15-213

"The course that gives CMU its Zip!"

Linking
October 11, 2006

Topics

- Static linking
- Dynamic linking
- Case study: Library interpositioning

class13.ppt

Example C Program

main.c

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

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

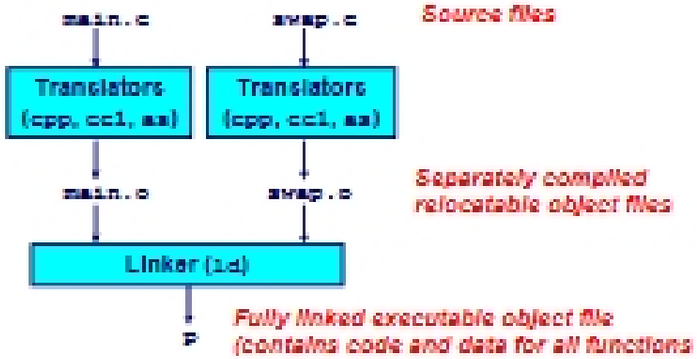
swap.c

```
extern int buf[];

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

void swap()
{
  int temp;

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


Static Linking

Programs are translated and linked using a *compiler driver*:

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



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

Why Linkers? (cont)

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 file...
  - Yet executable files and running memory images contain only code for the functions they actually use.

What Do Linkers 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 symbol xp, reference x */`
- Symbol definitions are stored (by compiler) in *symbol table*.
  - Symbol table is an array of structs
  - Each entry includes name, size, and location of symbol.
- Linker associates each symbol reference with exactly one symbol definition.

What Do Linkers Do? (cont)

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 (Modules)

1. Relocatable object file (`.o` file)

- Contains code and data in a form that can be combined with other relocatable object files to form executable object file.
  - Each `.o` file is produced from exactly one source (`.c`) file

2. Executable object file

- Contains code and data in a form that can be copied directly into memory and then executed.

3. Shared object file (`.so` file)

- Special type of relocatable object file that can be loaded into memory and linked dynamically, at either load time or run-time.
- Called *Dynamic Link Libraries* (DLLs) by Windows

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
- 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

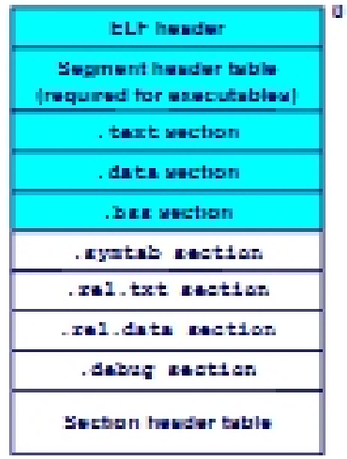
- Code

`.data` section

- Initialized 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
- Instructions for modifying.

`.rel.data` section

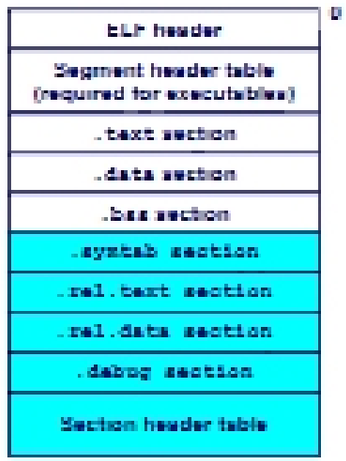
- Relocation info for `.data` section
- Addresses of pointer data that will need to be modified in the merged executable

`.debug` section

- Info for symbolic debugging (`gcc -g`)

Section header table

- Offsets and sizes of each section



Linker Symbols

Global symbols

- Symbols defined by module *m* that can be referenced by other modules.
- Ex: 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*.
- Ex: C functions and variables defined with the `static` attribute.

Key Point: Local linker symbols are *not* local program variables