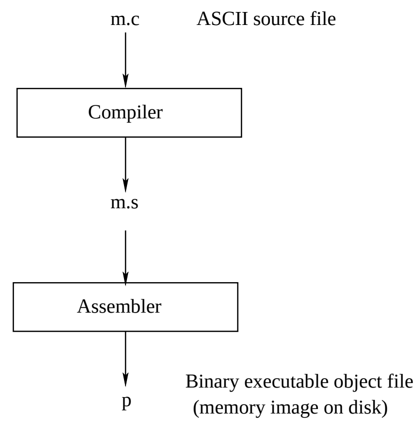
CS429: Computer Organization and Architecture
Linking I & II

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

Last updated: April 5, 2018 at 09:23

A Simplistic Translation Scheme

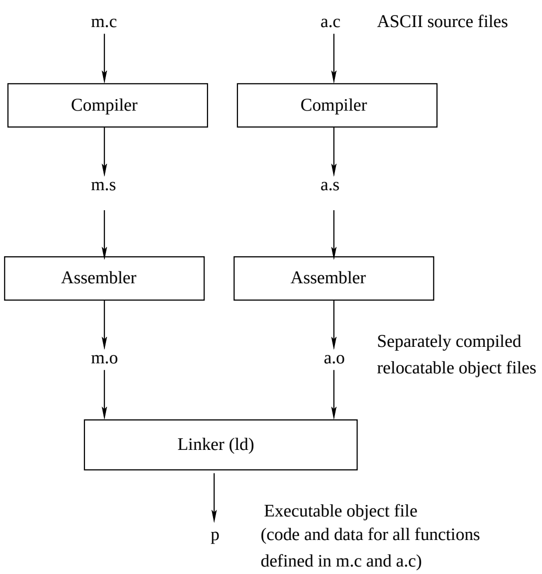


Problems:

- *Efficiency:* small change requires complete re-compilation.
- *Modularity:* hard to share common functions (e.g., printf).

Solution: Static linker (or linker).

Better Scheme Using a Linker



Linking is the process of combining various pieces of code and data into a single file that can be *loaded* (copied) into memory and executed.

Linking could happen at:

- compile time;
- load time;
- run time.

Must somehow tell a module about symbols from other modules.

Linking

A *linker* takes representations of separate program modules and combines them into a single *executable*.

This involves two primary steps:

1. *Symbol resolution:* associate each symbol reference throughout the set of modules with a single symbol definition.
2. *Relocation:* associate a memory location with each symbol definition, and modify each reference to point to that location.

Translating the Example Program

A *compiler driver* coordinates all steps in the translation and linking process.

- Typically included with each compilation system (e.g., gcc).
- Invokes the preprocessor (cpp), compiler (cc1), assembler (as), and linker (ld).
- Passes command line arguments to the appropriate phases

Example: Create an executable p from m.c and a.c:

```
> gcc -O2 -v -o p m.c a.c
cpp [args] m.c /tmp/cca07630.i
cc1 /tmp/cca07630.i m.c -O2 [args] -o /tmp/cca07630.s
as [args] -o /tmp/cca076301.o /tmp/cca07630.s
<similar process for a.c>
ld -o p [system obj files] /tmp/cca076301.o /tmp/
    cca076302.o
>
```


Role of the Assembler

- Translate assembly code (compiled or hand generated) into machine code.
- Translate data into binary code (using directives).
- Resolve symbols—translate into relocatable offsets.
- Error checking:
  - Syntax checking;
  - Ensure that constants are not too large for fields.

What Does a Linker Do?

Merges object files

- Merges multiple relocatable (.o) object files into a single executable object file that can be loaded and executed.

Resolves external references

- As part of the merging process, resolves external references.
- *External reference:* reference to a symbol defined in another object file.

Relocates symbols

- Relocates symbols from their relative locations in the .o files to new absolute positions in the executable.
- Updates all references to these symbols to reflect their new positions.
- References can be in either code or data:
  - code: `a();` `/* reference to symbol a */`
  - data: `*xp = &x;` `/* reference to symbol x */`

Why Linkers?

Modularity

- Programs can be written as a collection of smaller source files, rather than one monolithic mass.
- Can build libraries of common functions shared by multiple programs (e.g., math library, standard C library)

Efficiency

- Time:
  - Change one source file, recompile, and then relink.
  - No need to recompile other source files.
- Space:
  - Libraries of common functions can be aggregated into a single file.
  - Yet executable files and running machine images contain only code for the functions they actually use.

Example C Program

m.c

```
int e = 7;

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

a.c

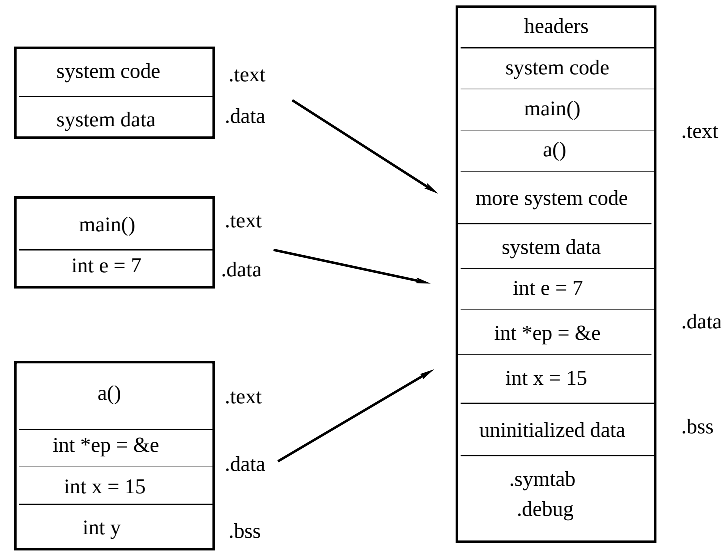
```
extern int e;

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

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


Merging Relocatable Object Files

Relocatable object files are merged into an executable by the Linker. Both are in ELF (Executable and Linkable Format).



Relocating Symbols and Resolving External References

- *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.

Relocating Symbols and Resolving External References (2)

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.o Relocation Info

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
```


a.o Relocation Info (.text)

a.c

```
extern int e;

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

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

Disassembly of section .text

```
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.o Relocation Info (.data)

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
```


Strong and Weak Symbols

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
   ...
}
```

Linker Symbol Rules

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 unitialized 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.

The Complete Picture