Linking

Today

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

Next time

- Exceptional control flows

Example C program

main.c

```
void swap();

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

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

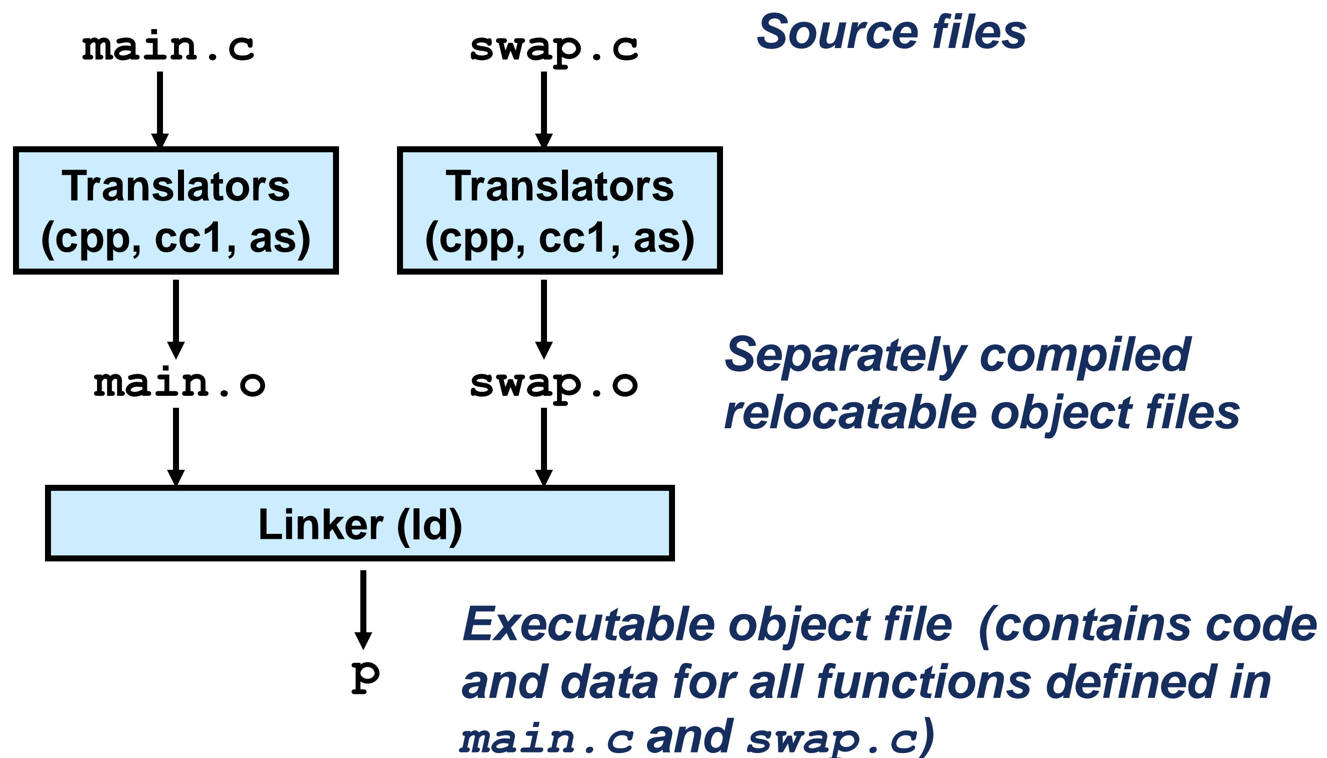
swap.c

```
extern int buf[];

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

void swap()
{
  int temp;

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

A better scheme using a linker

- Programs are translated and linked using a compiler driver:

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



Translating the example program

- *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 `main.c` and `swap.c`:

```
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>
```

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.

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

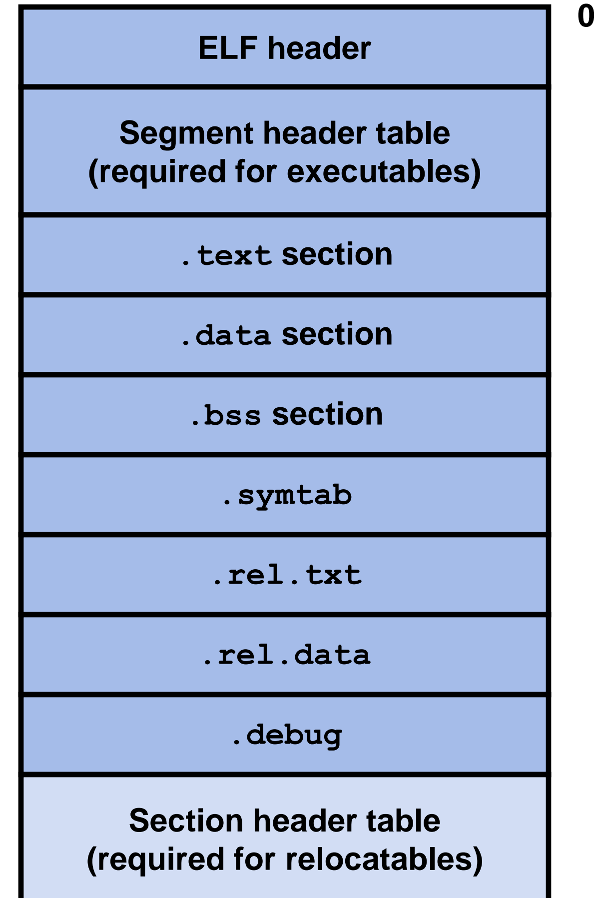
- Generated by compilers and assemblers
  - Relocatable object file
    - Contains code and data in a form that can be combined with other relocatable object files to form an executable
    - Each .o file is produced from exactly one source (.c) file
  - Shared object file
    - Special type of relocatable 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

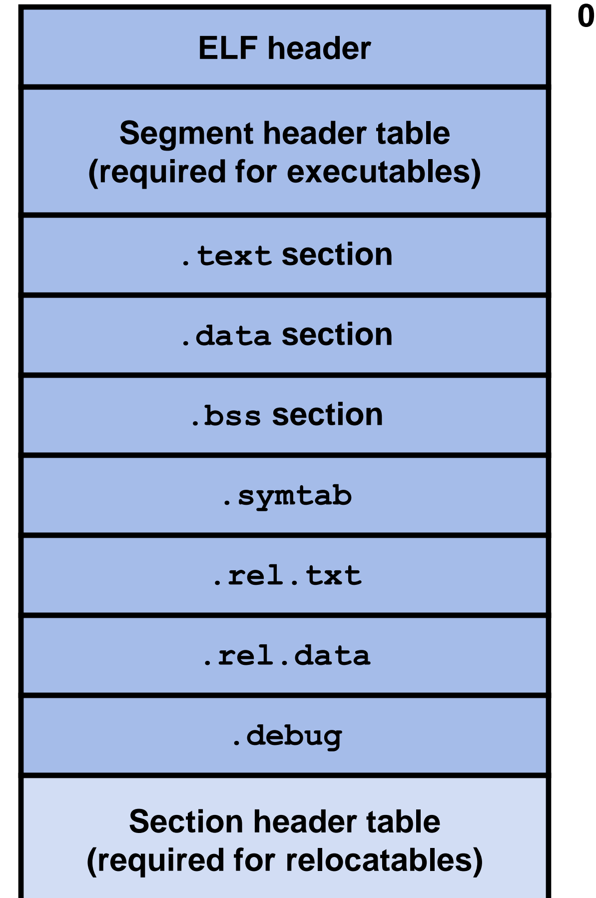
ELF object file format

- ELF header
  - Magic number, type (.o, exec, .so), machine, byte ordering, 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?

```
void swap();

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

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

main.c

```
extern int buf[];

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

void swap()
{
  int temp;

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

swap.c

- 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*

ELF symbol table example

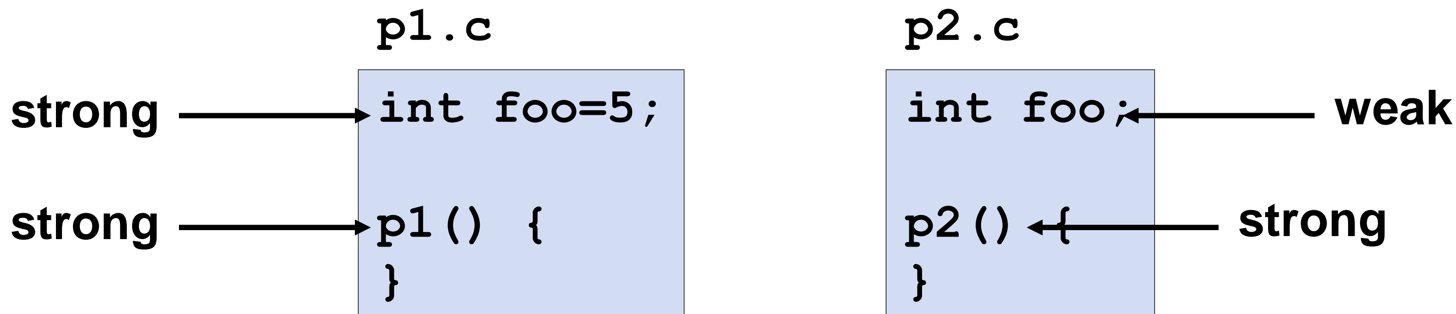
ELF symbol

```
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
```

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

Module 1	Module 2	Result
`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.

Nightmare scenario: two identical weak structs, compiled by different compilers with different alignment rules.

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 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 into executable.

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
…
```

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

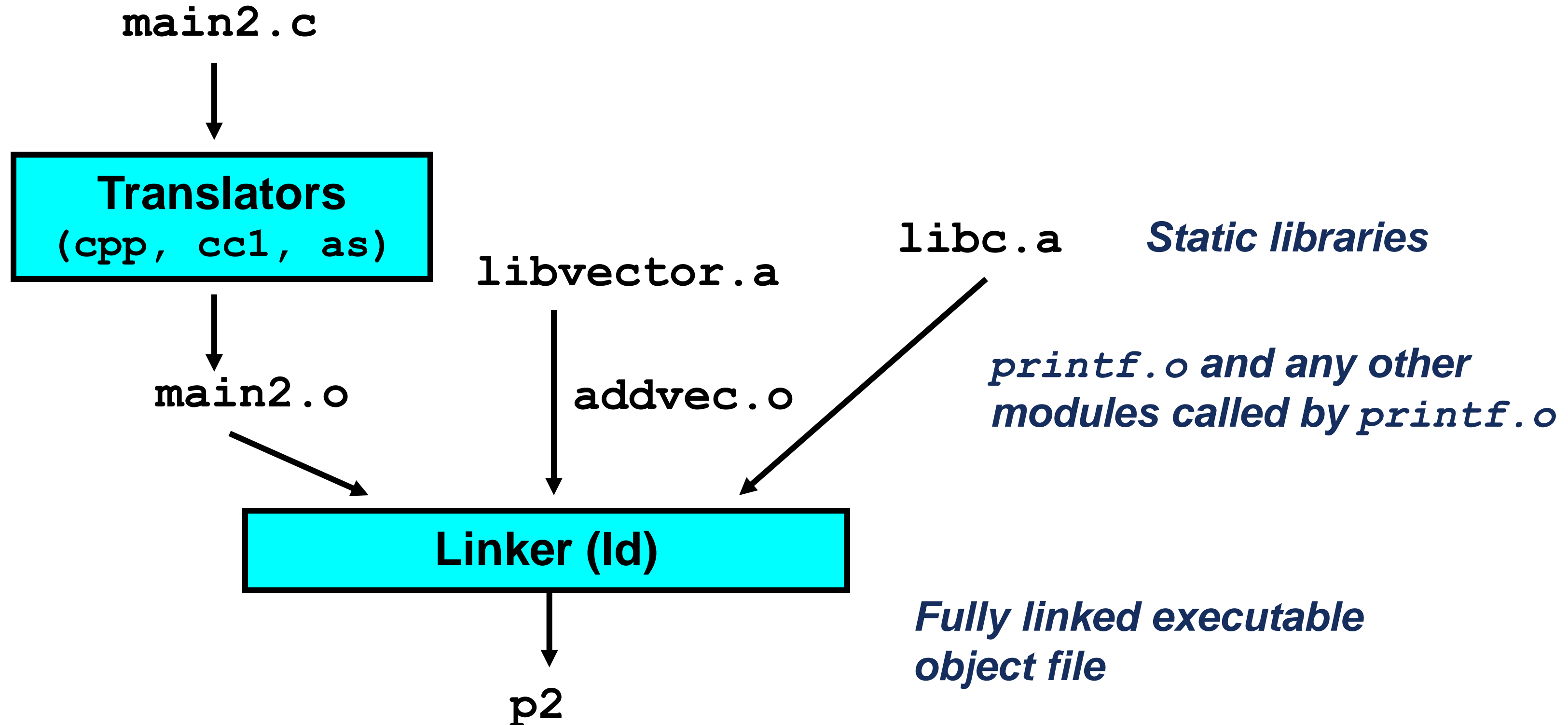
```
void addvec(int *x, int *y,
            int *z, int n)
{
  int i;

  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



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 any entries in the unresolved list at end of scan, then error.
- Problem:
  - Command line order matters!
  - Moral: put libraries at the end of the command line

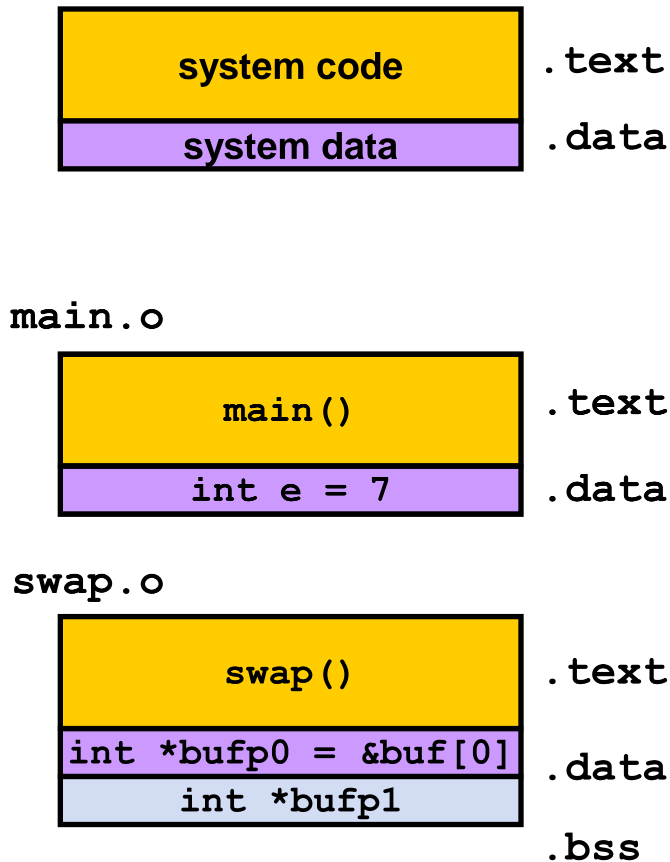
```
unix% gcc -static ./libvector.a main2.c
/tmp/ccC19pHI.o: In function `main':
main2.c:(.text+0x29): undefined reference to `addvec'
collect2: ld returned 1 exit status
```

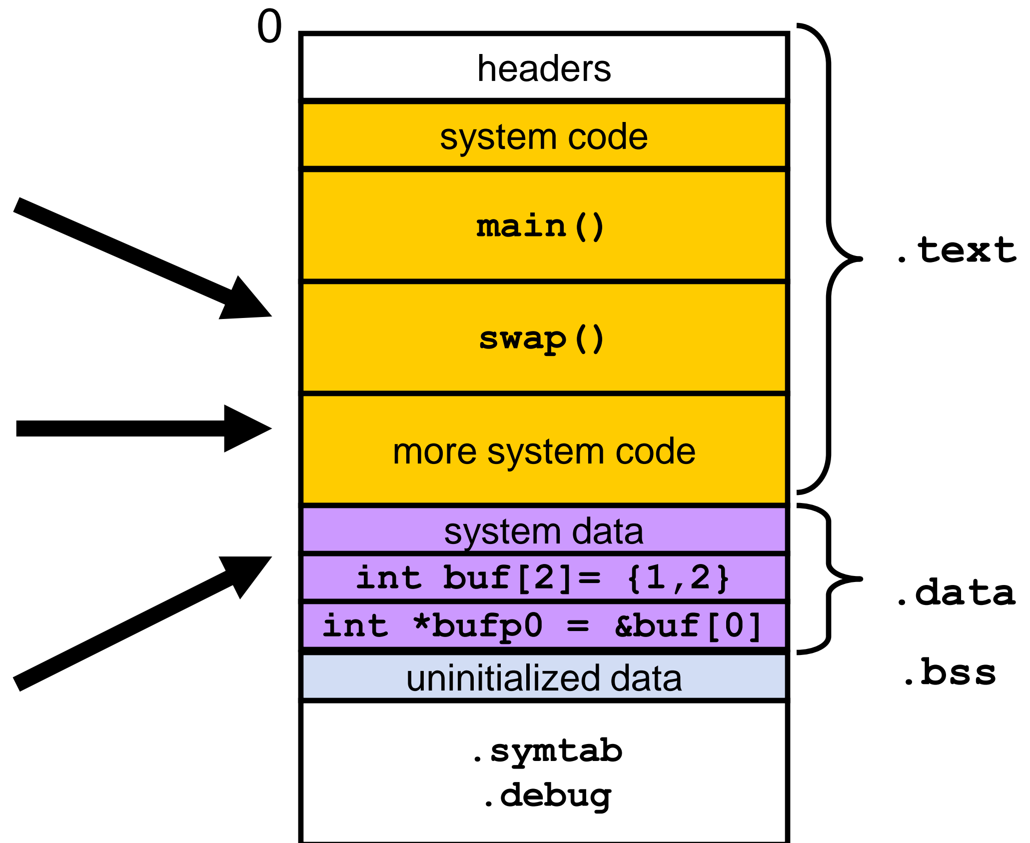
Relocation

- 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





Relocation info (main)

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:      b8 00 00 00 00              mov     $0x0,%eax
   d:      89 ec                       mov     %ebp,%esp
   f:      5d                          pop     %ebp
  10:      c3                          ret
```

```
Disassembly of section .data:

00000000 <buf>:
   0:      01 00 00 00 02 00 00 00 ...
```

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

source: `objdump –dr` **and** `objdump –dr –j .data`

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) =
0x80483b4

ADDR(r.symbol) = ADDR(swap) =
0x80483c8
```

```
refaddr = 0x80483b4 + 0x7 = 0x80483bb
*refptr = unsigned (0x80483c8 + (-4) – 0x80483bb) = 0x9
```

```
080483b4 <main>:
 80483b4:       55                      push    %ebp
 80483b5:       89 e5                   mov     %esp,%ebp
 80483b7:       83 ec 08                sub     $0x8,%esp
 80483ba:       e8 09 00 00 00          call    80483c8 <swap>
 80483bf:       b8 00 00 00 00          mov     $0x0,%eax
 80483c1:       89 ec                   mov     %ebp,%esp
 80483c3:       5d                      pop     %ebp
 80483c4:       c3                      ret
 80483c5:       90                      nop
 80483c6:       90                      nop
 80483c7:       90                      nop
...
```

At run time `call` instruction is at `0x80483ba` and `PC` has `0x80483bf;`
To execute the call CPU

1. `push PC onto stack`
2. `PC <- PC+0x9 = 0x80483c8`

Relocation info (swap, .text)

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:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   c7 05 00 00 00 00 04    movl   $0x4,0x0
   a:   00 00 00
                        5: R_386_32     bufp1
                        9: R_386_32     buf
   d:   a1 00 00 00 00          mov    0x0,%eax
                        e: R_386_32     bufp0
  12:   8b 10                   mov    (%eax),%edx
  14:   8b 0d 04 00 00 00       mov    0x4,%ecx
                        16: R_386_32    buf
  1a:   89 08                   mov    %ecx,(%eax)
  1c:   a1 00 00 00 00          mov    0x0,%eax
                        1d: R_386_32    bufp1
  21:   89 10                   mov    %edx,(%eax)
  23:   5d                      pop    %ebp
  24:   c3                      ret
```

Relocation info (swap, .data)

```
extern int buf[];

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

void swap()
...
```

swap.c

```
Disassembly of section .data:

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

```
r.offset = 0x0          ADDR(r.symbol) = ADDR(buf) = 0x804a014
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 (0x804a014 + 0) = 0x804a014
```

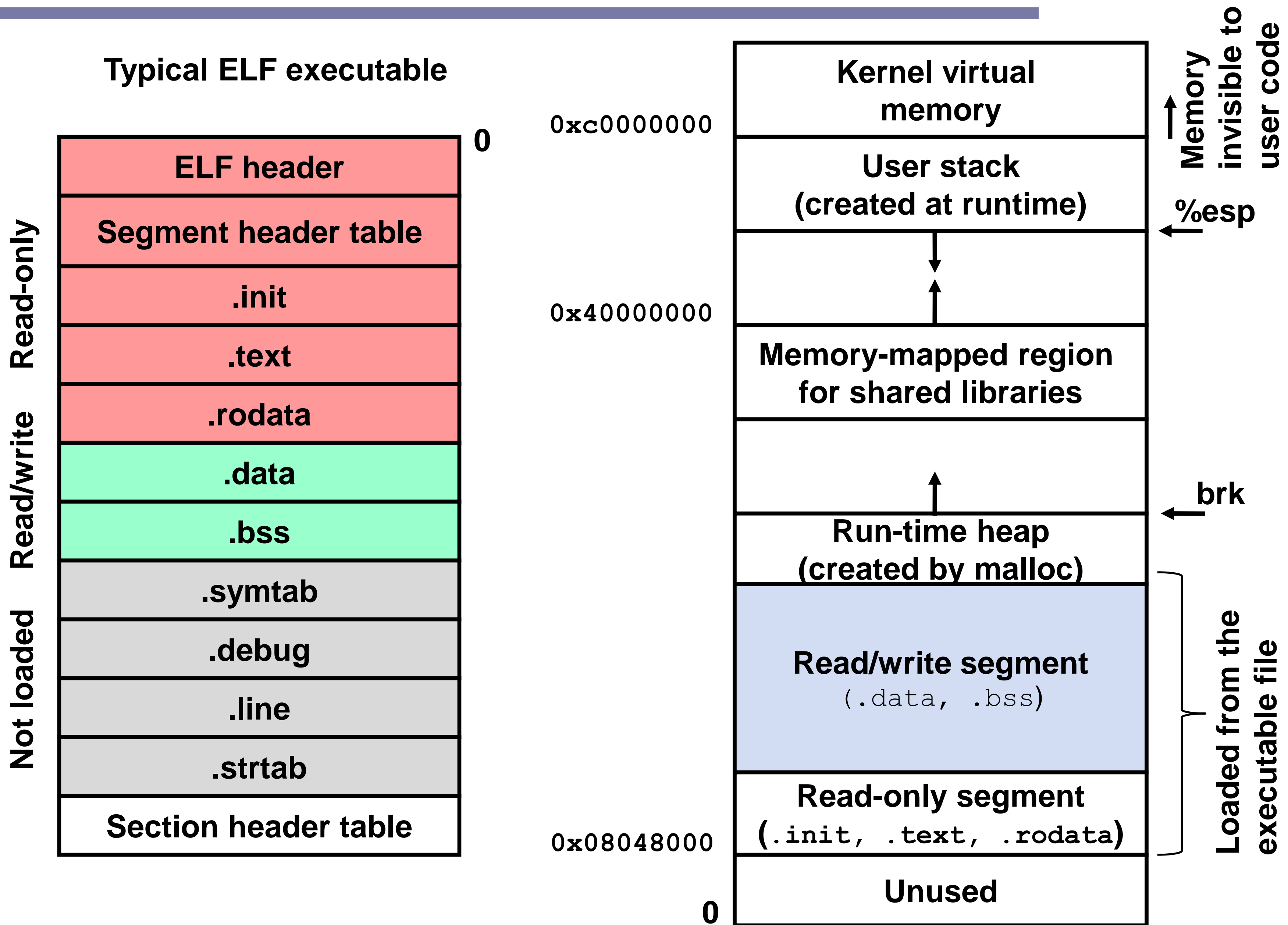
Executable after relocation (.data)

```
Disassembly of section .data:

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

0804a00c <__dso_handle>:
 804a00c: 00 00 00 00                                                ....

0804a010 <bufp0>:
 804a010: 14 a0 04 08                                                ....

0804a014 <buf>:
 804a014: 01 00 00 00 02 00 00 00                                    ........
```

Executable after relocation (.text)

```
080483dc <main>:
 80483dc: 55                        push   %ebp
 80483dd: 89 e5                     mov    %esp,%ebp
 80483df: 83 e4 f0                  and    $0xfffffff0,%esp
 80483e2: e8 cd ff ff ff            call   80483b4 <swap>        swap();
 80483e7: b8 00 00 00 00            mov    $0x0,%eax
 80483ec: 89 ec                     mov    %ebp,%esp
 80483ee: 5d                        pop    %ebp
 80483ef: c3                        ret
080483b4 <swap>:
 80483b4: 55                        push   %ebp
 80483b5: 89 e5                     mov    %esp,%ebp
 80483b7: c7 05 24 a0 04 08 18      movl   $0x804a018,0x804a024  bufp1 = &buf[1];
 80483be: a0 04 08
 80483c1: a1 10 a0 04 08            mov    0x804a010,%eax        Get *bufp0;
 80483c6: 8b 10                     mov    (%eax),%edx
 80483c8: 8b 0d 18 a0 04 08         mov    0x804a018,%ecx        Get bufp[1];
 80483ce: 89 08                     mov    %ecx,(%eax)
 80483d0: a1 24 a0 04 08            mov    0x804a024,%eax        Get *bufp1
 80483d5: 89 10                     mov    %edx,(%eax)
 80483d7: 5d                        pop    %ebp
 80483d8: c3                        ret
```

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* (DLLs) – members are dynamically loaded into memory and linked into apps at run-time

Shared libraries

- Shared libraries (`.so` on Unix)
  - 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.
  - 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

Dynamically linked at load time

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

main2.c **vector.h**

Translators (cc1, cc1,as)

Relocatable object file

main2.o

libc.so
libvector.so

Relocation and symbol table info

Linker (ld)

Partially linked executable object file

p2

libc.so
libvector.so

Loader (execve)

Code and data

Fully linked executable in memory

Dynamic Linker (ld-linux.so)

Dynamic linking from applications

- Why?
  - Distributing software
  - Building high-performance web servers

```
#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);
  }

  return 0;
}
```

Summary

- Linking
  - Linker mechanics
  - Shared libraries
  - Dynamic libraries
- Next time
  - Exceptional control flow