

CSE320 System Fundamentals II Linking

YOUNGMIN KWON / TONY MIONE

Announcements

Final Exam:

- Cumulative!
- Tue, Jun 14, 9:00-11:30 AM

Linking

Linking is the process of

- collecting and combining pieces of code and data
- building a single file that can be loaded into memory and executed

Benefits of doing linking

- Separates compilation process
- Checks missing variables, functions, libraries, incompatible library versions

Linking can be performed at

- Compile time (explicitly running linker after compile)
- load time (linking with shared libraries)
- run time (dlopen(), dlsym(), dlerror(), dlclose())

Linking Example

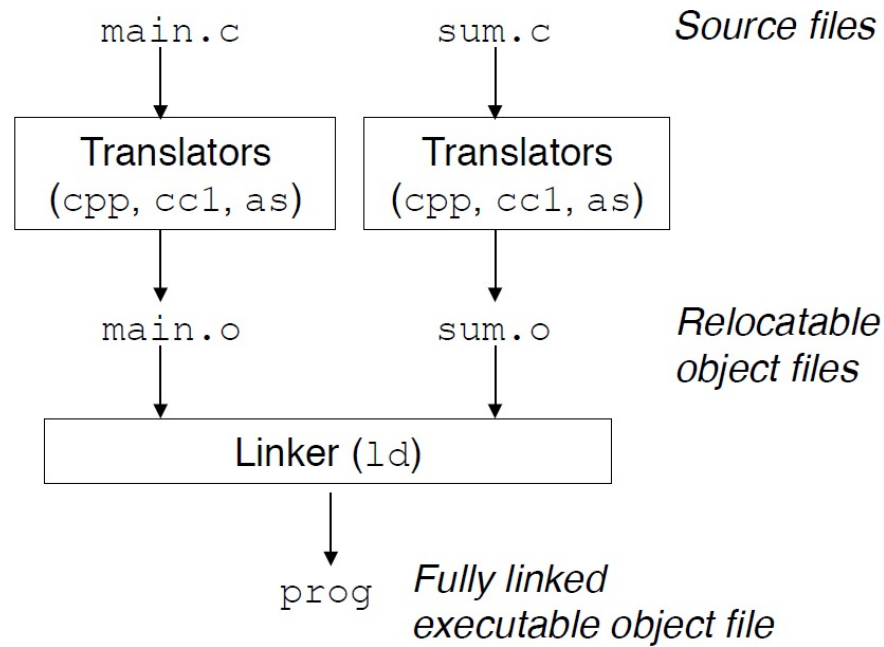
```
/* sum.c */
int sum(int *a, int n)
{
    int i, s = 0;
    for(i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}
```

```
/* main.c */
int sum(int *a, int n);
int array[2] = { 1, 2 };

int main()
{
    int val = sum(array, 2);
    return val;
}
```

```
// gcc -c sum.c main.c
// ld [system object files and args] sum.o main.o
// gcc -o prog sum.o main.o
```

Linking



Static Linking

Static linkers (e.g. ld)

- Input: a collection of relocatable object files
- Output: a fully linked executable object file

Linking steps

1) Symbol resolution step

- Associate each **symbol reference** with exactly one **symbol definition**
- Symbols: functions, global variables, static variables

2) Relocation step

- Compiler generates code and data sections starting at address 0
- **Allocate a memory location** with each symbol definition
- **Modify all references** to those symbols to memory locations

Object Files

Relocatable object file

- Binary code and data that can be combined with other relocatable object files at compile time to create an executable object file

Executable object file

- Binary code and data that can be copied directly into memory and executed

Shared object file

- A special type of relocatable object file that can be loaded into memory and linked dynamically at either load time or run time

Relocatable Object Files

ELF (Executable and Linkable Format)

.text: machine code

.rodata: read-only data

.data: initialized data

.bss: uninitialized data,
no actual space in the object file

.symtab: symbol table

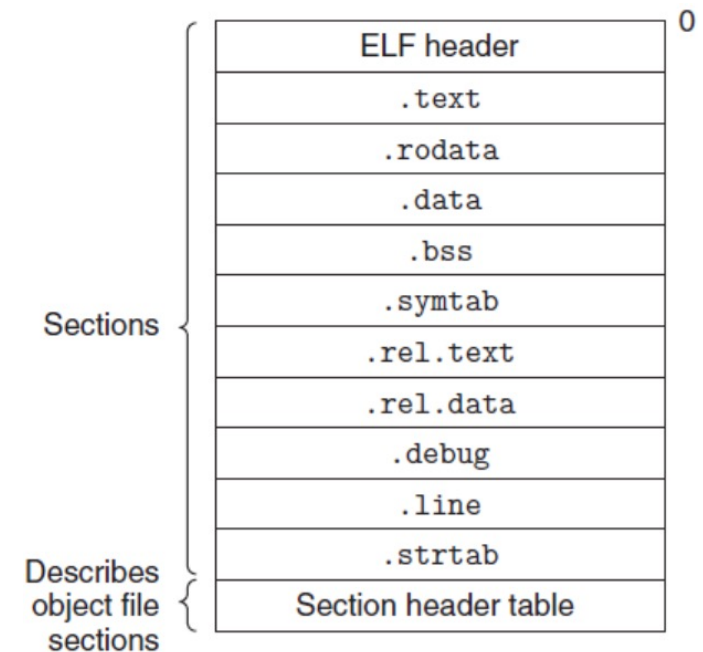
.rel.text: a list of locations in **.text** that needs to be modified
when combined with other object files

.rel.data: relocation info for any global variables that are
referenced or defined by the module

.debug: a debugging symbol table

.line: source file line number info for codes in **.text**

.strtab: string table for the symbol tables in **.symtab** and **.debug**



Relocatable Object Files

readelf command

```
$ readelf -s main.o
```

Symbol table '.symtab' contains 17 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	0000000000000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	0000000000000000	0	FILE	LOCAL	DEFAULT	ABS	main.c
2:	0000000000000000	0	SECTION	LOCAL	DEFAULT	1	
3:	0000000000000000	0	SECTION	LOCAL	DEFAULT	3	
4:	0000000000000000	0	SECTION	LOCAL	DEFAULT	4	
5:	0000000000000000	0	SECTION	LOCAL	DEFAULT	5	
6:	0000000000000000	0	SECTION	LOCAL	DEFAULT	7	
7:	0000000000000000	0	SECTION	LOCAL	DEFAULT	8	
8:	0000000000000000	0	SECTION	LOCAL	DEFAULT	9	
9:	0000000000000000	0	SECTION	LOCAL	DEFAULT	11	
10:	0000000000000000	0	SECTION	LOCAL	DEFAULT	13	
11:	0000000000000000	0	SECTION	LOCAL	DEFAULT	15	
12:	0000000000000000	0	SECTION	LOCAL	DEFAULT	16	
13:	0000000000000000	0	SECTION	LOCAL	DEFAULT	14	
14:	0000000000000000	8	OBJECT	GLOBAL	DEFAULT	3	array
15:	0000000000000000	31	FUNC	GLOBAL	DEFAULT	1	main
16:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	sum

Relocatable Object Files

readelf command

```
$ readelf -s sum.o
```

Symbol table '.symtab' contains 9 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	0000000000000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	0000000000000000	0	FILE	LOCAL	DEFAULT	ABS	sum.c
2:	0000000000000000	0	SECTION	LOCAL	DEFAULT	1	
3:	0000000000000000	0	SECTION	LOCAL	DEFAULT	2	
4:	0000000000000000	0	SECTION	LOCAL	DEFAULT	3	
5:	0000000000000000	0	SECTION	LOCAL	DEFAULT	5	
6:	0000000000000000	0	SECTION	LOCAL	DEFAULT	6	
7:	0000000000000000	0	SECTION	LOCAL	DEFAULT	4	
8:	0000000000000000	62	FUNC	GLOBAL	DEFAULT	1	sum

Symbol Resolution

Symbol resolution

- Associating each **symbol reference** with exactly one **symbol definition** from the symbol table of relocatable object files
- Local and static local variables in a module
 - Compiler ensures a unique name
- Global symbols
 - When a compiler encounters an undefined symbol, it assumes that it is defined in some other module

Symbol Resolution

When linker cannot find a definition

```
/* undef.c */  
void foo();  
extern int ivar;
```

```
int main() {  
    ivar = 0;  
    foo();  
    return 0;  
}
```

```
$ gcc undef.c
```

```
/tmp/cckHPc0h.o:undef.c:function main: error: undefined reference to 'ivar'  
/tmp/cckHPc0h.o:undef.c:function main: error: undefined reference to 'foo'  
collect2: ld returned 1 exit status
```

Symbol Resolution

Duplicated symbols in different modules

- Compiler exports each symbol as either strong or weak
- **Strong Symbols**: functions, initialized global variables
- **Weak Symbols**: uninitialized global variables

Resolution rules for duplicated symbols

- Multiple strong symbols are not allowed
- Given a strong symbol and multiple weak symbols, choose the strong symbol
- Given multiple weak symbols, any one can be chosen

Symbol Resolution

Multiple strong symbols

```
/* dup2.c */  
int foo() {  
    return 2;  
}
```

```
/* dup1.c */  
int foo() {  
    return 1;  
}
```

```
int main() {  
    return foo();  
}
```

```
$ gcc dup1.c dup2.c
```

```
/usr/bin/ld: error: /tmp/cc5gGlFB.o: multiple definition of 'foo'  
/usr/bin/ld: /tmp/ccvdInRf.o: previous definition here  
collect2: ld returned 1 exit status
```

Symbol Resolution

Strong and weak symbols

```
/* weak.c */
#include <stdio.h>
int x;
void foo();
int main()
{
    printf("before: %d\n", x);
    foo();
    printf("after:  %d\n", x);
}
```

```
/* strong.c */
int x = 100;
void foo() {
    x = 200;
}
```

```
$ gcc weak.c strong.c
$ ./a.out
before: 100
after:  200
```

Symbol Resolution

Multiple weak symbols

```
/* weak1.c */
#include <stdio.h>
int x;
void foo();
int main()
{
    x = 2;
    printf("before: %d\n", x);
    foo();
    printf("after:  %d\n", x);
}
```

```
/* weak2.c */
double x;
void foo() {
    x = 0.2;
}
```

```
$ gcc weak1.c weak2.c
$ ./a.out
before: 2
after: -1717986918
```


Linking with Static Libraries

Different approaches

A single relocatable module with all related functions

- Every executable file contains a copy of all functions
- Any changes to the module require library developers to recompile the entire source codes
- `gcc main.c /usr/lib/libc.o`

Separate relocatable files for each function

- App programmers need to link each modules => time consuming and error prone
- `gcc main.c /usr/lib/printf.o /usr/lib/scanf.o ...`

Static library

- Related functions are compiled into separate object modules and **packaged into a single file** called static library
- Linker will copy only the modules that are actually referenced
- `gcc main.c /usr/lib/libc.a`

Creating Static Libraries

```
/* addvec.c */
int addcnt = 0;
void addvec(int *x, int *y, int *z,
            int n) {
    int i;
    addcnt++;
    for(i = 0; i < n; i++)
        z[i] = x[i] + y[i];
}
```

```
/* mulvec.c */
int mulcnt = 0;
void mulvec(int *x, int *y, int *z,
            int n) {
    int i;
    mulcnt++;
    for(i = 0; i < n; i++)
        z[i] = x[i] * y[i];
}
```

```
/* main2.c */
#include <stdio.h>
void addvect(int *x, int *y, int *z,
             int n);
void mulvect(int *x, int *y, int *z,
             int n);

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;
}
```

```
$ gcc -c addvec.c
$ gcc -c mulvec.c
$ ar rcs libvector.a addvec.o mulvec.o
$ gcc -static main2.c ./libvector.a
```

Creating Static Libraries

```
$ gcc main2.c ./libvector.a (without -static option)
$ readelf -s a.out
```

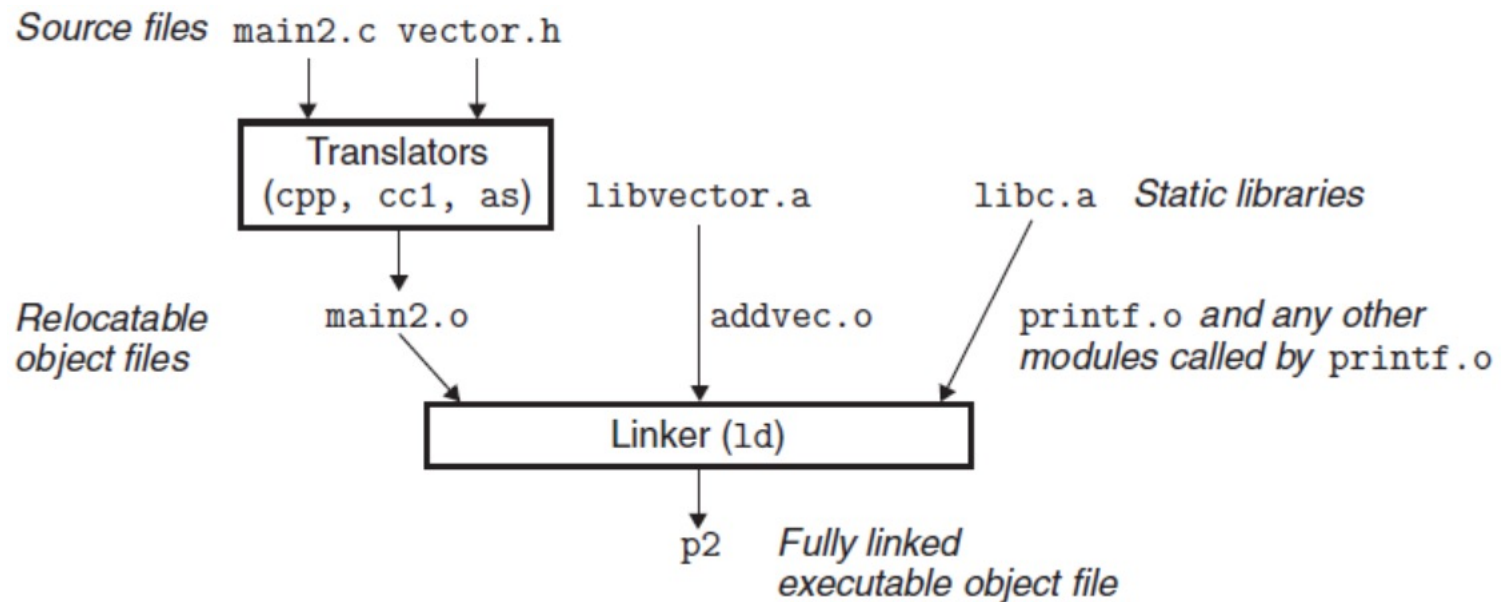
...

Symbol table '.symtab' contains 44 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
...							
10:	0000000000000000	0	FILE	LOCAL	DEFAULT	ABS	main2.c
11:	0000000000000000	0	FILE	LOCAL	DEFAULT	ABS	addvec.c
...							
25:	0000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	printf
...							
27:	0000000000400524	73	FUNC	GLOBAL	DEFAULT	13	main
...							
36:	0000000000402020	8	OBJECT	GLOBAL	DEFAULT	24	x
37:	0000000000402028	8	OBJECT	GLOBAL	DEFAULT	24	y
38:	0000000000402044	8	OBJECT	GLOBAL	DEFAULT	25	z
39:	0000000000400570	106	FUNC	GLOBAL	DEFAULT	13	addvec
40:	0000000000402040	4	OBJECT	GLOBAL	DEFAULT	25	addcnt

...

Linking with Static Libraries



Relocation

Once the symbol resolution step is done

- Linker knows the exact sizes of code and data sections

Relocation steps

1) Relocating **sections** and **symbol definitions**

- Merge all sections of the same type into an aggregate section of the same type
- Assign run-time memory address to sections and symbols

2) Relocating **symbol references** within sections

- Modify symbol references in code and data sections so that they point to the run-time addresses

Relocating Symbol References

ELF Relocation Entry

- Relocation entry is generated when an assembler encounters a reference whose location is unknown.
- Relocation entries for code and data are placed in `.rel.text` and `.rel.data` sections
- Of many relocation types, we are interested in
 - `R_X86_64_PC32`: 32 bit PC relative address
 - `R_X86_64_32`: 32 bit absolute address type

```
typedef struct {  
    long offset;        // Offset of the reference to relocate  
    long type:32,        // Relocation type (R_X86_64_PC32, R_X86_64_32)  
        symbol:32;      // Symbol table index  
    long addend;        // Constant part of relocation expression  
} Elf64_Rela;
```

Relocating Symbol References

```
1: foreach section s {
2:     foreach relocation entry r {
3:         refptr = s + r.offset; // reference to be relocated
4:
5:         // Relocate a PC-relative reference
6:         if (r.type == R_X86_64_PC32) {
7:             refaddr = ADDR(s) + r.offset; // ref's run-time address
8:             *refptr = (unsigned)(ADDR(r.symbol) + r.addend - refaddr);
9:         }
10:
11:         // Relocate an absolute reference
12:         if (r.type == R_X86_64_32)
13:             *refptr = (unsigned)(ADDR(r.symbol) + r.addend);
14: }
```

s: address of section

refptr (line 3) points to the data that need to be relocated

ADDR(s): *run-time* address for the section *s*

ADDR(r.symbol): *run-time* address for the symbol *r.symbol*

Relocating Symbol References

```
0: 48 83 ec 08          sub    $0x8, %rsp
4: be 02 00 00 00      mov    $0x2, %esi    %esi = 2
```

```
9: bf 00 00 00 00      mov    $0x0, %edi    %edi = &array
                        a: R_X86_64_32 array    Relocation entry
```

```
e: e8 00 00 00 00      callq  13 <main+0x13> sum()
                        f: R_X86_64_PC32 sum-0x4 Relocation entry
```

```
13: 48 83 c4 08          add    $0x8, %rsp
17: c3                  retq
```

```
-----
int main()
{
    int val = sum(array, 2);
    return val;
}
```

Code and relocation entries from main.o

Relocating PC-Relative References

callq 13 at address e:

- Relocation entry: {r.offset = 0xf, r.symbol = sum, r.type = R_X86_64_PC32, r.addend = -4}
- ADDR(s) = ADDR(.text) = 0x4004d0
- ADDR(r.symbol) = ADDR(sum) = 0x4004e8
- refaddr = ADDR(s) + r.offset = 0x4004df
- *refptr = ADDR(r.symbol) + r.addend - refaddr
= 0x4004e8 - 4 - 0x4004df = 0x5

In the executable object file

- 4004de: e8 05 00 00 00 callq 4004e8 <sum>
- Push PC onto stack
- $PC \leftarrow PC + 5$

Relocating Absolute References

`mov $0x0, %edi` at address 9:

- Relocation entry: $\{r.offset = 0xa, r.symbol = array, r.type = R_X86_64_32, r.addend = 0\}$
- $ADDR(r.symbol) = ADDR(array) = 0x601018$
- $*refptr = ADDR(r.symbol) + r.addend$
 $= 0x601018 + 0 = 0x601018$

In the executable object file

- `4004d9: bf 18 10 60 00      mov $0x601018, %edi`

Questions?
