15-410

"Nobody else reads these quotes anyway…"

Linking
February 13, 2004

Dave Eckhardt

Bruce Maggs

Some slides taken from 15-213 S'03 (Goldstein, Maggs).

Original slides authored by Randy Bryant and Dave O'Hallaron.

Synchronization

Upcoming events

- 2/18 (tonight) – Project 2 due
- 2/20 (Friday) – Project 3 out
- 2/25 (Wednesday) – Homework 1 due
- 2/25 (Wednesday) – Midterm exam (evening)
- March 3 – Project 3 checkpoint 1
- March 5 – Mid-semester/spring break

Spring break

- We do *not* plan for you to work on Project 3
- It can be an excellent time for some "light reading"

Pop Quiz

Q1. What does this program do?

```
[bmm@bmm bmm]$ cat pf.c

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

Q2. What does the Unix “ld” program do?

Pop Quiz

Q1. What does this program do?

```
[bmm@bmm bmm]$ cat pf.c

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

[bmm@bmm bmm]$ gcc pf.c -o pf
[bmm@bmm bmm]$ pf
134513416
```

Outline

What *is* printf()?

Where addresses come from

Executable files vs. Memory Images

- Conversion by “program loader”
- You will write one for exec() in Project 3

Object file linking (answer to Q2)

- Loader bugs make programs execute *half*-right
- You will need to characterize what's broken
    1. (*Not*: “every time I call printf() I get a triple fault”)
- You will need to how the parts *should* fit together

Where do addresses come from?

Program linking, program loading

- ... means getting bits in memory at the right addresses

Who *uses* those addresses?

- (Where did that “wild access” come from?)

Code addresses: program counter (%cs:%eip)

- Straight-line code
- Loops, conditionals
- Procedure calls

Stack area: stack pointer (%ss:%esp, %ss:%ebp)

Data regions (data/bss/heap)

- Most pointers in general purpose registers (%ds:%ebx)

How are they initialized?

Program counter

- Set to “entry point” by OS program loader

Stack pointer

- Set to “top of stack” by OS program loader

Registers

- How does my code know the address of `thread_table[]`?
- Some pointers are stored in the instruction stream

```
for (tp = thread_table,
  tp < &thread_table[n_threads], ++tp)
```

- Some pointers are stored in the data segment

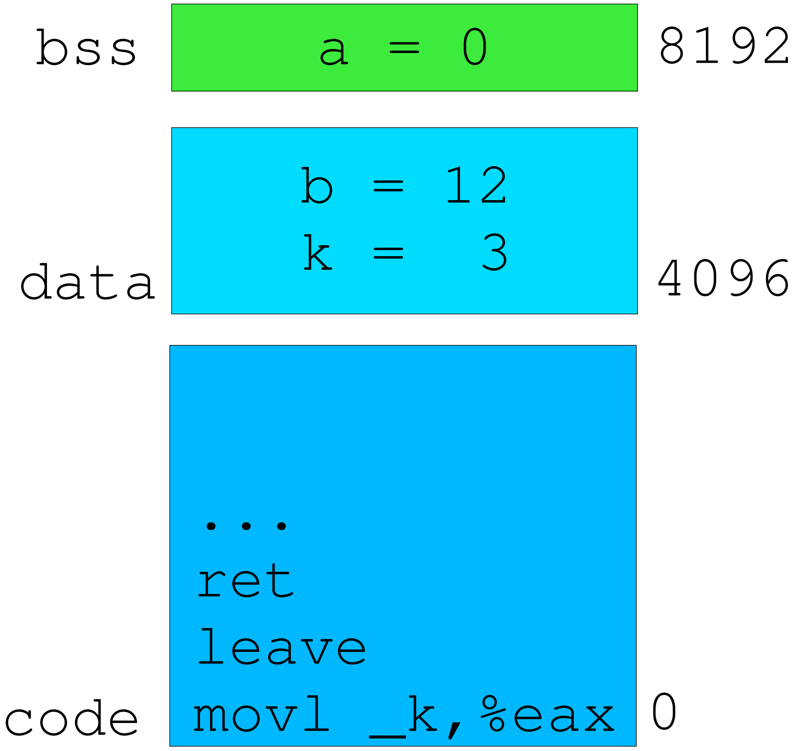
```
struct thread *thr_base = &thread_table[0];
```

- How do these all point to the right places?

Where does an int live?

```
int k = 3;
int foo(void) {
  return (k);
}

int a = 0;
int b = 12;
int bar (void) {
  return (a + b);
}
```



Loader: Image File ⇒ Memory Image

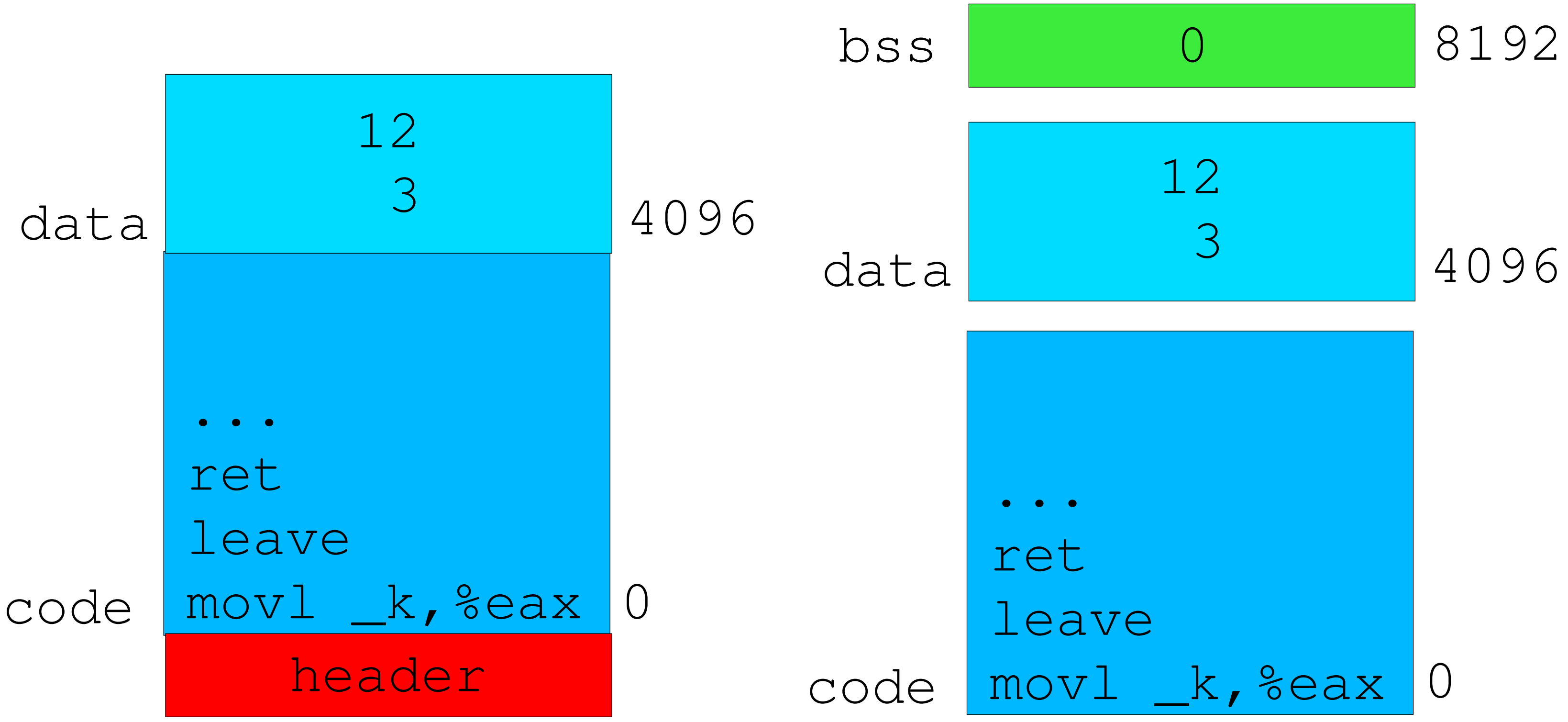


Image file has header (tells loader what to do)
Memory image has bss segment!

Programs are Multi-part

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

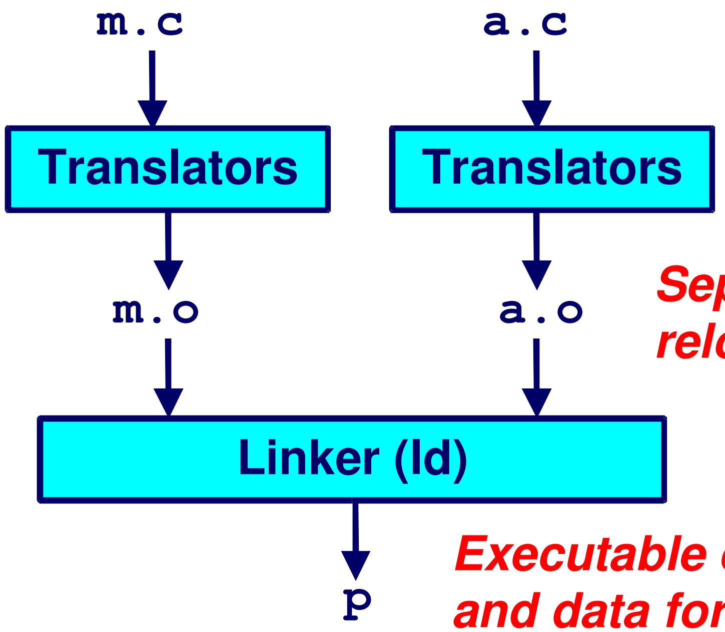
Efficiency (time)

- Change one source file, compile, and then relink.
- No need to recompile other source files.

"Link editor" combines objects into one image file

- Unix "link editor" called "ld"

Combining Objects: Link Editor



Separately compiled relocatable object files

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

Linker Todo List

Merge object files

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

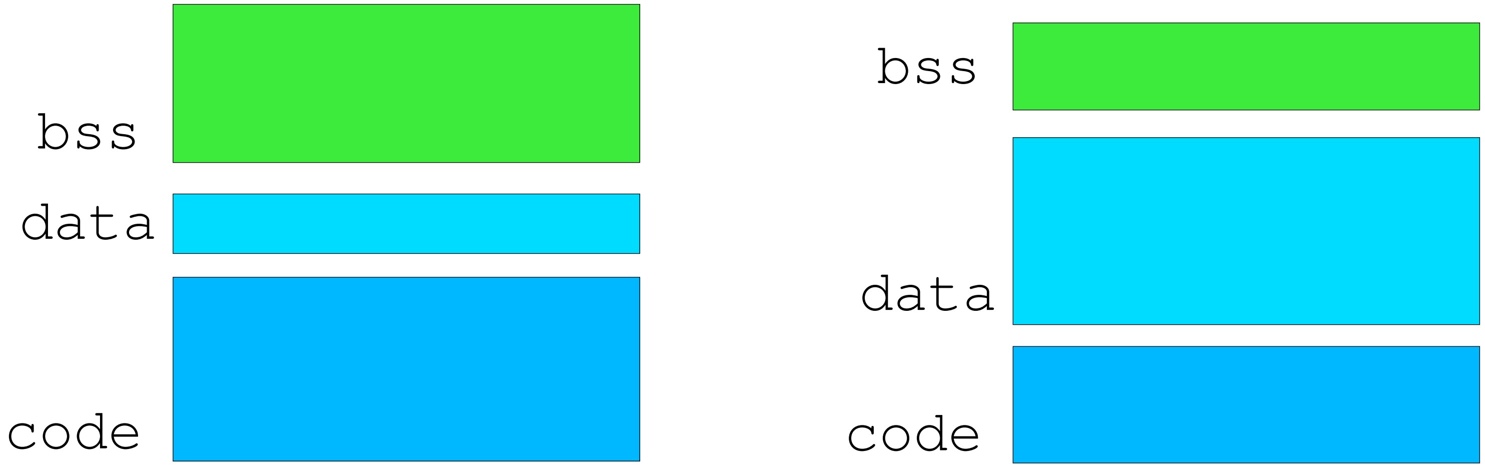
Resolve external references

- As part of the merging process, resolves external references.
  - *External reference*: reference to a symbol defined in another object file.

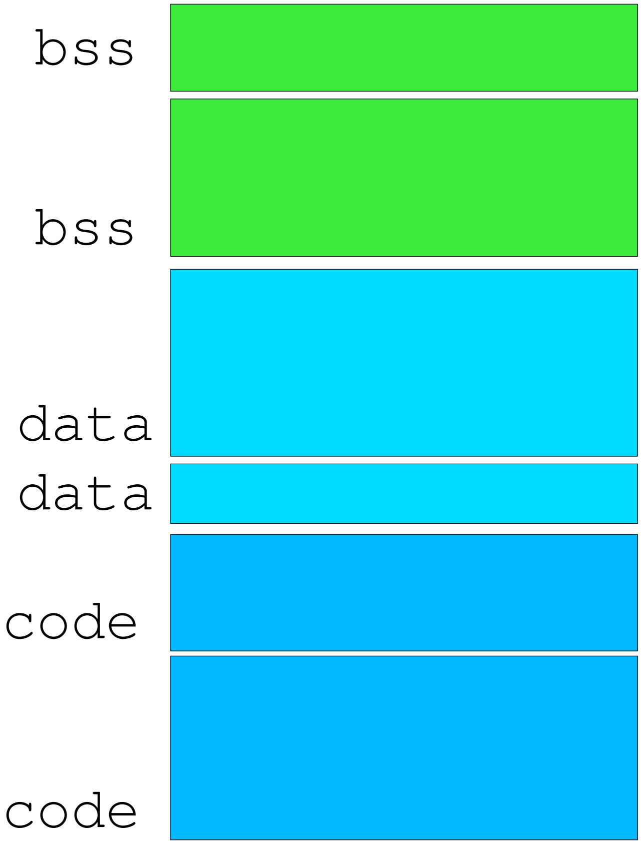
Relocate symbols

- 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.
- What does this mean??

Every .o uses same address space



Combining .o's Changes Addresses



Linker uses *relocation information*

Field

- address, bit field size

Field type

- relative, absolute

Field reference

- symbol name

Example

- “Bytes 1024..1027 of foo.o refer to absolute address of _main”

Example C Program

m.c

```c
int e=7;

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

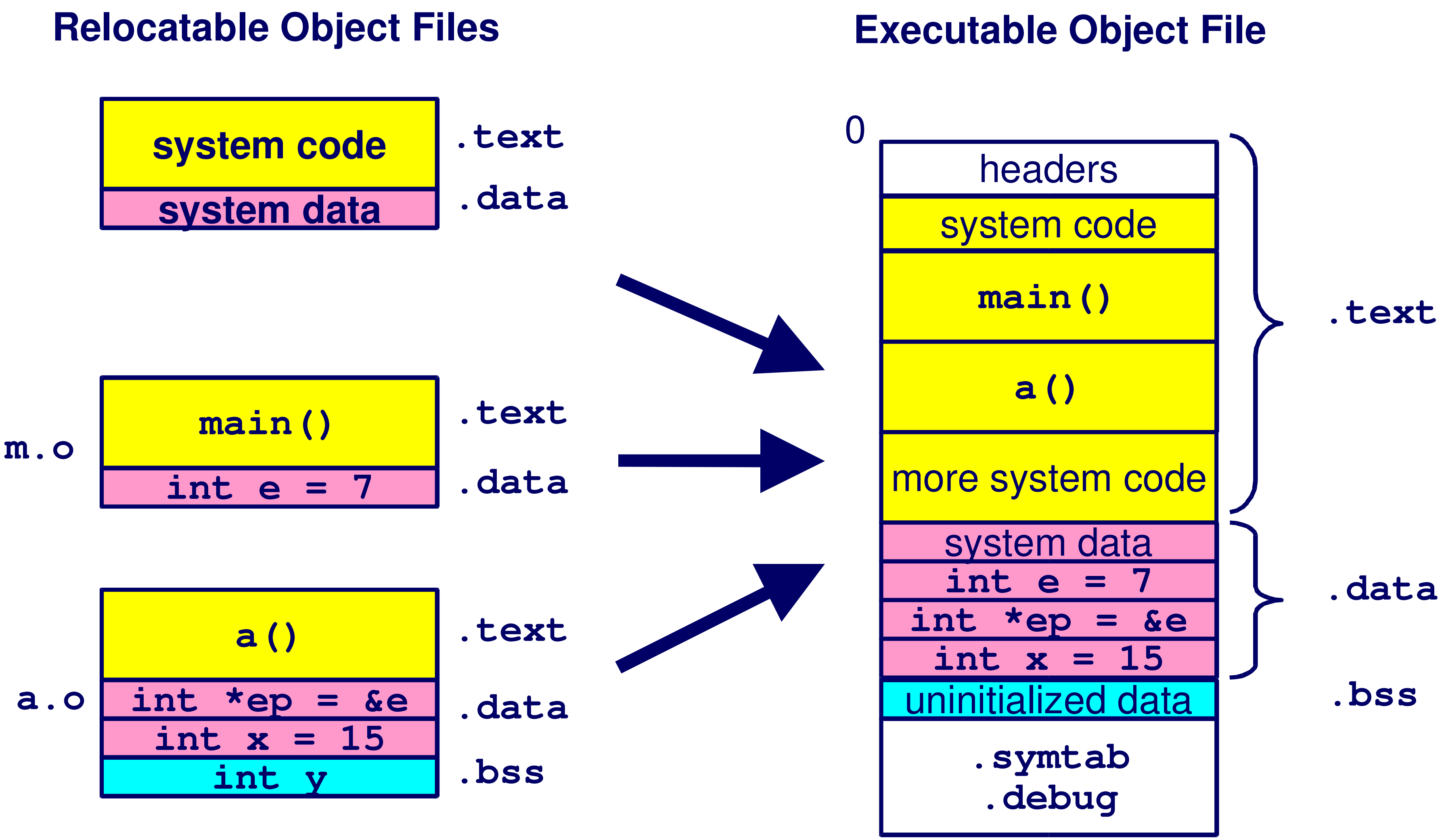
a.c

```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



Relocatable Object Files

system code .text

system data .data

m.o

main() .text

int e = 7 .data

a.o

a() .text

int *ep = &e .data

int x = 15

int y .bss

Executable Object File

0

headers

system code

main()

a()

more system code

.text

system data

int e = 7

int *ep = &e

int x = 15

.data

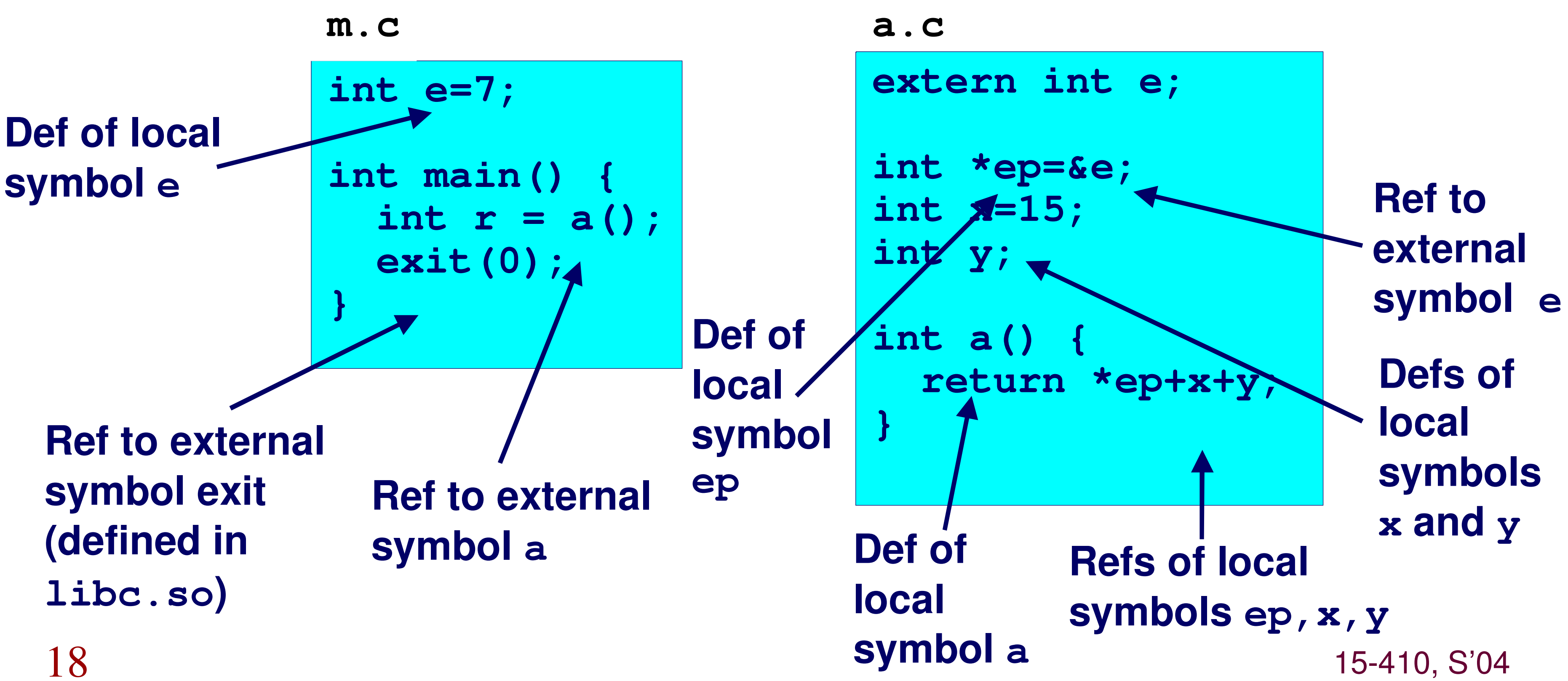
uninitialized data .bss

.symtab

.debug

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);
}
```

```
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

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:

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



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

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

Executable File / Image File

Linked program consists of multiple “sections”

- Section properties
  - Type
  - Memory address

Common Executable File Formats

- a.out - “assembler output” (primeval Unix format: 70's, 80's)
- Mach-O – Mach Object (used by MacOS X)
- ELF – Executable and Linking Format
  - (includes “DWARF” - Debugging With Attribute Record Format)

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

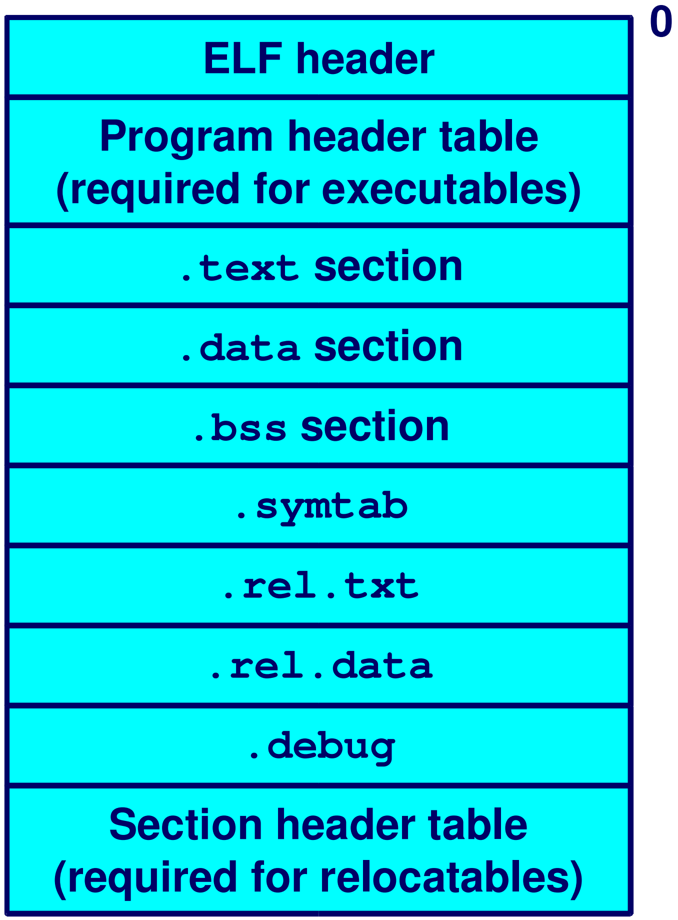
- Code

`.data` section

- Initialized (static) data

`.bss` section

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



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 (cont)

`.symtab` section

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

`.rel.text` section

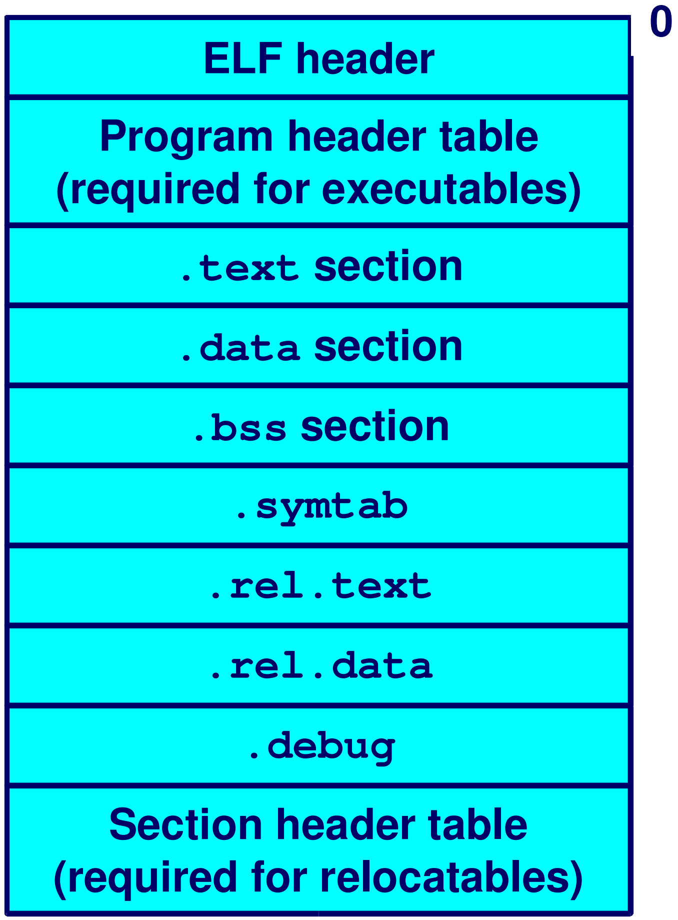
- Relocation info for `.text` section
- Addresses of instructions that will need to be modified in the executable
- Instructions for modifying.

`.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)

“Not needed on voyage”

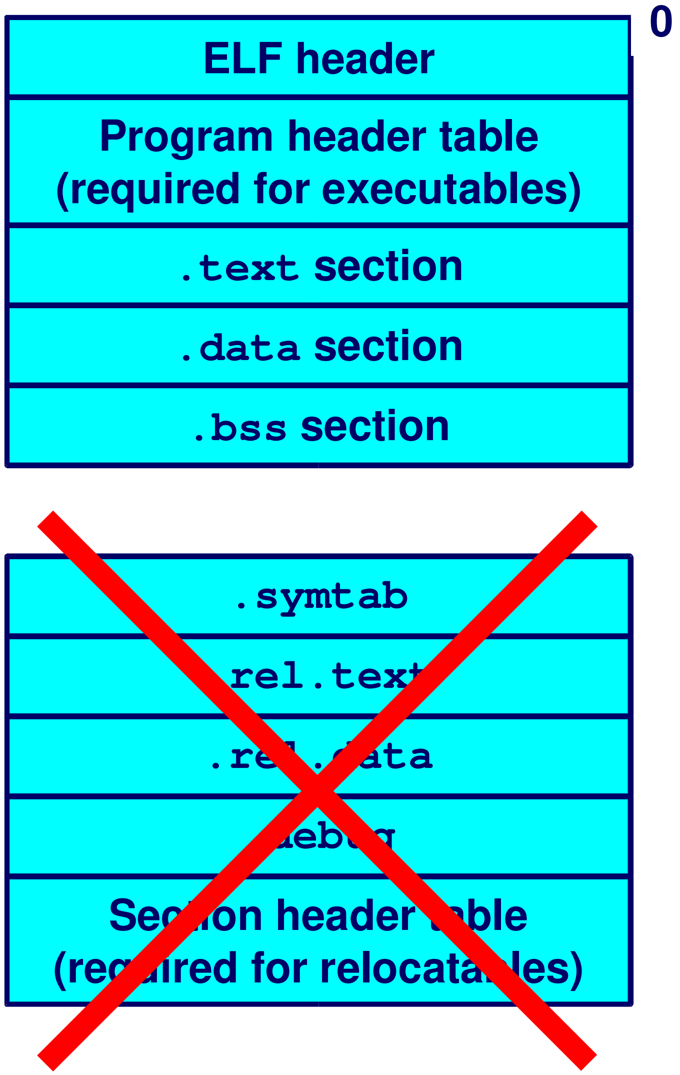
Some sections not needed for execution

- Symbol table
- Relocation information
- Symbolic debugging information

These sections not loaded into memory

May be removed with “strip” command

- Or retained for future debugging



0

ELF header

Program header table
(required for executables)

`.text` section

`.data` section

`.bss` section

`.symtab`

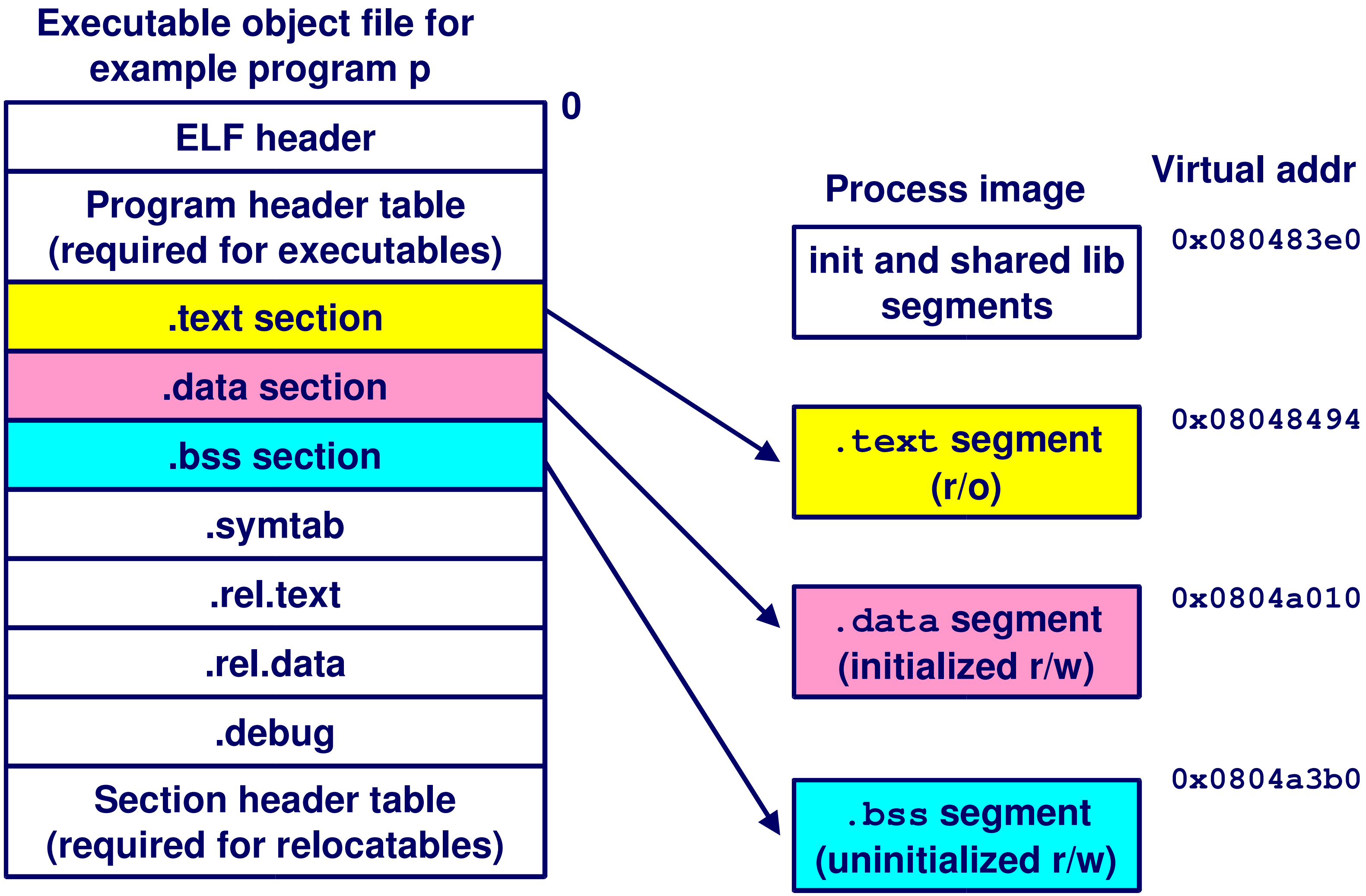
`.rel.text`

`.rel.data`

`.debug`

Section header table
(required for relocatables)

Loading ELF Binaries


Executable object file for example program p
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)
Process image
Virtual addr
init and shared lib segments
0x080483e0
.text segment (r/o)
0x08048494
.data segment (initialized r/w)
0x0804a010
.bss segment (uninitialized r/w)
0x0804a3b0

Packaging Commonly Used Functions

How to package functions commonly used by programmers?

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

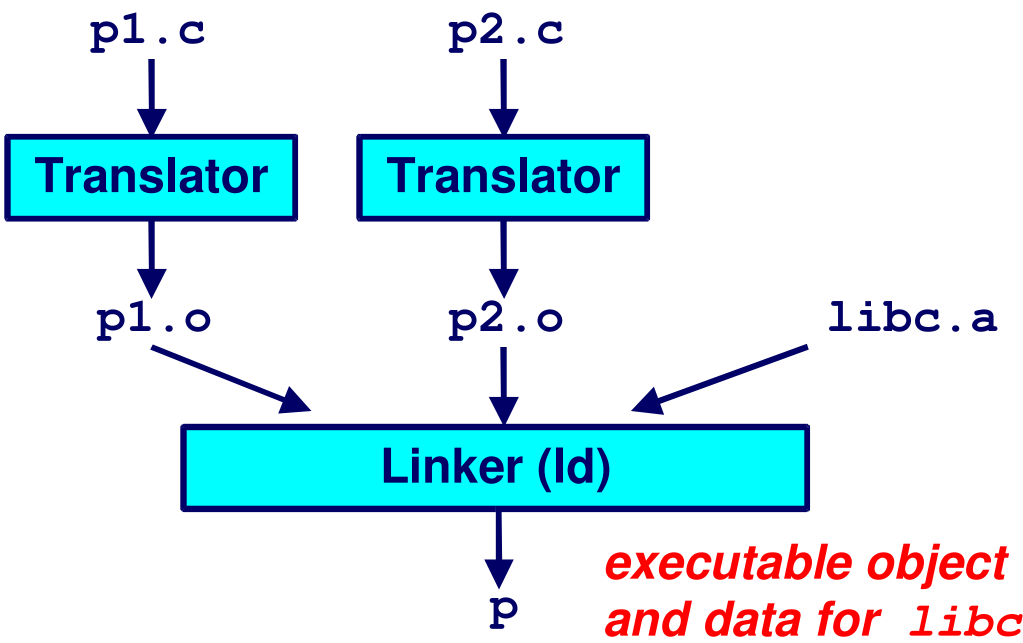
Awkward, given the linker framework so far:

- 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)



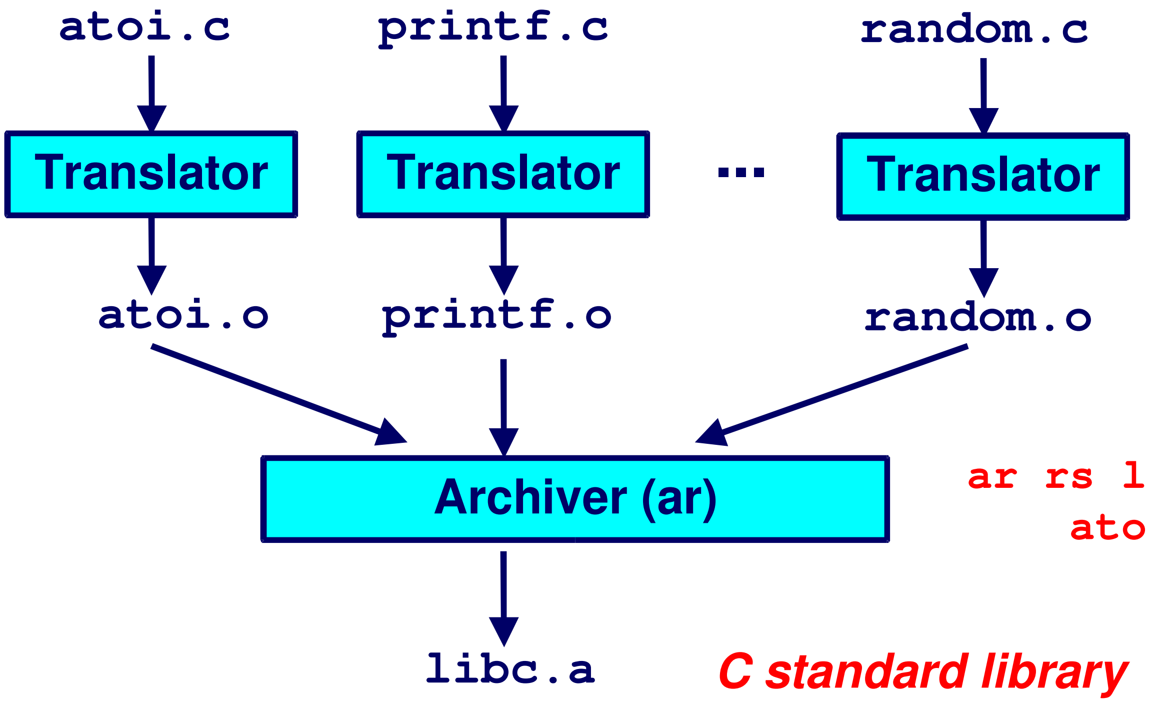
static library (archive) of relocatable object files concatenated into one file.

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 includes only those `.o` files in the archive that are actually needed by the program.

Creating Static Libraries



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

C standard library

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'
```

Dynamic Linking

Goal

- Build program
- Don't commit to particular version of library now
- Don't bloat program with 1024'th copy of printf() on disk

Defer “final link” as much as possible

- The instant before execution

Program startup invokes “shared object loader”

- Locates library files
- Adds files into address space
- Links files to program, often incrementally
  - Self-modifying “stub” routines
  - First call looks up routine address in symbol table
  - Later calls go directly to start of routine

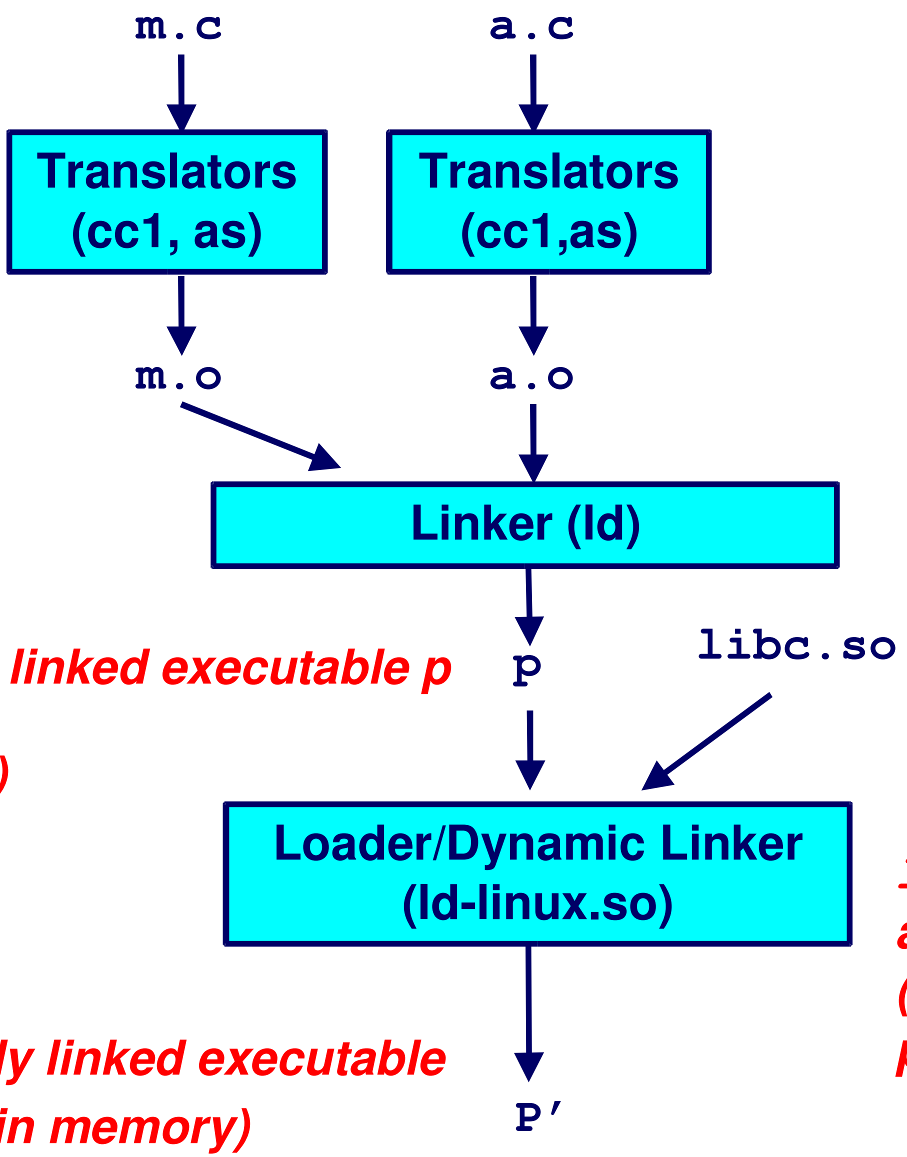
“Shared libraries”

Extension/optimization of dynamic linking

Basic idea

- Why have N copies of printf() *in memory?*
- Allow processes to share memory pages
  - “Intelligent” mmap()
- Must avoid address-map conflicts
  - Can issue each library on a system an address range, or
  - Can build libraries from *position-independent code*
    - » (out of scope for this class)

Dynamically Linked Shared Libraries



`m.c`

`a.c`

Translators (cc1, as)

Translators (cc1,as)

`m.o`

`a.o`

Linker (ld)

`p`

`libc.so`

Loader/Dynamic Linker (ld-linux.so)

`p'`

Partially linked executable p

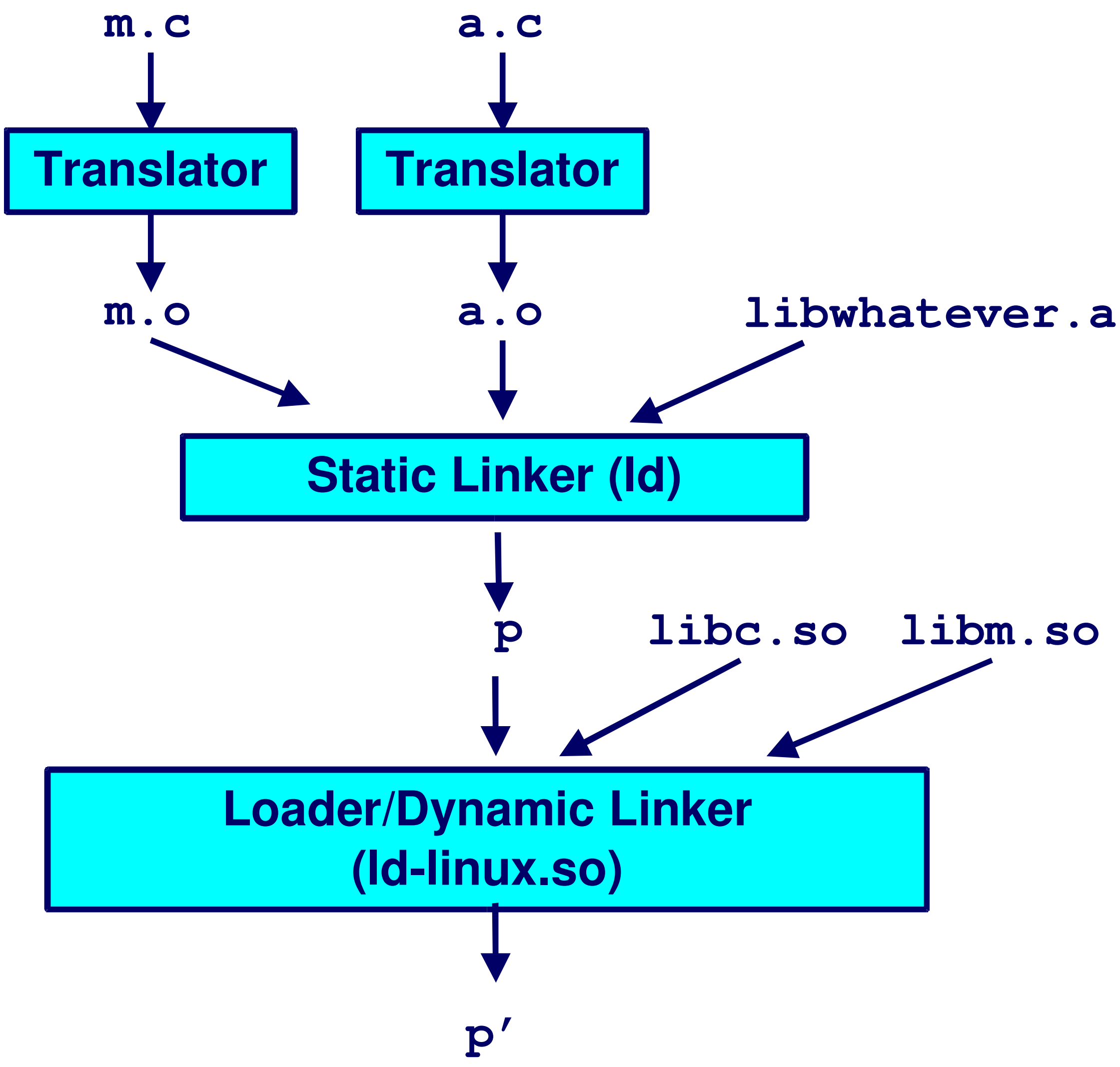
(on disk)

Shared library of dynamically relocatable object files

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

Fully linked executable p' (in memory)

The Complete Picture



Summary

Where do addresses come from?

Where does an int live?

Image file vs. Memory image

Linker

- What, why
- Relocation

ELF structure

Static libraries

Dynamic / Shared libraries

Back to the mystery of pf.c

```
[bmm@bmm bmm]$ cc -S pf.c -o pf.S
[bmm@bmm bmm]$ as -c pf.S -o pf.o
[bmm@bmm bmm]$ gcc pf.o -o pf
```

Back to the mystery of pf.c

```
[bmm@bmm bmm]$ cc -S pf.c -o pf.S
[bmm@bmm bmm]$ as -c pf.S -o pf.o
[bmm@bmm bmm]$ ld -static pf.o /usr/lib/crt1.o /usr/lib/crti.o \
/usr/local/libexec/gcc-2.95.3/lib/gcc-lib/i686-pc-linux-gnu/2.95.3/crtbegin.o \
/usr/local/libexec/gcc-2.95.3/lib/gcc-lib/i686-pc-linux-gnu/2.95.3/crtend.o \
/usr/lib/crtn.o -lc -o pf
```

```
[bmm@bmm bmm]$ cc -S pf.c -o pf.S
[bmm@bmm bmm]$ cat pf.S

.section        .rodata
.LC0:
        .string "%d"
.text
        .align 4
.globl main
        .type    main,@function
main:
        pushl %ebp
        movl %esp,%ebp
        pushl $printf
        pushl $.LC0
        call printf
        addl $8,%esp
.L1:
        leave
        ret
```

```
[bmm@bmm bmm]$ as pf.S -o pf.o
[bmm@bmm bmm]$ objdump -D --disassemble-zeroes pf.o

pf.o:     file format elf32-i386

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 c4 f8             add    $0xfffffff8,%esp
   9:   68 00 00 00 00       push   $0x0
   e:   68 00 00 00 00       push   $0x0
  13:   e8 fc ff ff ff       call   14 <main+0x14>
  18:   83 c4 10             add    $0x10,%esp
  1b:   89 ec                mov    %ebp,%esp
  1d:   5d                   pop    %ebp
  1e:   c3                   ret

Disassembly of section .data:
Disassembly of section .rodata:

00000000 <.rodata>:
   0:   25
   1:   64
   2:   00
```

```
[bmm@bmm bmm]$ gcc pf.o -o pf
[bmm@bmm bmm]$ objdump -D --disassemble-zeroes pf

pf:     file format elf32-i386

Disassembly of section .text:

080483e4 <main>:
 80483e4:       55                      push   %ebp
 80483e5:       89 e5                   mov    %esp,%ebp
 80483e7:       83 ec 08                sub    $0x8,%esp
 80483ea:       83 c4 f8                add    $0xfffffff8,%esp
 80483ed:       68 08 83 04 08          push   $0x8048308
 80483f2:       68 68 84 04 08          push   $0x8048468
 80483f7:       e8 0c ff ff ff          call   8048308 <_init+0x70>
 80483fc:       83 c4 10                add    $0x10,%esp
 80483ff:       89 ec                   mov    %ebp,%esp
 8048401:       5d                      pop    %ebp
 8048402:       c3                      ret

Disassembly of section .rodata:

08048464 <_IO_stdin_used>:
 8048464:       01 00
 8048466:       02 00
 8048468:       25
 8048469:       64
 804846a:       00
```

Linker Puzzles

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

```
p1() {}
```

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

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

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

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

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

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

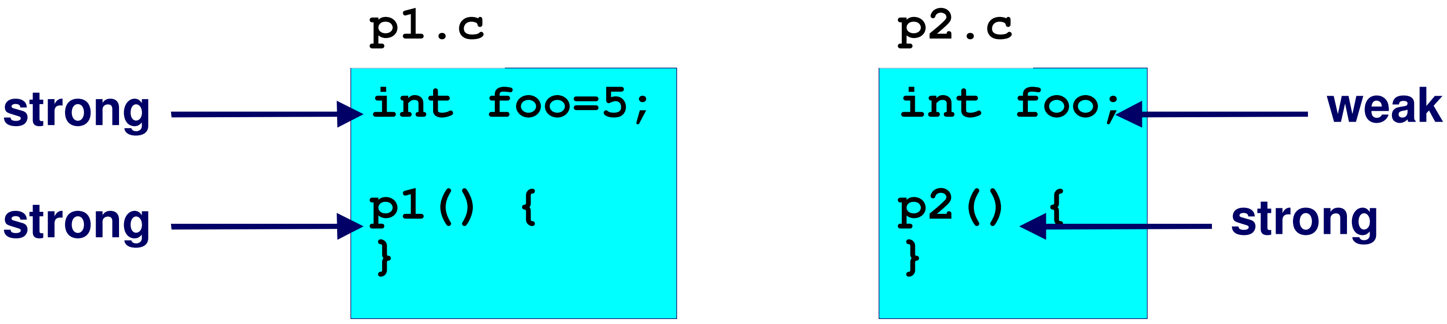
```
int x=7;
p1() {}
```

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

Strong and Weak Symbols

Program symbols are either strong or weak

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



p1.c

```
int foo=5;

p1() {
}
```

p2.c

```
int foo;

p2() {
}
```

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.