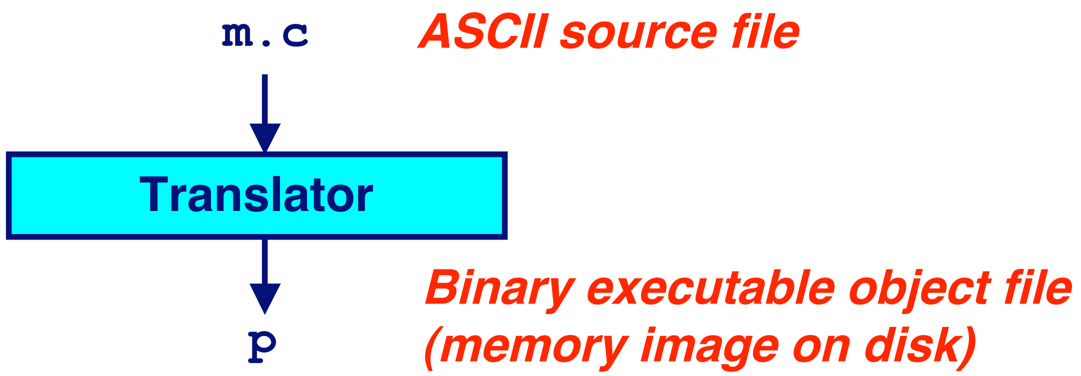
CS 201

Linking

Gerson Robboy
Portland State University

A Simplistic Program Translation Scheme



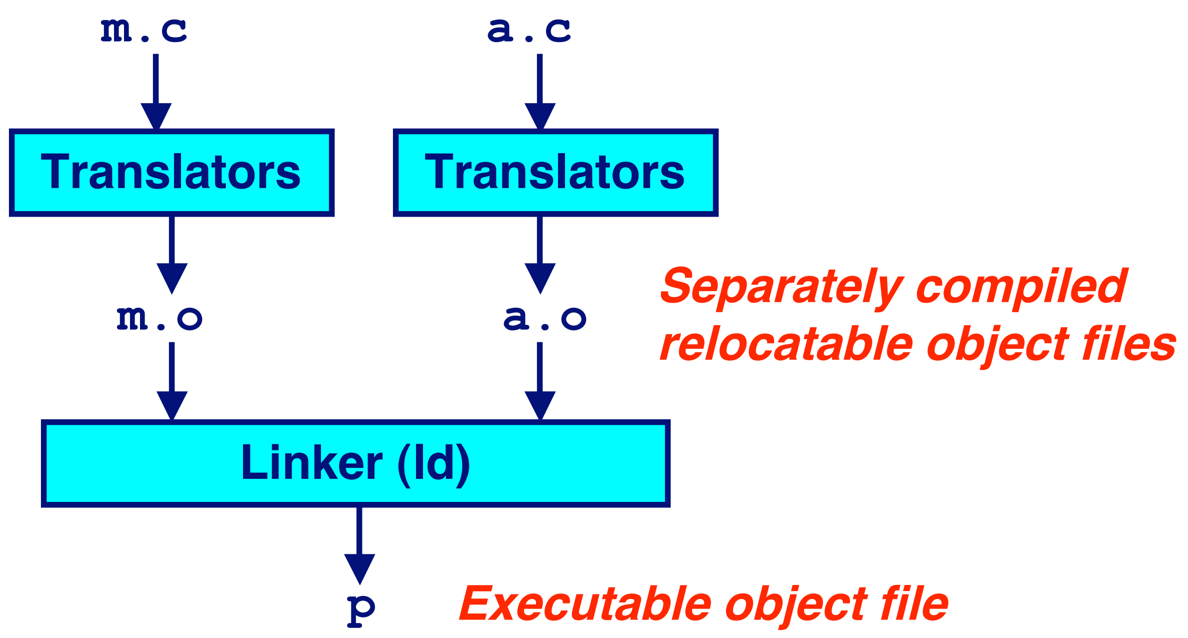
Problems:

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

Solution:

- ***LInker***

A Better Scheme Using a Linker



Translating the Example Program

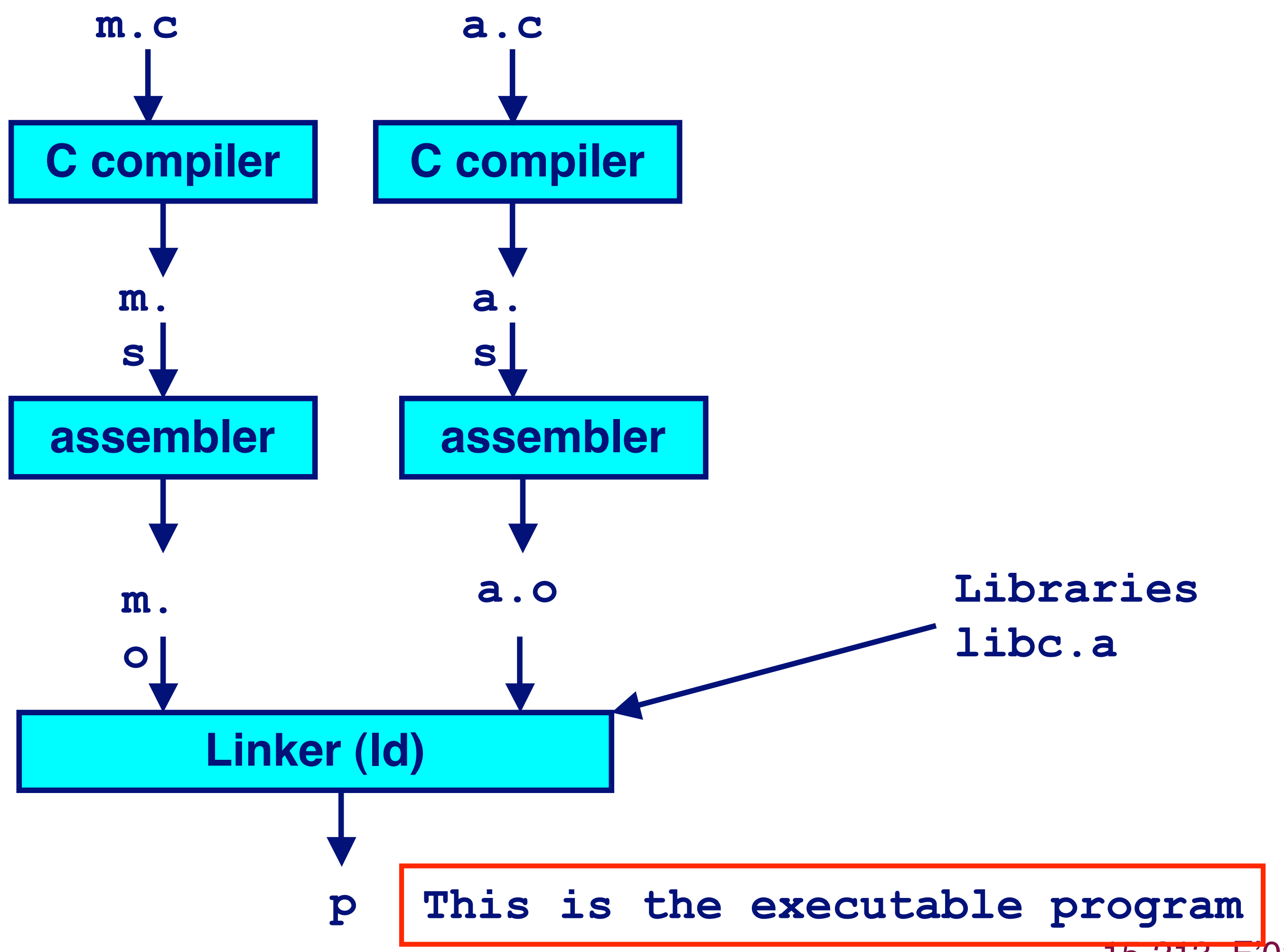
Compiler driver **coordinates all steps in the translation and linking process.**

- **Included with each compilation system (`cc or gcc`)**
- **Invokes preprocessor (`cpp`), compiler (`cc1`), assembler (`as`), and linker (`ld`).**
- **Passes command line arguments 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>
```

A picture of the tool set



What Does a Linker Do?

Merges object files

- Merges multiple relocatable (`.o`) object files into a single executable program.

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 in both code and data
    - » code: `a();            /* reference to symbol a */`
    - » data: `int *xp=&x;  /* reference to symbol x */`

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.

Questions for you

When a linker combines relocatable object files into an executable file, why does the linker have to modify instructions in the actual code?

How does the linker know what values to put into the code?

How does the linker know exactly where to insert those values?

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.

Also better, more complete information for debuggers.

ELF Object File Format

Elf header

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

Program header table

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

`.text` section

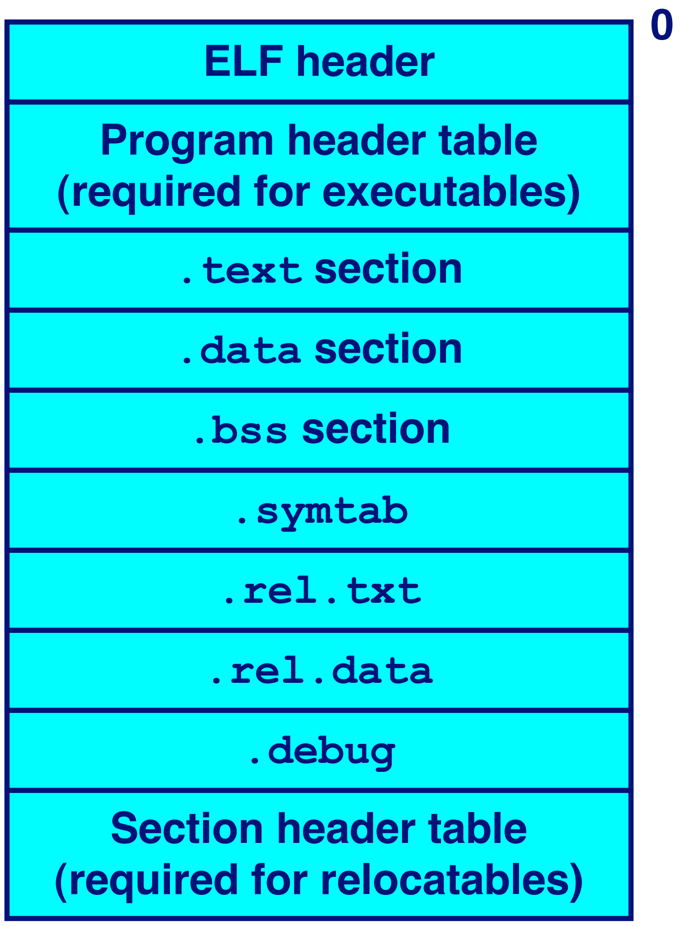
- Code

`.data` section

- Initialized (static) data

`.bss` section

- Uninitialized (static) data
- “Block Started by Symbol”
- Has section header but occupies no space in the disk file


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**

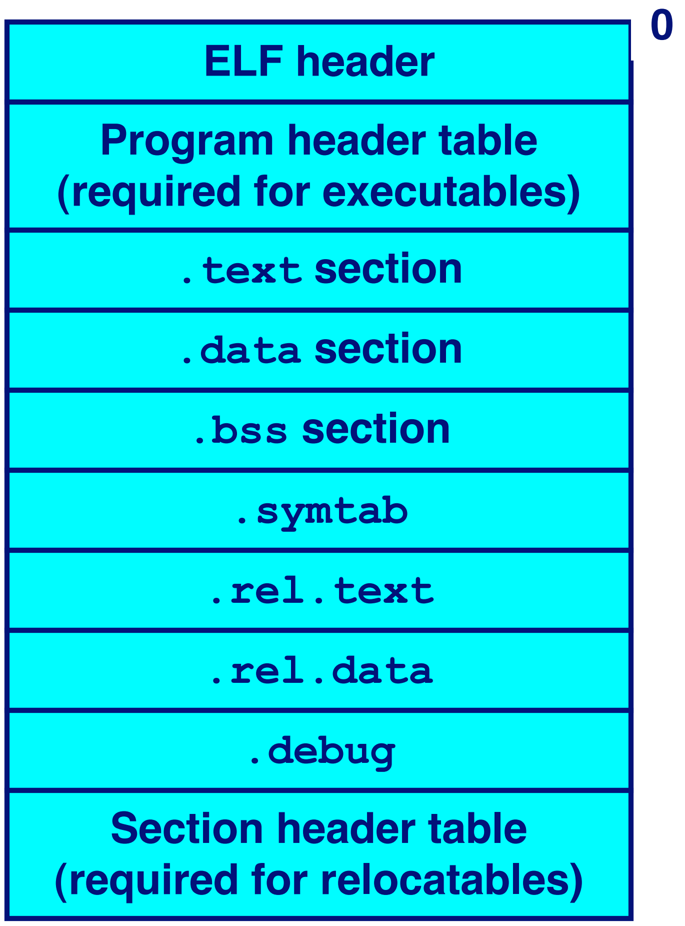
- Relocation info for `.text` section
- Addresses of instructions that will need to be modified in the executable
- Instructions for modifying.

`.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)

Example C Program

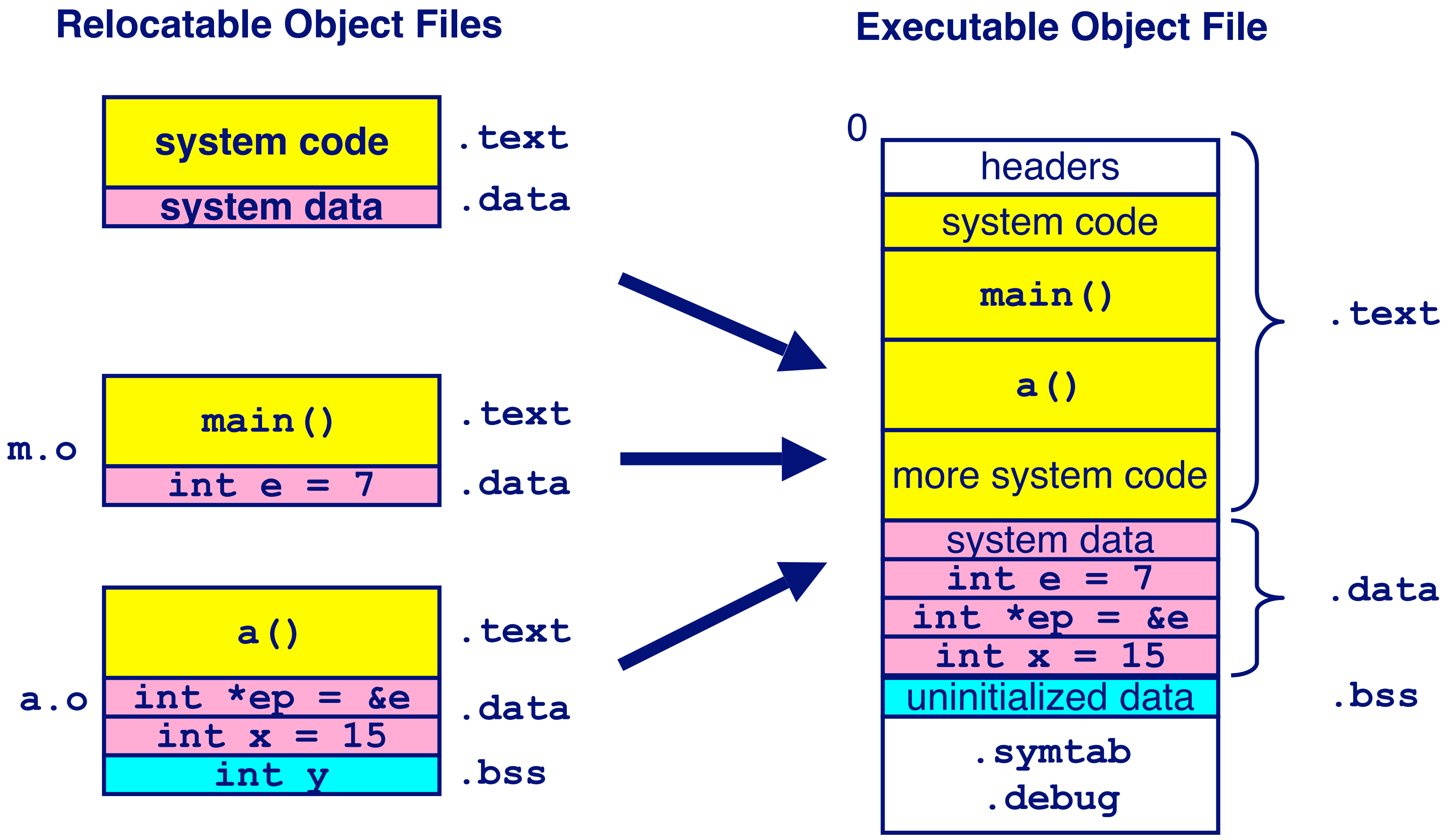
m.c

```
int e=7;

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

a.c

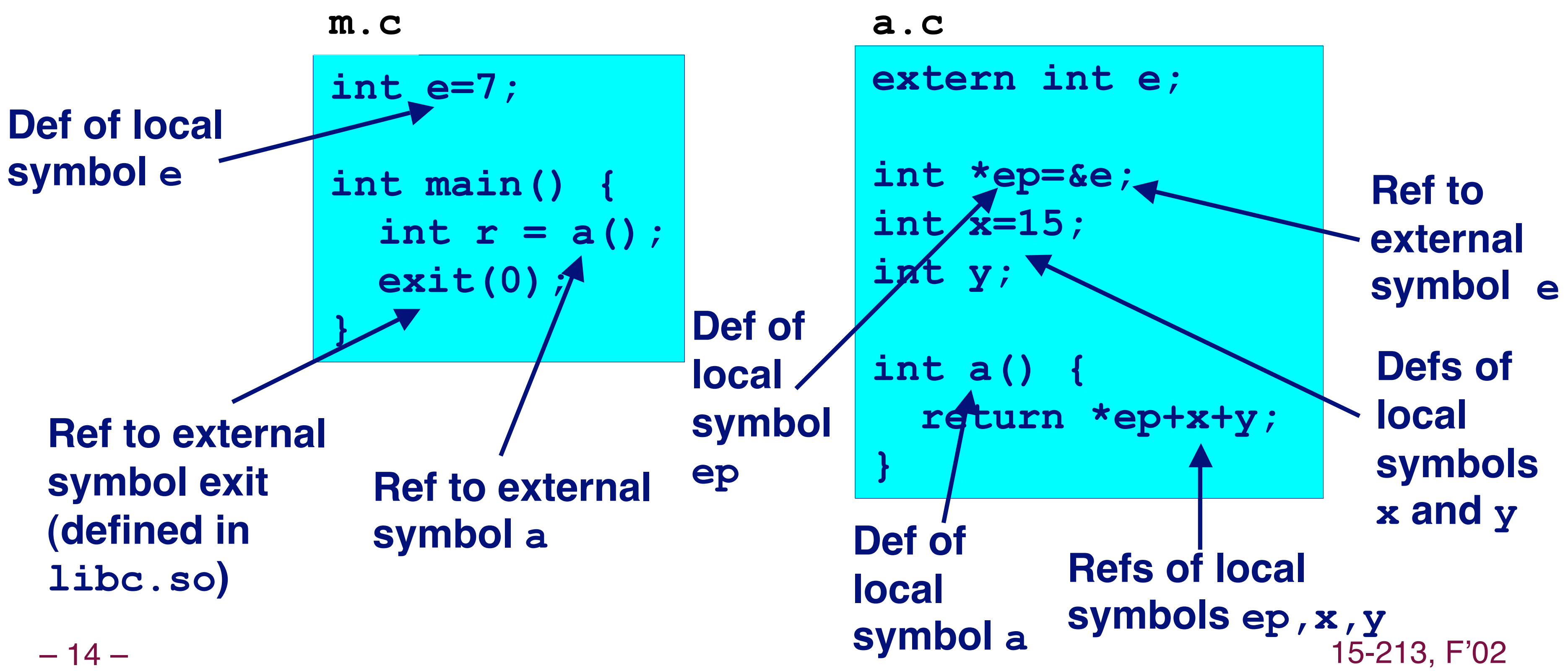
```
extern int e;

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;
}
```

Questions for you

In the function main on the previous slide, why is there no arrow pointing to the variable *r* ?

Does *r* have to be relocated when the program is linked?

What information about *r* has to be in the symbol table?

What does the debugger need to know about *r* ?

External functions

In main, notice that the names *a* and *exit* are external symbols.

The compiler knows they are functions, and the linker will resolve the references.

Exit is just another function call

- The compiler doesn’t know anything about Unix system calls
- The compiler knows about names and data types

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
```

Question

On the previous slide, the variables ep, x, and y are local in the same source file.

So why can’t the compiler just generate completed code? Why is relocation information necessary?

`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(.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
```

Exercise

m.c

```
void f(void);
int x = 15213;
int y = 15212;

int main() {
  f();
  printf("x = %d,y = %d\n", x, y);
  return 0;
}
```

a.c

```
double x;

void f() {
  x = 0.0;
}
```

Will the C compiler accept this code without an error?
Will the C compiler give a warning?
Will the C compiler overload the two symbols "x" because they have different data types?
Will the linker link these modules, or abort with an error?
What will this program print when it runs?

In the previous slide, what can you change to make the program work “correctly;” i. e., print the initialized values of x and y?

Exercise

m.c

```
int f()
{
   static int x = 0;
   return x;
}
int g()
{
   static int x = 1;
   return x;
}
```

Will the C compiler accept this code without an error?
Are the two variables *x* temporary?
What is their scope?
Is there a conflict between the two variables *x*?
How does the compiler handle these two variables?
How does the linker handle them?

Relocation

In a relocatable file, each section (text, data, bss) starts at address zero. Offsets in the section are relative to zero.

In an executable file, each section is bound to the absolute address at which it will be loaded in memory.

How does the linker know what address to bind each section to?

- That is, how does the linker know where the program will be loaded in memory?

Where are programs loaded in memory?

To start with, imagine a primitive operating system.

- Single tasking.
- Physical memory addresses go from zero to N.
- The problem of loading is simple: load the program starting at address zero
  - Use as much memory as it takes.
- The linker binds the program to absolute addresses
  - Code starts at zero
  - Data concatenated after that
  - etc.

Where are programs loaded, cont'd

Next imagine a multi-tasking operating system on a primitive computer.

- A physical memory space, from zero to N.
- Memory must be allocated at load time.
- The linker does not know where the program will be loaded.
  - The linker binds together all the modules, but keeps them relocatable.

How does the operating system load this program?

- Not a pretty solution.

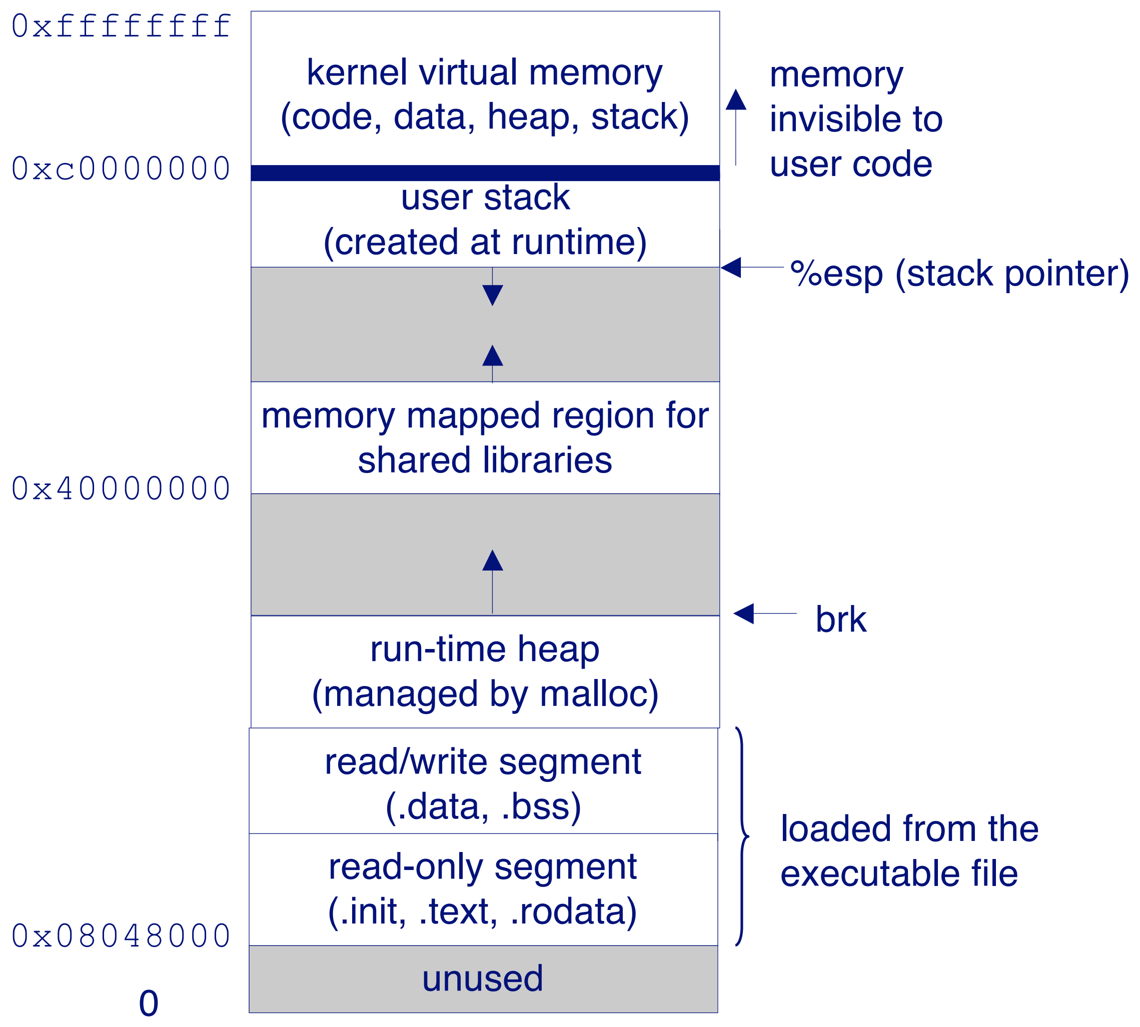
Where are programs loaded, cont'd

Next, imagine a multi-tasking operating system on a modern computer, with hardware-assisted dynamic relocation.

- The O. S. creates a virtual memory space for each user's program.
  - As though there is a single user with the whole memory all to itself.
- Now we're back to the simple model
  - The linker statically binds the program to virtual addresses
  - At load time, the operating system allocates memory, creates a virtual address space, and loads the code and data.
  - More about how this is done in a few weeks.

The linker binds programs to absolute addresses

Nothing is left relocatable, no relocation at load time.



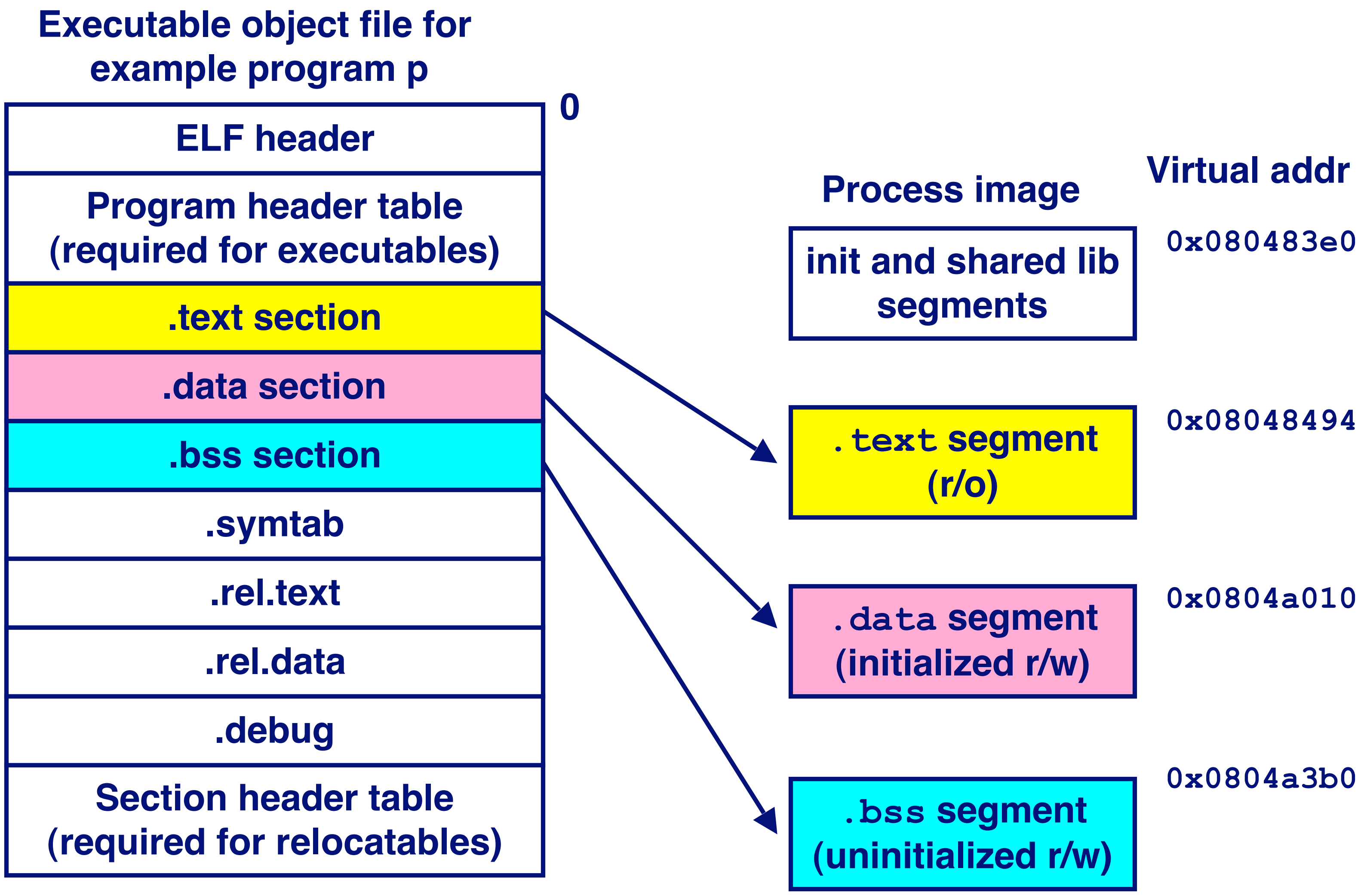
0xffffffff
kernel virtual memory (code, data, heap, stack)
memory invisible to user code
0xc0000000
user stack (created at runtime)
%esp (stack pointer)
memory mapped region for shared libraries
0x40000000
brk
run-time heap (managed by malloc)
read/write segment (.data, .bss)
loaded from the executable file
read-only segment (.init, .text, .rodata)
0x08048000
unused
0

More details on program loading

How does the O. S. know where to load the program and how much memory to allocate?

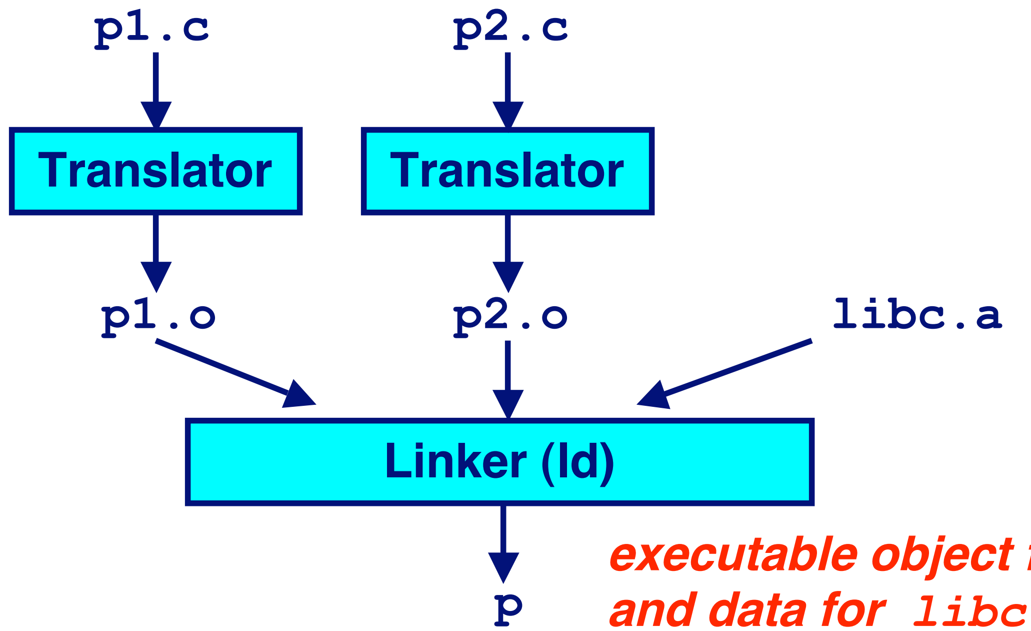
- The linker and the O. S. loader must agree on an object module format.
  - The linker writes an executable file
  - The O. S. loader reads that file to load the program
  - The O. S. allocates the appropriate memory, and reads the program code and data into memory.
- More on this in CS 333.

Loading Executable 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

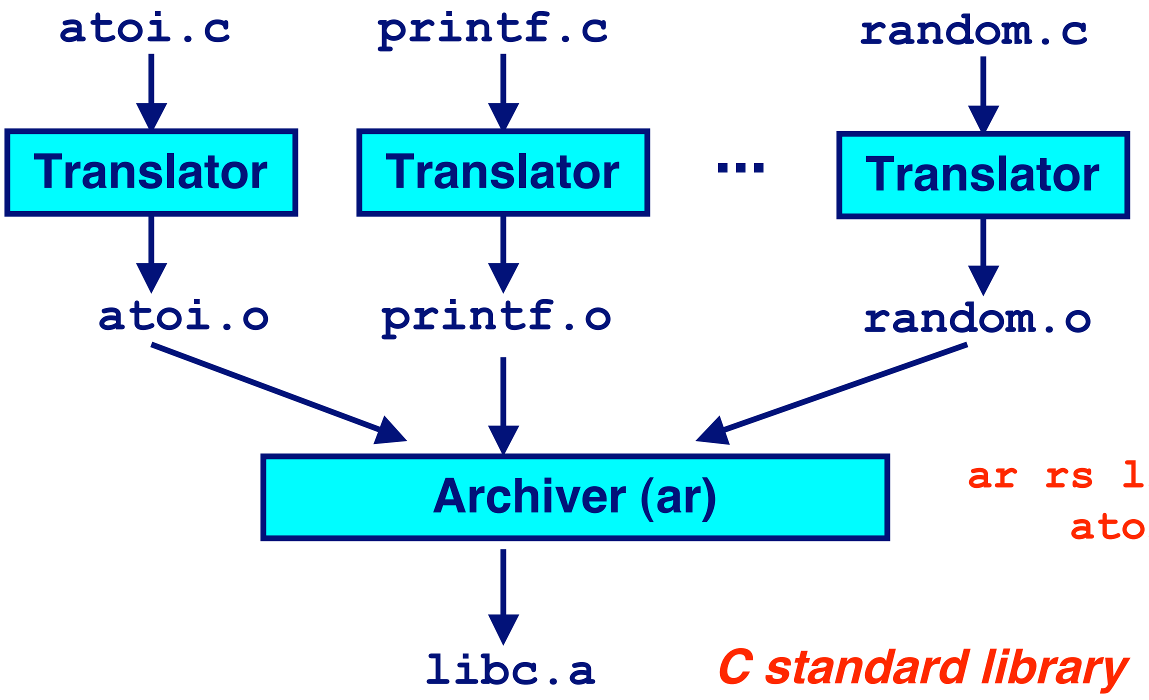
Static Libraries (archives)



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`)

Creating Static Libraries



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

C standard library

Why do we need static libraries?

Why not just use ld to link atoi.o, printf.o, random.o, … into a big relocatable file, *libc.o* instead of an archive, *libc.a ?*

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

The linker tries to resolve all references by scanning the files on the command line, in order

- As each new .o or .a file obj is encountered, try to resolve each unresolved reference in the list against the symbols in obj.

Command line order matters!

- In your Makefile, where should libraries go on 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'
```

Exercise

Suppose you write a program that uses the *pow* function from libm.a, which takes two double arguments:

```
double pow(double x, double y);
```

`Suppose you pass it an integer for y, instead of a double, and you find that it works correctly.`

1. How can you tell if pow implemented as a macro or a library function call? List three different ways you can find out.

2. When you assign an integer value to a float or double variable, the compiler does the conversion for you. Does the compiler do that when you pass an integer as an argument to a function that takes a double? How can you tell?

3. If pow or some other function is implemented as a macro that takes a double as an argument, and the programmer passes it an int instead of a double, then how can the macro still work correctly?

Shared Libraries

Invented by AT&T in 1986 for Unix System V on PCs

- In 1986 the Intel 386 came out
- The PC was at last capable of meaningfully running Unix

Microsoft later copied the idea: DLLs

What problem was AT&T trying to solve?

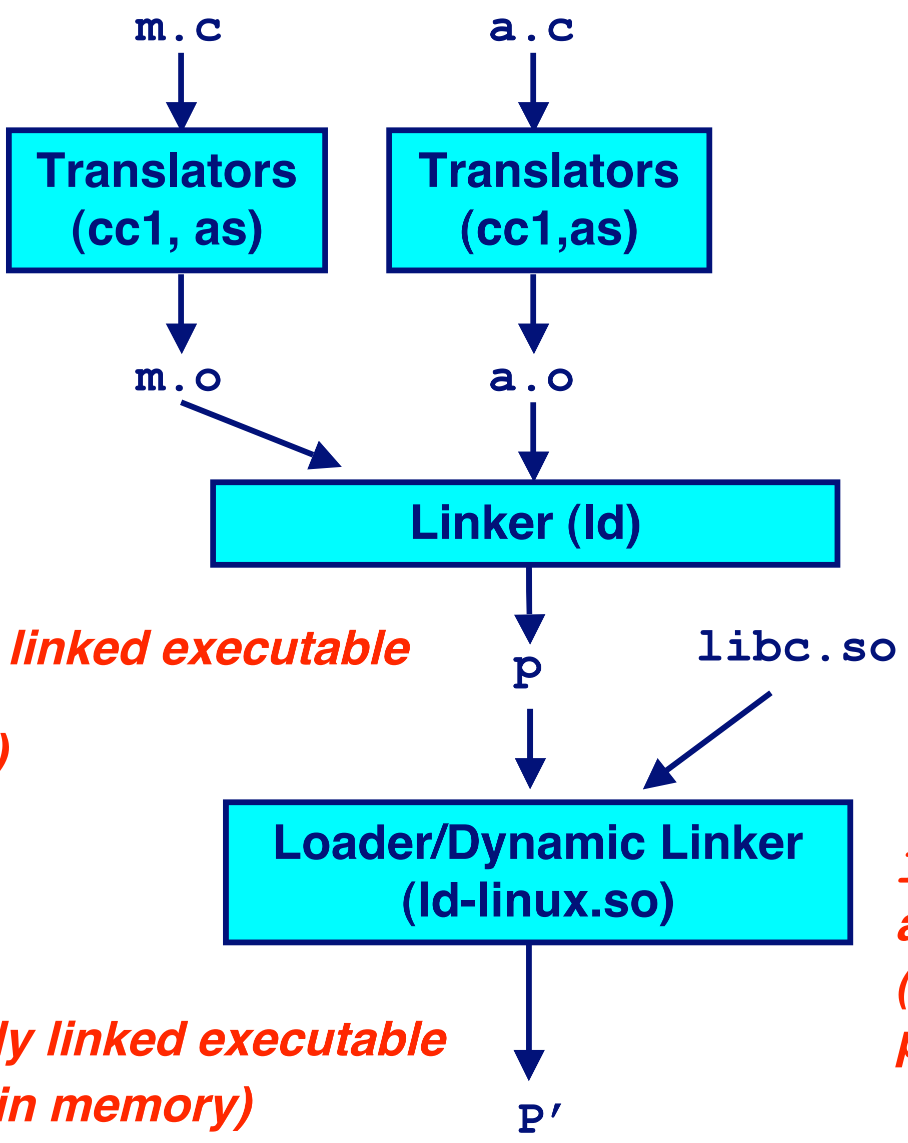
- PC distribution of Unix was on floppy disks
  - Lots and lots of floppy disks
- Reduce the aggregate size of the distribution
- Also conserve memory at run time

Shared Libraries

What problems do shared libraries solve today?

- **Avoid duplicating code in the virtual memory space of many processes.**
- **Minor bug fixes of system libraries don't require a relink of all the user space programs**

Dynamically Linked Shared Libraries



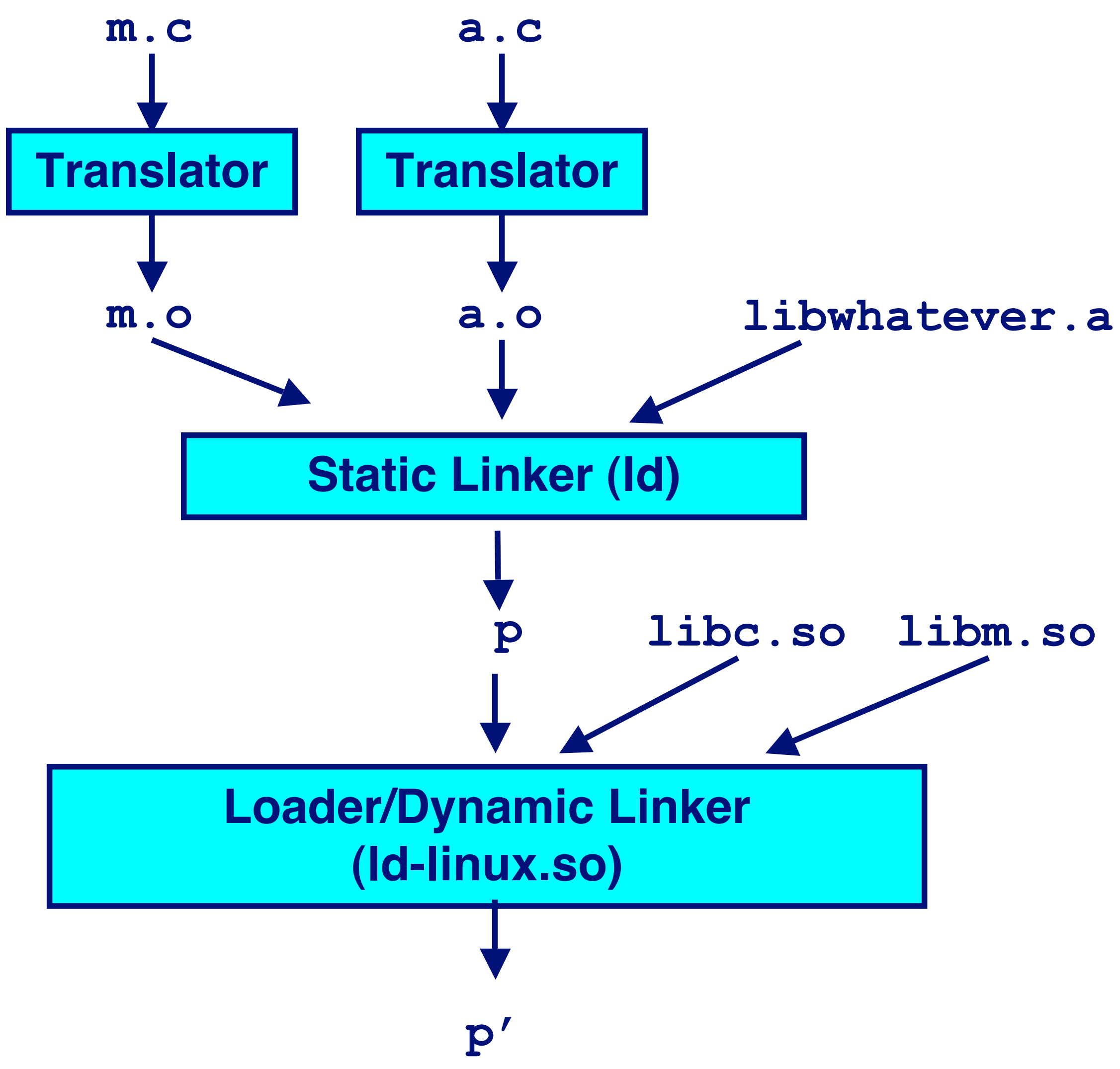
Partially linked executable p (on disk)

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



Problems to solve with shared libraries

Where do you put them in memory?

- Solution: Reserve a region of virtual memory for shared libraries

What's the problem if each shared library function has its own reserved fixed address?

What's the problem if shared libraries can be relocated when loaded?

Problems with dynamic relocation

Within your own program:

- Where are the shared library functions?
- How do you call them?

Within the shared library code itself:

- How to call other functions within the shared library?
- How to call functions in other shared libraries?
- How to access global variables if they are relocated?
  - External global variables
  - Defined in the same file, but relocated

Version Control

The biggest problem with shared libraries is version control.

Is a newly installed program compatible with the shared libraries that came with the O. S.?

A hassle on linux:

- Copy a binary program from another linux system
- It won't run because of different version of shared libraries

Are shared libraries worth the hassle?

Do they really solve a problem today?

A note on installable device drivers

By-product of shared library technology

These are cool.

- **Buy commodity components, retail**
- **Install a device vendor's driver from a CD or Internet**
- **No need to compile or link the kernel**
- **Anyone can do it at home**

PIC

problem with shared libraries

could assign a chunk of address space to each possible shared lib

- but this is pain in posteriori

therefore use "Position Independent Code"

code more or less references addresses via table lookup plus offset

more expensive - but solves shared lib problem on UNIX

maybe 5 instructions per single instruction for non-shared approach

PIC workings with gcc

- compile and link to create shared lib (.so)
- gcc -shared -fPIC -o libmumble.so a.c b.c .. z.c
- link to previous program that wants to use shared lib
- gcc -o program mainprog.c ./libmumble.so
- NOTE: table lookup info but no code added to program
- load and run (execvp), however first dynamic linking is done so that program can find libmumble.so

list of tools mentioned in B/O

1. ar - library archive tool (libc.a <- *.o)
2. strings - find C strings in a binary file.
3. strip - throw out the symbol table
4. nm - list the symbol table (enemy of strip)
5. readelf - display structure of object file, including elf header
6. objdump - dump binary file in various ways
7. size - simple info about text/bss etc for size of binary file
8. hexdump/od - hex/octal dumper
9. don’t leave out ld - linker, and ldd - dynamic loader