Linking

Kai Shen

1

Example C Program

main.c

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

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

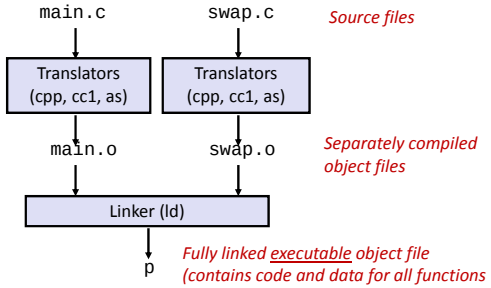
swap.c

```
extern int buf[];

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

void swap()
{
  int temp;

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

2

Static Linking

- Programs are translated and linked using a *compiler driver*:
  - unix> *gcc -O2 -g -o p main.c swap.c*
  - unix> *./p*



3

Why Linkers?

- Reason 1: 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, or my own toolbox
- Reason 2: Efficiency
  - Time: Separate compilation
    - Change one source file, compile, and then relink.
    - No need to recompile other source files.
  - Space: Libraries
    - Common functions can be aggregated into a single library.
    - Multiple running programs can share one library.

4

Linker Does Symbol Resolution

- Programs define and reference *symbols* (variables and functions):

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

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

5

Linker Does Relocation

- Merges separate code sections from multiple object files into single code section
- Same for data sections
- Cannot determine the absolute locations/addresses of symbols before linking. Why?
- Relocation
  - Relocates symbols from their relative locations in the object files to their final absolute memory locations in the executable.
  - Updates all references to these symbols to reflect their absolute positions.

6

Three Kinds of Object Files (Modules)

- Relocatable object file (`.o` file or `.a` file)
  - Contains code and data in a form that can be combined with other relocatable object files to form executable object file.
- Executable object file (`a.out` file)
  - Contains code and data in a form that can be copied directly into memory and then executed.
- Dynamic link object file (`.so` file)
  - Special type of relocatable object file that can be loaded into memory and linked dynamically, at run-time.
  - Called *Dynamic Link Libraries* (DLLs) by Windows

7

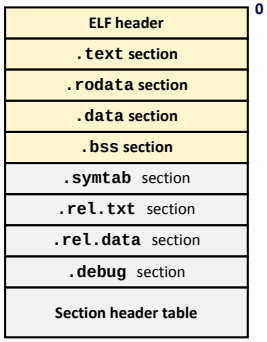
Executable and Linkable Format (ELF)

- Standard binary format for object files
- Originally proposed by AT&T System V Unix
  - Later adopted by BSD Unix variants and Linux
- One unified format for
  - Relocatable object files (`.o`),
  - Executable object files (`a.out`)
  - Dynamic link object files (`.so`)
- Generic name: ELF binaries

8

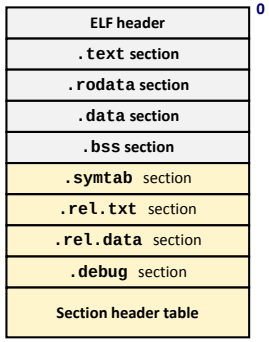
ELF Object File Format

- ELF header
  - Word size, byte ordering, file type (.o, exec, .so), machine type, etc.
- `.text` section
  - Code
- `.rodata` section
  - Read only data: jump tables, ...
- `.data` section
  - Initialized global variables
- `.bss` section
  - Uninitialized global variables
  - "Block Started by Symbol"
  - "Better Save Space"
  - Has section header but occupies no space



ELF Object File Format (cont.)

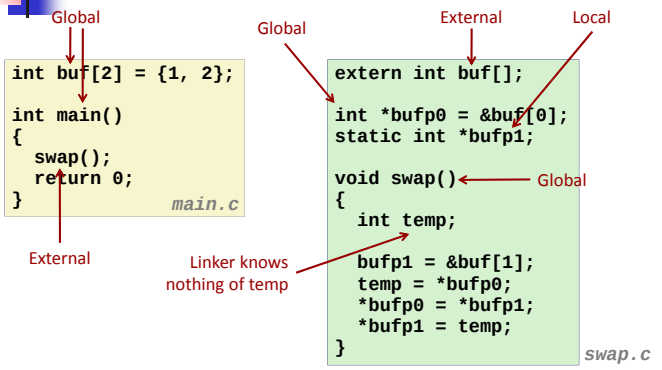
- `.symtab` section
  - Symbol table
  - Procedure and static variable names
  - Section names and locations
- `.rel.text` section
  - Relocation info for `.text` section
  - Addresses of instructions that will need to be modified in the executable
- `.rel.data` section
  - Relocation info for `.data`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



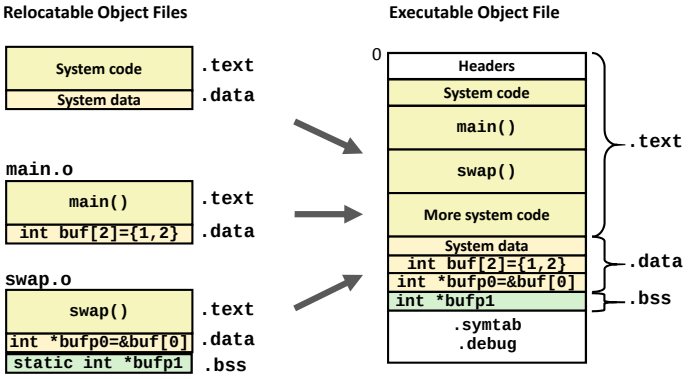
Linker Symbols

- Global symbols
  - Symbols defined by module *m* 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 module *m* but defined by some other module.
- Local symbols
  - Symbols that are defined and referenced exclusively by module *m*.
  - E.g.: C functions and variables defined with the `static` attribute.

Resolving Symbols



Relocating Code and Data



13

Symbol Resolution and Relocation

- Code section change due to relocation at link time

main.c

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

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

14

Symbol Resolution and Relocation

- Data section change due to relocation at link time

swap.c

```
extern int buf[];

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

void swap() {
  int temp;

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

15

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 into 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

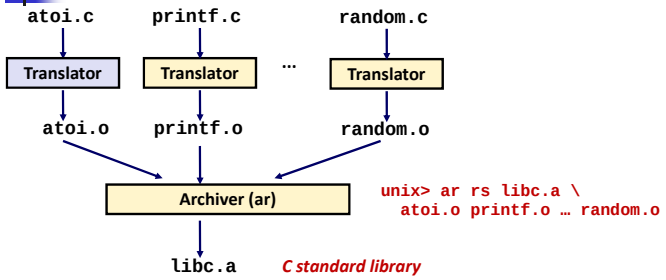
16

Solution: Libraries

- Concatenate related relocatable object files into a 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 it into the executable.
- Static libraries

17

Creating Static Libraries



- Archiver allows incremental updates
- Recompile function that changes and replace .o file in archive.

18

Commonly Used Libraries

`libc.a` (the C standard library)

- 8 MB archive of 1392 object files.
- I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math

`libm.a` (the C math library)

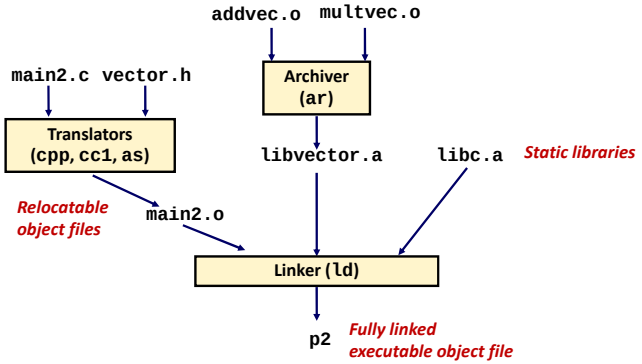
- 1 MB archive of 401 object files.
- floating point math (sin, cos, tan, log, exp, sqrt, …)

```
% 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
…
```

19

Linking with Static Libraries



20

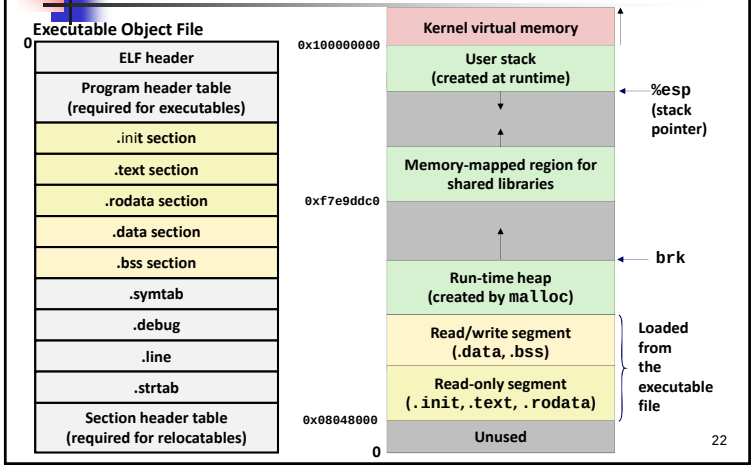
Using Static Libraries

- 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 **`.a`** file is encountered, try to resolve each unresolved reference in the list against the symbols defined in the library.
  - If any entries in the unresolved list at end of scan, then error.
- Problem:
  - Command line order matters!
  - Always put libraries at the end.

```
unix> gcc -L. libtest.o -lmine
unix> gcc -L. -lmine libtest.o
libtest.o: In function `main':
libtest.o(.text+0x4): undefined reference to `libfun'
```

21

Loading Executable Object Files



22

Dynamic Link Libraries

- Static libraries have the following disadvantages:
  - Duplication (every function needs std libc)
    - In stored executables
    - In the running executables
  - Minor bug fixes of system libraries require each application to explicitly relink
- Modern solution: Dynamic link libraries
  - Object files that contain code and data that are loaded and linked into an application *dynamically,* at either *load-time* or *run-time*

23

Dynamic Link Libraries (cont.)

- Dynamic linking can occur when executable is first loaded and run (load-time linking).
  - Common case for Linux, handled automatically by the dynamic linker.
  - Standard C library (**`libc.so`**) usually dynamically linked.
- Dynamic linking can also occur after program has begun (run-time linking).
  - In Linux, this is done by calls to the **`dlopen()`** interface.
  - High-performance web servers.

24

Position-Independent Code

- Dynamic link library routines can be shared by multiple processes.
  - But may be loaded to different addresses of the processes
- Need position-independent code
  - Code that can be loaded and executed at any address without being modified by the linker
  - One approach: indirect reference through a per-process *global offset table*

25

Disclaimer

These slides were adapted from the CMU course slides provided along with the textbook of "Computer Systems: A programmer's Perspective" by Bryant and O'Hallaron. The slides are intended for the sole purpose of teaching the computer organization course at the University of Rochester.

26