CS 4400
Computer Systems

LECTURE 15

Static libraries

Relocation

Shared libraries and dynamic linking

Looking at Object File Symbol Tables

- Use the “nm” command
- Example

Static Libraries

- ***Static library***: A collection of .o files glued together
  - linker copies only the object modules that the program refs
  - C example: defs of `printf`, `strcpy`, and `rand` are in `libc.a`

```
> gcc main.c /usr/libm.a /usr/libc.a
```

Static Libraries

- Why not put all library functions in a single module?
- Why not put each library function in its own module?

Resolving References

- The linker scans relocatable objects ***left to right*** as they appear on the command line.
  - Compiler driver automatically translates any `.c` files to `.o` files

Resolving References

- During the scan, the linker maintains:
  - **E**: a set of relocatable objects to be merged
  - **U**: a set of unresolved symbols (referred to but not yet defined)
  - **D**, a set of symbols that have been defined previously
  - initially sets **E**, **U**, and **D** are empty

Scanning Input Object Files

- For each input object file or library file `f`
  - if `f` is an <u>object file</u>
    - add `f` to **E**
    - update **U** and **D** to reflect the symbols and references in `f`

E = {relocatable objects}
U = {unresolved symbols}
D = {symbols defined}

Scanning Input Object Files

- For each input object file or library file `f`
  - if `f` is an <u>object file</u>
    - add `f` to **E**
    - update **U** and **D** to reflect the symbols and references in `f`
  - if `f` is a <u>library</u>
    - if member `m` defines a symbol in **U**
      - add `m` to **E**
      - update **U** and **D** to reflect defs and refs in `m`
    - iterate over all members until **U** and **D** no longer change
    - Discard any objects from `f` not contained in **E**

E = {relocatable objects}
U = {unresolved symbols}
D = {symbols defined}

Scanning Input Object Files

- For each input object file or library file `f`
  - if `f` is an object file
    - add `f` to **E**
    - update **U** and **D** to reflect the symbols and references in `f`
  - if `f` is a library
    - if member `m` defines a symbol in **U**
      - add `m` to **E**
      - update **U** and **D** to reflect defs and refs in `m`
    - iterate over all members until **U** and **D** no longer change
    - Then discard any objects from `f` not contained in **E**
- If **U** is nonempty when linker finishes scanning, ERROR

E = {relocatable objects}
U = {unresolved symbols}
D = {symbols defined}

Example: Scanning Input Files

```
unix> gcc ./libvector.a main2.c
/tmp/cc9XH6Rp.o: In function `main':
/tmp/cc9XH6Rp.o(.text+0x18): undefined reference to `addvec'
```

- If the library (which defines a symbol) appears on the command line before the object file (which references the symbol), the reference cannot be resolved.

Example: Scanning Input Files

- Libraries can be repeated on the command line as needed to satisfy dependencies.
  - Suppose that `foo.c` calls a function in `libx.a` that calls a function in `liby.a` that calls a function in `libx.a`.

```
unix> gcc foo.c libx.a liby.a libx.a
```

Exercise: Scanning Input Files

- Let a → b denote that a depends on b
  - i.e., b defines a symbol that is referenced by a
- Give the minimal command line that will allow the static linker to resolve all symbol references.

`p.o` → `libx.a`

`p.o` → `libx.a` → `liby.a`

`p.o` → `libx.a` → `liby.a`
and `liby.a` → `libx.a` → `p.o`

Exercise: Scanning Input Files

- Let a → b denote that a depends on b
  - i.e., b defines a symbol that is referenced by a
- Give the minimal command line that will allow the static linker to resolve all symbol references.

`p.o` → `libx.a`	`> gcc p.o libx.a`
`p.o` → `libx.a` → `liby.a`	`> gcc p.o libx.a liby.a`
`p.o` → `libx.a` → `liby.a` and `liby.a` → `libx.a` → `p.o`	`> gcc p.o libx.a liby.a libx.a`

Relocation

- The linker merges the input modules and assigns run-time addresses to each symbol

Relocation

- Step 1: relocate sections and symbol definitions
  - merge all sections of the same type into a new aggregate section
  - assign run-time addresses to new aggregate sections
  - assign run-time addresses to each symbol defined

Relocation

- Step 2: relocate symbol references within sections
  - modify every symbol reference in bodies of the code and data sections so that they point to the correct run-time addresses
    - linker relies on relocation entries `.rel.text` and `.rel.data` to perform this step

Relocating Symbol References

```
typedef struct {
  int offset;     /* offset of ref to relocate */
  int symbol:24,  /* symbol ref should point to */
      type:8;     /* relocation type */
} Elf32_Rel;
```

format of ELF relocation entry

relocation algorithm

```
foreach section s
  foreach relocation entry r {
    refptr = s + r.offset; /* ptr to ref to be relocated */

    if(r.type == R_386_32) /* relocate an absolute addr */
      *refptr = (unsigned) (ADDR(r.symbol) + *refptr);

    if(r.type == R_386_PC32) { /* relocate a PC-relative ref */
      refaddr = ADDR(s) + r.offset; /* ref's run-time addr */
      *refptr = (unsigned) (ADDR(r.symbol) + *refptr – refaddr);
    }
  }
```

<u>Assumptions</u>: `s` is an array of bytes, `r` has type `Elf32_Rel`, and linker has already chosen run-time addresses for each section (`ADDR(s)`) and each symbol (`ADDR(r.symbol)`)

Relocating Absolute References

```
int* bufp0 = &buf[0];
```

- `bufp0` will be:
  - stored in `.data` of `swap.o`
  - initialized to the address of a global array
- Thus, the value of `bufp0` must be relocated

Relocating Absolute References

```
int* bufp0 = &buf[0];
```

- `bufp0` will be:
  - stored in `.data` of `swap.o`
  - initialized to the address of a global array
- Thus, the value of `bufp0` must be relocated

```
00000000 <bufp0>:
  0: 00 00 00 00    int* bufp0 = &buf[0];
  0: R_386_32 buf   relocation entry
```

Disassembled listing of the `.data` *section (from* `swap.o`*)*

```
r.offset = 0
r.symbol = buf
r.type = R_386_32
```

Relocating Absolute References

`r.offset = 0`, `r.symbol = buf`, `r.type = R_386_32`

- Assume:

  `ADDR(buf) = 0x8049454`

Relocating Absolute References

`r.offset = 0`, `r.symbol = buf`, `r.type = R_386_32`

- Assume:

  `ADDR(buf) = 0x8049454`

- Linker updates the reference:

```
*refptr = (unsigned)(ADDR(r.symbol) + *refptr)
        = (unsigned)(0x8049454 + 0) = 0x8049454
```

Relocating Absolute References

`r.offset = 0`, `r.symbol = buf`, `r.type = R_386_32`

- Assume:

  ```
  ADDR(buf) = 0x8049454
  ```

- Linker updates the reference:

  ```
  *refptr = (unsigned)(ADDR(r.symbol) + *refptr)
          = (unsigned)(0x8049454 + 0) = 0x8049454
  ```

- Linker decides that at run time `bufp0` will be located at `0x804945c` and will be initialized to `0x8049454`, the run-time address of the `buf` array.

```
0804945c <bufp0>:
 804945c: 54 94 04 08
```

disassembled `.data` *listing (from executable object file)*

Relocating PC-Relative References

opcode

reference (-4) biased because PC always points to next instruction

```
6: e8 fc ff ff ff   call 7 <main+0x7>   swap();
7: R_386_PC32 swap                      relocation entry
```

disassembled `call` *instruction (from* `main.o`*)*

```
r.offset = 7
r.symbol = swap
r.type = R_386_PC32
```

Relocating PC-Relative References

`r.offset = 7`, `r.symbol = swap`, `r.type = R_386_PC32`

- Assume:

```
ADDR(.text)  = 0x80483b4
ADDR(swap)   = 0x80483c8
```

Relocating PC-Relative References

`r.offset = 7`, `r.symbol = swap`, `r.type = R_386_PC32`

- Assume:

```
ADDR(.text)  = 0x80483b4
ADDR(swap)   = 0x80483c8
```

- First, linker computes run-time address of the reference:

```
refaddr = ADDR(s) + r.offset
        = 0x80483b4 + 0x7
        = 0x80483bb
```

Relocating PC-Relative References

`r.offset = 7`, `r.symbol = swap`, `r.type = R_386_PC32`

- Assume:
```
ADDR(.text)  = 0x80483b4
ADDR(swap)   = 0x80483c8
```
- First, linker computes run-time address of the reference:
```
refaddr = ADDR(s) + r.offset
        = 0x80483b4 + 0x7
        = 0x80483bb
```
- Then, linker updates the reference from its current value (-4) so that it will point to the swap routine at run time:
```
*refptr = (unsigned)(ADDR(r.symbol) + *refptr – refaddr)
        = (unsigned)(0x80483c8 + (-4) – 0x80483bb)
        = 0x9
```

Relocating PC-Relative References

`r.offset = 7`, `r.symbol = swap`, `r.type = R_386_PC32`

- Assume:
```
ADDR(.text)  = 0x80483b4
ADDR(swap)   = 0x80483c8
```

- First, linker computes run-time address of the reference:
```
refaddr = ADDR(s) + r.offset
        = 0x80483b4 + 0x7
        = 0x80483bb
```

- Then, linker updates the reference from its current value (-4) so that it will point to the swap routine at run time:
```
*refptr = (unsigned)(ADDR(r.symbol) + *refptr – refaddr)
        = (unsigned)(0x80483c8 + (-4) – 0x80483bb)
        = 0x9
```

```
80483ba: e8 09 00 00 00 call 80483c8 <swap>
```

disassembled `call` *instruction (from executable object file)*

ELF Executable Object File Format

ELF header	describes overall format, includes entry point (addr of 1st instruction)
Segment header table	maps contiguous file sections to run-time memory sections
`.init`	describes function `_init`, to be called by program's initialization code
`.text`	read-only memory segment (code segment)
`.rodata`	
`.data`	read/write memory segment (data segment)
`.bss`	
`.symtab`	symbol table, debug info not loaded into memory
`.debug`	
`.line`	
`.strtab`	
section header table	**Notice the lack of `.rel.text` and `rel.data` sections. Why?

Loading Executable Object Files

```
$ ./p
```

- Because p is not a built-in shell command, the shell assumes that p is an executable object file.

Loading Executable Object Files

`$ ./p`

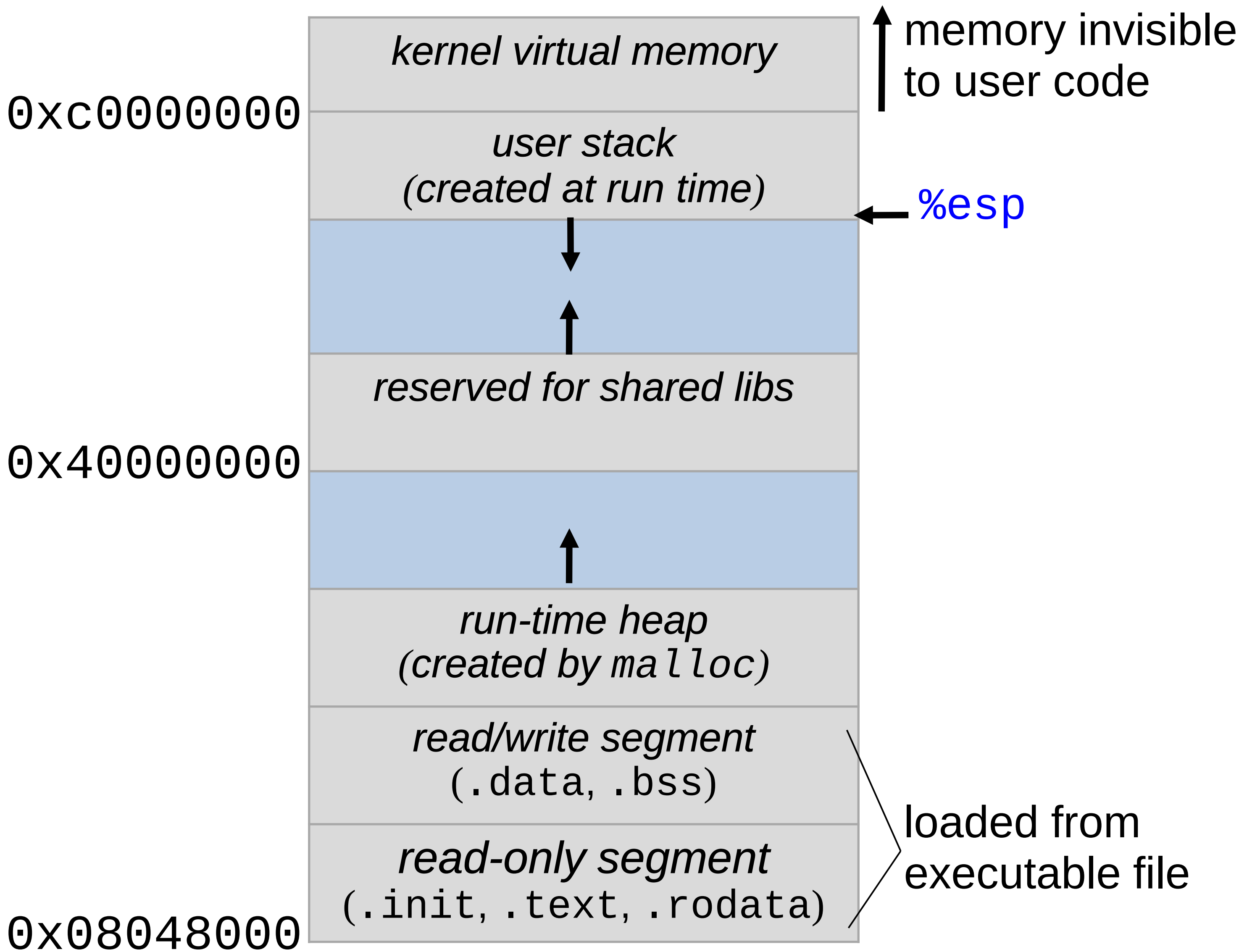
- Because `p` is not a built-in shell command, the shell assumes that `p` is an executable object file.
- The shell invokes loader (by calling function `execve`)
  - copy the code and data from `p` into memory and
  - run the program by jumping to the "entry point"

Loading Executable Object Files

```
$ ./p
```

- Because `p` is not a built-in shell command, the shell assumes that `p` is an executable object file.
- The shell invokes loader (by calling function `execve`)
  – copy the code and data from `p` into memory and
  – run the program by jumping to the “entry point”
- When the loader runs, it creates a memory image and copies chunks of the executable into the code and data segments.

Unix Run-Time Memory Image



Shared Libraries

- Static libraries sometimes need to be updated
  - Bummer because all programs using the lib need to re-link
- Almost all C programs reference standard I/O functions, and code for these functions appears in the text segment of nearly every running program—waste of memory.

Shared Libraries

- ***Shared library***—an object module that can be loaded and linked with a program in memory, all at load or run time.
  - The process of linking a shared library is called dynamic linking.
- AKA shared objects (.so Unix suffix, DLLs on Microsoft)

Dynamic Linking

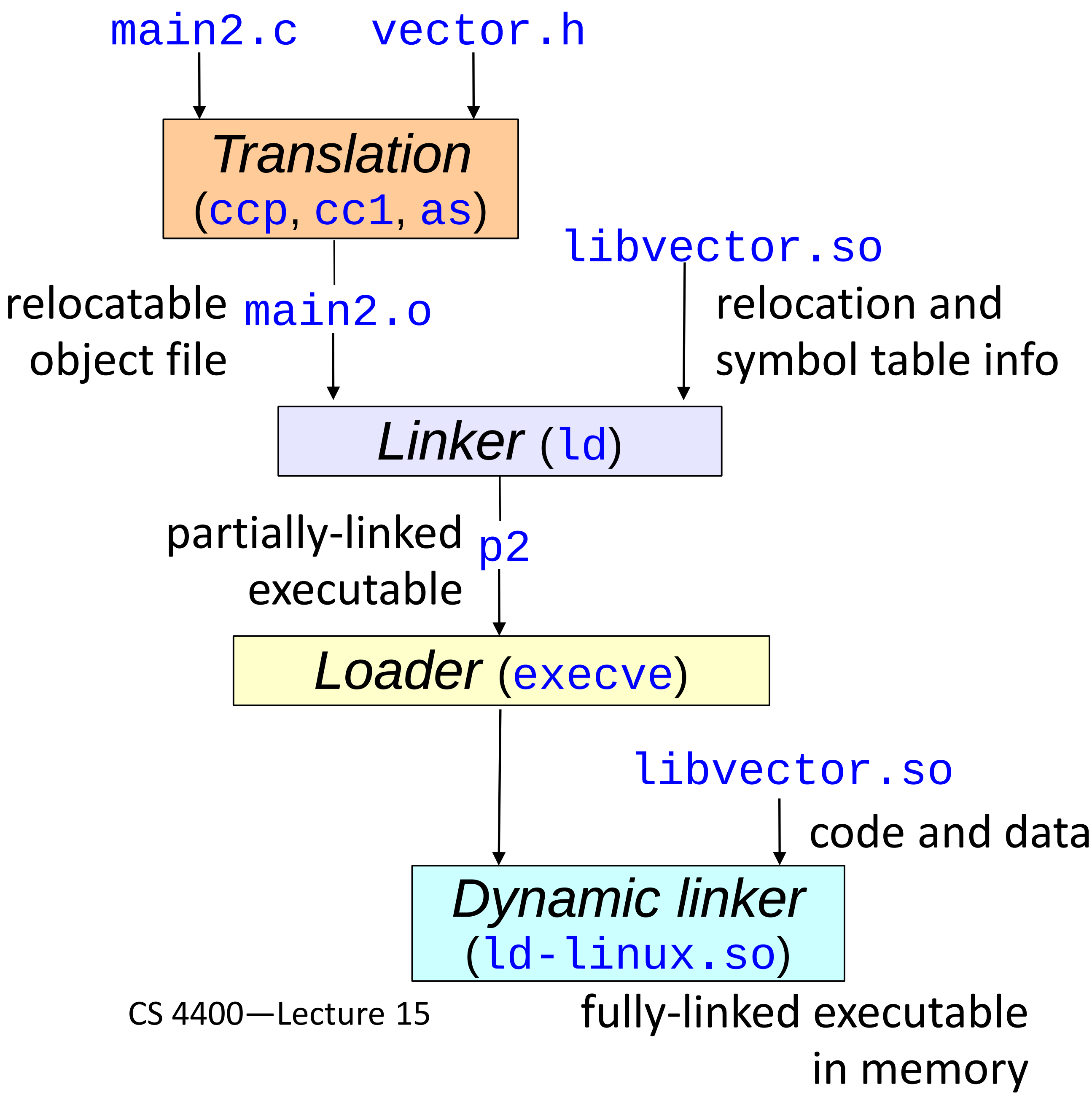
- Shared libraries must be compiled as "position independent code" using the `-fpic` compiler option
  - PIC uses a dedicated relocation register to figure out where things are
  - Slight loss of performance

```
unix> gcc -o p2 main2.c ./libvector.so
```

  - creates `p2` in a form to be linked with `libvector.so` at load time

- Does some of the linking statically and then completes linking process when the program is loaded.

Dynamic Linking w/ Shared Libs



- None of code and data from `libvector.so` is copied into `p2`
- Instead it copies only some relocation and symbol table info to allow references to be resolved at run time.
- Loader notices an `.interp` section with the dynamic linker's path. It passes control there to finish linking. Then passes control to the program.

Run-Time Loading and Linking

- A program requests that the dynamic linker load and link shared libraries while the program is running
  - without having to (partially) link in the libraries at compile time
- Microsoft uses shared libraries to distribute SW updates
  - users download updates and the next time their application runs, it will automatically link and load the new shared library
- Web servers generate dynamic content
  - the appropriate function is loaded/linked at run time
- (See the text for a discussion of the simple interface for the dynamic linker that is provided on Unix systems.)

Summary

- Linkers manipulate object files at compile time (relocatable, static linking), load time or run time (shared libraries, dynamic linking)

- Two main tasks: symbol resolution and relocation
  - each global symbol in an object file is bound to a unique definition
  - the ultimate memory address for each symbol is determined

Summary

- Static linkers combine multiple relocatable object files into a single executable object file at compile time.
- Dynamic linkers are invoked at load or run time to resolve references in user code with definitions in shared libraries.

Summary

- The left-to-right scan of input object files can also be confusing
- The rules linkers use for silently resolving multiple definitions can introduce subtle bugs