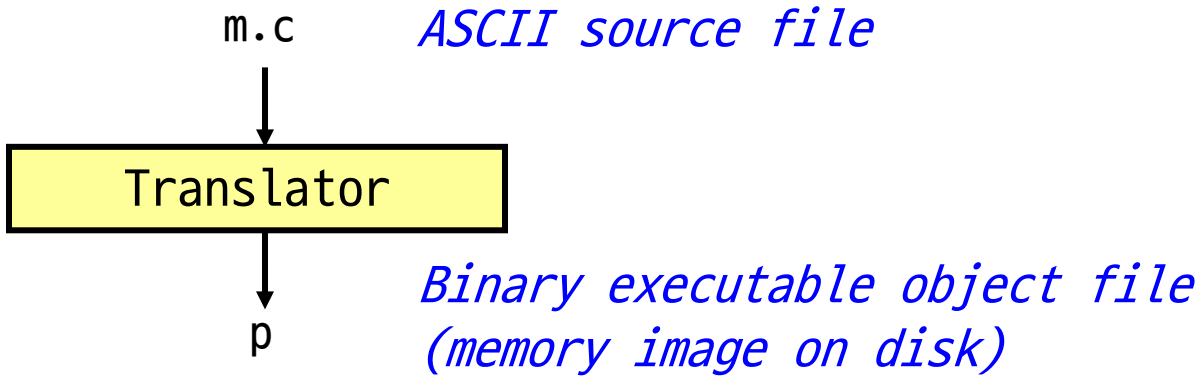
Linking

Jo, Heeseung

Program Translation (1)

A simplistic program translation scheme



Problems:

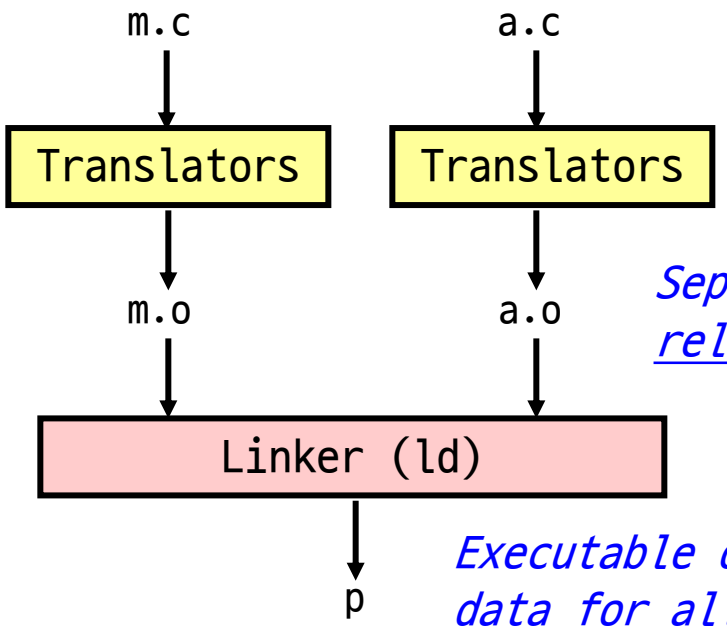
- Efficiency: small change requires complete recompilation
- Modularity: hard to share common functions (e.g. printf)

Solution:

- Static linker (or linker)

Program Translation (2)

A better scheme using a linker



Separately compiled relocatable object files

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

Program Translation (3)

Compiler driver coordinates all steps in the translation and linking process

- Typically included with each compilation system (gcc)
- Invokes preprocessor (cpp), compiler (cc1), assembler (as), and linker (ld)
- Passes command line arguments to appropriate phases
- Example: create executable p from main.c and swap.c

```
$ gcc -O2 -v -o p main.c swap.c
  cpp [args] main.c /tmp/main.i
  cc -S /tmp/main.i -O2 [args] -o /tmp/main.s
  as [args] -o /tmp/main.o /tmp/main.s
  <similar process for swap.c>
  ld -o p [system obj files] /tmp/main.o /tmp/swap.o
$
```

Linker (1)

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
    - e.g., math library, standard C library
- Efficiency in time
  - Change one source file, compile, and then relink
  - No need to recompile other source files
  - e.g., Linux kernel, MySQL, ...
- Efficiency in space
  - Libraries of common functions can be aggregated into a single file (storage space)
  - Yet executable files and running memory image contain only code for the functions they actually use (memory space)

Linker (2)

What does a linker do?

- Merges object files
  - Multiple relocatable (.o) object files -> 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
- Relocates symbols
  - Relocates symbols from their relative locations in the .o files to new absolute positions in the executable
  - Updates all references to these symbols to reflect their new positions
  - References can be in either code or data:
    - `code: func();  // reference to symbol func`
    - `data: int *xp = &x;  // reference to symbol x`

ELF (1)

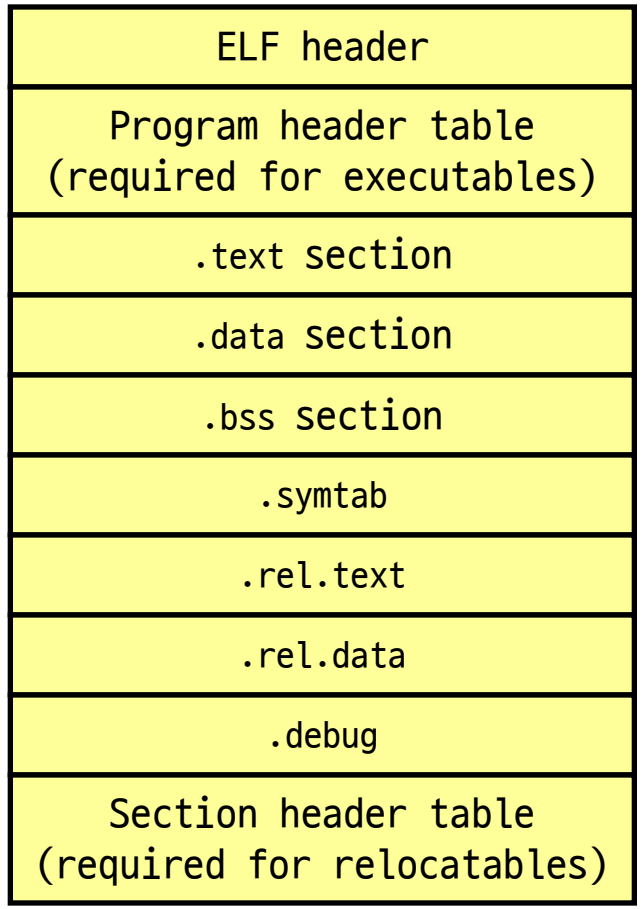
Executable and Linkable Format

- Standard library format for object files
- Derived 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 (2)

ELF object file format

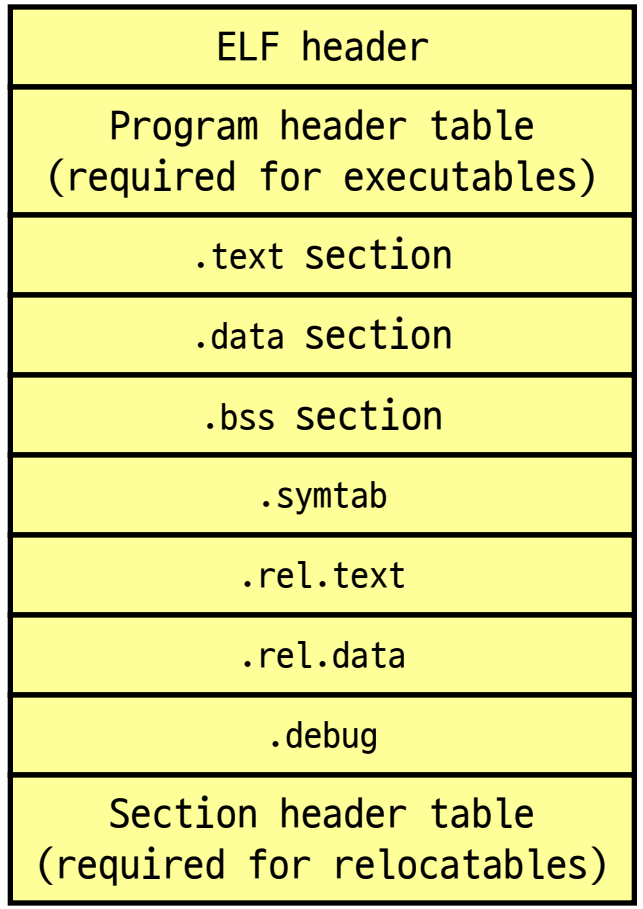
- ELF header
  - Magic number (\177ELF), type (.o, exec, .so), machine, byte ordering, etc.
- Program header table
  - Page size, virtual addresses memory segments (sections), segment sizes
- **.text** section
  - Code
- **.data** section
  - Initialized (static) data
- **.bss** section
  - Uninitialized (static) data
  - Better for space saving
  - Has section header but occupies no space



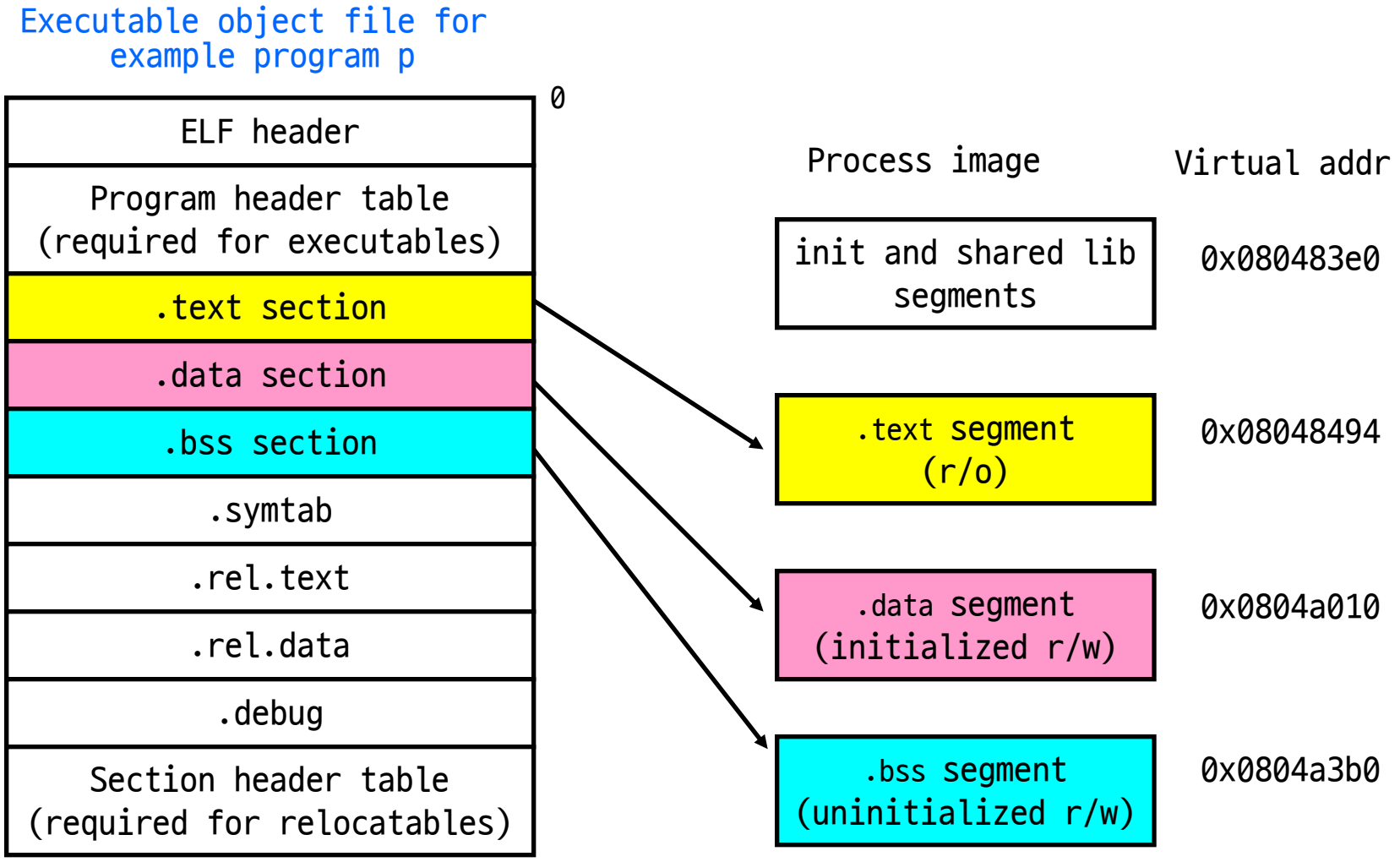
ELF (3)

ELF object file format (cont'd)

- **.symtab** section
  - Symbol table
  - Procedures 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)



Loading Executable Binaries



Linking Example (1)

Example C program

m.c

```
int e=7;

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

a.c

```
extern int e;

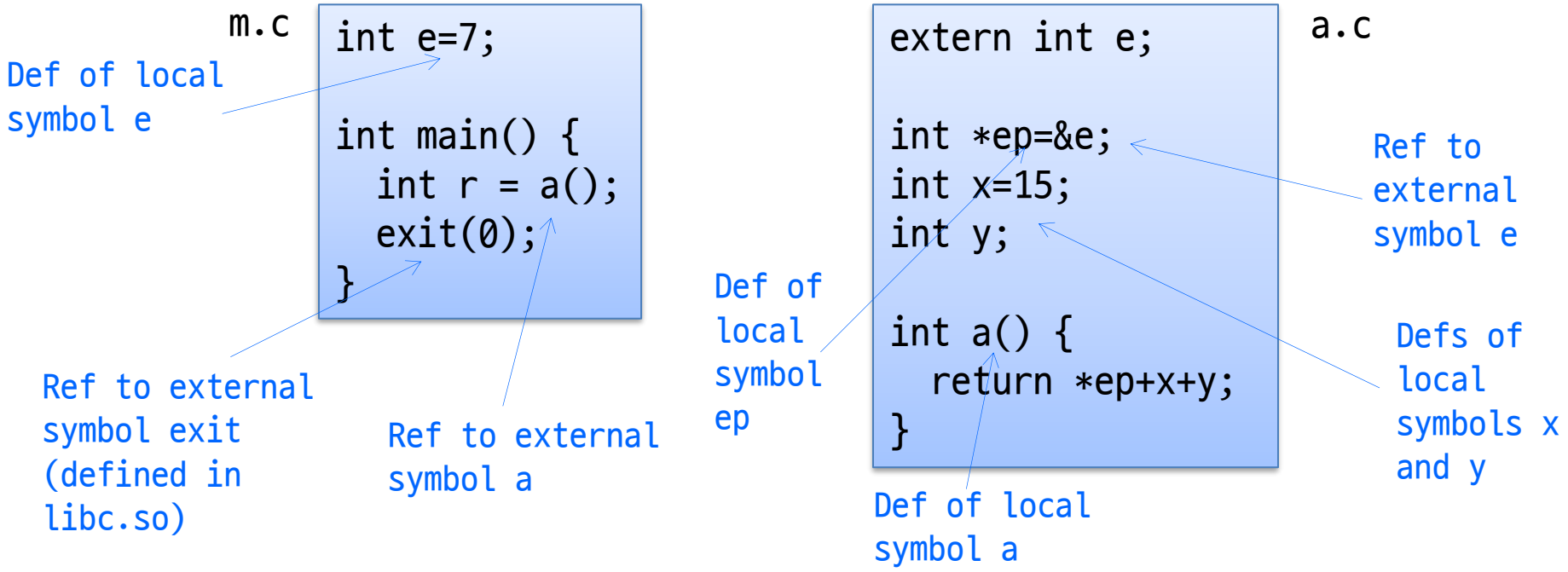
int *ep=&e;
int x=15;
int y;

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

Linking Example (2)

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



Linking Example (3)

m.o object file

m.c

```
int e=7;

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

objdump -D m.o

```
Disassembly of section .text:

00000000 <main>:
   0:   55                      pushl  %ebp
   1:   89 e5                   movl   %esp,%ebp
   3:   e8 00 00 00 00          call   4 <main+0x4>
   8:   6a 00                   pushl  $0x0
   a:   e8 00 00 00 00          call   b <main+0xb>
   f:   90                      nop

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

Relocation section [.rel.text]:
00000004   R_386_PC32   a
0000000b   R_386_PC32   exit
```

Linking Example (4)

a.o object file

a.c

```
extern int e;

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

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

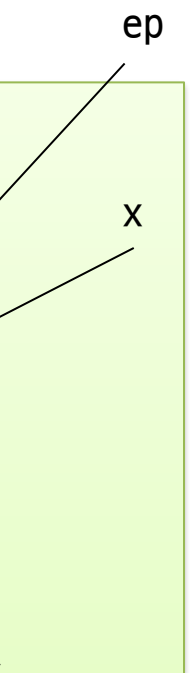
objdump -D a.o

```
Disassembly of section .text:

00000000 <a>:
 0:    55                   pushl  %ebp
 1:    8b 15 00 00 00 00    movl   0x0,%edx
 7:    a1 00 00 00 00       movl   0x0,%eax
 c:    89 e5                movl   %esp,%ebp
 e:    03 02                addl   (%edx),%eax
 10:   89 ec                movl   %ebp,%esp
 12:   03 05 00 00 00 00    addl   0x0,%eax
 18:   5d                   popl   %ebp
 19:   c3                   ret

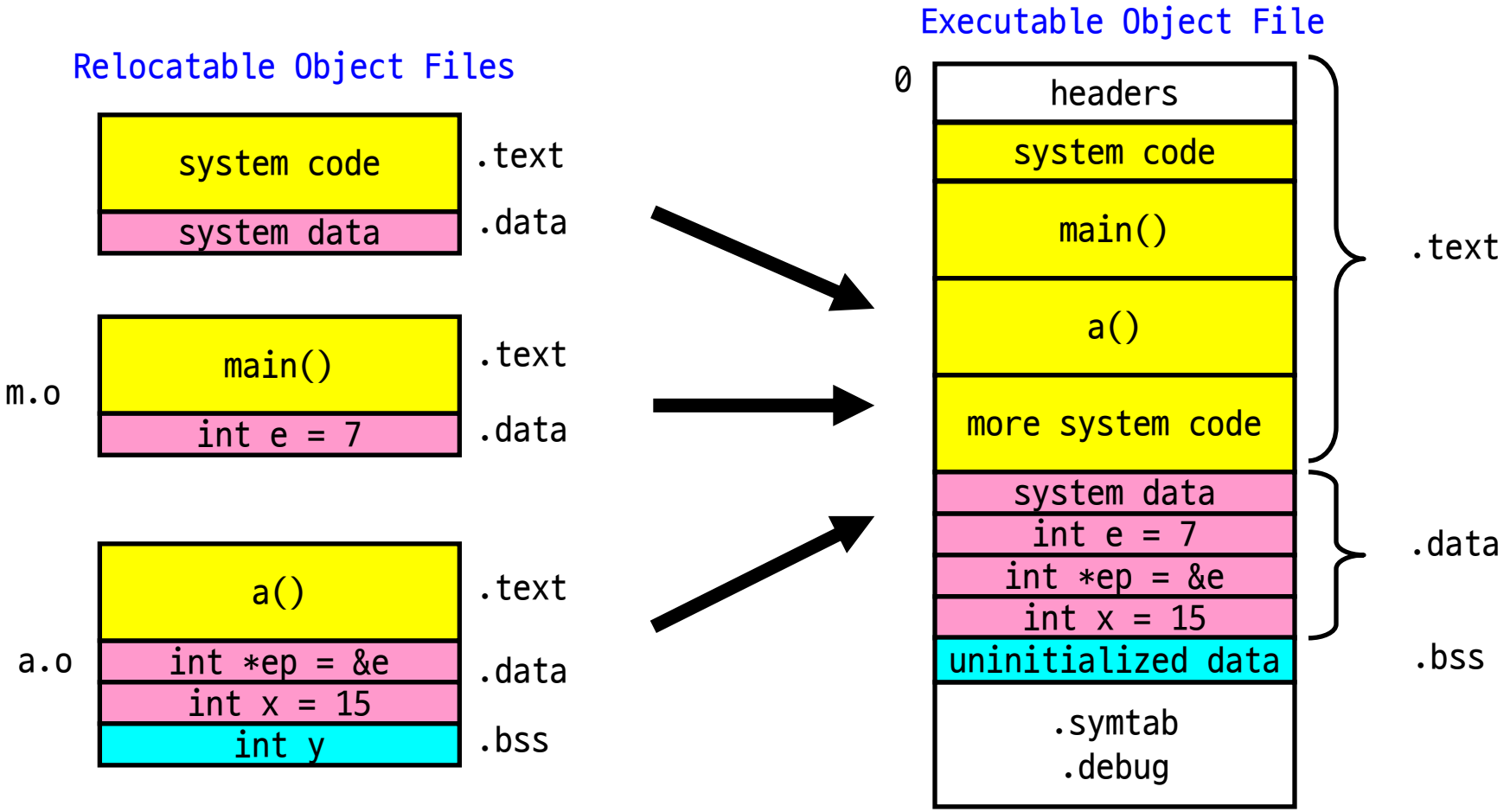
Disassembly of section .data:
00000000 <ep>:
 0: 00 00 00 00
00000004 <x>:
 4: 0f 00 00 00

Relocation section [.rel.data]:
00000000   R_386_32      e
```



Linking Example (6)

Merging relocatable object files into an executable object file



Linking Example (7)

After relocation and external reference resolution

```
08048530 <main>:
 8048530:       55                      pushl  %ebp
 8048531:       89 e5                   movl   %esp,%ebp
 8048533:       e8 08 00 00 00          call   8048540 <a>
 8048538:       6a 00                   pushl  $0x0
 804853a:       e8 35 ff ff ff          call   8048474 <_init+0x94>
 804853f:       90                      nop

08048540 <a>:
 8048540:       55                      pushl  %ebp
 8048541:       8b 15 1c a0 04 08       movl   0x804a01c,%edx
 8048547:       a1 20 a0 04 08          movl   0x804a020,%eax
 804854c:       89 e5                   movl   %esp,%ebp
 804854e:       03 02                   addl   (%edx),%eax
 8048550:       89 ec                   movl   %ebp,%esp
 8048552:       03 05 d0 a3 04 08       addl   0x804a3d0,%eax
 8048558:       5d                      popl   %ebp
 8048559:       c3                      ret
```

ep

x

15

y

return value

```
Disassembly of section .data:

0804a018 <e>:
 804a018:       07 00 00 00

0804a01c <ep>:
 804a01c:       18 a0 04 08

0804a020 <x>:
 804a020:       0f 00 00 00

Disassembly of section .bss:

0804a3d0 <y>:
 804a3d0:       00 00 00 00  ??
```

m.c

```
int e=7;

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

a.c

```
extern int e;

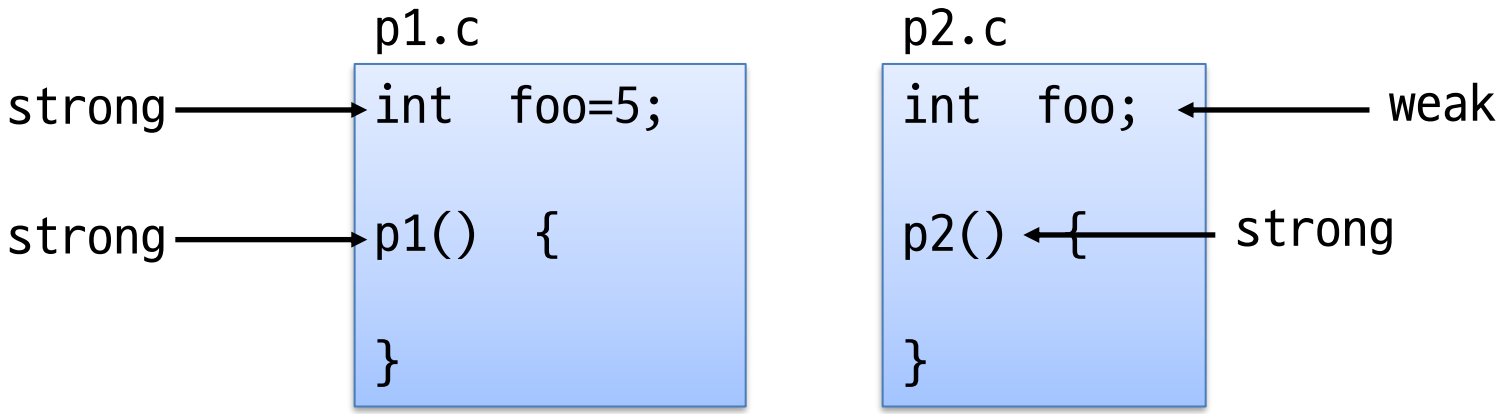
int *ep=&e;
int x=15;
int y;

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

Symbol Resolution (1)

Program symbols are either strong or weak

- Strong symbols: procedures and initialized globals
- Weak symbols: uninitialized globals



Symbol Resolution (2)

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

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

Symbol Resolution (3)

Examples

`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

Static Libraries (1)

Packaging commonly used functions

- How to package functions commonly used by programmers?
  - Math, I/O, memory management, string manipulation, etc.

- 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

Static Libraries (2)

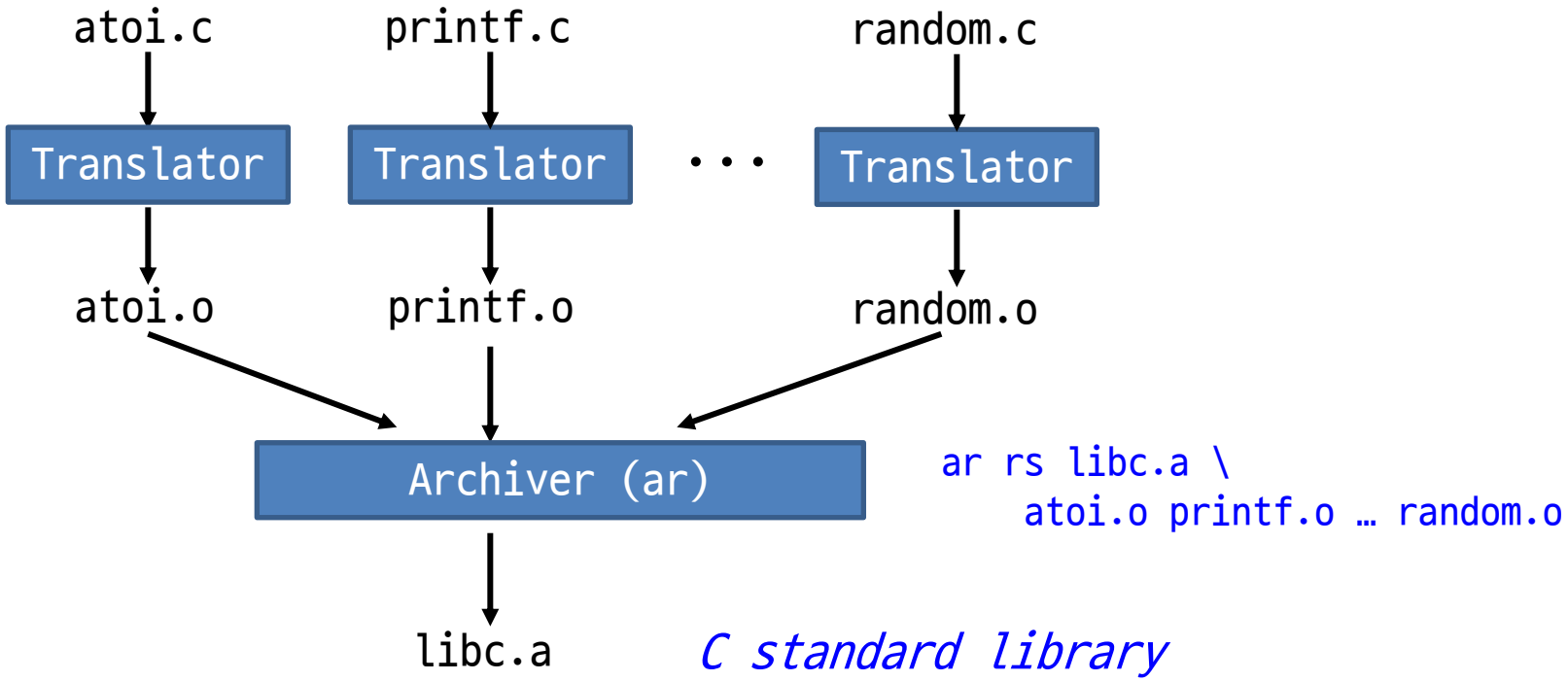
Solution: Static libraries (.a archive files)

- Concatenate related relocatable object files into a single file
  - With an index (called an archive)
- Linker 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
- Further improves modularity and efficiency by packaging commonly used functions
  - e.g. C standard library (**libc**), math library (**libm**), etc.

Static Libraries (3)

Creating static libraries

- Archiver (ar) allows incremental updates:
  - Recompile function that changes and replace .o file in archive



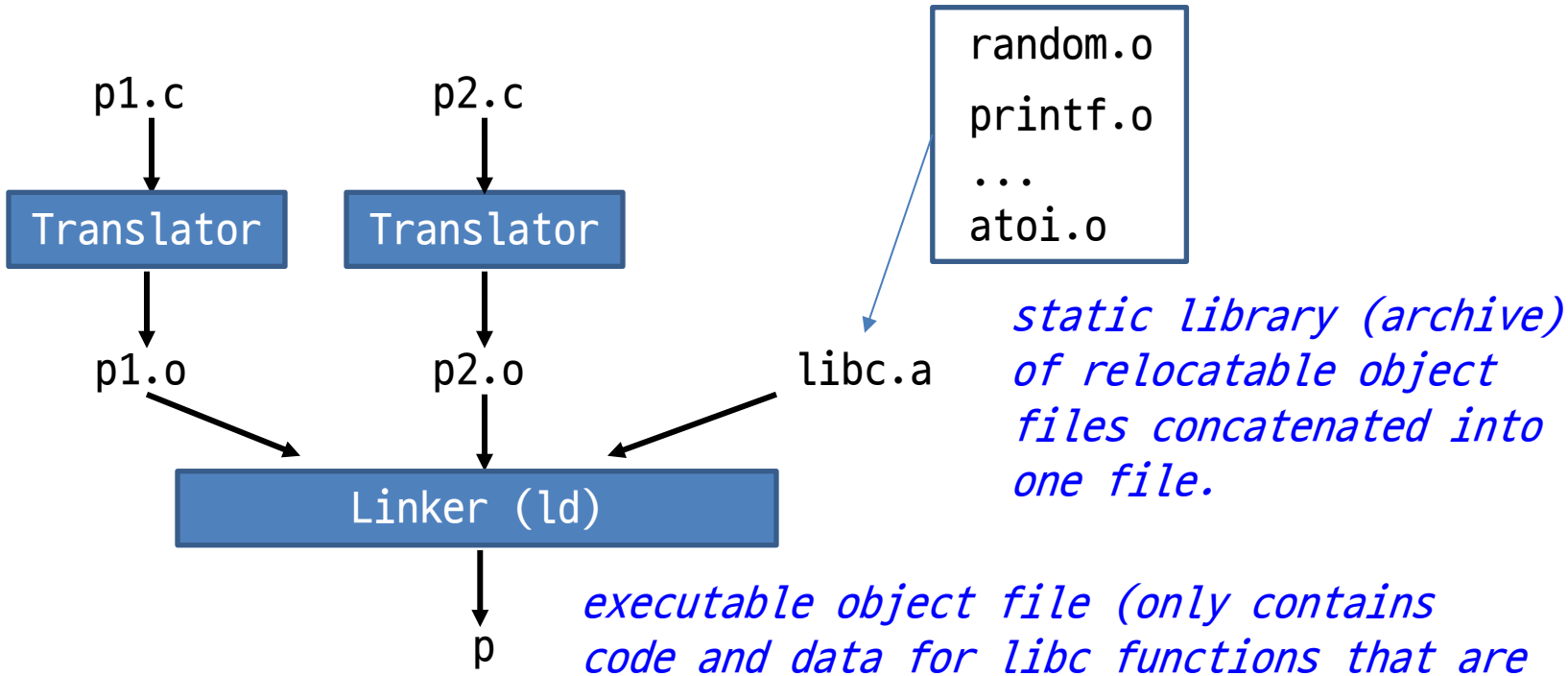

```
ar rs libc.a \
    atoi.o printf.o … random.o
```

C standard library

Static Libraries (4)

Mechanism

- Link selectively only the .o files in the archive that are actually needed by the program



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)

Static Libraries (5)

Commonly used libraries

- libc.a (C standard library)
  - 8MB archive of 900 object files
  - I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math, etc.
- libm.a (C math library)
  - 1MB archive of 226 object files
  - Floating point math (sin, cos, tan, log, exp, sqrt, etc.)

```
% ar -t /usr/lib/libc.a ¦ sort
…
fork.o
fprintf.o
fputc.o
freopen.o
…
```

```
% ar -t /usr/lib/libm.a ¦ sort
…
e_acos.o
e_acosf.o
e_acosh.o
e_acoshf.o
…
```

Static Libraries (6)

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 unresolved references
  - As each new .o or .a file is encountered, try to resolve each unresolved reference in the list against the symbols in the object file
  - 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. -lmine main.o
main.o: In function `main':
main.o(.text+0x4): undefined reference to `func1'


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

```
main.c

int main()
{
  ...
  func1();
  ...
}
```

```
libmine.c

void func1()
{
  ....
  ....
}
```



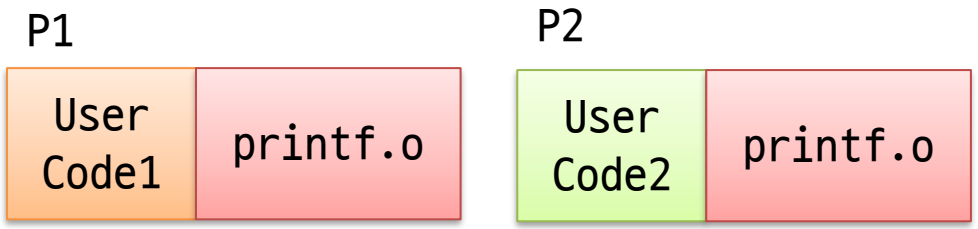
```
libmine.a

....
func1.o
....
```

Shared Libraries (1)

Static libraries have the following disadvantages:

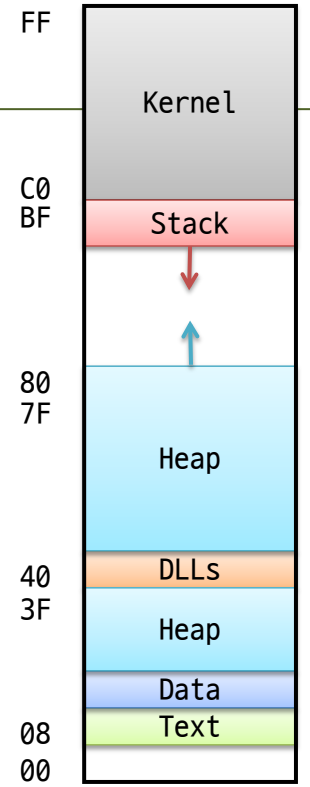
- Potential for duplicating lots of common code in the executable files on a filesystem
  - e.g., every C program needs the standard C library
- Potential for duplicating lots of code in the virtual memory space of many processes
- Minor bug fixes of system libraries require each application to explicitly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

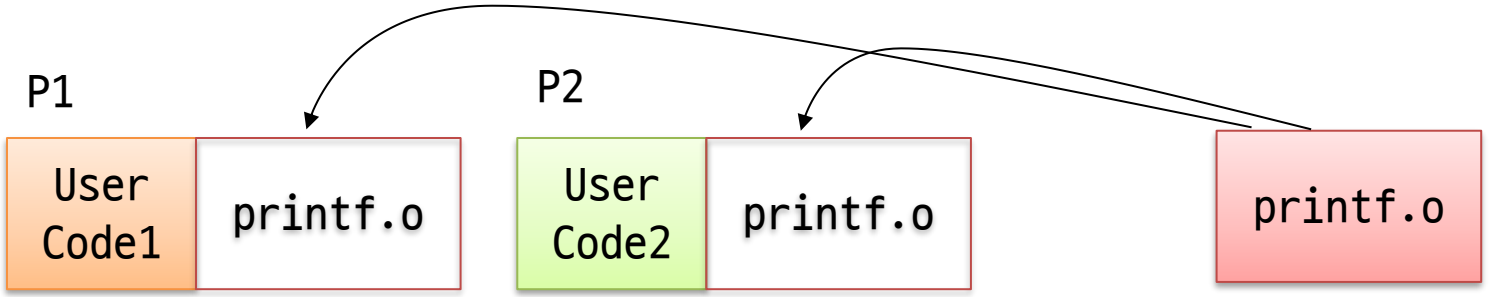


Shared Libraries (2)

Solution: shared libraries

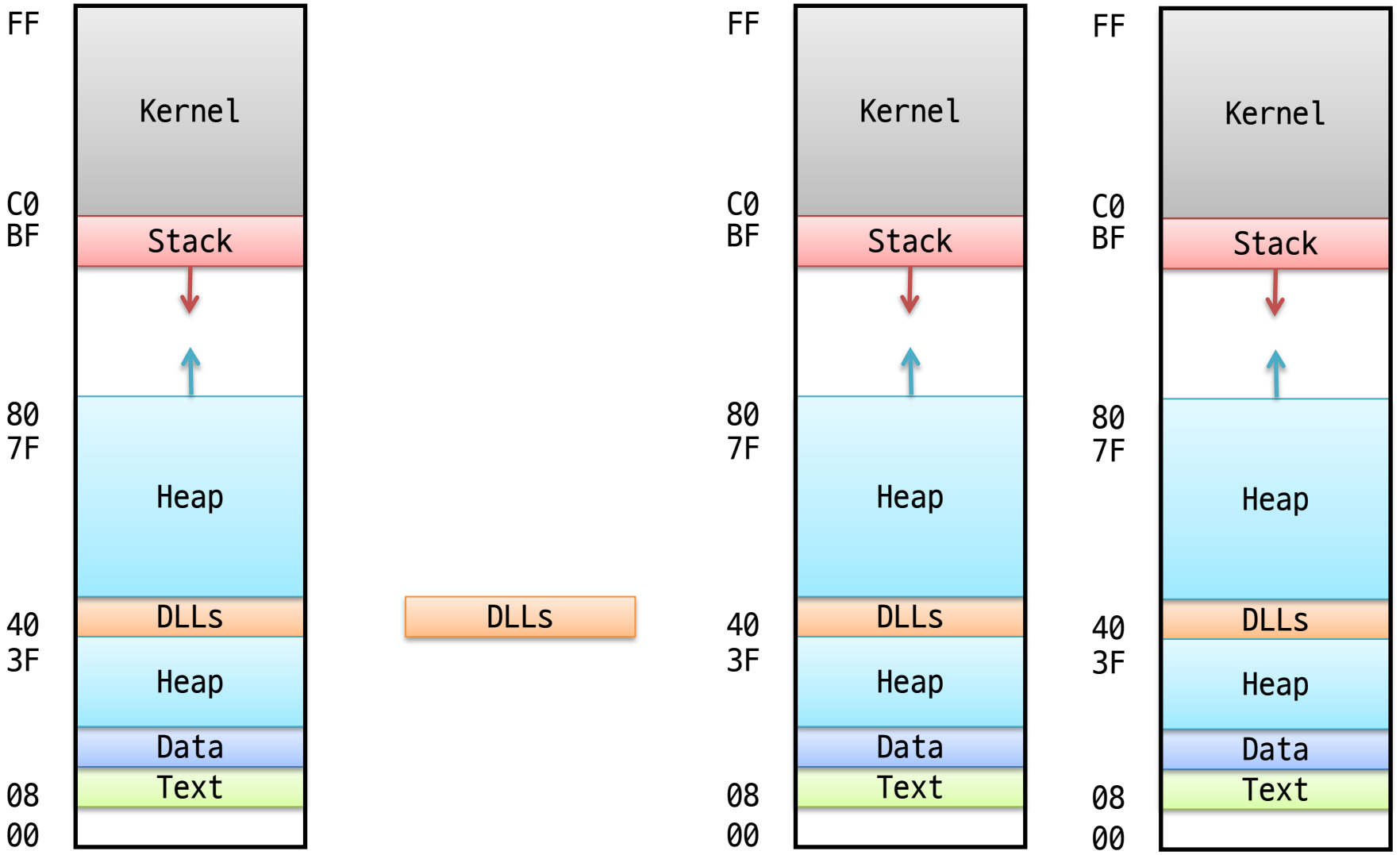
- Dynamic link libraries or DLLs
- Members are dynamically loaded into memory and linked into an application at run-time
- Dynamic linking can occur ...
  - When executable is first loaded and run
  - Common case for Linux, handled automatically by **ld-linux.so**
  - Also after program has begun
    - In Linux, this is done explicitly by user with **dlopen()**
- Shared library routines can be shared by multiple processes



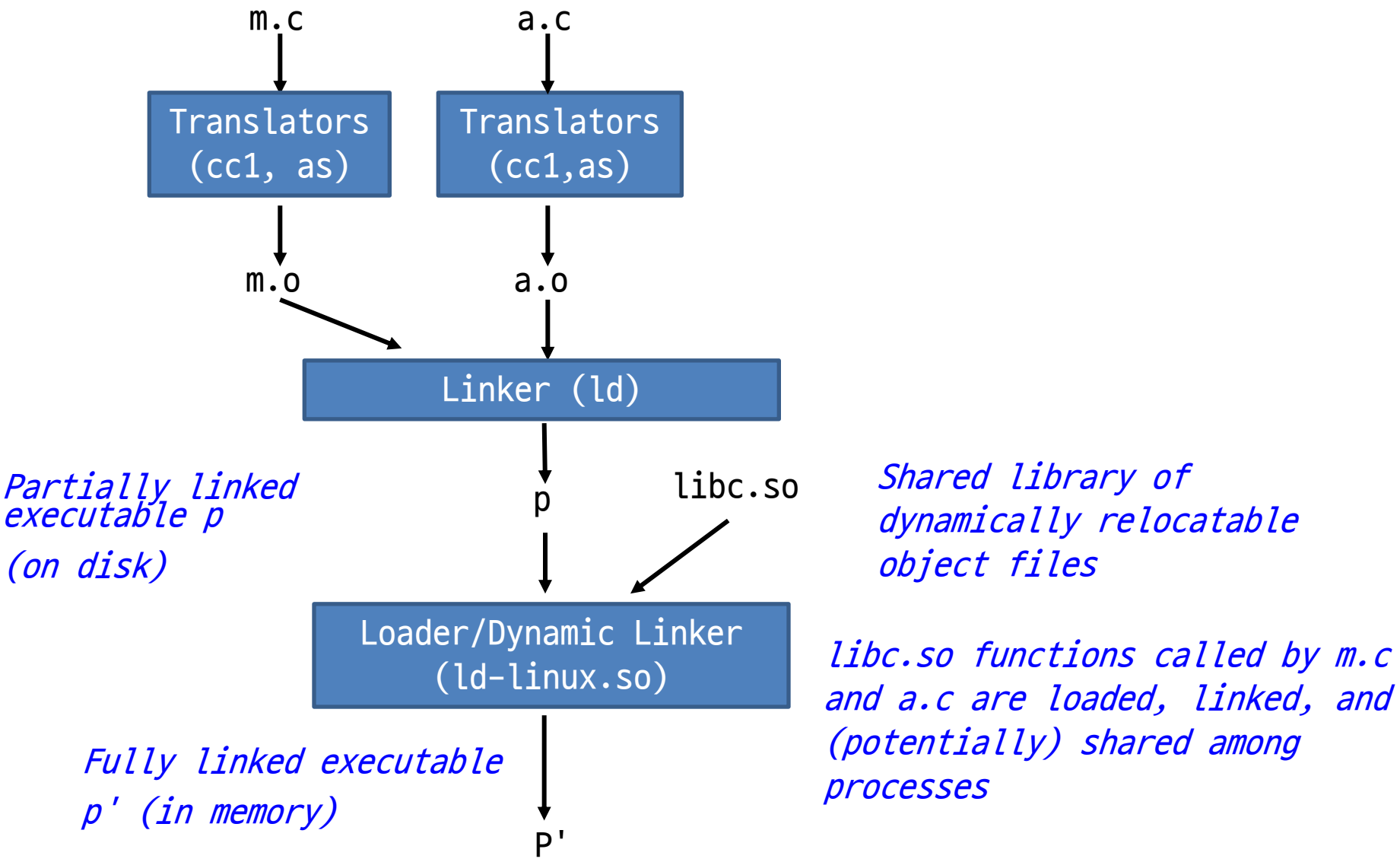


Shared Libraries (3)

Physical memory area of DLLs is shared



Shared Libraries (4)



The Complete Picture

