

CS429: Computer Organization and Architecture

Linking II

Dr. Bill Young
Department of Computer Sciences
University of Texas at Austin

Last updated: January 13, 2017 at 08:55

- *Symbols* are lexical entities that name functions and variables.
- Each symbol has a *value* (typically a memory address).
- Code consists of symbol *definitions* and *references*.
- References can be either *local* or *external*.

m.c

```
int e = 7;           // def of global e

int main() {
    int r = a();     // ref to external symbol a
    exit(0);         // ref to external symbol exit
                    // (defined in libc.so)
}
```

Note that *e* is *locally* defined, but *global* in that it is visible to all modules. Declaring a variable *static* limits its scope to the current file module.

CS429 Slideset 25: 1

Linking II

CS429 Slideset 25: 2

Linking II

Relocating Symbols and Resolving External References (2)

m.o Relocation Info

a.c

```
extern int e;

int *ep = &e;       // def of global ep, ref to
                    // external symbol e

int x = 15;         // def of global x
int y;              // def of global y

int a() {           // def of global a
    return *ep+x+y; // refs of globals ep, x, y
}
```

m.c

```
int e = 7;

int main() {
    int r = a();
    exit(0);
}
```

Source: objdump

Disassembly of section .text

```
00000000 <main>:
0: 55                pushl %ebp
1: 89 e5             movl %esp, %ebp
3: e8 fc ff ff ff   call 4<main+0x4>
4: R_386_PC32      a
8: 6a 00            pushl $0x0
a: e8 fc ff ff ff   call b<main+0xb>
b: R_386_PC32      exit
f: 90                nop
```

Disassembly of section .data

```
00000000 <e>:
0: 07 00 00 00
```

Disassembly of section .text

a.c

```
extern int e;

int *ep = &e;
int x = 15;
int y;

int a() {
    return
        *ep + x + y;
}
```

```
00000000 <a>:
0: 55                pushl %ebp
1: 8b 15 00 00 00    movl 0x0, %edx
6: 00

3: R_386_32 ep
7: a1 00 00 00 00    movl 0x0, %eax
8: R_386_32 x
c: 89 e5            movl %esp, %ebp
e: 03 02            addl (%edx), %eax
10: 89 ec            movl %ebp, %esp
12: 03 05 00 00 00    addl 0x0, %eax
17: 00

14: R_386_32 y
18: 5d                popl %ebp
19: 3c                ret
```

a.c

```
extern int e;

int *ep = &e;
int x = 15;
int y;

int a() {
    return *ep + x + y;
}
```

Disassembly of section .data

```
00000000 <ep>:
0: 00 00 00 00
0: R_386_32 e

00000004 <x>:
4: 0f 00 00 00
```

Executable After Relocation (.text)

```
08048530 <main>:
08048530: 55                pushl %ebp
08048531: 89 e5            movl %esp, %ebp
08048533: e8 08 00 00 00    call 8048540 <a>
08048538: 6a 00            pushl $0x0
0804853a: e8 35 ff ff ff    call 8048474 <_init+0x94>
0804853f: 90                nop

08048540 <a>:
08048540: 55                pushl %ebp
08048541: 8b 15 1c a0 04    movl 0x804a01c, %edx
08048546: 08

08048547: a1 20 a0 04 08    movl 0x804a020, %eax
0804854c: 89 e5            movl %esp, %ebp
0804854e: 03 02            addl (%edx), %eax
08048550: 89 ec            movl %ebp, %esp
08048552: 03 05 d0 a3 04    addl 0x804a3d0, %eax
08048557: 08

08048558: 5d                popl %ebp
08048559: c3                ret
```

Executable After Relocation (.data)

m.c

```
int e = 7;

int main() {
    int r = a();
    exit(0);
}
```

a.c

```
extern int e;

int *ep = &e;
int x = 15;
int y;

int a() {
    return *ep + x + y;
}
```

Disassembly of section .data

```
0804a018 <e>:
804a018: 07 00 00 00

0804a01c <ep>:
804a01c: 18 a0 04 08

0804a020 <x>:
804a020: 0f 00 00 00
```

Program symbols are either *strong* or *weak*.

strong: procedures and initialized globals

weak: uninitialized globals

This doesn't apply to purely local variables.

p1.c

```
int foo = 5; // foo: strong
p1() {      // p1: strong
  ...
}
```

p2.c

```
int foo;     // foo: weak here
p2() {      // p2: strong
  ...
}
```

Rule 1: A strong symbol can only appear once.

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 one arbitrarily.

Linker Puzzles

What happens in each case?

File 1	File 2	Result
int x; p1() {}	p1() {}	
int x; p1() {}	int x; p2() {}	
int x; int y; p1() {}	double x; p2() {}	
int x=7; int y=5; p1() {}	double x; p2() {}	
int x=7; p1() {}	int x; p2() {}	

Linker Puzzles

Think carefully about each of these.

File 1	File 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. What you wanted?
int x; int y; p1() {}	double x; p2() {}	Writes to x in p2 might overwrite y! That's just evil!
int x=7; int y=5; p1() {}	double x; p2() {}	Writes to x in p2 might overwrite y! Very 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.

How to package functions commonly used by programmers?
(Math, I/O, memory management, string manipulation, etc.)

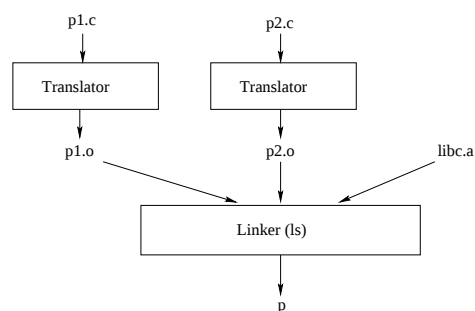
Awkward, given the linker framework so far:

- 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* (.a archive files)

- Concatenate related relocatable object files into a single repository with an index (called an archive).
- Enhance the linker so that it tries to resolve unresolved external reference by looking for symbols in one or more archives.
- If an archive member resolves the reference, link into the executable.

Static Libraries (archives)



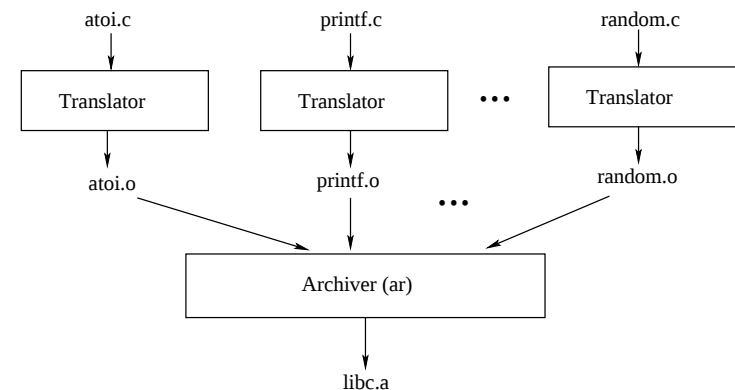
libc.a is a static library (archive) of relocatable object files concatenated into one file.

The output p is an executable object file that only contains code and data for libc functions called from p1.c and p2.c.

This further improves modularity and efficiency by packaging commonly used functions, e.g., C standard library (libc) or math library (libm).

The linker extracts only those .o files from the archive that are actually needed by the program.

Creating Static Libraries



Command: `ar rs libc.a atoi.o printf.o ... random.o`

Archiver allows incremental updates: Recompile a function that changes and replace the .o file in the archive.

libc.a (the C standard library)

- 8MB 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)

- 1MB 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
...
```

```
% 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
...
```

CS429 Slideset 25: 17

Linking II

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 in obj.
- If there are any entries in the unresolved list at the end of the scan, then error.

Problem:

- Command line order matters.
- Moral: put libraries at the end of the command line.

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

CS429 Slideset 25: 18

Linking II

Executable object file for
example program p:

ELF header
Program header tables (required for executables)
.text section
.data section
.bss section
.symtab
.rel.text
.rel.data
.debug
Section header table (required for relocatables)

Loaded segments:

Process image	Virtual addr
init and shared lib segments	0x080483e0
.text segment (r/o)	0x08048494
.data segment (initialized r/w)	0x0804a010
.bss segment (uninitialized r/w)	0x0804a3b0

CS429 Slideset 25: 19

Linking II

Static libraries have some disadvantages:

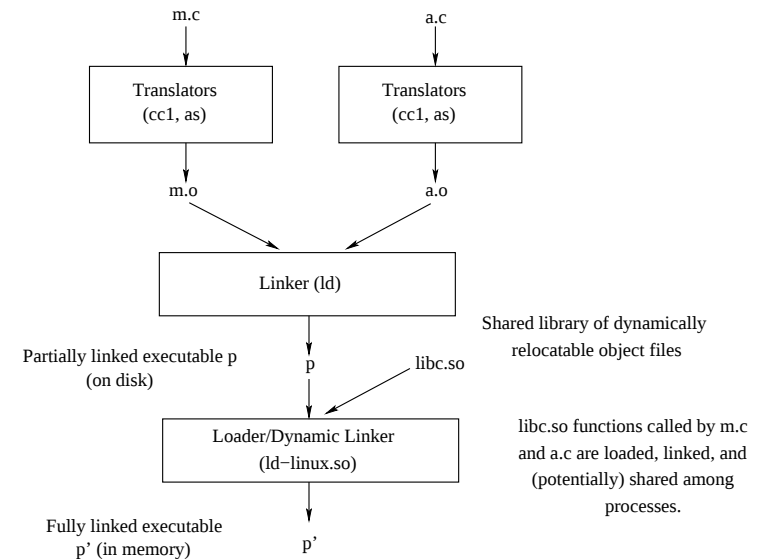
- Potential for duplicating lots of common code in the executable files on a file system. (e.g., every program needs the standard C library).
- Potential for duplicating lots of code in the virtual memory space of many processes.
- Minor bug fixes of system libraries require each application to explicitly relink.

CS429 Slideset 25: 20

Linking II

Solution:

- *Shared libraries* (dynamic link libraries DLLs) whose members are dynamically loaded into memory and linked into an application at run-time.
- Dynamic linking can occur when an executable is first loaded and run. (The common case for Linux, handled automatically by `ld-linux.so`.)
- Dynamic linking can also occur after the program has begun.
 - In Linux, this is done explicitly by user with `dlopen()`.
 - Basis for High-Performance Web Servers.
- Shared library routines can be shared by multiple processes.

**The Complete Picture**