CS 3214
Computer Systems

Some of the following slides are taken with permission from
Complete Powerpoint Lecture Notes for
Computer Systems: A Programmer's Perspective (CS:APP)

Randal E. Bryant and David R. O'Hallaron

http://csapp.cs.cmu.edu/public/lectures.html

Part 1

LINKING AND LOADING

Topics

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

- Linking is a mundane topic
  - Full of quirks and seemingly arbitrary rules
  - But worth learning, IMO
  - Essential skill for any practicing C programmer
  - Necessary skill for productive practice in mixed-language and inter-language environments

High-Level Issues

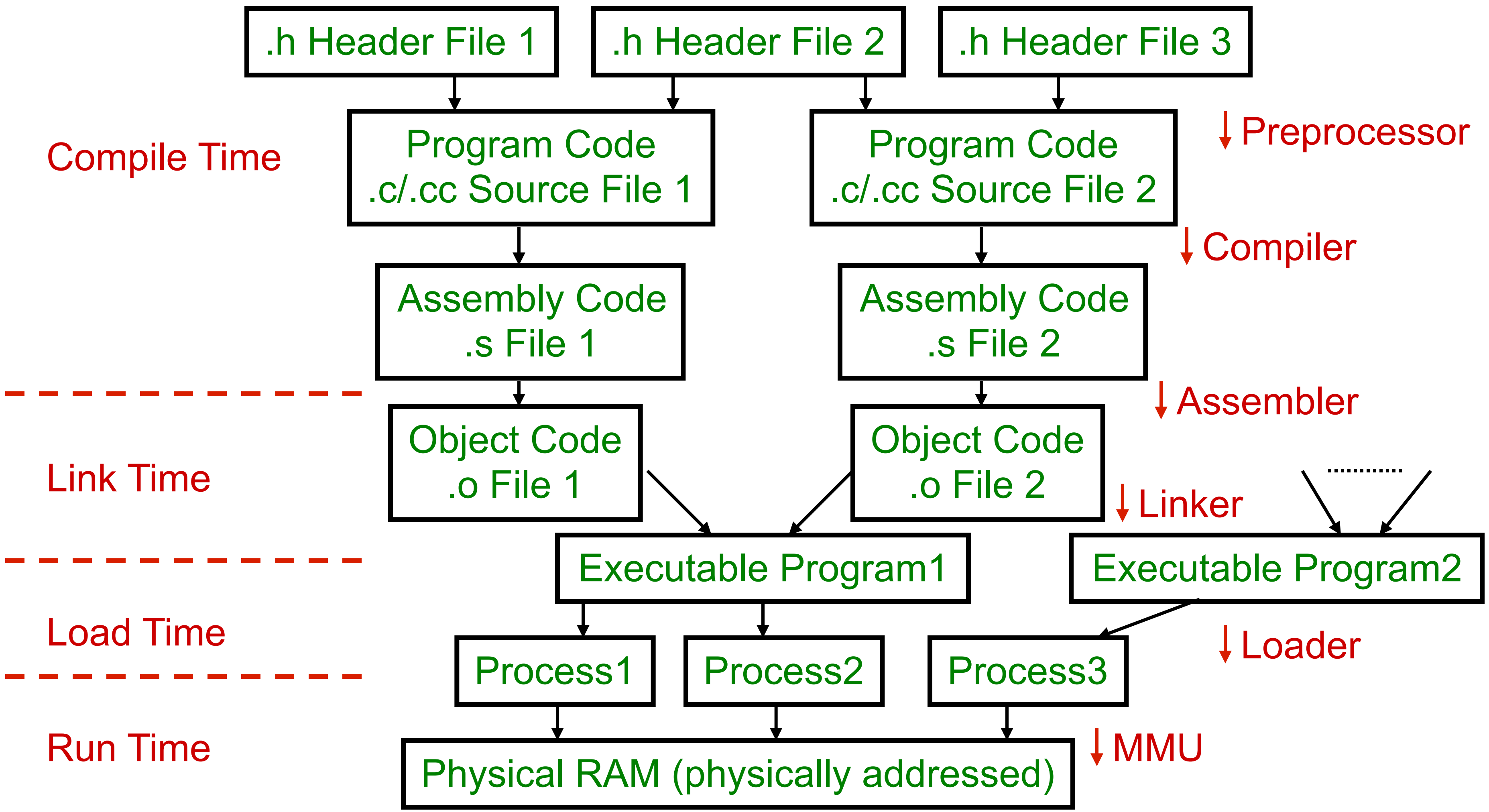
- How are (large) programs being built and executed
  - How do compiler, linker, and loader cooperate
- SE angle:
  - What “module” system does the standard toolchain provide, particularly wrt encapsulation and code reuse.
  - Difference between declaration and definition
  - Best practices in using it when writing large programs

Compiling and Linking

- *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 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      cpp has been integrated into cc1
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>
```

The Big Picture



From High To Low Level: Resolving Symbolic Names

- Compiler, Assembler, Linker all resolve symbolic names
- Compiler: (function-)local variables, field names, control flow statements
- Assembler: resolves labels
- Linker resolves references to (file-)local ("static") and global variables and function

Linker Puzzles

Question for each example:
Will it link?
If so, will it run?
If so, what will happen?

code1.c	code2.c
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() {}

What Does a Linker Do?

- Merges object files
  - Merges multiple relocatable (`.o`) object files into a single executable object file that can 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();            /* reference to symbol a */`
      - data: `int *xp=&x;   /* reference to symbol x */`

Why Linkers?

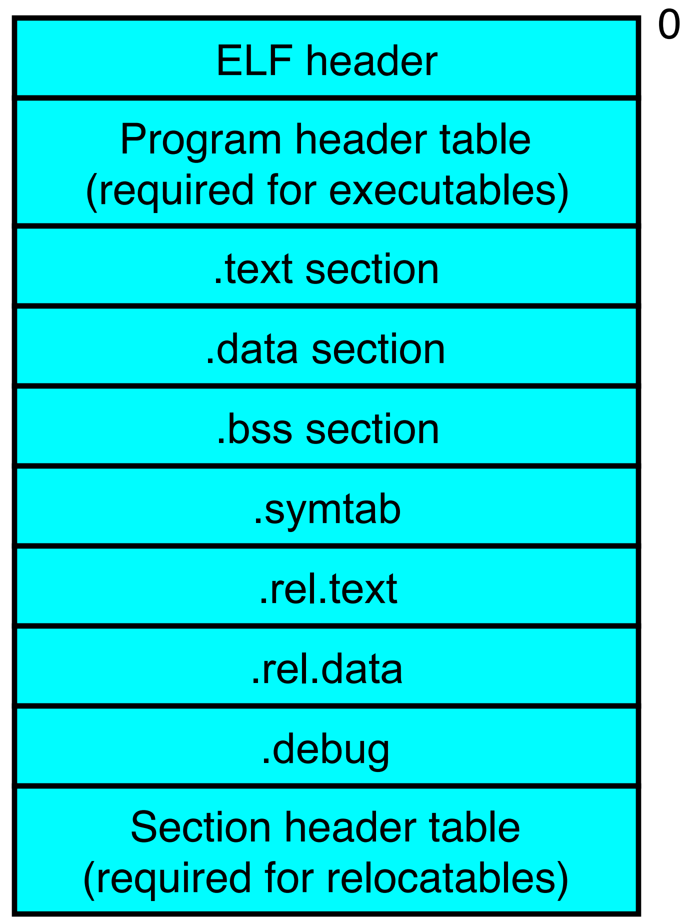
- Modularity
  - Program can be written as a collection of smaller source files, rather than one monolithic mass.
  - Can build libraries of common functions (more on this later)
    - e.g., Math library, standard C library
- Efficiency
  - Time:
    - Change one source file, compile, and then relink.
    - No need to recompile other source files.
  - Space:
    - Libraries of common functions can be aggregated into a single archive file...
    - Yet executable files and running memory images contain only code for the modules whose 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
  - Shared object files (.so)
- Generic name: ELF binaries
- Better support for shared libraries than old a.out formats.

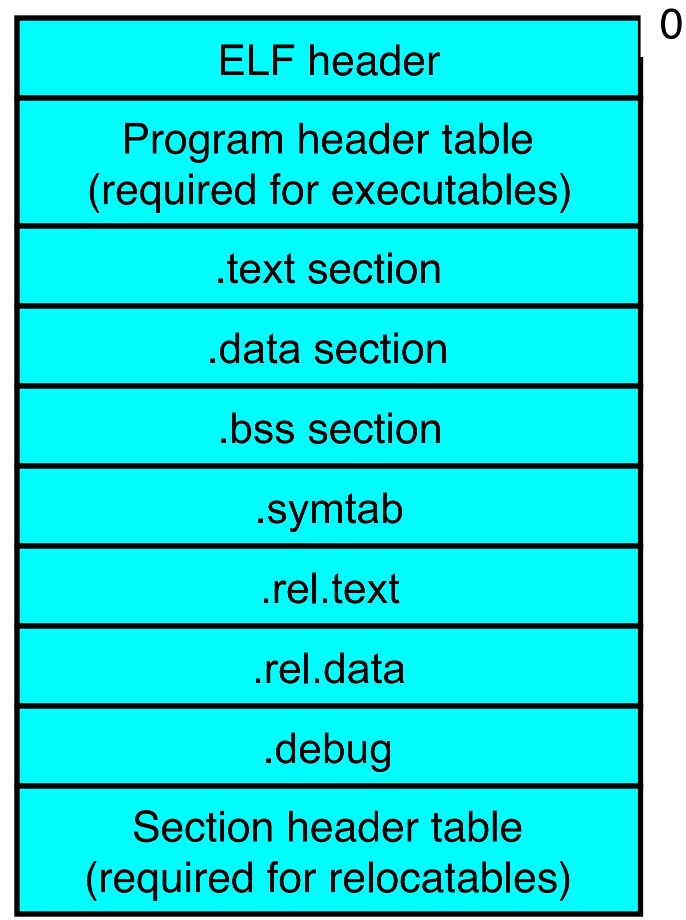
ELF Object File Format

- 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)

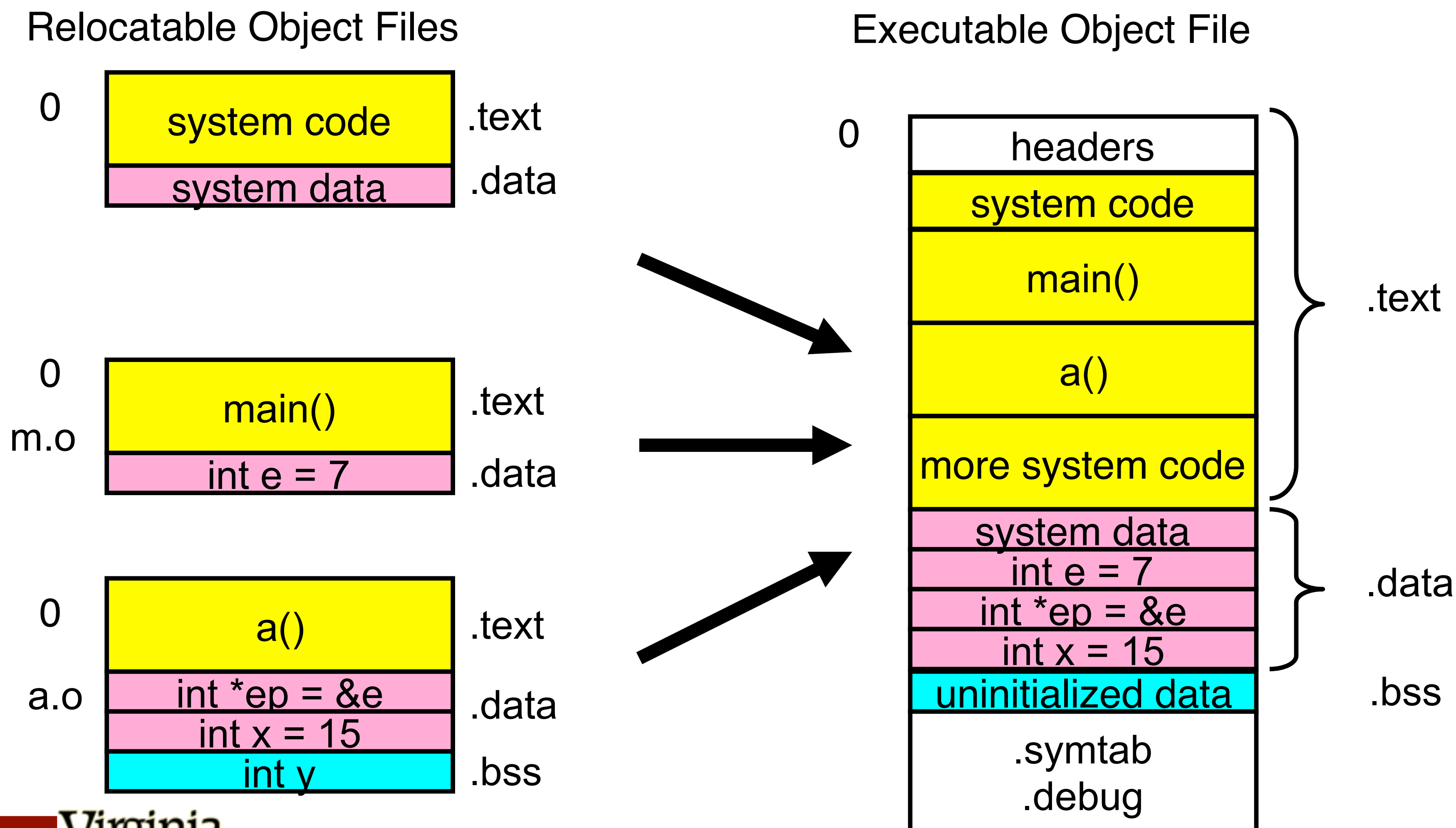
- `.symtab` section
  - Symbol table
  - Procedure and static variable names
  - Section names and locations
- `.rel.text` section
  - Relocation info for `.text` section
  - Addresses of instructions that will need to be modified in the executable
  - Instructions for modifying.
- `.rel.data` section
  - 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`)



Relocatable object files

- Contains multiple sections denoted in header
- Constructed as if text & data started at address 0
- Can be moved anywhere in memory
- Includes place holders for where values of symbols will go (e.g., as part of jump/call or mov instructions)
- Includes "patchup" instructions for linker – aka relocation records that describe where final addresses must appear after linking

Merging Relocatable Object Files into an Executable Object File



Linker Symbols

- Global symbols
  - Defined by a module, can be used by other modules
  - C global variables and functions (non-static!)
- External symbols
  - Used by a module, defined by another module
- Local symbols
  - Defined by a module, used exclusively in that module
  - C static functions and static variables
    - Don't confuse with local variables!
  - Different modules can use the same local symbol without conflict – different copies!

Example C Program

m.c

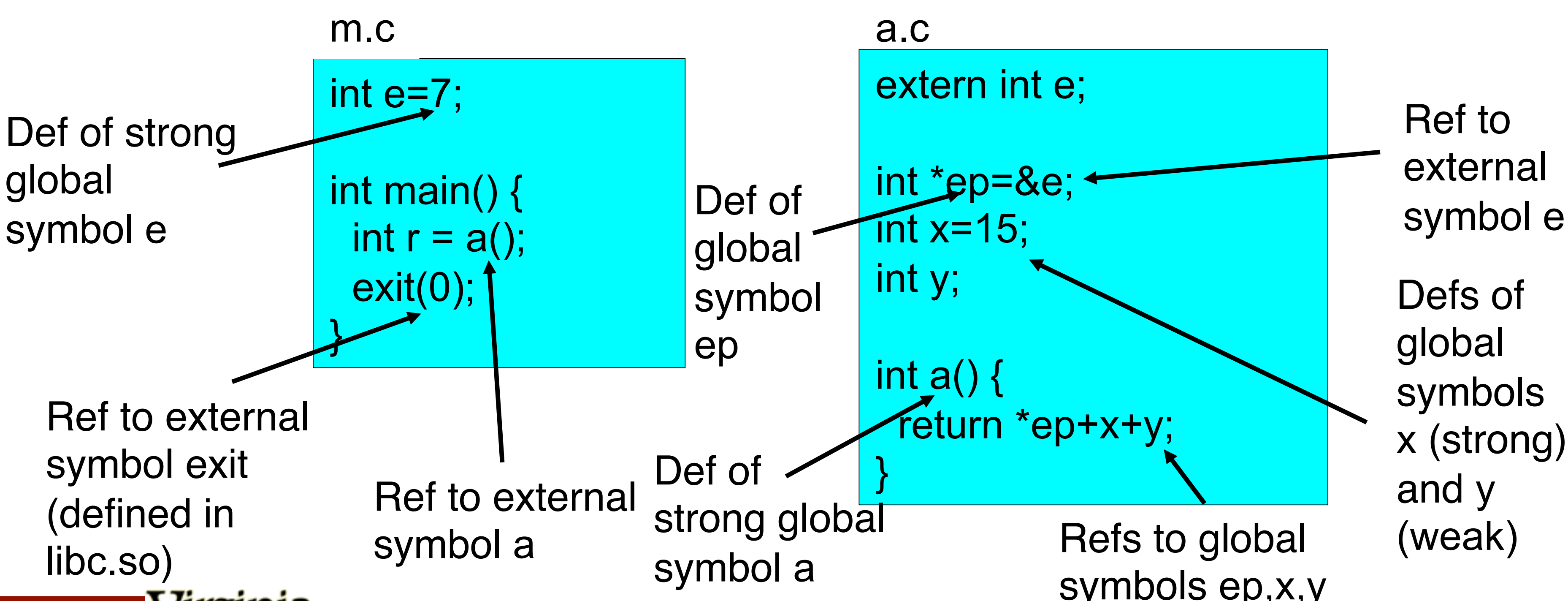
```c
int e=7;

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;
}
```

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*.
- Definitions can be *local* or *global*. **Local is local to a .o file/module!**
- Global definitions can be *strong* or *weak*.
- 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
```

R_386_PC32:
PC-relative relocation

```
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
```

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 of the same name, the linker can pick an arbitrary one.

Linker Puzzles, Resolved

Module 1	Module 2	Resolution
`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.

Linker Symbols

- Global symbols
  - Defined by a module, can be used by other modules
  - C global variables and functions (non-static!)
- External symbols
  - Used by a module, defined by another module
- Local symbols
  - Defined by a module, used exclusively in that module
  - C static functions and static variables
    - Don't confuse with local variables!
  - Different modules can use the same local symbol without conflict – different copies!

Mapping C Names To Symbols

Functions

- static void f(void) { … }
  - Defines local symbol 'f'
- void g(void) { … }
  - Defines (strong) global symbol 'g'
- static void f(void);
  - Defines no symbol
- void e(void);
  extern void e(void);
  - Make 'e' an external reference
- Undefined functions are assume to be external by default

Variables

- static int lf = 4;
  static int lf2;
  - Defines local symbols 'lf', 'lf2'
- int wgl;
  - Defines weak global symbol aka 'common' symbol
- int gl = 4;
  - Defines strong global symbol
- extern int ef;
  - 'ef' is defined somewhere else
- No default

Aside: Assembler Rules

- Symbols became labels at the assembler level
- Default here is local, must say ".global" to make it global
  - Reversed from C, where default is global and must say "static" to make local

Practical Considerations

- Variables:
  - Avoid global variables (of course!)
  - Place 'extern' declaration in header file, choose exactly one .c file for definition
    - (If appropriate) initialize global variable to make symbol strong
  - If you followed these rules, **`-Wl,--warn-common`** should be quiet
- Functions
  - Make static whenever possible
  - Use consistent prefix for public functions, e.g.:
    - file.c exports file_open, file_close, file_read, file_write
    - strlen.c contains strlen()
  - Don't ignore "implicit declaration warnings"
  - Declare global functions in header files
    - Consider **`–Wmissing-prototypes`**

Practical Considerations (2)

- Never define variables in header files
- Consider
  - If defined non-static
    - w/o initialization, e.g. int x;
      - will link and refer to same variable – unclear if this is intended (because of multiple weak symbol rule)
    - w/ initialization, e.g. int x = 5;
      - gives linker error "multiply defined" once .h is included in multiple .c files
  - If defined static, e.g. static int x;
    - w/ or w/o initialization
      - Will compile and link, but each .c file has its own copy
      - This is practically <u>never</u> what you want

Practical Considerations (3)

- Defining functions in header files
- Ok for static functions
  - (non-static would give conflicting strong symbols)
- Potential disadvantage: if function is not inlined, code will be duplicated in each .o module
  - If inlined, no penalty
- Remember that compiler needs to see function definition in same compilation unit to inline
  - Solution: define all functions you wish to inline statically in .h files
  - Define all small functions in .h files

Some of the following slides are taken with permission from
Complete Powerpoint Lecture Notes for
Computer Systems: A Programmer's Perspective (CS:APP)

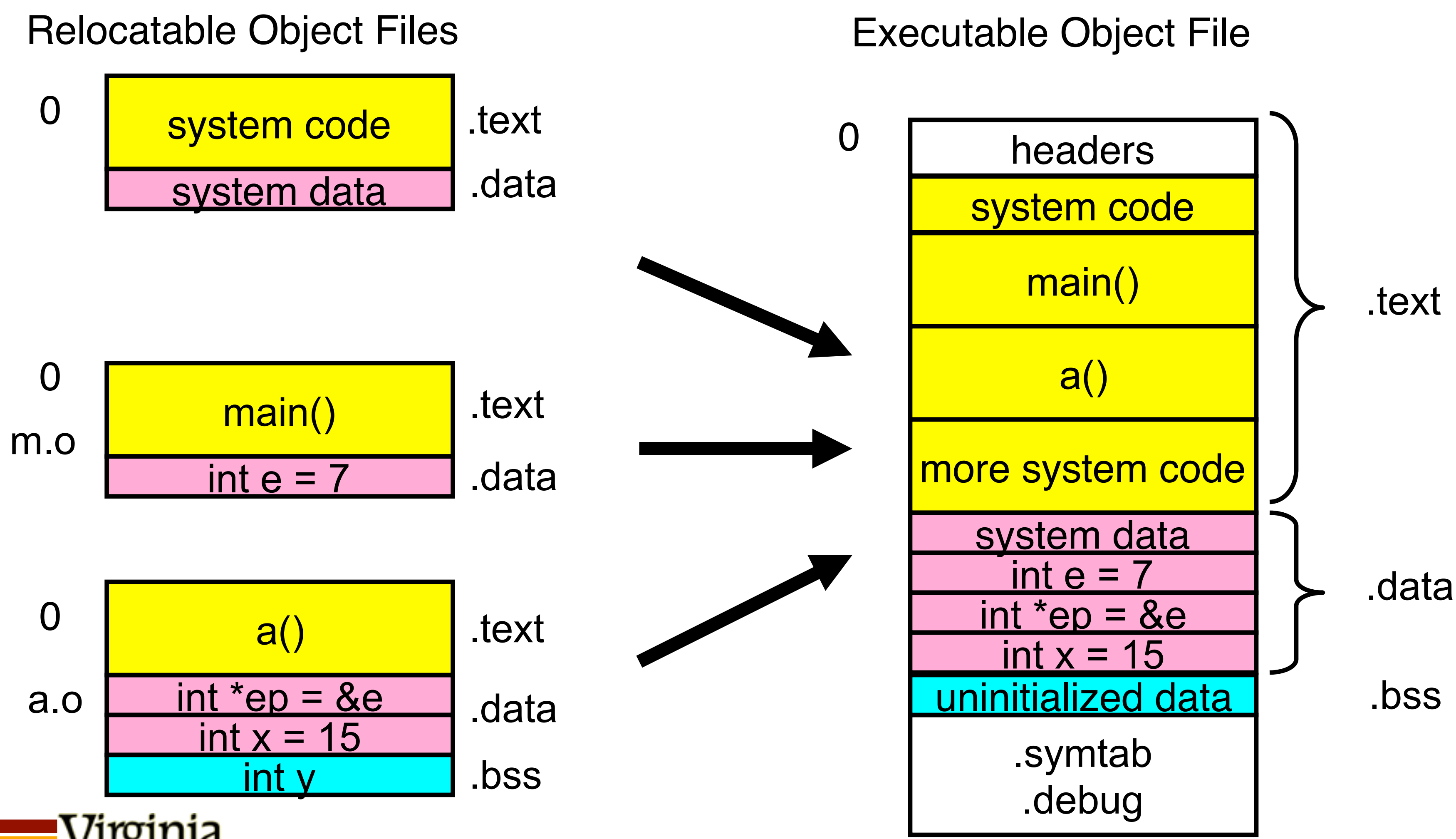
Randal E. Bryant and David R. O'Hallaron

http://csapp.cs.cmu.edu/public/lectures.html

Part 2

LINKING AND LOADING

Merging Relocatable Object Files into an Executable Object File



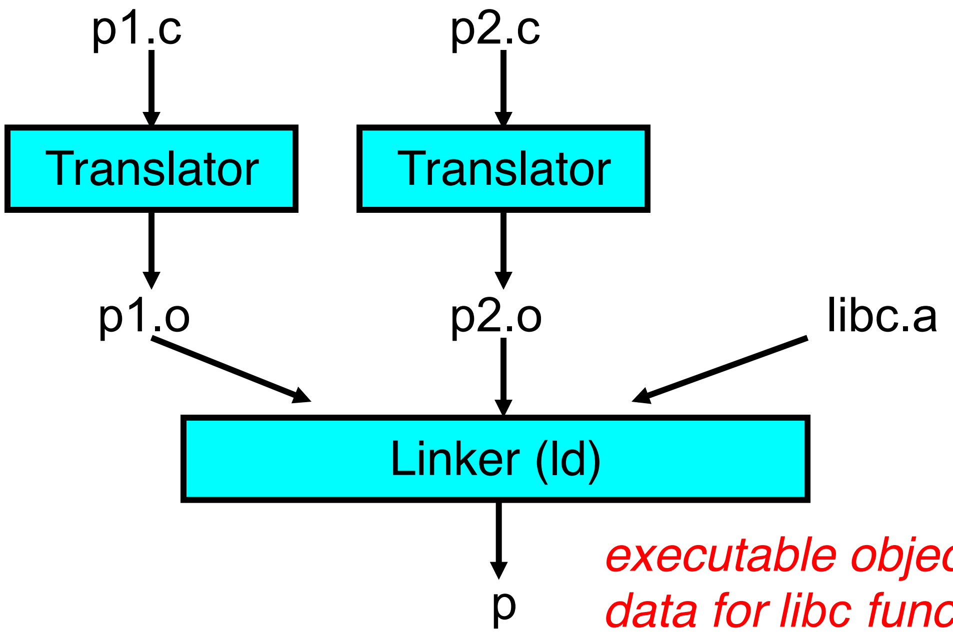
Packaging Commonly Used Functions

- 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

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 member file 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)

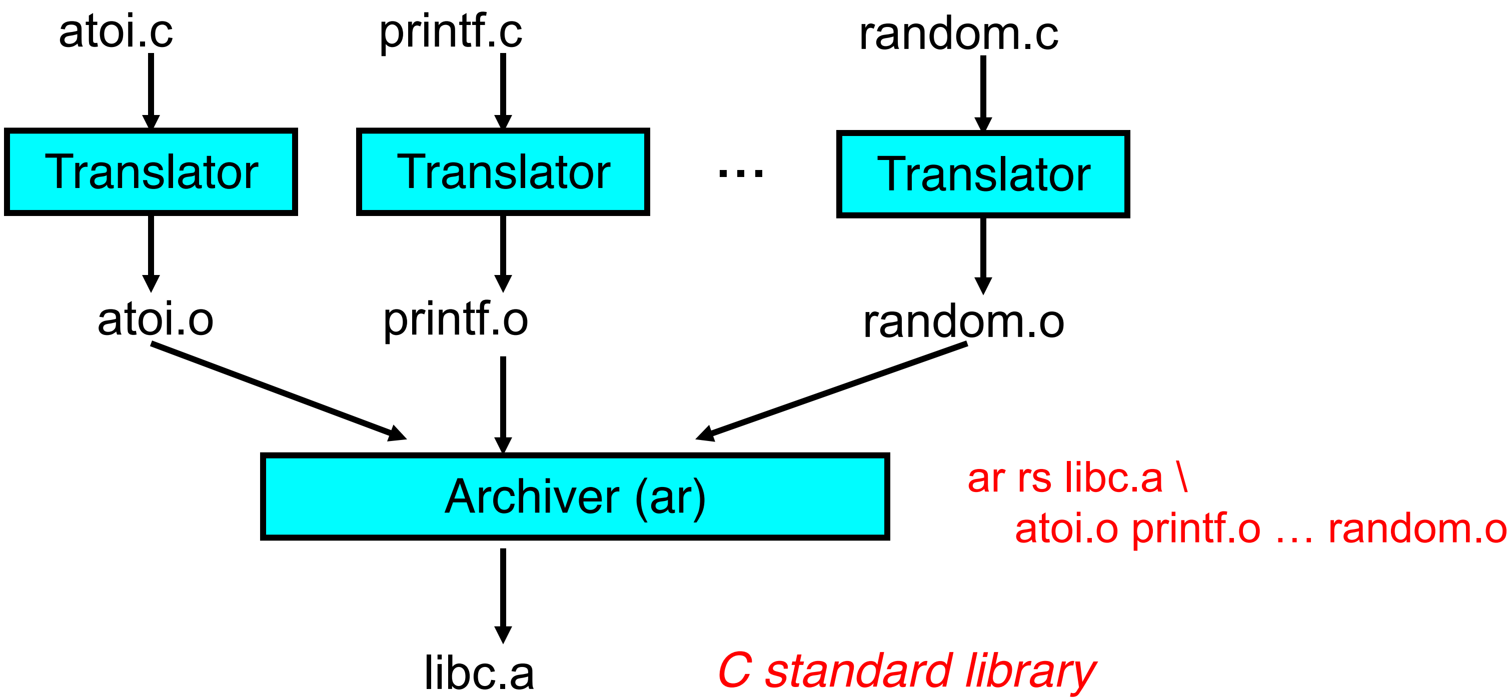
Further improves modularity and efficiency by packaging commonly used functions [e.g., C standard library (libc), math library (libm)]

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

How to link with static libraries

- Suppose have /some/path/to/libfun.a
  - Two options
- Specify library directly
  - **`gcc ... /some/path/to/libfun.a`**
- Use **`-L`** and **`-l`** switch
  - **`gcc ... -L/some/path/to -lfun`**
  - Driver adds 'lib[name].a'
  - -L must come before -l
- Example:
  - **`-liberty`** links in libiberty.a

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, date 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 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
  - Add any new external symbols not yet resolved to list
  - If any entries in the unresolved list at end of scan, then error.

Using Static Libraries (2)

- 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'
```

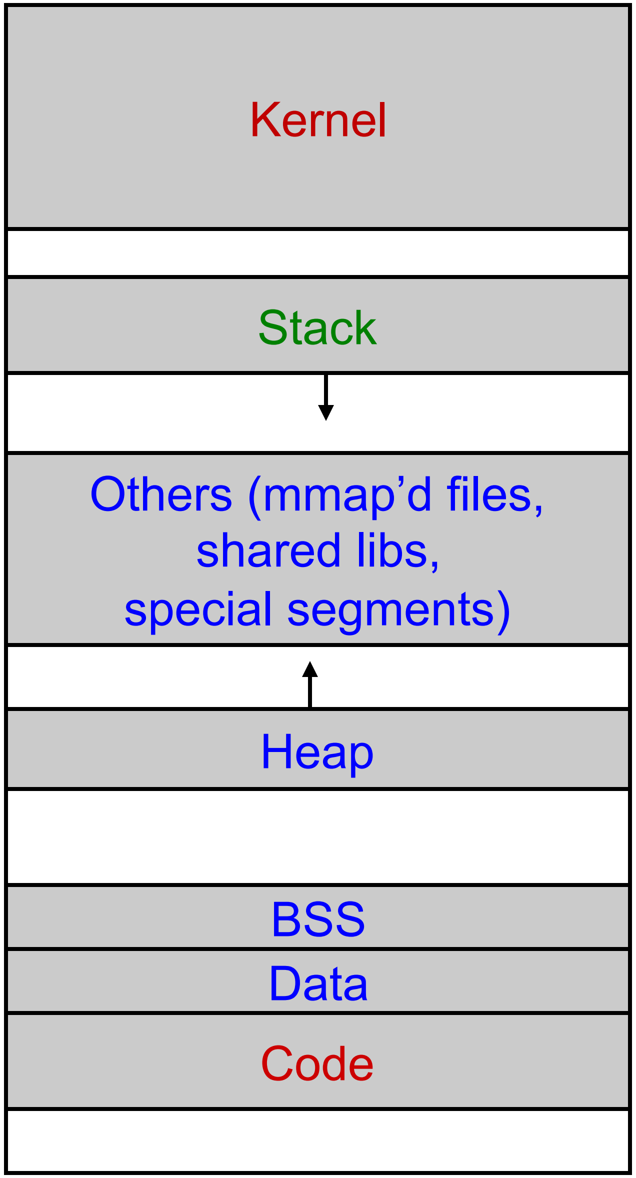
- Granularity of linking is .o file/module, not function
  - When designing libraries, place functions that may be used independently in different files
  - Otherwise, conflicts or unintended resolutions may result
- Hint: create a linker map!
  - gcc –Wl,Map –Wl,filename

Loading

- When starting a program, OS loads executable from disk and creates a new process
- Linker and loader cooperate:
  - Linker: creates ELF files
  - Loader: interprets them
- Process's information appears at virtual addresses
  - Start with segments defined by executable
  - More added as program executes

Virtual Address Space Layout in Linux for IA32

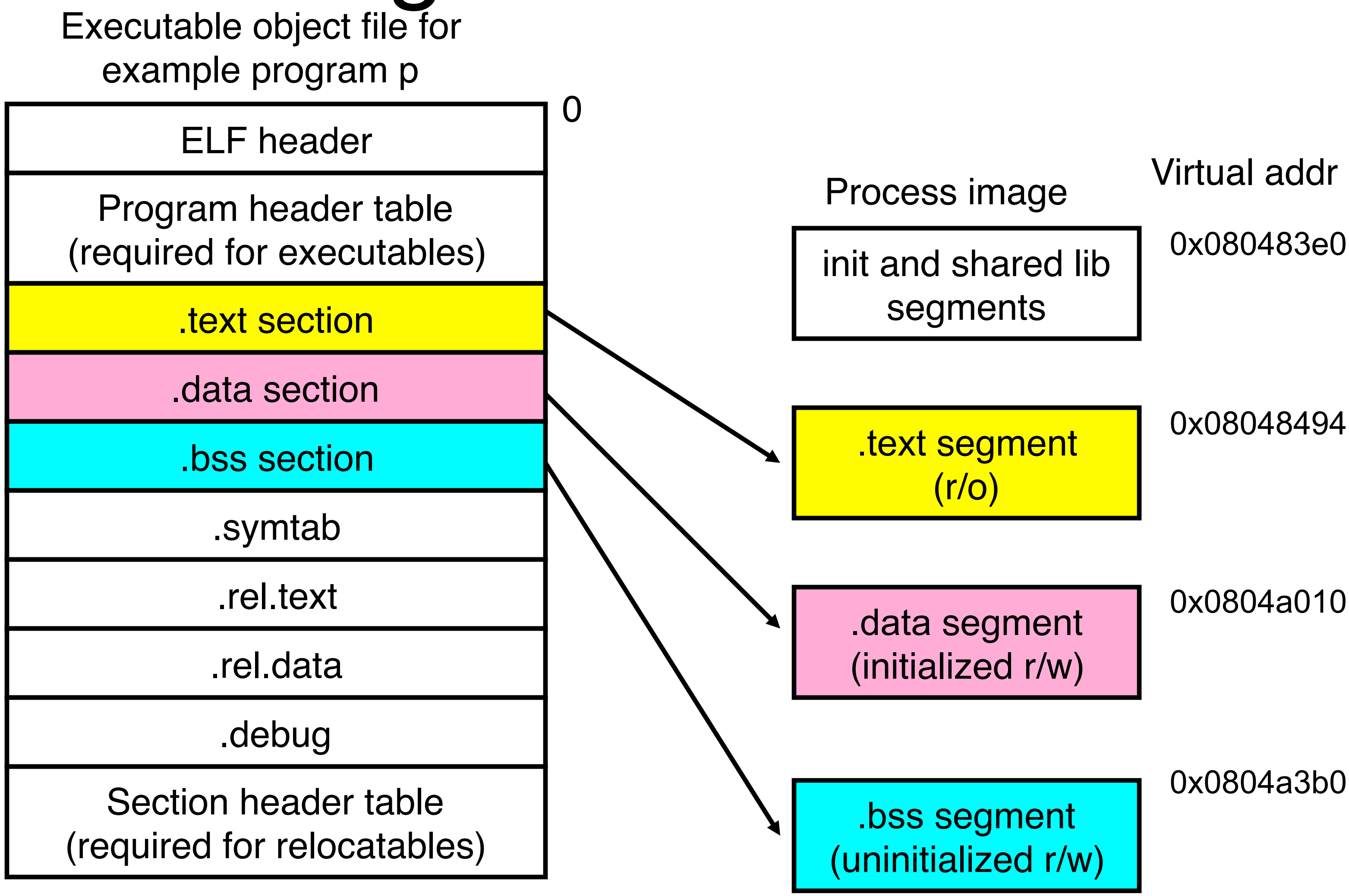
- All of this is by convention
  - Changing over time
- Recent systems randomize everything that is not fixed during static linking phase
  - Start of stack, heap, mmap, …
- Can tell linker to link at any address
  - Built-in instructions: ld --verbose



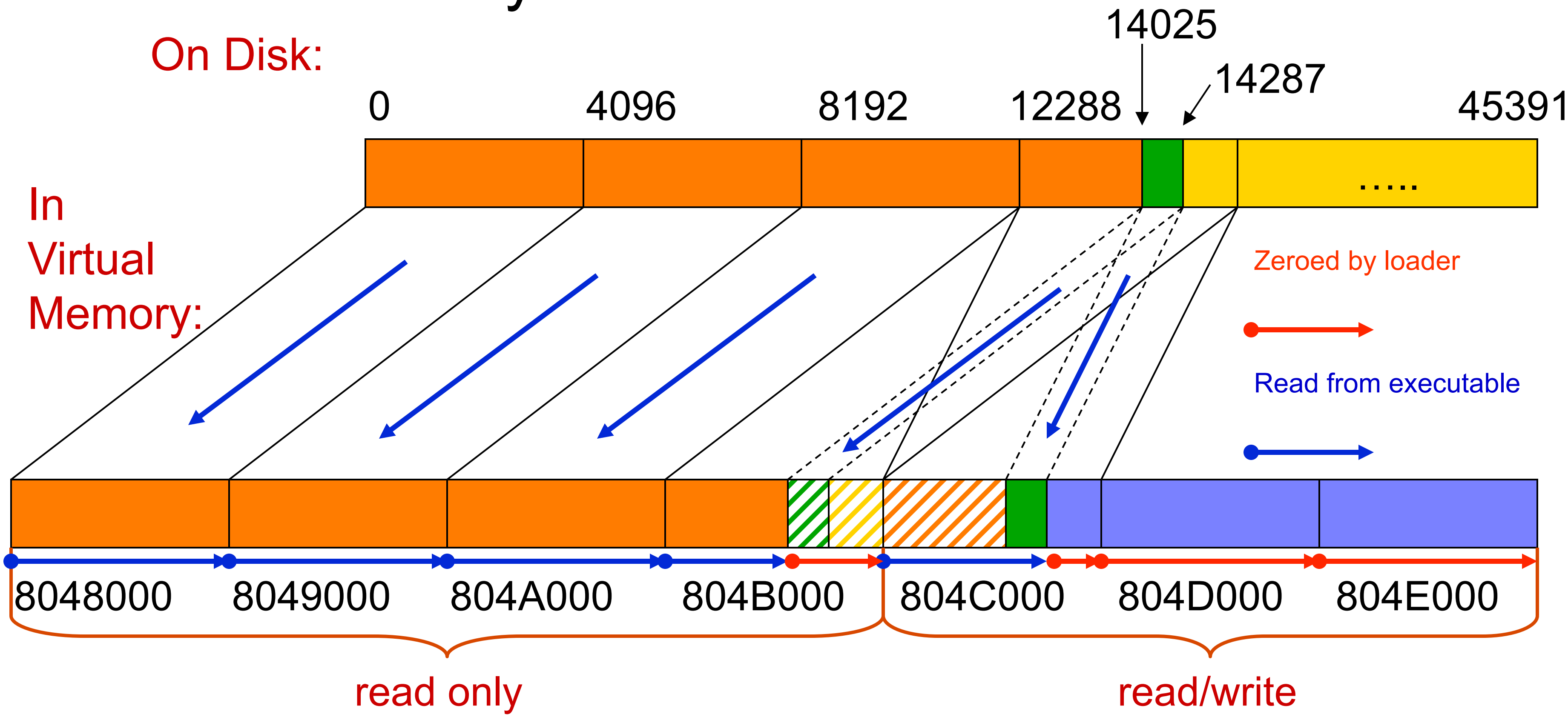
ASLR

- Address Space Layout Randomization
- Addresses known at link time facilitate certain types of attacks (e.g. buffer overflow, return-to-libc, etc.)
- Modern systems (since ca. 2004) aggressively randomize addresses of various segments (shared libs, heap, stack);
  - Some go further, i.e., OpenBSD: PIE
- Trade-off: costs vs. gain

Loading Executable Binaries



Aside: Anatomy of an Executable



Code and Read-Only Data

Other sections: Debugging Info, etc.

Initialized Global Data

Uninitialized Global Data (Must Zero)

"impurities" – the content of these areas is undefined as far as the program is concerned. For simplicity, they can be filled with data from the executable

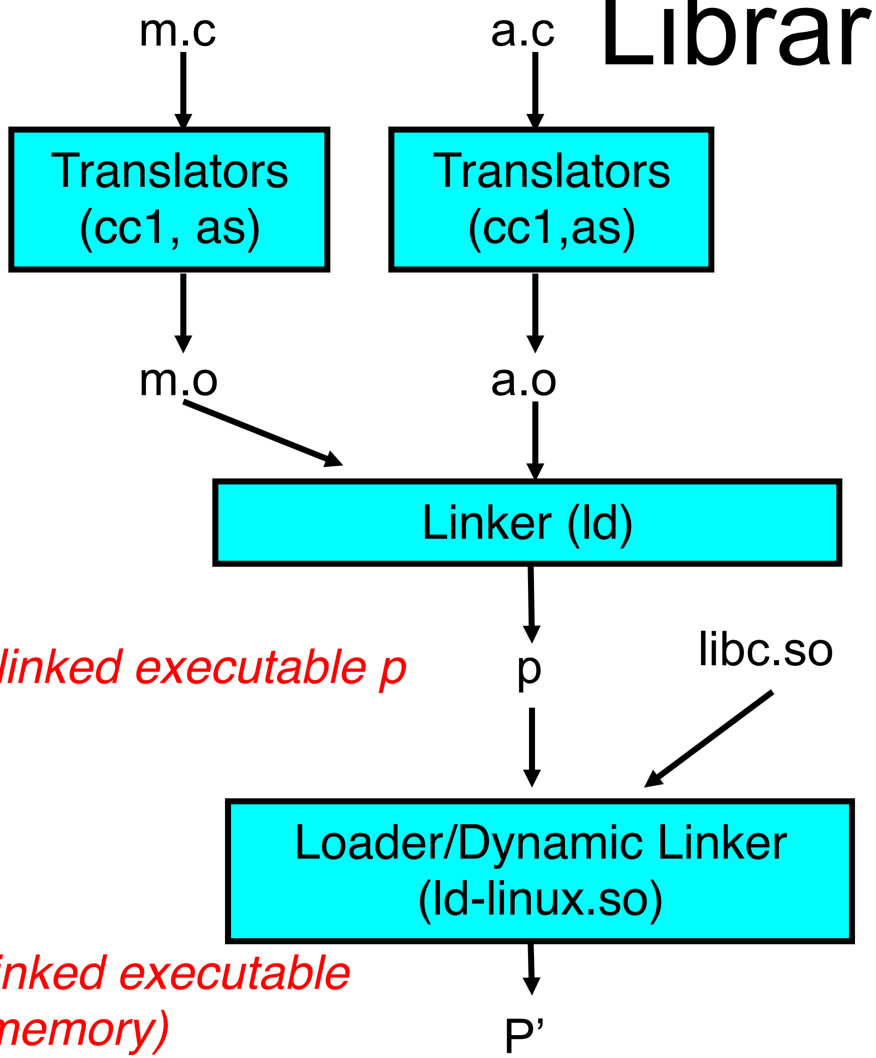
Drawbacks of Static Libraries

- 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

Shared Libraries

- 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.
    - Explore 'ldd' command
  - Dynamic linking can also occur after program has begun.
    - In Linux, this is done explicitly by user with dlopen()
    - Basis for many plug-in systems
  - Shared library routines can be shared by multiple processes.

Dynamically Linked Shared Libraries *(Simplified)*



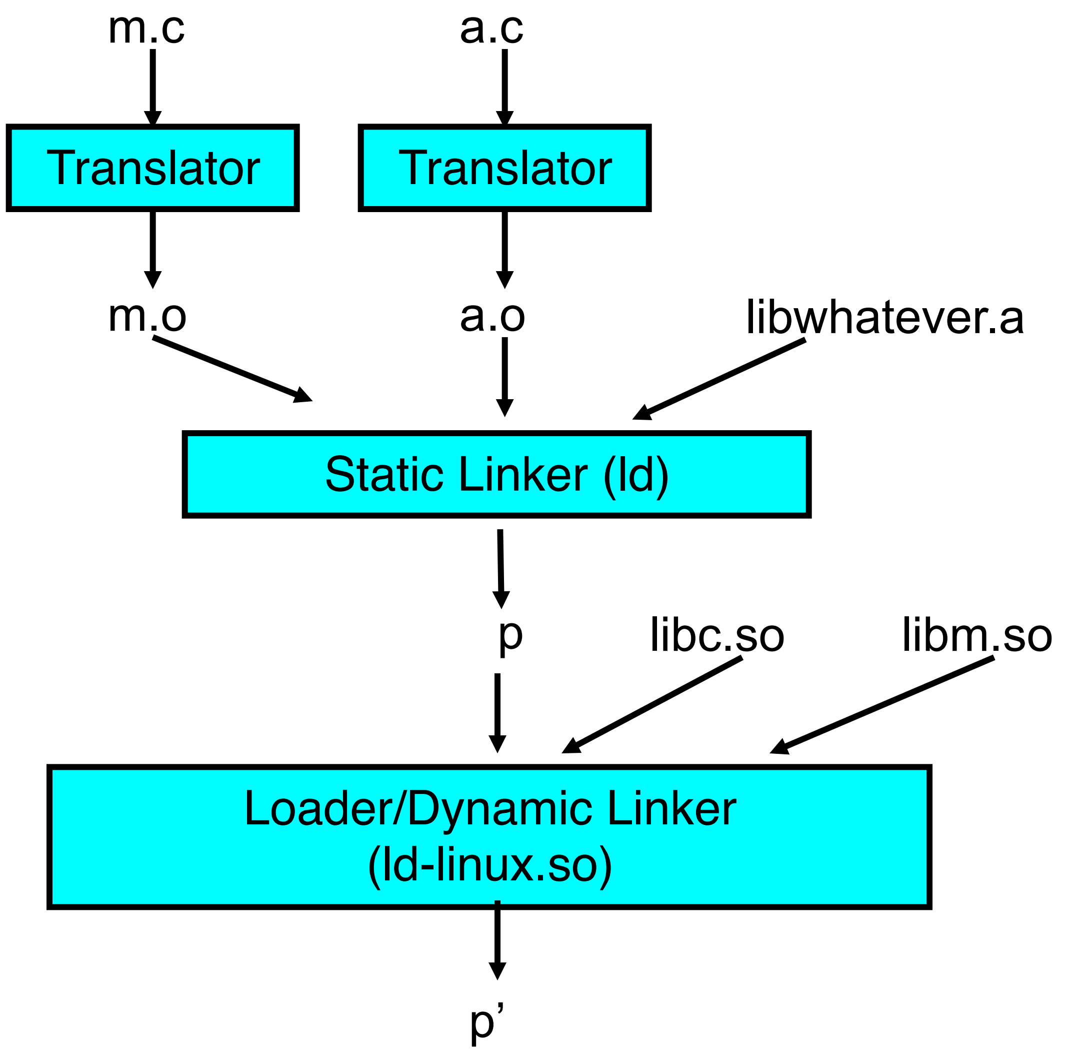
Partially linked executable p (on disk)

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)

A More Complete Picture



Shared Libraries

- Loaded on demand
  - Using a trampoline (next slide)
- Want to load shared libraries
- Goal: want to load shared libraries into multiple processes at different addresses
- Note:
  - Code segment should not change during dynamic linking
  - Each process needs own data segment
- Solution: Position-Independent Code

Use of Trampolines

```
int main()
{
    exit(printf("Hello, World\n"));
}
```

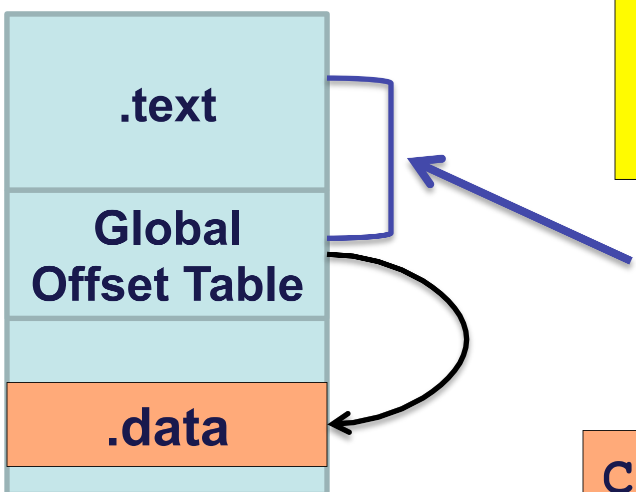
0x08

```
0x80482be <printf@plt+6>:        push   $0x10
0x80482c3 <printf@plt+11>:       jmp    0x8048288

0x8048288:      pushl  0x80495a8
0x804828e:      jmp    *0x80495ac

0x80495ac <_GLOBAL_OFFSET_TABLE_+8>:    0x003874c0

0x3874c0 <_dl_runtime_resolve>: push   %eax
```

0x
0x

Position-Independent Code

- How to encode accesses to global variables without requiring relocation?
  - Use indirection

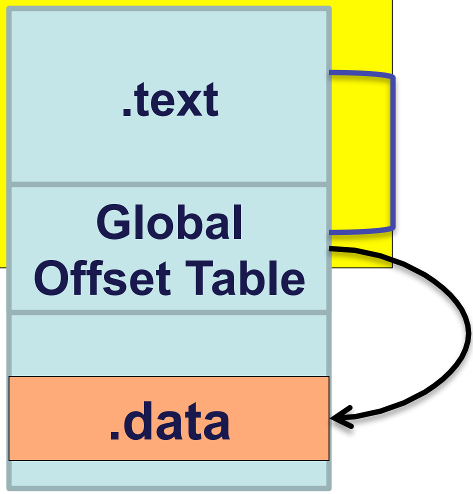



```
int x = 0xdeadbeef;

int getx() { return x; }
```

```
Contents of section .data:
 1530 efbeadde
```

Position-Independent Code (2)

```
000003bc <getx>:
 3bc:    55                    push   %ebp
 3bd:    89 e5                 mov    %esp,%ebp
 3bf:    e8 10 00 00 00        call   3d4 <__i686.get_pc_thunk.cx>
 3c4:    81 c1 58 11 00 00     add    $0x1158,%ecx
 3ca:    8b 81 f8 ff ff ff     mov    0xfffffff8(%ecx),%eax
 3d0:    8b 00                 mov    (%eax),%eax
 3d2:    5d                    pop    %ebp
 3d3:    c3                    ret

000003d4 <__i686.get_pc_thunk.cx>:
 3d4:    8b 0c 24              mov    (%esp),%ecx
 3d7:    c3                    ret
```



- All access to global variables are indirect

Relevant gcc switches

- -shared
  - build a shared object
- -fpic, -fPIC
  - Generate position-independent code

Dynamically loading objects at runtime

```
#include <dlfcn.h>

void *dlopen(const char *filename, int flag);
char *dlerror(void);
void *dlsym(void *handle, const char *symbol);
int dlclose(void *handle);
```

NB: each dynamically loaded module gets its own private link chain

Using link-time interposition

- LD_PRELOAD=./mylib.so
- Preloaded libraries can define symbols before libraries that would normally define are loaded
  - Redirection
- Can implement interposition by dynamically loading original libraries and forwarding the calls
  - Very powerful

Linking C with other languages

- Many scripting languages need to load and then call compiled functions
- Easy part:
  – Use dlopen() + dlsym() and call the function
- Harder part:
  – Access arguments compatibly & safely
- Each language has its own API
- Use SWIG – Simplified Wrapper and Interface Generator

SWIG

Embedding Interpreter: Python, Perl, Java, C#, Tcl, …

Glue Code/ Wrappers

(Unchanged) C Code

Shared library (.so, .dll)

- Wrapper manage parameter passing, type representations, etc.