CS 4400
Computer Systems

LECTURE 15

Static libraries

Relocation

Shared libraries and dynamic linking

Review

- What is linking? when does it happen?
- What is an object file? types? sections?
- What is a symbol? types? symbol resolution?
- What is contained in the symbol table?
- What if there are multiple definitions of a global symbol?

Static Libraries

- *Static library*—a collection of related object modules.
  - linker copies *only* the object modules that the program refs
  - *C example*: defs of `printf`, `strcpy`, `rand` are in `libc.a`
  - > `gcc main.c /usr/libm.a /usr/libc.a`
- Why doesn't the compiler recognize calls to standard functions and generate the appropriate code directly?
- Why not put all standard functions in a single module?
- Why not put each standard function in its own module?
- (See the text for how to create a static library.)

Resolving References

- The way in which the Unix linker uses static libraries to resolve external refs can be a source of confusion.
- During symbol resolution, the linker scans relocatable objects *left to right* as they appear on the command line.
    - driver automatically translates any `.c` files to `.o` files
- During the scan, the linker maintains
    - *E*, a set of relocatable object files to be merged into executable
    - *U*, a set of unresolved symbols (referred to but not yet defined)
    - *D*, a set of symbols that have been defined (in previous files)
    - initially sets *E*, *U*, and *D* are empty

Scanning Input Object Files

For each input file *f*,

- if *f* is an object file: add *f* to *E* and update *U* and *D* to reflect the symbol definitions and references in *f*
- if *f* is a library: if member *m* defines a symbol in *U*, add *m* to *E* and 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 member object files not contained in *E*
- if *U* is nonempty when linker finishes scanning, ERROR

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.
- 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`

Relocation

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

- *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

- *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

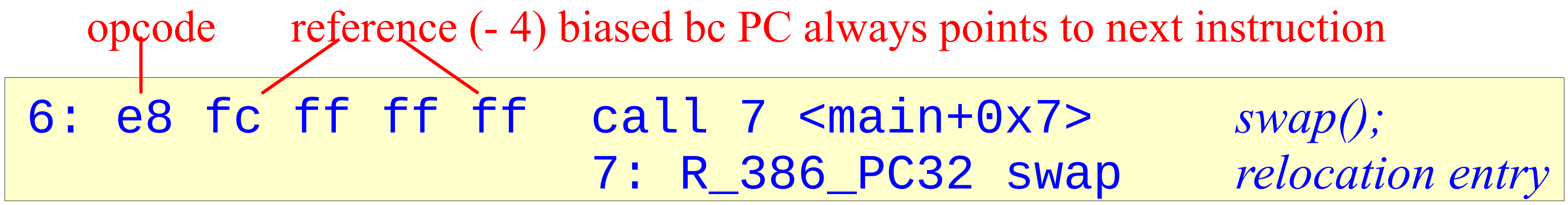
```
foreach section s
  foreach relocation entry r {
    refptr = s + r.offset; /* ptr to ref to be relocated */

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

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

Assume: `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 PC-Relative References



opcode   reference (- 4) biased bc 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`

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)*

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

Assume: ADDR(buf) = 0x8049454

Linker updates the reference:

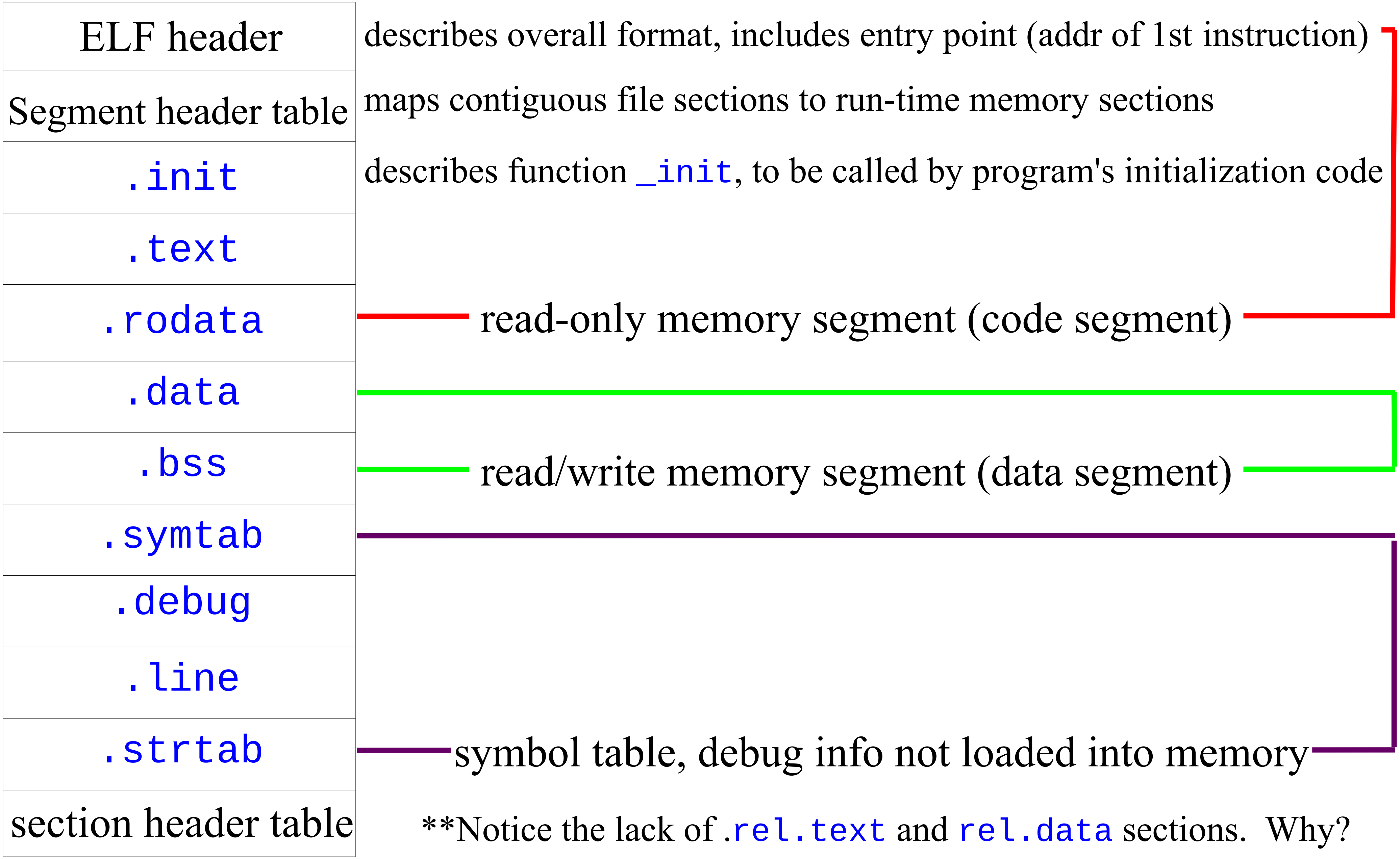
```
    *refptr = (unsigned)(ADDR(r.symbol) + *refptr)
            = (unsigned)(0x8049454 + 0) = 0x8049454
```

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)*

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

.rodata

read-only memory segment (code segment)

.data

.bss

read/write memory segment (data segment)

.symtab

.debug

.line

.strtab

symbol table, debug info not loaded into memory

section header table

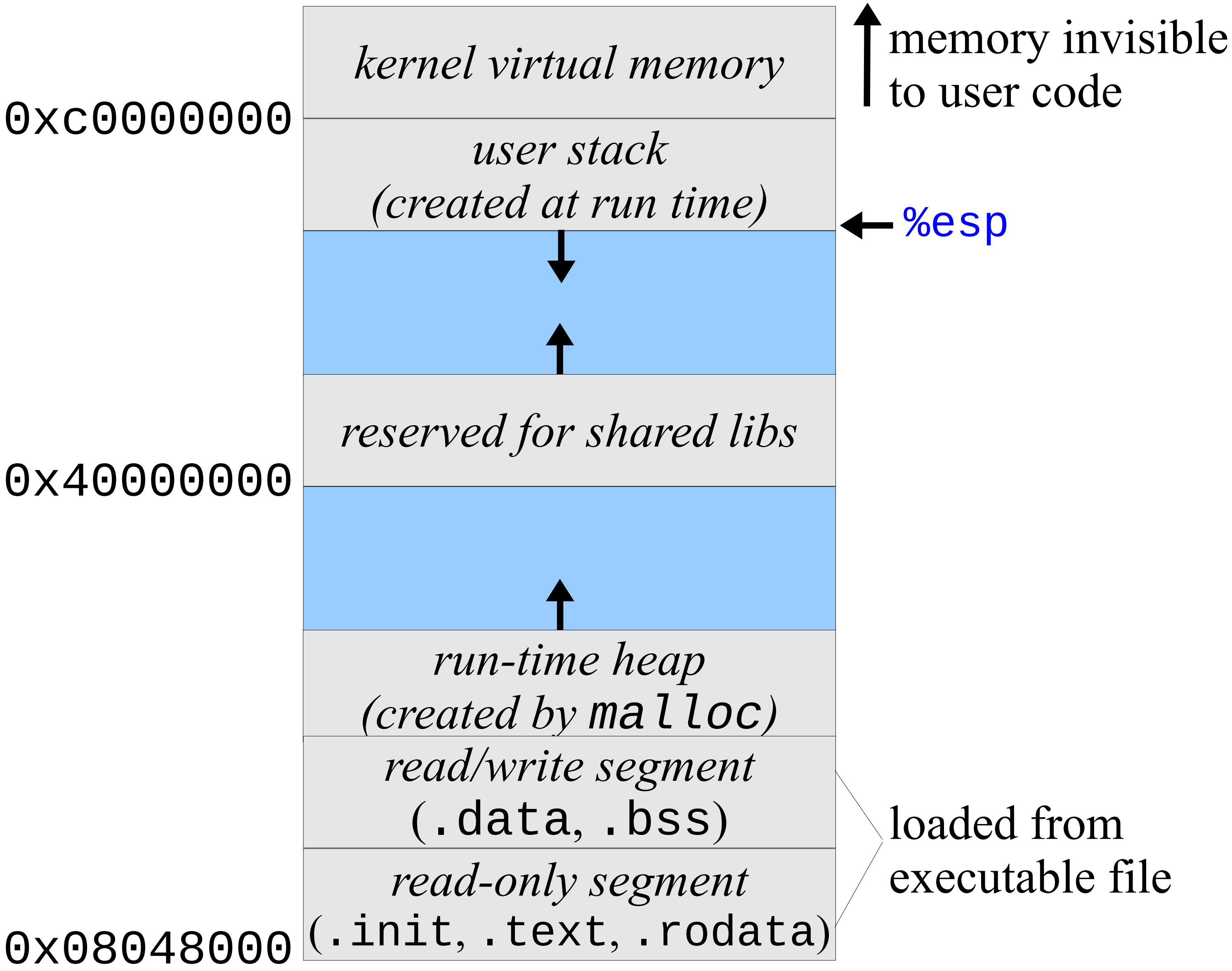
**Notice the lack of .rel.text and rel.data sections. Why?

Loading Executable Object Files

```
unix> ./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`) to
  - 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 (next slide) and copies chunks of the executable into the code and data segments.

Unix Run-Time Memory Image



kernel virtual memory
memory invisible to user code
0xc0000000
user stack (created at run time)
%esp
reserved for shared libs
0x40000000
run-time heap (created by malloc)
read/write segment (.data, .bss)
loaded from executable file
read-only segment (.init, .text, .rodata)
0x08048000

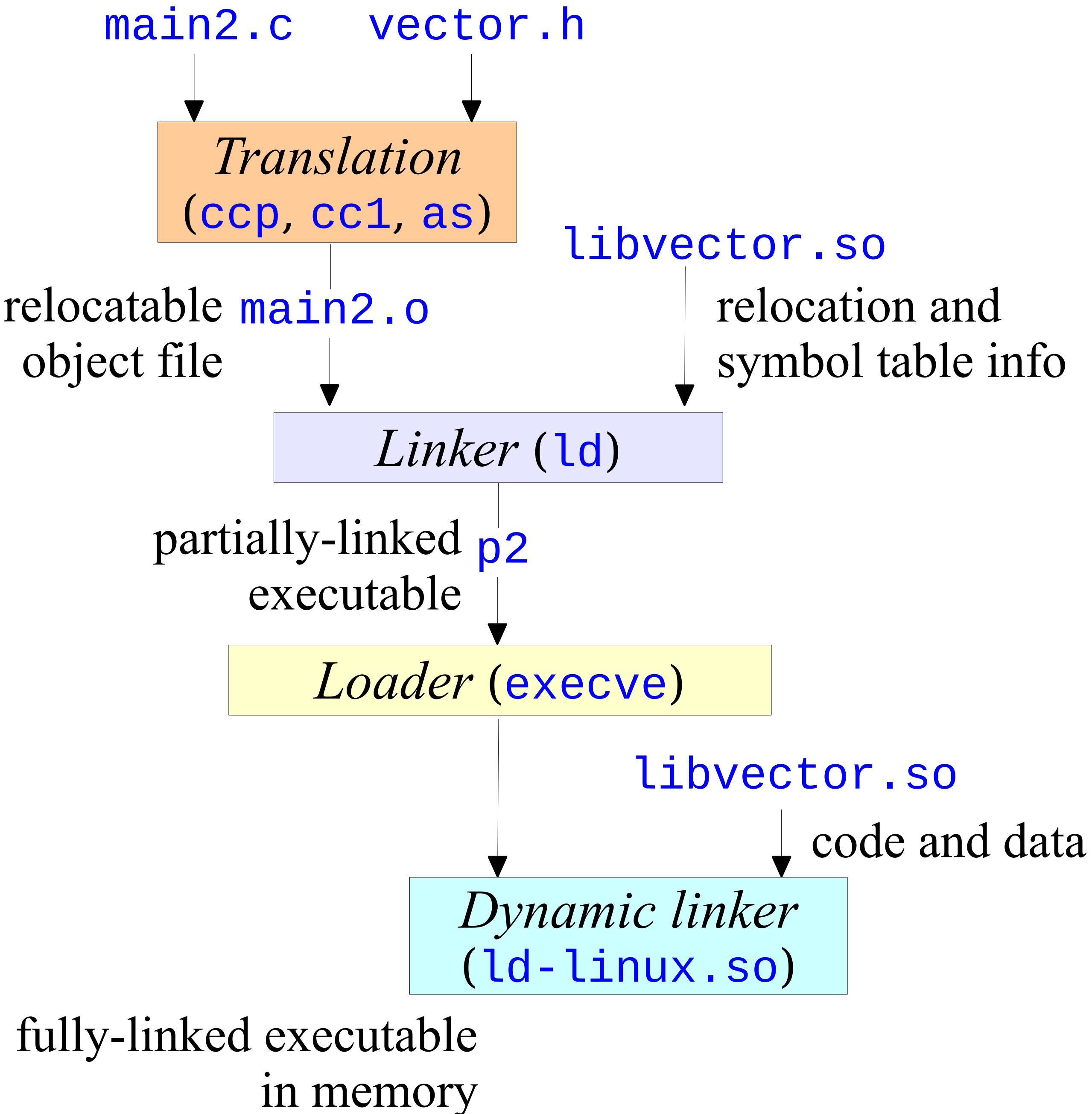
Shared Libraries

- Static libraries must be updated periodically.
  - programmer must be aware of change and explicitly relink
- 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 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

- Why “shared”?
  - The code and data in exactly one `.so` file is shared by all executable object files that reference the library. How is this different from static libraries?
  - A single copy of a shared library's `.text` section in memory can be shared by different running processes.
- `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



main2.c vector.h

Translation (ccp, cc1, as)

relocatable main2.o object file

libvector.so

relocation and symbol table info

Linker (ld)

partially-linked p2 executable

Loader (execve)

libvector.so

code and data

Dynamic linker (ld-linux.so)

fully-linked executable in memory

- None of code and data from libvector.so is copied into p2, 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
- The rules linkers use for *silently* resolving multiple definitions can introduce subtle bugs.
- The left-to-right scan of input object files can also be confusing.
- 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.