

15-213

Linking

October 5, 2002

Topics

- static linking
- object files
- static libraries
- loading
- dynamic linking of shared libraries
- Reading: Chapter 7
- Problems: 7.8 and 7.12

Linker puzzles

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

```
p1() {}
```

link time error: two strong symbols (p1)

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

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

both instances of x refer to the same uninitialized int.

```
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() {}
```

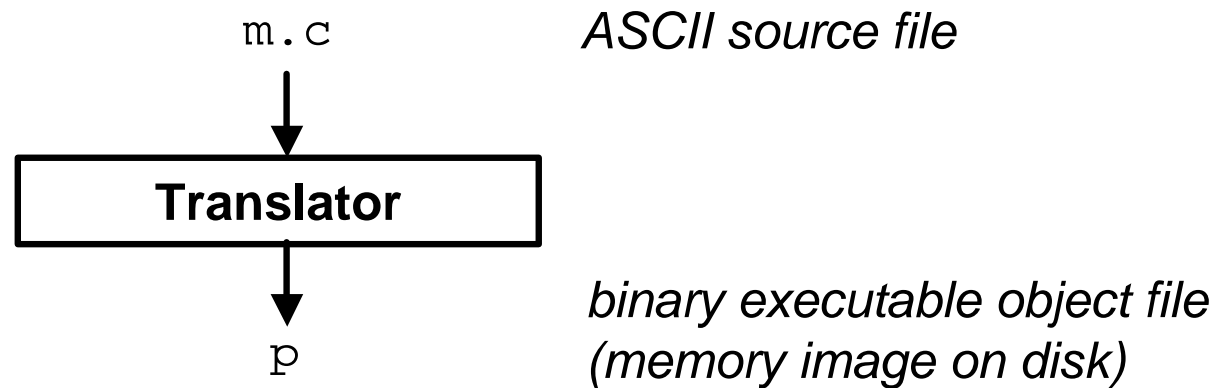
writes to x in p2 will overwrite something!
Nasty!

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

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

references to x refer to the same initialized variable.

A simplistic program translation scheme



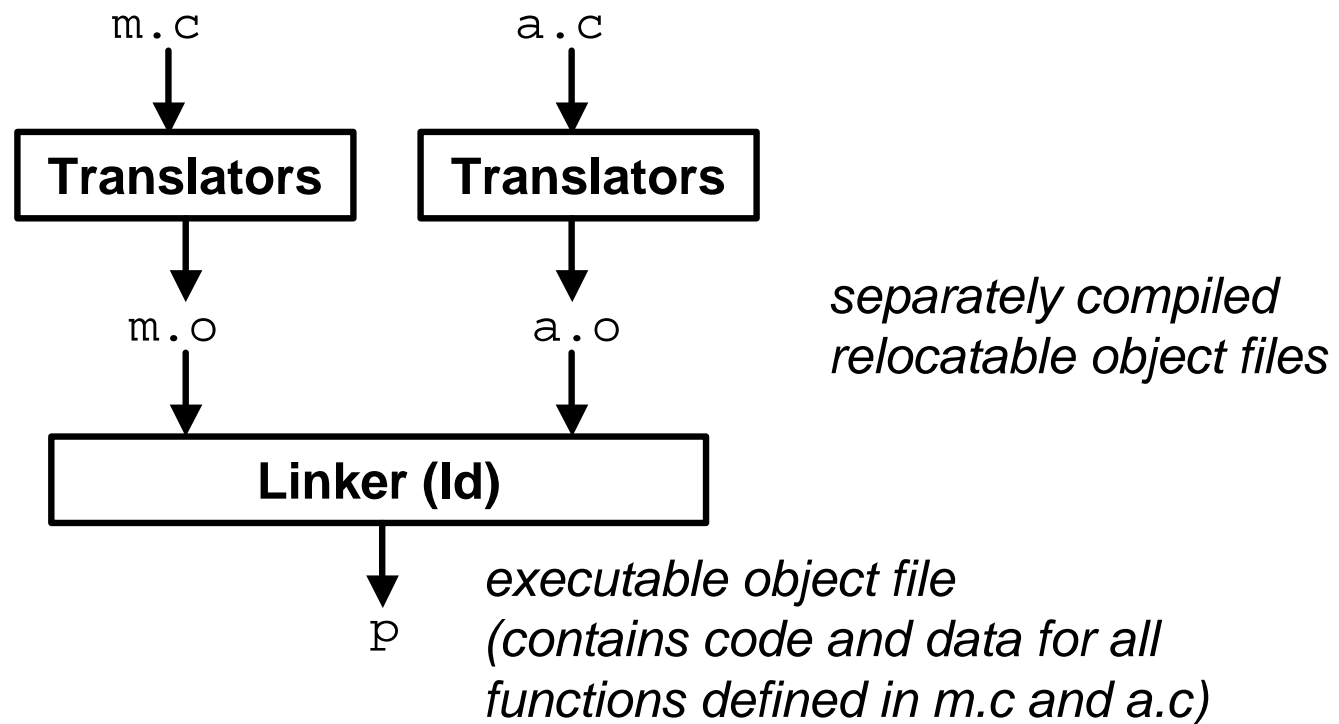
Problems:

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

Solution:

- *static linker (or linker)*

Linkers



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 (cpp), compiler (cc1), assembler (as), and linker (ld).
- Passes command line args to appropriate phases

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

```
bass> gcc -O2 -v -o p m.c a.c
cpp [args] m.c /tmp/cca07630.i
cc1 /tmp/cca07630.i 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
bass>
```

What does a linker do?

Merges object files

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

Resolves external references

- as part of the merging process, 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:** `a();` `/* ref to symbol a */`
 - » **data:** `int *xp=&x;` `/* ref to symbol x */`
 - because of this modifying, linking is sometimes called *link editing*.

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.

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, and 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 for 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

0	ELF header
	Program header table (required for executables)
	.text section
	.data section
	.bss section
	.symtab
	.rel.txt
	.rel.data
	.debug
	Section header table (required for relocatables)

ELF object file format

.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)

ELF header	0
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);
}
```

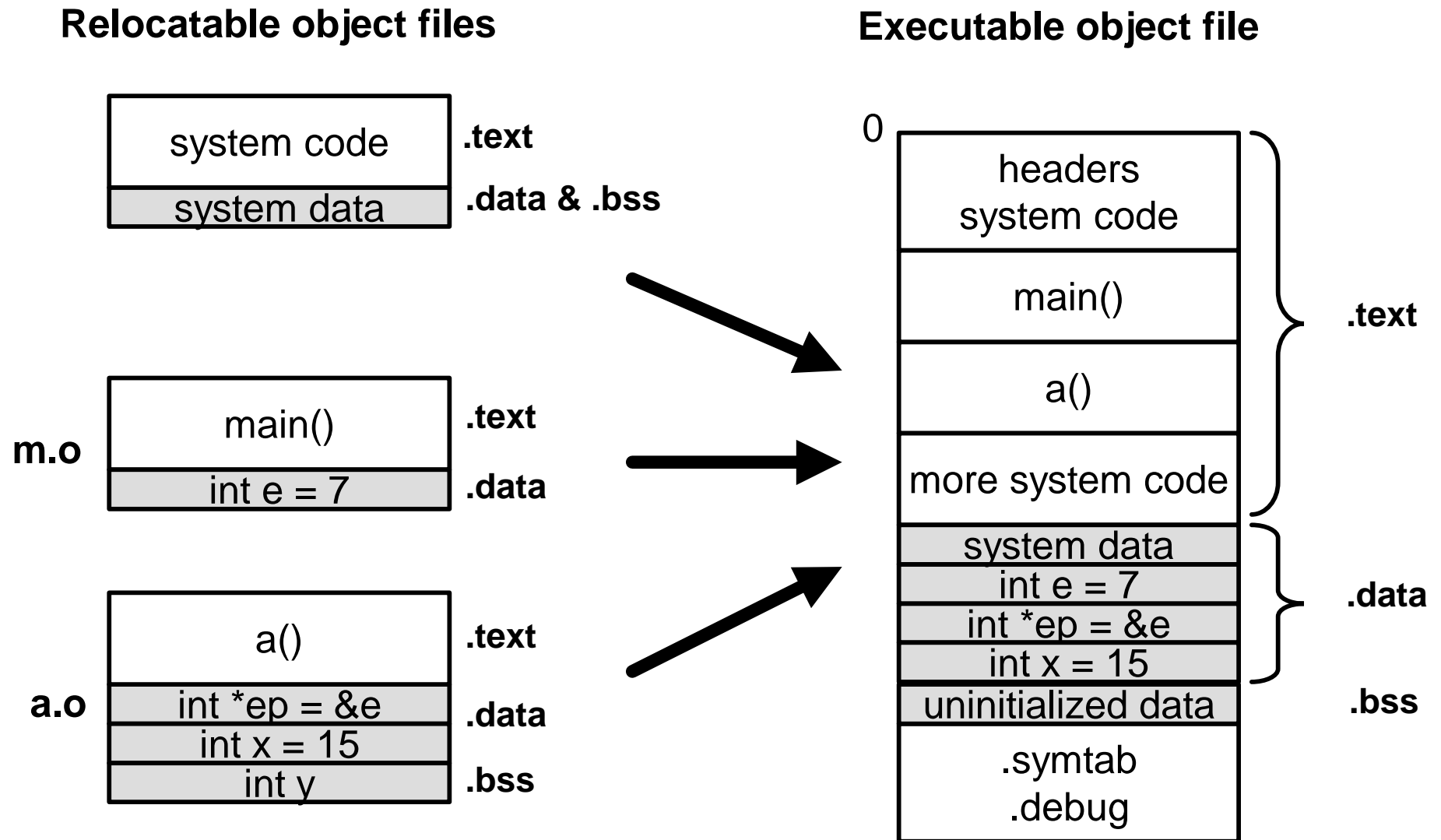
a.c

```
extern int e;

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

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

Merging .o files into an executable



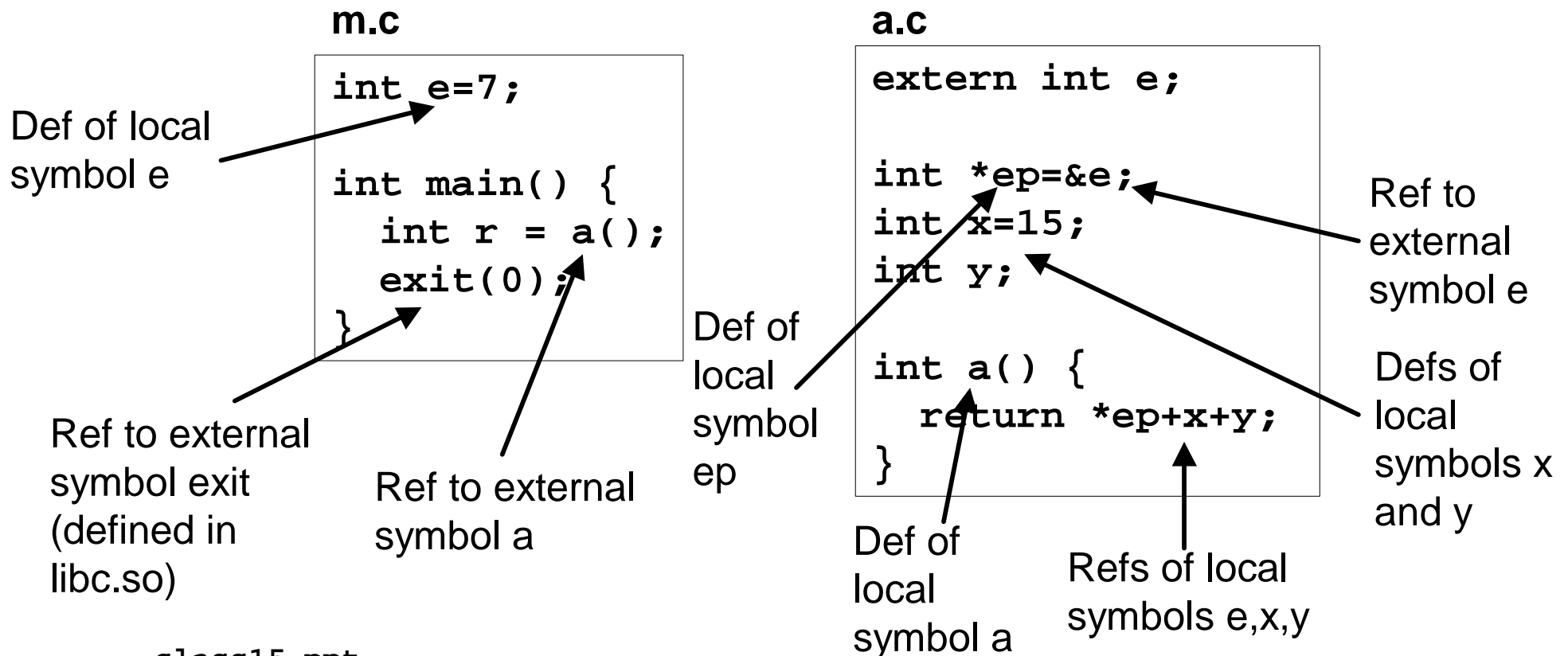
Relocating symbols and resolving external references

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.o relocation info

m.c

```
int e=7;

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

Disassembly of section .text:

```
00000000 <main>: 00000000 <main>:
    0:   55                      pushl   %ebp
    1:   89 e5                   movl    %esp,%ebp
    3:   e8 fc ff ff ff       call    4 <main+0x4>
                                4: R_386_PC32    a
    8:   6a 00                   pushl   $0x0
    a:   e8 fc ff ff ff       call    b <main+0xb>
                                b: R_386_PC32    exit
    f:   90                      nop
```

Disassembly of section .data:

```
00000000 <e>:
    0:   07 00 00 00
```

source: objdump

class15.ppt

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	pushl	%ebp
1:	8b 15 00 00 00	movl	0x0,%edx
6:	00		
3:	R_386_32	ep	
7:	a1 00 00 00 00	movl	0x0,%eax
8:	R_386_32	x	
c:	89 e5	movl	%esp,%ebp
e:	03 02	addl	(%edx),%eax
10:	89 ec	movl	%ebp,%esp
12:	03 05 00 00 00	addl	0x0,%eax
17:	00		
14:	R_386_32	y	
18:	5d	popl	%ebp
19:	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)

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	movl	0x804a01c,%edx
8048546:	08		
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	addl	0x804a3d0,%eax
8048557:	08		
8048558:	5d	popl	%ebp
8048559:	c3	ret	

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;
}
```

Disassembly of section .data:

```
0804a010 <__data_start>:
804a010:      00 00 00 00

0804a014 <p.2>:
804a014:      f8 a2 04 08

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

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

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

Strong and weak symbols

Program symbols are either *strong* or *weak*

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

p1.c:

```
strong → int foo=5;  
strong → p1() {  
        }  
        }
```

p2.c:

```
int foo; ← weak  
p2() ← { strong  
    }
```

Linker's symbol rules

1. A strong symbol can only appear once.
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.
3. If multiple weak symbols, the linker can pick either one.

Linker puzzles

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

```
p1() {}
```

link time error: two strong symbols (p1)

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

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

both instances of x refer to the same uninitialized int.

```
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 refer to the same initialized variable.

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

Packaging commonly used functions

How to package functions commonly used by programmers?

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

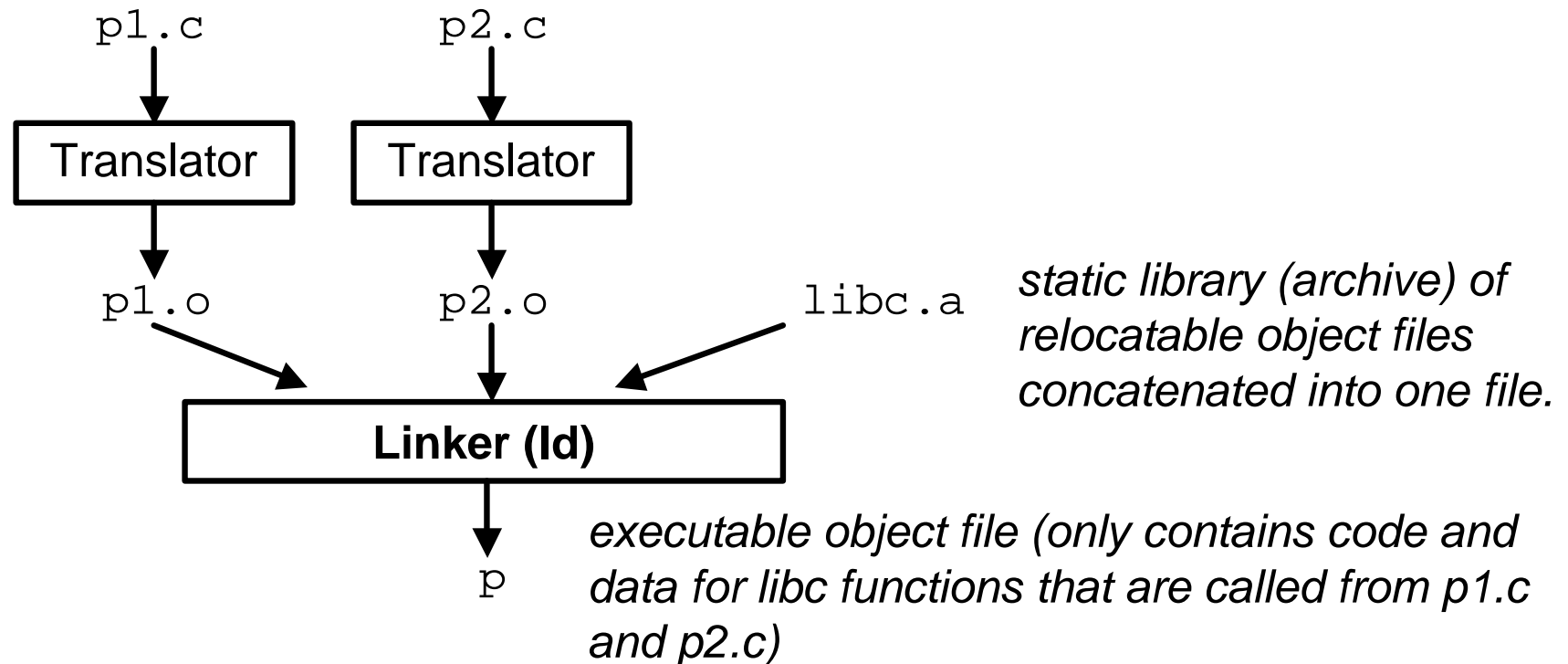
Awkward, given the linker framework so far:

- **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.

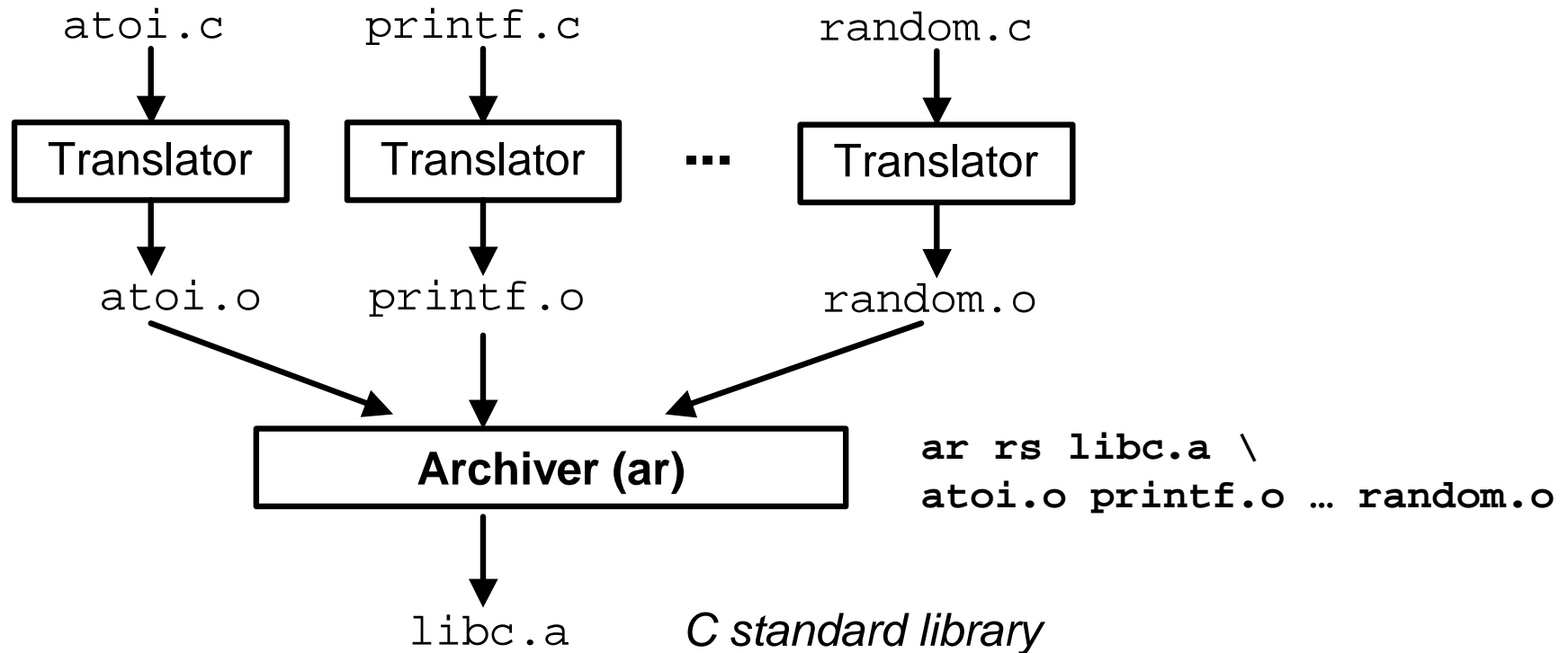
Static libraries (archives)



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

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

Creating static libraries



Archiver allows incremental updates:

- recompile function that changes and replace .o file in archive.

Commonly used libraries

libc.a (the C standard library)

- 8 MB archive of 900 object files.
- I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math

libm.a (the C math library)

- 1 MB archive of 226 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:

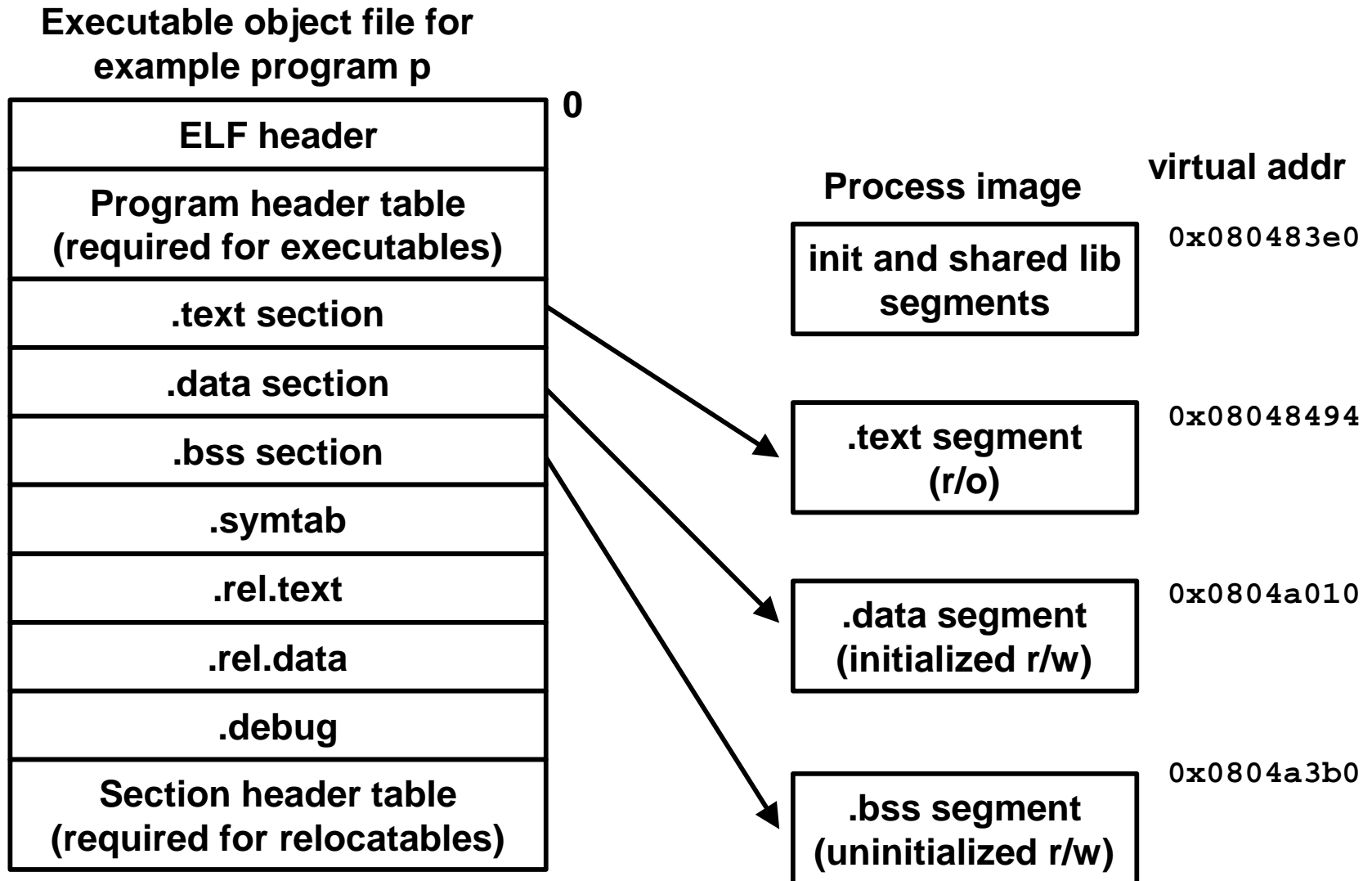
- Scan .o files and .a files in the command line order.
- During the scan, keep a list of the current unresolved references.
- As each new .o or .a file *obj* is encountered, try to resolve each unresolved reference in the list against the symbols in *obj*.
- If any entries in the unresolved list at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

Loading executable binaries



Shared libraries

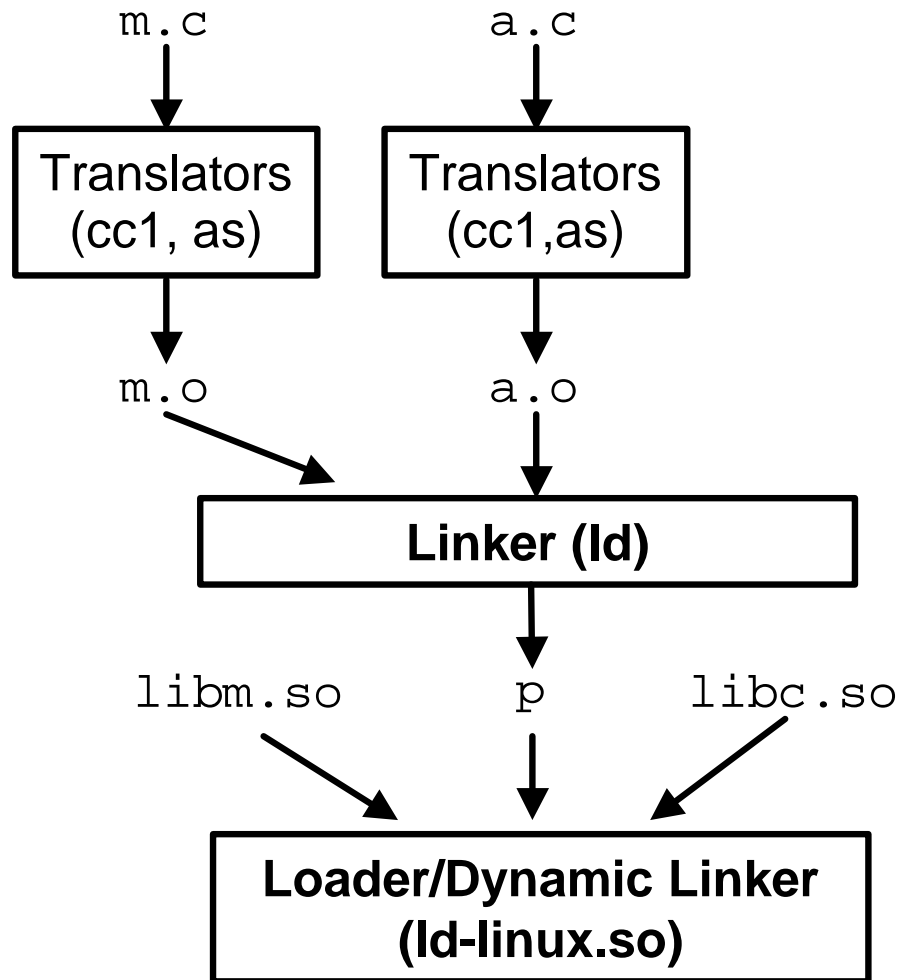
Static libraries have the following disadvantages:

- **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().
 - shared library routines can be shared by multiple processes.

Dynamically linked shared libraries



shared libraries of dynamically relocatable object files

libc.so functions called by m.c and a.c are loaded, linked, and (potentially) shared among processes.

Stubs for Dynamic Library Calls

- A *stub* replaces the actual procedure call within the code. Below, for example, is the stub for `malloc()`.

```
8048552: e8 9d fe ff ff call malloc@@SYSVABI_1.3
```

- This stub doesn't actually call a function. Instead, it makes a call into the *Procedure Linkage Table (PLT)*.
- This table contains one entry for each dynamically linked function. If the function is present, this code will invoke it. If not, it will invoke the dynamic linker.
- The dynamic linker, in turn, will map the dynamically linked function and invoke it. The code will remain mapped for the next call.

The Role of the PLT and GOT

- Each entry in the PLT makes an indirect jump through the *Global Object Table (GOT)*.
- Initially, the GOT, which is nothing more than an array of addresses, is initialized so that each functions entry is the next instruction in the PLT. In other words, the jump amounts to a “no op”.
- The rest of the code of the PLT invokes the dynamic linker. Once the dynamic linker runs, it changes the GOT entry so that it is the address of the newly mapped dynamically linked function.
- After that point, the first jump in the PLT will invoke the function instead of the linker.

An Example PLT Entry

- The code below is one entry from the PLT. Recall that there is one entry per function. Notice the first line -- 0x8049600 is the address of the corresponding GOT entry.
- The subsequent pushl and jmp invoke the dynamic linker. The push is pushing a parameter identifying the index of the functions relocation entry.
- Notice the jmp is relative to the current PC. This is because the first two entries in the PLT are used for the dynamic linker, itself.

```
80483f4: ff 25 00 96 04 08 jmp *0x8049600
80483fa: 68 20 00 00 00 pushl $0x20
80483ff: e9 a0 ff ff ff jmp .-0x5b [0x80483a4]
```


An Example GOT Entry

- This is an example of an entry from the GOT. Notice that there isn't much to it.
- The GOT is a simple array of pointers into the text area. Each one is either the address of the dynamic function, or the address of the next part of the PLT.
- The table, itself, is nothing more than an array of these.

```
0x8049600 <_GLOBAL_OFFSET_TABLE_+16>: 0x80483fa
```

- Below is the same entry, after relocation. Notice how the value changed.

```
0x8049600 <_GLOBAL_OFFSET_TABLE_+16>: 0x40068ae
```

- Recall from our earlier discussions that 0x80xxxxx addresses are in the static text area, whereas 0x40xxxxx addresses are in the DLL area.

The complete picture

