Sandeep K. S. Gupta
Arizona State University
18th lecture
Linking
(Slides adapted from CSAPP)
Computer system topics

- Program structure and execution
- Running programs on a system
  - Linkers (this class)
  - Process
  - Virtual memory
- Interconnection and communication between programs
  - System level I/O (This class)
- Next topics:
  - Network programming
  - Concurrent programming
Outline

- Why linker?
- What do the linkers do?
- Executable and Linkable Format (ELF)
- Static libraries
- Dynamic libraries
Lifetime of a simple C program

Ex: insert studio.h into the program text

Ex: merging printf.o and hello.o
Example C Program

main.c

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

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

swap.c

```c
extern int buf[];

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

![Diagram of static linking process]

- Source files
- Separately compiled relocatable object files
- Fully linked executable object file (contains code and data for all functions defined in main.c and swap.c)
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 we expect linkers to do?

- Merging data
  - Global variables
- Merging code
- Relocating (adjusting addresses)
What Do Linkers Do?

- Step 1. 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 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)

- 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

- Executable object file (e.g., a.out file)
  - Contains code and data in a form that can be copied directly into memory and then executed.

- 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 (a.out)
  - Shared object files (.so)

- Generic name: ELF binaries
ELF Object File Format

- Elf header
  - Word size, byte ordering, file type (.o, exec, .so), machine type, etc.

- Segment header table
  - Page size, virtual addresses memory segments (sections), segment sizes.

- .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

Diagram:

<table>
<thead>
<tr>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELF header</td>
</tr>
<tr>
<td>Segment header table</td>
</tr>
<tr>
<td>.text section</td>
</tr>
<tr>
<td>.rodata section</td>
</tr>
<tr>
<td>.data section</td>
</tr>
<tr>
<td>.bss section</td>
</tr>
<tr>
<td>.symtab section</td>
</tr>
<tr>
<td>.rel.txt section</td>
</tr>
<tr>
<td>.rel.data section</td>
</tr>
<tr>
<td>.debug section</td>
</tr>
<tr>
<td>Section header table</td>
</tr>
</tbody>
</table>
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.
  - 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.
  - Local linker symbols are *not* local program variables
Resolving Symbols

```c
int buf[2] = {1, 2};
int main()
{
    swap();
    return 0;
}
``` 

```c
extern int buf[];
int *bufp0 = &buf[0];
static int *bufp1;
void swap()
{
    int temp;
    bufp1 = &buf[1];
    temp = *bufp0;
    *bufp0 = *bufp1;
    *bufp1 = temp;
}
```
Program symbols are either strong or weak

- **Strong**: procedures and initialized globals
- **Weak**: uninitialized globals

```c
int foo = 5;
p1() {
}
```

```c
int foo;
p2() {
}
```
Linker’s Symbol Rules

- Rule 1: Multiple strong symbols are not allowed
  - Each item can be defined only once
  - Otherwise: Linker error

- Rule 2: Given a strong symbol and multiple weak symbol, choose the strong symbol
  - References to the weak symbol resolve to the strong symbol

- Rule 3: If there are multiple weak symbols, pick an arbitrary one
  - Can override this with `gcc -fno-common`
Linker Puzzles

```c
int x;
p1() {}  

p1() {}  
```

Link time error: two strong symbols (p1)

```c
int x;
p1() {}  

int x;
p2() {}  
```

References to x will refer to the same uninitialized int. Is this what you really want?

```c
int x;
int y;
p1() {}  

double x;
p2() {}  
```

Writes to x in p2 might overwrite y!

```c
int x=7;
p1() {}  

int x;
p2() {}  
```

References to x will refer to the same initialized variable.
Linking problem example

/* file1.c */
#include <stdio.h>
Void f()
Int x;
Int main() {
    x=100
    f();
    printf("%d",x)
    return 1;
}

/* file2.c */
#include <stdio.h>
Int x;
Void f(){
    x=200;
}
### Linking problem example

/* file1.c */
#include <stdio.h>
void f();
int x;
int y;
main()
{
    x=10;
    y=9999999;
    f();
    printf("y:%d x:%d\n",y,x);
}

/* file2.c */
#include <stdio.h>
double x;
void f();
void f()
{
    x=45.76763;
}

Output
y:9999999 x:-1289177384
Relocating

- Relocating sections and symbol definitions:
  - Merging all sections of the same type into a new aggregate section of the same type
    - E.g., .data section
  - Assigning run-time memory addresses to the new aggregate sections and symbols
    - E.g., global variables to have a unique run-time address

- Relocating symbol references
  - Modifying every symbol references in the bodies of the code and data to point to the correct run-time address
Relocating Code and Data

Relocatable Object Files

```
main.o
```

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

```
swap.o
```

```
int *bufp0 = &buf[0]
```

```
static int *bufp1
```

Executable Object File

```
0
```

```
Headers
```

```
System code
```

```
main()
```

```
swap()
```

```
More system code
```

```
System data
```

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

```
int *bufp0 = &buf[0]
```

```
int *bufp1
```

```
.bss
```

Even though private to swap, requires allocation in .bss
Global Variables

- Avoid if you can

- Otherwise
  - Use `static` if you can
  - Initialize if you define a global variable
  - Use `extern` if you use external global 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:** 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
Solution: Static Libraries

- **Static libraries** (.a archive files)
  - Concatenate/package 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 it into the executable.
Creating Static Libraries

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

unix> ar rs libc.a \n   atoi.o printf.o ... random.o

C standard library
Linking with Static Libraries

**Translators**
- `cpp`, `cc1`, `as`

**Main Source Files**
- `main2.c`  
- `vector.h`

**Object Files**
- `main2.o`
- `printf.o`
- `addvec.o`
- `multvec.o`

**Archiver** (`ar`)
- `libvector.a`
- `libc.a`

**Linker** (`ld`)
- `addvec.o`
- `main2.o`

**Static Libraries**
- `printf.o` and any other modules called by `printf.o`

**Fully Linked Executable**
- `p2`

**Command**
- `gcc example.c /usr/lib/libc.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 the symbols defined 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 -L. libtest.o -lmine
unix> gcc -L. -lmine libtest.o
libtest.o: In function `main':
libtest.o(.text+0x4): undefined reference to `libfun'
```
Executable Object File

- ELF header
- Program header table (required for executables)
- .init section
- .text section
- .rodata section
- .data section
- .bss section
- .symtab
- .debug
- .line
- .strtab
- Section header table (required for relocatables)

Kernel virtual memory

- User stack (created at runtime)
- Memory-mapped region for shared libraries
- Run-time heap (created by malloc)
- Read/write segment (.data, .bss)
- Read-only segment (.init, .text, .rodata)
- Unused

Memory outside 32-bit address space

%esp (stack pointer)

brk

Loaded from the executable file

CEN591 Fall 2011
Shared Libraries

- Static libraries have the following disadvantages:
  - Duplication in the stored executables. Every function needs std libc (e.g., printf, scanf)-Disk overhead
  - Duplication in the running executables-Memory overhead
  - Minor bug fixes of system libraries require each application to explicitly relink

- Modern solution: Shared Libraries
  - Object files that contain code and data that are loaded and linked into an application *dynamically*, at either *load-time* or *run-time*
  - Also called: dynamic link libraries, DLLs, *so* files
Shared Libraries (cont.)

- Dynamic linking can occur when executable is first loaded and run (load-time linking).
  - Standard C library (libc.so) usually dynamically linked.

- Dynamic linking can also occur after program has begun (run-time linking).
  - Distributing software.
  - High-performance web servers.
  - Runtime library interpositioning.

For the schemes the library file is shared among all executable objects

- Shared library routines can be shared by multiple processes.
  - Recall page-sharing in virtual memory
Dynamic Linking at Load-time

Translators (cpp, cc1, as)
main2.c  vector.h

Linker (ld)
main2.o

Loader (execve)
p2

Dynamic linker (ld-linux.so)

Partially linked executable object file

Relocatable object file

Relocation and symbol table info

Fully linked executable in memory

Code and data

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

libc.so
libvector.so

unix> gcc -shared -o libvector.so \ addvec.c multvec.c
Dynamic Linking at Run-time

```c
#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 the shared lib that contains addvec() */
    handle = dlopen("./libvector.so", RTLD_LAZY);
    if (!handle) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
```
Dynamic Linking at Run-time

...  

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

/* Now we can call addvec() just like any other function */
addvec(x, y, z, 2);
printf("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;