Linking

Today

- Static linking
- Object files
- Static & dynamically linked libraries

Next time

- No time left!

Example C program

main.c

```
void swap();

int buf[2] = {1, 2};

int main()
{
  swap();
  return 0;
}
```

swap.c

```
extern int buf[];

int *bufp0 = &buf[0];
int *bufp1;

void swap()
{
  int temp;

  bufp1 = &buf[1];
  temp = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}
```

Building an executable from multiple files

- Programs are translated and linked using a compiler driver
- *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
- Eg: create executable `p` from `main.c` and `swap.c`:

```
unix> gcc -O2 -g -o p main.c swap.c
unix> ./p
```

Translating the example program

```
unix> gcc -O2 -v -o p main.c swap.c
Reading specs from /usr/lib/gcc/i386-redhat-linux/3.4.6/specs
Configured with: ../configure --prefix=/usr --mandir=/usr/share/man --infodir=/
usr/share/info --enable-shared --enable-threads=posix --disable-checking --with-
system-zlib --enable-__cxa_atexit --disable-libunwind-exceptions --enable-java-
awt=gtk --host=i386-redhat-linux
Thread model: posix
gcc version 3.4.6 20060404 (Red Hat 3.4.6-11)
 /usr/libexec/gcc/i386-redhat-linux/3.4.6/cc1 -quiet -v main.c -quiet -dumpbase
main.c -auxbase main -O2 -version -o /tmp/ccUck8xa.s
…
unix>
```

```
cpp [other args] main.c /tmp/main.i
```

```
cc1 /tmp/main.i main.c -O2 [other args] -o /tm/main.s
```

```
as [other args] -o /tmp/main.o /tmp/main.s
```

```
ls -o p [sys obj files and args] /tmp/main.o /tmp/
swap.o
```

Why linkers?

- Modularity
  - Large program can be written as a collection of smaller 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 re-link
    - No need to recompile other source files
  - Space:
    - Libraries of common functions can be put in a single file...
    - Yet executable files and running memory images contain only code for the functions they actually use

What does a linker do?

- Step 1: Symbol resolution
  - Programs define and reference symbols (variables and functions)

```
void swap() {…}   /* define symbol swap */
swap();           /* reference symbol swap */
int *xp = &x;     /* define xp, reference x */
```

  - Symbol definitions are stored (by compilers) in a *symbol table*
    - Symbol table is an array of structs
    - Each entry includes name, type, size, and location of symbol
  - Linker associates each symbol reference with exactly one symbol definition

What does a linker do?

- Step 2: Relocation
  - Merges separate code and data sections into single sections
  - Relocates symbols from their relative locations in the .o files to their final absolute memory locations in the executable
  - Updates all references to these symbols to reflect their new positions

Three kinds of object files

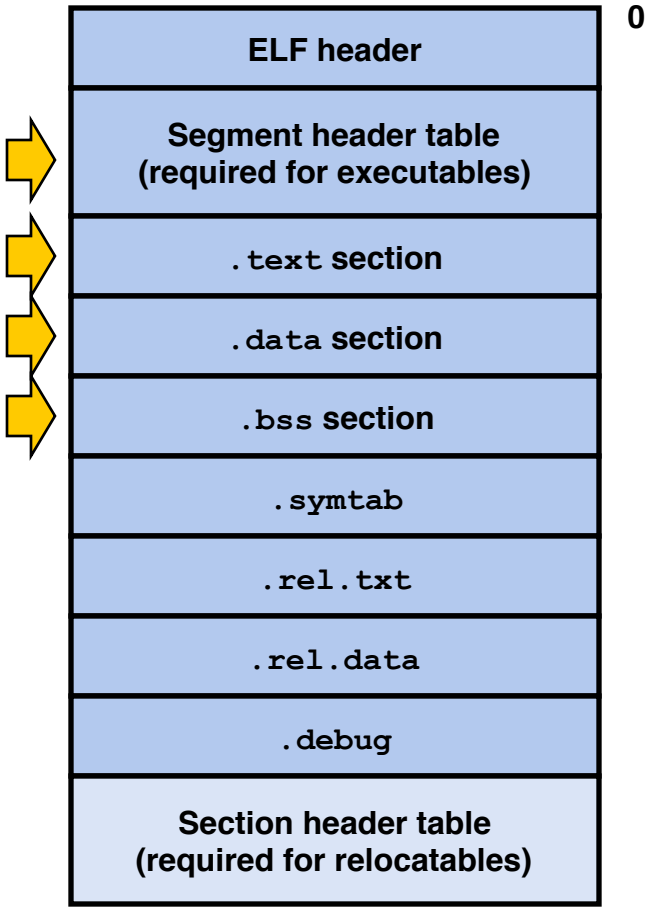
- Generated by compilers and assemblers
  - Relocatable object file
    - Contains code and data in a form that can be combined with other relocatable object files to form an executable
    - Each .o file is produced from exactly one source (.c) file
  - Shared object file
    - Special type of relocatable object file that can be loaded into memory and linked dynamically at either load or run time
    - Called Dynamic Link Libraries (DLLs) in Windows
- Generated by linkers
  - Executable object file
    - Contains code and data in a form that can be copied directly into memory and executed

Executable and Linkable Format (ELF)

- Standard binary format for object files
- Derives from AT&T System V Unix (Common Object File Format – COFF)
  - Later adopted by BSD Unix variants and Linux
- One unified format for
  - Executable object files
  - Relocatable object files (`.o`),
  - Shared object files (`.so`)
- Generic name: ELF binaries

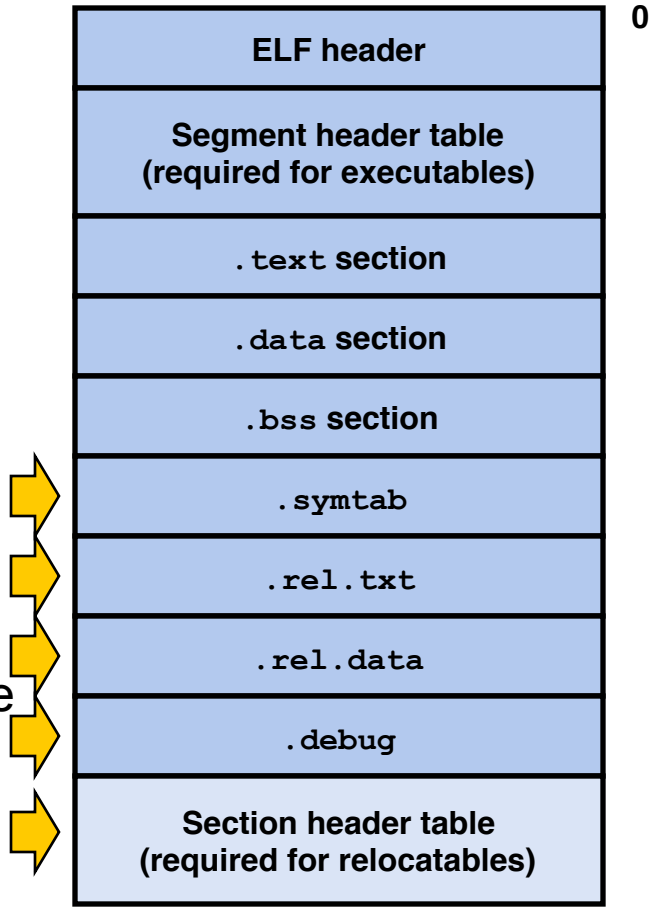
ELF object file format

- ELF header
  - Magic number, type (.o, exec, .so), machine, byte ordering, offset of section header table, etc.
- Segment header table
  - Page size, virtual addresses memory segments (sections), segment sizes.
- `.text` section
  - Code
- `.data` section
  - Initialized (static) data
- `.bss` section
  - Uninitialized (static) data
  - Originally an IBM 704 assembly instruction; “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`)
- Section header table
  - Offsets and sizes of each section



Linker symbols

- Every relocatable object module has a symbol table
  - Global symbols
    - Symbols defined by a module that can be referenced by other modules
    - E.g. non-static C functions and non-static global variables
  - External symbols
    - Global symbols that are referenced by a module but defined by some other module
  - Local symbols
    - Symbols that are defined and referenced exclusively by a module
    - E.g. C functions and variables defined with the static attribute
    - *Local linker symbols are not local program variables* (no symbols for local nonstatic program variables that are managed at runtime)

Global, external or local?

main.c

```
void swap();

int buf[2] = {1, 2};

int main()
{
  swap();
  return 0;
}
```

swap.c

```
extern int buf[];

int *bufp0 = &buf[0];
int *bufp1;

void swap()
{
  int temp;

  bufp1 = &buf[1];
  temp = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}
```

- In main.c
  - `buf` Global
  - `main` Global
  - `swap` External (def. by swap.c)
- In swap.c
  - `buf` External (def. by main.c)
  - `bufp0` / `bufpp1` Global
  - `swap` Global
  - `temp` *A local variable; not a local symbol*

Global: defined by module, use by others (non-static C functions & global vars)

External: referenced here, defined by others

Local: defined & ref exclusively here (static vars and C functions)

ELF symbol table example

ELF symbol

```
int name;         /* String table offset */
int value;        /* Section offset, or VM address */
int size;         /* Object size in bytes */
char type:4,      /* Data, func, section or src file name (4 bits) */
     binding:4;   /* Local or global (4 bits) */
char reserved;    /* Unused */
char section;     /* Section header index, ABS, UNDEF or COMMON */
```

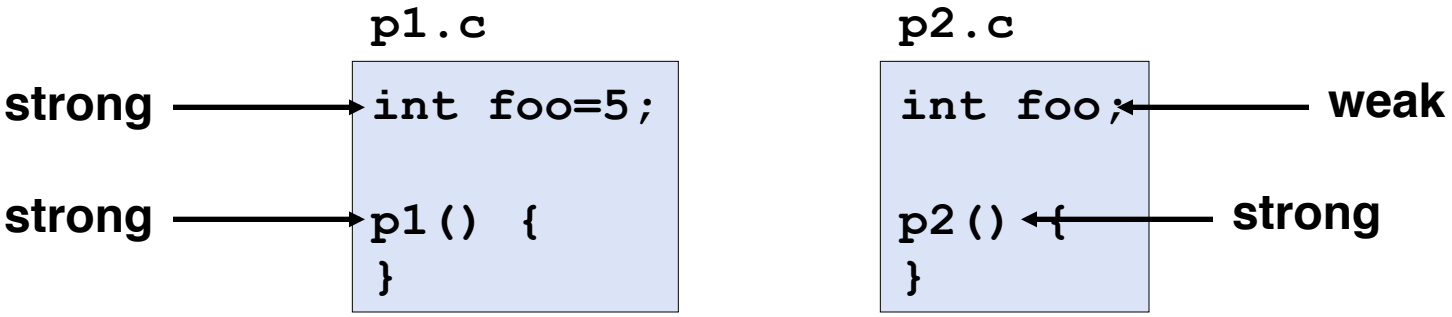
```
unix% readelf -s main.o

Symbol table '.symtab' contains 10 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 FILE    LOCAL  DEFAULT  ABS main.c
     2: 00000000     0 SECTION LOCAL  DEFAULT    1
     3: 00000000     0 SECTION LOCAL  DEFAULT    3
     4: 00000000     0 SECTION LOCAL  DEFAULT    4
     5: 00000000     0 SECTION LOCAL  DEFAULT    6
     6: 00000000     0 SECTION LOCAL  DEFAULT    5
     7: 00000000    20 FUNC    GLOBAL DEFAULT    1 main
     8: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND swap
     9: 00000000     8 OBJECT  GLOBAL DEFAULT    3 buf
```

Local symbols for use by the linker

Symbol resolution

- Linker associates each reference with exactly one symbol definition from symbol tables of its input files
  - Easy for references to local symbols
  - Trickier with global symbols – compiler assumes it is defined only once, *somewhere*, and that the linker will take care of it
    - If not anywhere, linker will complain
    - If more than once, maybe it will complain
- Program symbols are either strong or weak
  - *strong*: procedures and initialized globals
  - *weak*: uninitialized globals



Linker's symbol rules

- Rule 1. Multiple strong symbols are not allowed
  - Each item can be defined only once
  - Otherwise: linker error
- 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.
  - Can override with `gcc -fno-common`

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.

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: Have the compiler generate the code (Pascal)
    - More complex compiler and a new version each time you add/ delete/modify a function
  - Option 2: Put all functions in a single source file
    - Programmers link big object file into their programs
    - Space and time inefficient
  - Option 3: 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

- Static libraries (`.a` archive files)
  - Concatenate related re-locatable 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

Commonly used libraries

- `libc.a` (the C standard library)
  - 5 MB archive of 1500 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 400 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
…
```

Creating static libraries

- To create the library

```
unix% gcc -c addvec.c mulvec.c
unix% ar rcs libvector.a addvec.o mulvec.o
```

libvector.a

```
void addvec(int *x, int *y,
            int *z, int n)
{
  int i;

  for (i = 0; i < n; i++)
    z[i] = x[i] + y[i];
}
```

```
void multvec(int *x, int *y,
             int *z, int n)
{
  int i;

  for (i = 0; i < n; i++)
    z[i] = x[i] * y[i];
}
```

main2.c

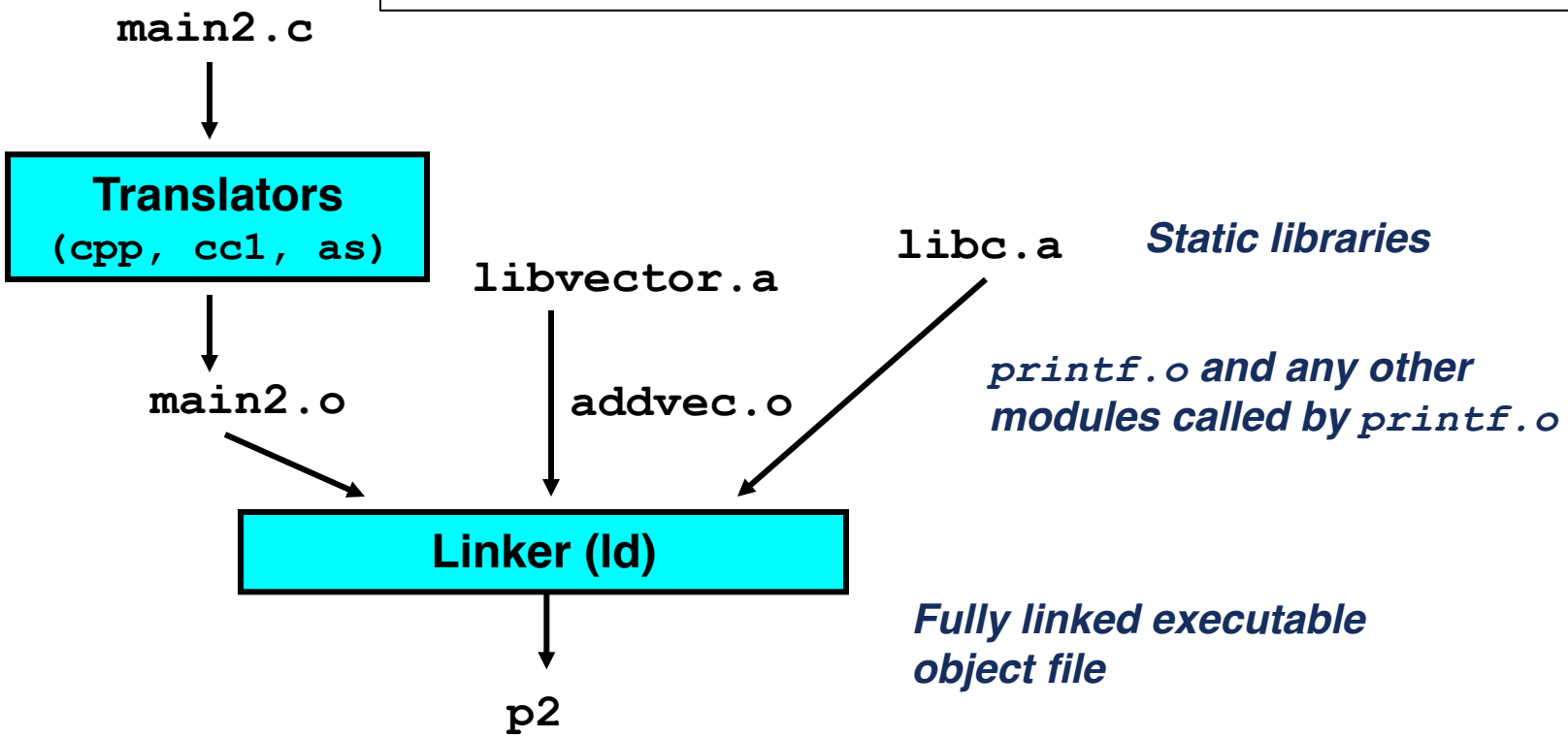
```
#include <stdio.h>
#include "vector.h"

int x[2] = {1, 2};
int y[2] = {3, 4};
int z[2];

int main()
{
  addvec(x, y, z, 2);
  printf("z = [%d %d]\n", z[0], z[1]);
  return 0;
}
```

Linking with static libraries

```
unix% gcc -O2 -c main2.c
unix% gcc -static -o p2 main2.o ./libvector.a
```



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 symbols in obj
  - If there are entries in the unresolved list at end, then error
- Problem:
  - Command line order matters!
  - Moral: put libraries at the end of the command line

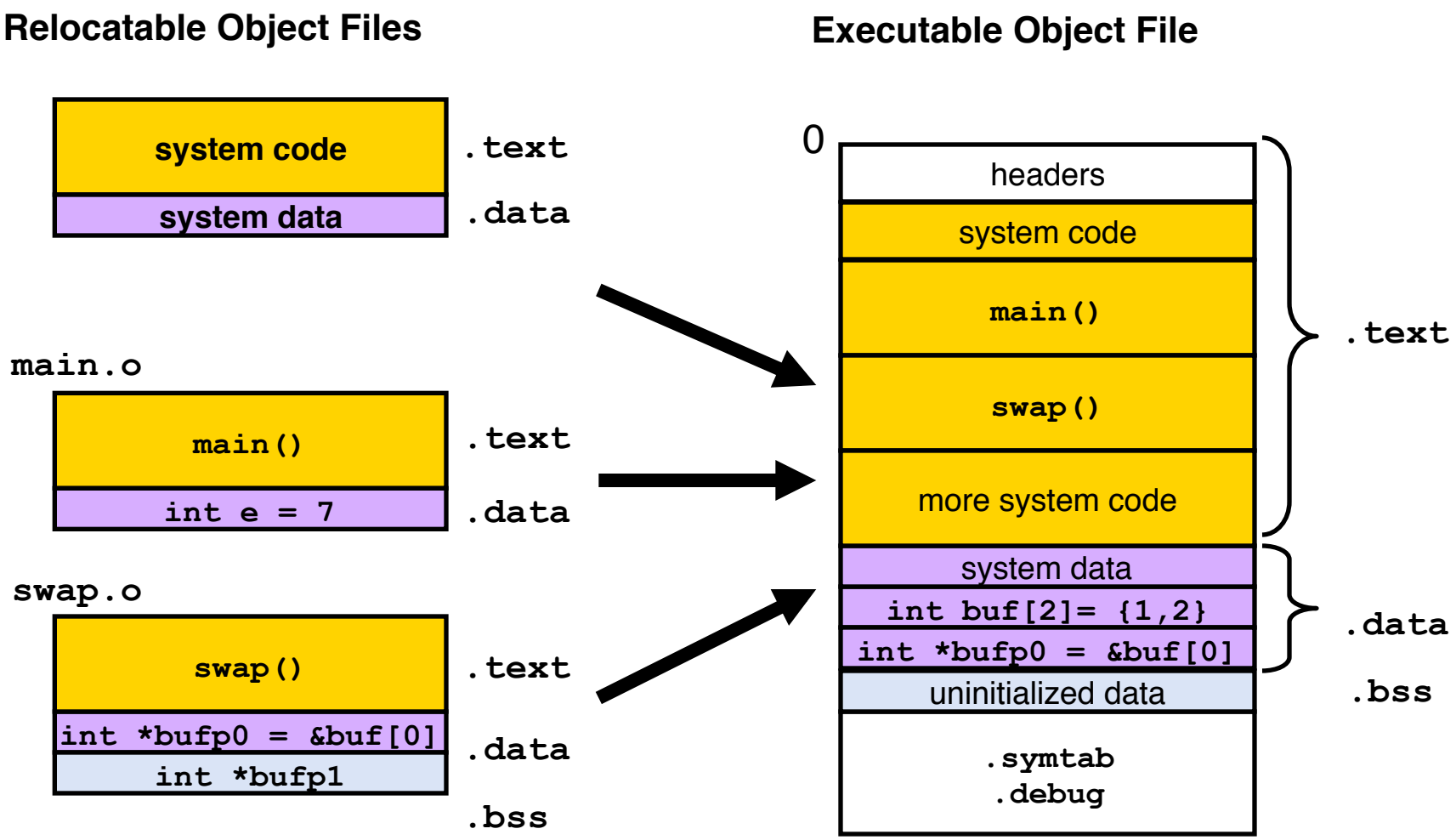
```
unix% gcc -static ./libvector.a main2.c
/tmp/ccC19pHI.o: In function `main':
main2.c:(.text+0x29): undefined reference to `addvec'
collect2: ld returned 1 exit status
```

Relocation

- Done with symbol resolution
  - Each symbol reference is associated with one definition
  - Every code and data sections of each module is known
- Relocation
  - Relocating sections and symbol definitions
    - Merge sections of same type and assign run-time addresses to each sections and symbols
  - Relocating symbol references within sections
    - Modify symbol reference to point to the right addresses
  - For relocation, assembler generates a relocation table showing how to modify references when merging
    - Relocation entries for code - .rel.text
    - Relocation entries for data - .rel.data

```
typedef struct {
   int offset;        /* Section offset of the ref to modif */
   int symbol: 24,    /* Symbol the ref should point to */
       type: 8;       /* How to modify it e.g. R_386_PC32 */
} Elf32_Rel;
```

Merging relocatable object files



Relocating symbol references

- Assume
  - Each section is an array of bytes
  - Each relocation entry is a struct as `Elf32_Rel`
  - Each section has a run-time address assigned – `ADDR(s)`
  - Each symbol has a run-time address – `ADDR(r.symbol)`

```
1. foreach section s {
2.    foreach relocation entry r {
3.       refptr = s + r.offset;  /* ptr to ref to be relocated) */
4.
5.       /* Relocate a PC-relative reference */
6.       if (r.type == R_386_PC32) {
7.          refaddr = ADDR(s) + r.offset; /* ref's run-time address */
8.          *refptr = (unsigned) (ADDR(r.symbol) + *refptr - refaddr);
9.       }
10.
11.      /* Relocate an absolute reference *.
12.      if (r.type == R_386_P2) {
13.         *refptr = (unsigned) (ADDR(r.symbol) + *refptr);
14.      }
15.  }
```

Relocation info (main)

- Relocating PC-relative references

main.c

```
void swap();

int buf[2] = {1, 2};

int main()
{
  swap();
  return 0;
}
```

```
Disassembly of section .text:

00000000 <main>:
   0:    55                      push   %ebp
   1:    89 e5                   mov    %esp,%ebp
   3:    83 e4 f0                and    $0xfffffff0,%esp
   6:    e8 fc ff ff ff          call   7 <main+0x7>
                        7: R_386_PC32        swap
   b:    31 c0                   xor    %eax,%eax
   d:    89 ec                   mov    %ebp,%esp
   f:    5d                      pop    %ebp
  10:    c3                      ret
```

Call instruction begins at section offset 0x6
Consists of 1-byte opcode (0xe8) and 32b ref 0xfffffffc (-4 decimal)

Relocation entry for the reference

```
r.offset = 0x7
r.symbol = swap
r.type = R_386_PC32
```

source: `objdump -dr`

Executable after relocation (.text)

```
foreach section s {
   foreach relocation entry r {
      refptr = s + r.offset;

      /* Relocate a PC-relative reference *.
      if (r.type == R_386_PC32) {
         refaddr = ADDR(s) + r.offset;
         *refptr = (unsigned) (ADDR(r.symbol) +
                *refptr - refaddr);
      …
```

```
r.offset = 0x7
r.symbol = swap
r.type = R_386_PC32

ADDR(s) = ADDR(.text) =
0x80483c0

ADDR(r.symbol) = ADDR(swap) =
0x80483e0
```

What to change `refaddr = 0x80483c0 + 0x7 = 0x80483c7`

What to change it to `*refptr = unsigned (0x80483e0 + (-4) - 0x80483c7) = 0x15`

```
080483c0 <main>:
 80483c0: 55                              push   %ebp
 80483c1: 89 e5                           mov    %esp,%ebp
 80483c3: 83 e4 f0                        and    $0xfffffff0,%esp
 80483c6: e8 15 00 00 00                  call   80483e0 <swap>
 80483cb: 31 c0                           xor    %eax,%eax
 80483cd: 89 ec                           mov    %ebp,%esp
 80483cf: 5d                              pop    %ebp
 80483d0: c3                              ret
 80483d1: 90                              nop
...
```

At run time `call` instruction is at `0x80483c8` and `PC` has `0x80483cb`;
To execute the call CPU

1. `push PC onto stack`
2. `PC <- PC+0x15 = 0x80483e0`

Relocation info (`swap, .text`)

- Relocating absolute references

swap.c

```
extern int buf[];

int *bufp0 = &buf[0];
int *bufp1;

void swap()
{
  int temp;

  bufp1 = &buf[1];
  temp = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}
```

```
00000000 <swap>:
   0:   a1 00 00 00 00              mov    0x0,%eax
                        1: R_386_32          bufp0
   5:   55                          push   %ebp
   6:   8b 0d 04 00 00 00           mov    0x4,%ecx
                        8: R_386_32          buf
   c:   89 e5                       mov    %esp,%ebp
   e:   c7 05 00 00 00 00 04        movl   $0x4,0x0
  15:   00 00 00
                        10: R_386_32         bufp1
                        14: R_386_32         buf
  18:   8b 10                       mov    (%eax),%edx
  1a:   89 08                       mov    %ecx,(%eax)
  1c:   5d                          pop    %ebp
  1d:   89 15 04 00 00 00           mov    %edx,0x4
                        1f: R_386_32         buf
  23:   c3                          ret
```

source: `objdump -dr`

Relocation info (swap, .data)

```
extern int buf[];

int *bufp0 = &buf[0];
int *bufp1;

void swap()
...
```

Because bufp0 is initialized it is stored in .data

```
Disassembly of section .data:

00000000 <bufp0>:
   0:     00 00 00 00                                  ....
                                0: R_386_32          buf
```

```
r.offset = 0x0          ADDR(r.symbol) = ADDR(buf) = 0x804a010
r.symbol = buf
r.type = R_386_32
```

```
foreach section s {
   foreach relocation entry r {
         refptr = s + r.offset;
         ...
         /* Relocate an absolute reference *.
         if (r.type == R_386_P2) {
             *refptr = (unsigned) (ADDR(r.symbol) + *refptr);
```

```
*refptr = unsigned (0x804a010 + 0) = 0x804a010
```

The run-time address of the buf array

source: `objdump -dr -j .data`

Executable after relocation (.data)

```
Disassembly of section .data:

0804a008 <__data_start>:
 804a008: 00 00                          add    %al,(%eax)
          ...

0804a
 804a                                                        ....

0804a010 <buf>:
 804a010: 01 00 00 00 02 00 00 00                            ........

0804a018 <bufp0>:
 804a018: 10 a0 04 08
........
```

The run-time address of the `buf` array

Executable after relocation (.text)

```
080483c0 <main>:
 80483c0: 55                            push   %ebp
 80483c1: 89 e5                         mov    %esp,%ebp
 80483c3: 83 e4 f0                      and    $0xfffffff0,%esp
 80483c6: e8 15 00 00 00                call   80483e0 <swap>
 80483cb: 31 c0                         xor    %eax,%eax
 80483cd: 89 ec                         mov    %ebp,%esp
 80483cf: 5d                            pop    %ebp
 80483d0: c3                            ret
 80483d1: 90                            nop
 ...
080483e0 <swap>:
 80483e0: a1 18 a0 04 08                mov    0x804a018,%eax
 80483e5: 55                            push   %ebp
 80483e6: 8b 0d 14 a0 04 08             mov    0x804a014,%ecx
 80483ec: 89 e5                         mov    %esp,%ebp
 80483ee: c7 05 24 a0 04 08 14          movl   $0x804a014,0x804a024
 80483f5: a0 04 08
 80483f8: 8b 10                         mov    (%eax),%edx
 80483fa: 89 08                         mov    %ecx,(%eax)
 80483fc: 5d                            pop    %ebp
 80483fd: 89 15 14 a0 04 08             mov    %edx,0x804a014
 8048403: c3                            ret
 80
...
```

swap();

*Get *bufp0;*

Get bufp[1];

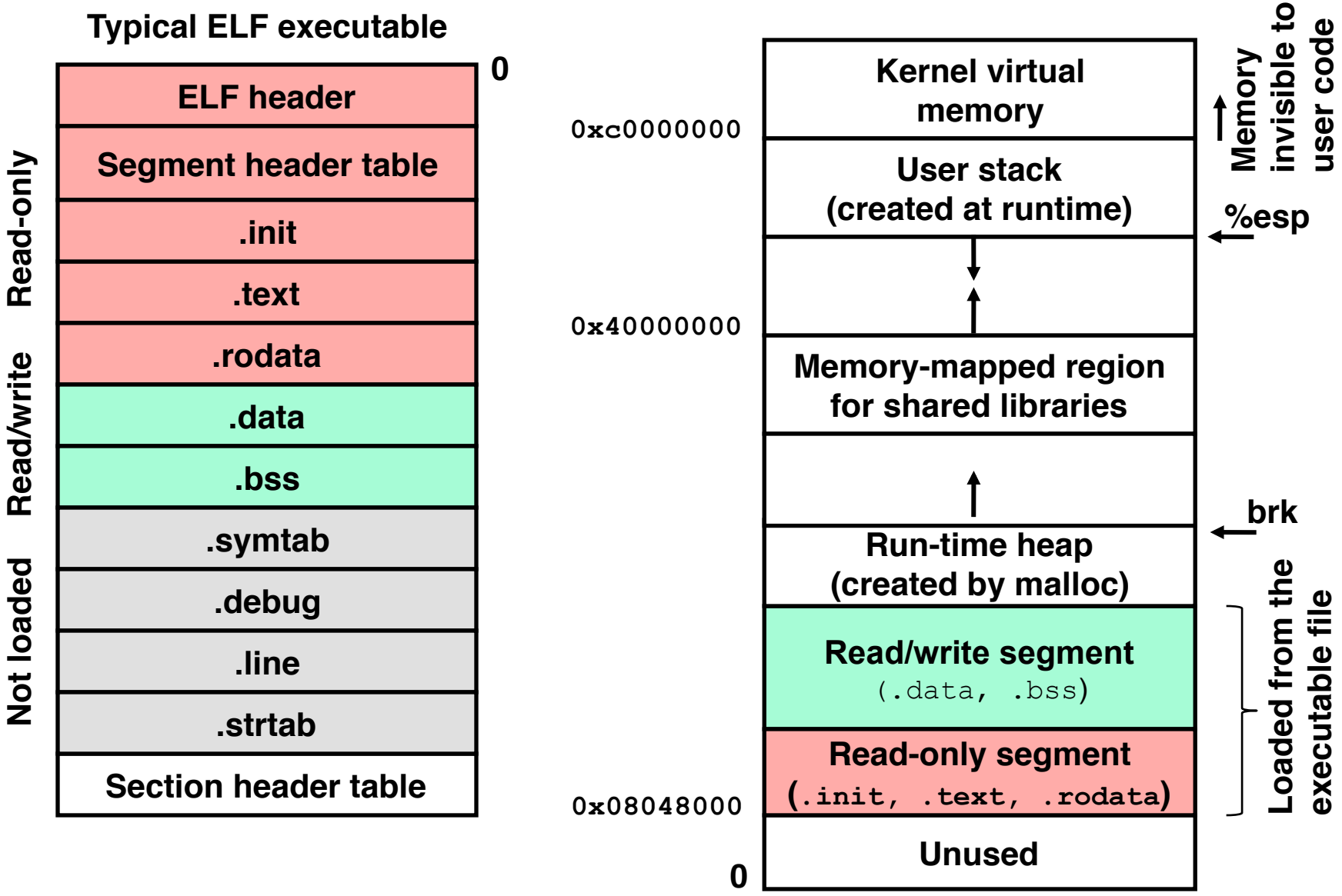
bufp1 = &buf[1];

*Get *bufp1*

```
...
0804a010 <buf>:
 804a010: 01 00 00 00 02 00 00 00                        ........

0804a018 <bufp0>:
 804a018: 10 a0 04 08
........
```

Loading executable binaries



Static and shared libraries

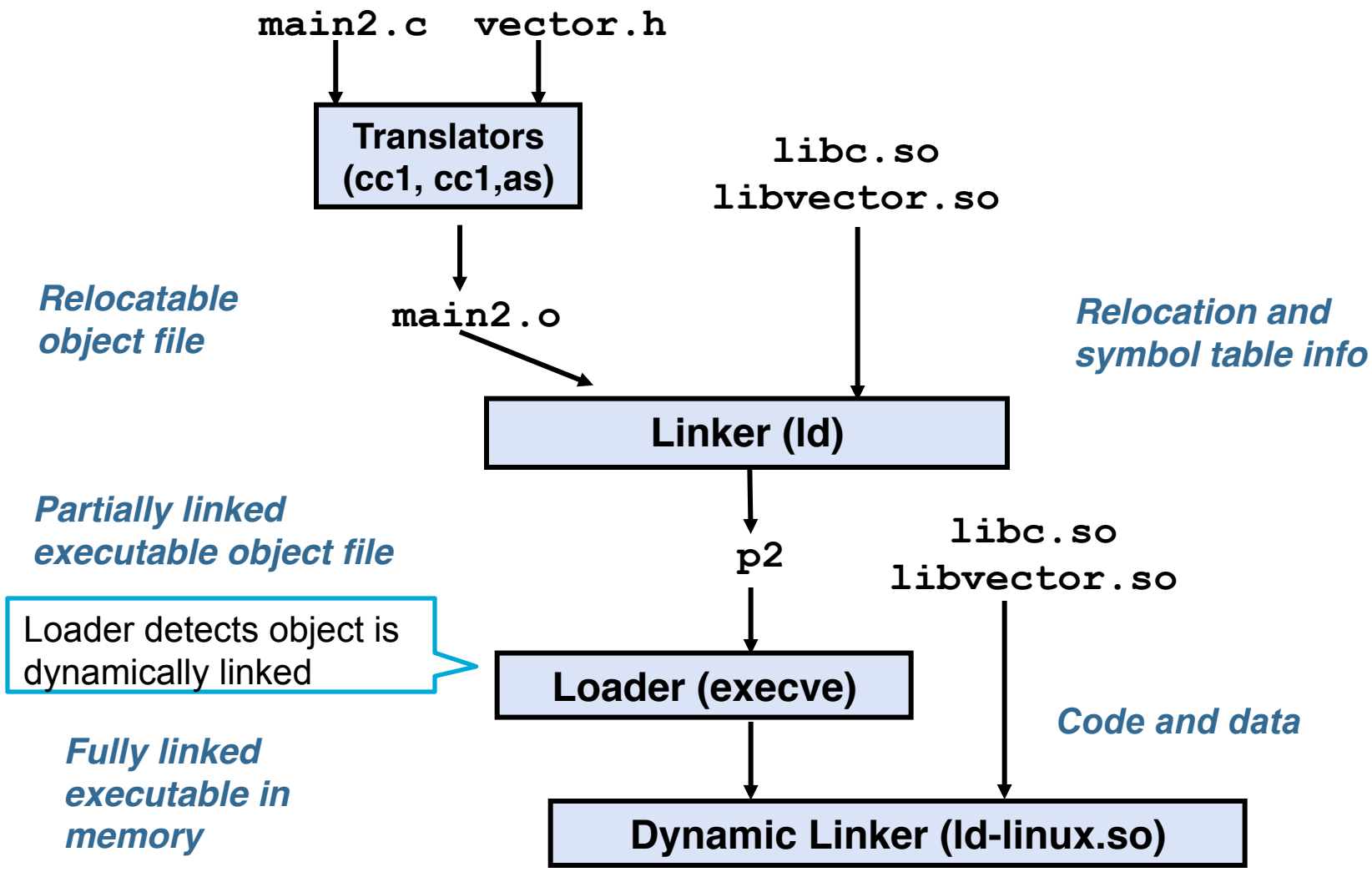
- Static libraries still have a few disadvantages:
  - Potential for duplicating common code in multiple exec files
    - e.g., every C program needs the standard C library
  - Potential for duplicating code in the virtual mem. space of many processes
  - Minor bug fixes of system libraries require each application to explicitly relink
- Shared libraries – members are dynamically loaded into memory and linked into apps at run-time

Shared libraries

- Forms of sharing
  - In any given file system, only one .so file for a particular library
  - A single copy of .text section of the library is shared by different processes
- Shared libraries (.so on Unix, DDL for MS)
  - Dynamic linking can occur when exec is first loaded and run
    - Common case for Linux, handled automatically by ld-linux.so.
  - Dynamic linking can also occur after program has begun
    - In Linux, this is done explicitly by user with dlopen()
    - Basis for High-Performance web servers.

Dynamically linked at load time

```
unix> gcc -shared -o libvector.so addvec.c multvec.c
```



Dynamic linking from applications

- Why?
  - Distributing software – new versions of shared libraries used as they become available (MS Windows)
  - Building high-performance web servers – functions that generate dynamic content available as dll

```
#include <stdio.h>
#include <dlfcn.h>

int x[2] = {1, 2};
int y[2] = {3, 4};
int z[2];

int main()
{
   void *handle;
   void (*addvec) (int *, int*, int *, int);
   char *error;

   /* dynamically load shared lib */
   handle = dlopen("./libvector.so",
    RTLD_LAZY);
   if (!handle) {
     fprintf(stderr, "%s\n", dlerror());
     exit(1);
   }
```

```
  /* get pointer to addvec() func loaded */
  addvec = dlsym(handle, "addvec");
  if ((error = dlerror()) != NULL) {
     fprintf(stderr, "%s\n", error);
     exit(1);
   }

   /* Now call addvec() as usual */
   addvec(x, y, z, 2);
   prtintf("z = [%d %d]\n", z[0], z[1]);

   /* unload the shared library */
if (dlclose(handle) < 0) {
   fprintf(stderr, "%s\n", dlerror());
   exit(1);
}
```

Summary

- Linking
  - Linker mechanics
  - Shared libraries
  - Dynamic libraries
- Next time
  - …