Topic 6 – Linkers

Key Ideas

- what the goal of a linker
- external symbol references (ESR)
- external symbol definitions (ESD)
- steps in linking files
- dynamic vs. static linking

Assemblers, Loaders and Linkers

What They Do

- Assemblers
  - *what:* need two passes to translate labels
  - *why:* so labels can be used before they are defined
- Loader
  - *what*: need to track and adjust labels that were used in a *.word* assembler directive.
  - *why:* allows a program to be loaded anywhere into RAM
- Linker
  - *what:* use multiple files for code
  - *why: …*

Linking

Why Link Object Code Files?

- Answer*: so we can break up a large program into several modules* (i.e. easier to manage pieces).
- Why break-up large programs?
- Answers: For the same reasons we do so for high level languages.
  - *Procedural Abstraction*: programmers just need to know interface not how the subroutine is implemented.
  - Collect related subroutines together.

Linking

Why Link Object Code Files?

- Why break-up large programs?
  - Create a collection of subroutines (i.e. a library) that can be used in many programs.
  - Errors are easier to track down.
  - Different people/ groups can be responsible for different modules.
  - Avoid duplication of effort (e.g. same print integer subroutine created many times)

Linking Files

How to Link: Attempt 1

- Recall Goal: *use multiple files for code.*
- Attempt 1: just combine (i.e. concatenate) all the small files assembly language files into one big one and then assemble.
- A small change in one small file would mean redoing everything.
- May just want to distribute the object code not the assembly language code.
- Requirement #1: *We need a tool that works with multiple MERL files.*

Linking Files

How to Link: Attempt 2

- Attempt 2: assemble all the MERL files then concatenate (i.e. join) together.
- When assembling, we start at address 0x0, so all files would start at the same location.
- If you concatenate two MERL files, the result is not a MERL file.
- Requirement #2: *We need a tool that outputs the MERL format.*
- Requirement #3: *We need a tool that works with labels defined in one file and used in another.*

Linking Files

How to Link: The External Symbol Reference (ESR)

- *Create a directive, .import, that tells the assembler that this symbol occurs in another file* (i.e. externally).
- The assembler does not translate this statement into an instruction. It provides information to the assembler.
- For example *.import notify_nsa* means that the symbol *notify_nsa* is defined in another file.
- When assembling, initially assign the value of 0 to this symbol, but make a note in the MERL file that this symbol is not yet defined.
- If you never find it, then report an error.

Linking Files

The External Symbol Reference (ESR) Format

- *In the third section of MERL file, create an ESR entry*.
- Note there is one ASCII char per word to represent the chars in the symbol (here a label).
- It is in the following format

word 1:	0x11	
word 2:	address	; where the symbol is used
word 3:	length	; of the symbol in bytes (say n)
word 4:	1st char of symbol (in ASCII)	
word 5:	2nd char of symbol (in ASCII)	
...	...	
word n+3:	last char of symbol (in ASCII)	

Linking Files

The External Symbol Reference (ESR) Format

- The first word is always 0x11 which signifies that whatever follows is an ESR.
- Concern: *What if multiple files use the same symbol?*

file1.asm

```
.import abc
lis $1
.word abc
```

file2.asm

```
; abc is a loop
abc:
  …
beq $1, $2, abc
```

file3.asm

```
; abc is a proc
abc:
  sw $1, -4($30)
  sw $2, -8($30)
```

Linking Files

The External Symbol Definition (ESD)

- Requirement: Need some sort of way to *provide information hiding.*
- We want to differentiate between a symbol meant for local use (within a file) and one meant for global use (external to the file).
- *Use the .export directive* to indicate that other files may use (i.e. refer to) this symbol.
- A symbol can only be defined once, but can be referenced many times.

Linking Files

The External Symbol Definition (ESD) Format

- Using *.export* is like declaring a variable global.
- The *.import .export* pair links the definition in one file to its reference in another.

file1.asm

```
.import abc
lis $1
.word abc
```

file2.asm

```
; abc is a loop
abc:
  …
beq $1, $2, abc
```

file3.asm

```
; abc is a proc
.export abc
abc:
   sw $1, -4($30)
   sw $2, -8($30)
```

Linking Files

The External Symbol Definition (ESD) Format

- *In the third section of MERL file, create an ESD entry.*
- It is similar in format to the ESR entry except the entry type is now 0x05 (rather than 0x01 or 0x11).

word 1:	0x05	
word 2:	address	; that symbol refers to
word 3:	length	; of the symbol in bytes (say n)
word 4:	1st char of symbol (in ASCII)	
word 5:	2nd char of symbol (in ASCII)	
…	…	
word n+3:	last char of symbol (in ASCII)	

Linking Files

Review: Modifications to Create a MERL Assembler

Pass 1 Changes

- record the size of the file
- when you encounter a `.word <label>` instruction
    - record the location

Pass 2 Changes

- first output header
- then the MIPS machine code
- finally output the relocation table

Linking Files

Modifications to Handle External References

Pass 1 Changes

- when you encounter a `.import <symbol>` directive record each symbol that needs importing
- when you encounter a `.export <symbol>` directive record each symbol that needs exporting

Pass 2 Changes

- create an ESR entry for each symbol that is imported
- create an ESD entry for each symbol that is exported

Linker Pseudocode

Goal: handle multiple files and external symbols

1. Concatenate the programs
2. Combine and adjust ESDs
3. Use new ESDs to update old ESRs
4. Relocate addresses (internally)

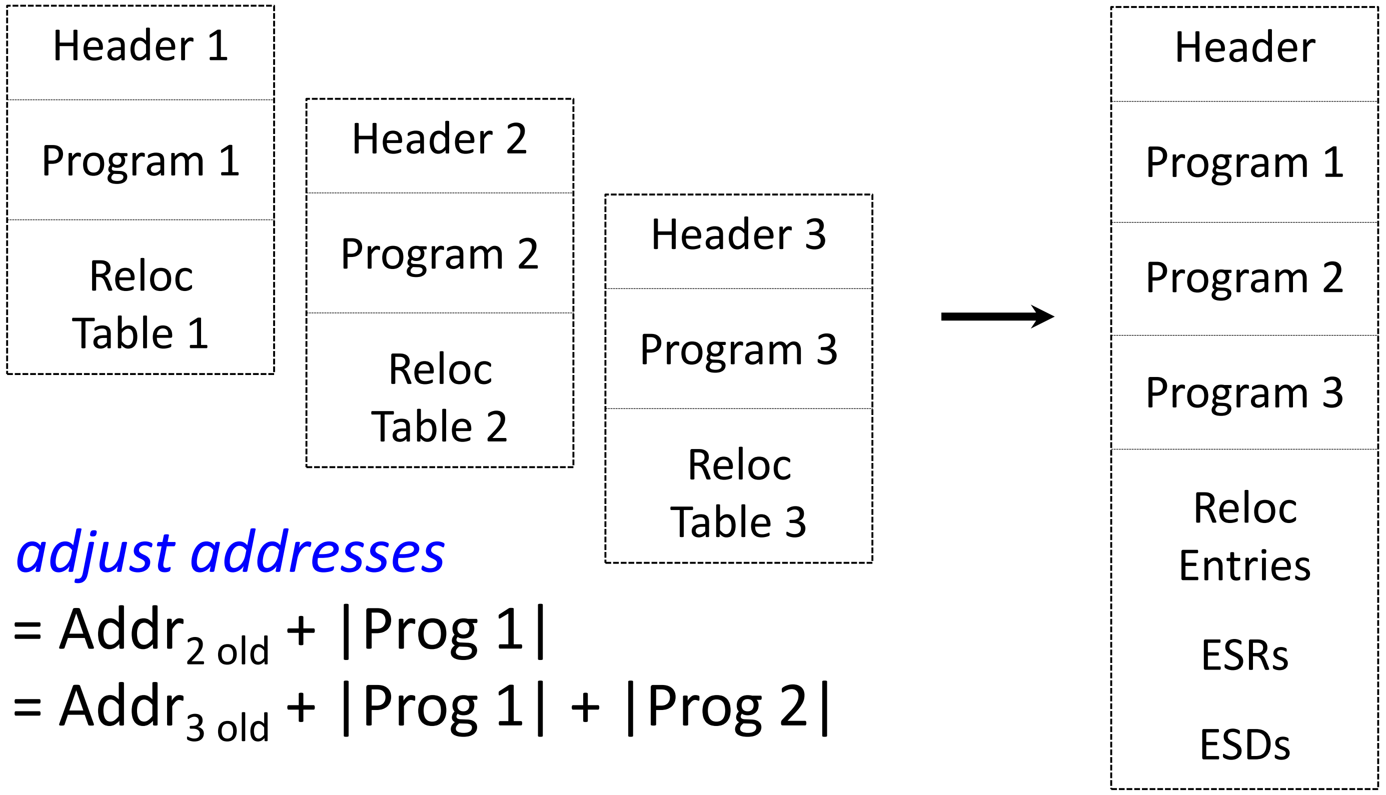
Key Task: like loading, addresses need to be adjusted.

If file2.asm is added to the end of file1.asm then the addresses in file2.asm need to be adjusted to take in account that they now occur after file1.asm.

Linker Pseudocode

Step 1: Concatenate Programs

- Note that you will not be able to finalize the header and the ESRs and ESDs initially.



Key Task: adjust addresses

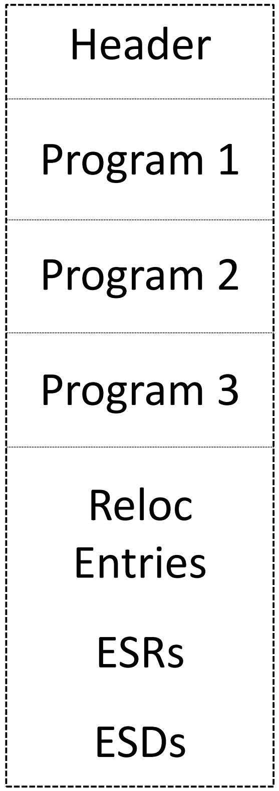
$Addr_{2\ new} = Addr_{2\ old} + |Prog\ 1|$

$Addr_{3\ new} = Addr_{3\ old} + |Prog\ 1| + |Prog\ 2|$

Linker Pseudocode

Step 2: Combine and Adjust ESDs

- Combine all the External Symbol Definitions (ESDs)
  - Program 1's ESDs have no change.
  - *Programs 2's ESDs have to be shifted down by the size of Program 1*, i.e.
    $ESD_{2\ new} = ESD_{2\ old} + |Prog\ 1|$
  - Programs 3's ESDs have to be shifted down by the size of Program 1 + the size of Program 2, i.e.
    $ESD_{3\ new} = ESD_{3\ old} + |Prog\ 1| + |Prog\ 2|$
- You can get the size of each program from its original header.



Linker Pseudocode

```
**Step 3:** Use new ESDs to update old ESRs
  for each old ESR
      look up the new ESD
      if found
          update the value at the location + offset
          (i.e. it is no longer referenced externally)
      else
          adjust the new ESRs with the new offset, e.g.
          $ESR_{2\ new} = ESR_{2\ old} + |Prog\ 1|$
          $ESR_{3\ new} = ESR_{3\ old} + |Prog\ 1| + |Prog\ 2|$ ...
```

Linker Pseudocode

Step 4: Relocate addresses (internally)

- just like what was done for loading, any *relocatable addresses in programs 2, 3, etc. need to be relocated*.
- for each relocation entry
  - add the appropriate offset in the code
  - add the appropriate offset in the relocation entry

$Addr_{2\ new} = Addr_{2\ old} + |Prog\ 1|$

$Addr_{3\ new} = Addr_{3\ old} + |Prog\ 1| + |Prog\ 2|$

Dynamic Linking

Static vs. Dynamic Linking

- What we have just described is called *static linking,* i.e. the files are all linked before the program is loaded.
- A contrasting approach, especially among commonly used libraries, is to use *dynamic linking.*
- *shared libraries*
  - many programs use the I/O or math libraries
  - several programs may be using it at the same time
  - idea: keep only one copy of object code in memory
  - reserve memory area for relocatable object code

Dynamic Linking

Dynamic Linking

- *dynamic libraries*
  - do not add the object code to executable file
  - combine object code at load time
- *dynamic linking*
  - relocate and resolve symbols at load time
  - a program may halt because it is “missing a DLL”

What are they called?

- dynamic link library (DLL), dll file, in Windows
- shared object file, so file, in Linux
- dylibs in Mac OS

Relocation

Assembly	**Machine Code**		**Loaded at 0x0**	
lis $1	0x0	0000 0814	0x0	0000 0814
.word 1	0x4	0000 0001	0x4	0000 0001
⋮	⋮	⋮	⋮	⋮
lis $3	0x20	0000 1814	0x20	0000 1814
.word p	0x24	0000 0040	0x24	0000 0040
jalr $3	0x28	0060 0009	0x28	0060 0009
⋮	⋮	⋮	⋮	⋮
p: sw $2, -4($30)	0x40	AFC2 FFFC	0x40	AFC2 FFFC
⋮	⋮	⋮	⋮	⋮
jr $31	0x5C	03E0 0008	0x5c	03E0 0008

Relocation

Assembly	Machine Code		Loaded at 0x100	
lis $1	0x0	0000 0814	0x100	0000 0814
.word 1	0x4	0000 0001	0x104	0000 0001
⋮	⋮	⋮	⋮	⋮
lis $3	0x20	0000 1814	0x120	0000 1814
.word p	0x24	0000 0040	0x124	0000 0140
jalr $3	0x28	0060 0009	0x128	0060 0009
⋮	⋮	⋮	⋮	⋮
p: sw $2, -4($30)	0x40	AFC2 FFFC	0x140	AFC2 FFFC
⋮	⋮	⋮	⋮	⋮
jr $31	0x5C	03E0 0008	0x15C	03E0 0008

Relocation

Assembly	Machine Code		Loaded at 0x2000	
lis $1	0x0	0000 0814	0x2000	0000 0814
.word 1	0x4	0000 0001	0x2004	0000 0001
⋮	⋮	⋮	⋮	⋮
lis $3	0x20	0000 1814	0x2020	0000 1814
.word p	0x24	0000 0040	0x2024	0000 2040
jalr $3	0x28	0060 0009	0x2028	0060 0009
⋮	⋮	⋮	⋮	⋮
p: sw $2, -4($30)	0x40	AFC2 FFFC	0x2040	AFC2 FFFC
⋮	⋮	⋮	⋮	⋮
jr $31	0x5C	03E0 0008	0x205C	03E0 0008

Dynamic Linking

f1.asm

```
            .import pr
0x0C       lis $1
              ⋮
0x30       lis $2
0x34       .word a
0x38       jalr $2
              ⋮
0x50       lis $3
0x54       .word pr
0x58       jalr $3
              ⋮
0x70  a:  sw $4, -4($30)
              ⋮
0x108       jr $31
0x10C      ; Epilogue

; code 0x100 bytes long
```

f1.merl

```
0x000   0x1000 0002
0x004         0x128
0x008         0x10C

0x00C   0x0000 0814
  ⋮          ⋮
0x108   0x03e0 0008

0x10C           0x1
0x110          0x34
0x114           0x5
0x118          0x54
0x11C           0x2
0x120          0x70
0x124          0x72
```

Header
cookie
file size: C + 100 + 1C
header + code = C+100

Code *f1*

Epilogue
relocation entry
relocation addr (a)
Ext Symbol Reference
ESR address (pr)
length of symbol
ASCII p
ASCII r

Dynamic Linking

f2.asm

```
           .export pr
0x0C      lis $1
              ⋮
0x20      lis $1
0x24      .word b
0x28      jalr $1
              ⋮
0x40  b:  sw $2 -4($30)
              ⋮
0x60  pr: sw $3 -4($30)
              ⋮
0x88       jr $31
0x8C     ; Epilogue
; code 0x80 bytes long
```

f2.merl		
		Header
0x000	0x1000 0002	cookie
0x004	0x108	file size: C + 80 + 1C
0x008	0x08C	header + code = C + 80
0x00C	0x0000 0814	**Code *f2***
⋮	⋮	
0x088	0x03e0 0008	
		Epilogue
0x08C	0x1	relocation entry
0x090	0x24	relocation addr (b)
0x094	0x11	Ext Symbol Definition
0x098	0x60	ESD address (pr)
0x09C	0x2	length of symbol
0x100	0x70	ASCII p
0x104	0x72	ASCII r

Dynamic Linking

	f.merl	Header
0x000	0x1000 0002	cookie
0x004	0x	file length
0x008	0x18C	code length C + 100 + 80
0x00C	0x0000 0814	Code *f1* - not shifted
⋮	⋮	⋮
0x108	0x03e0 0008	
0x10C	0x0000 0814	Code *f2* - shifted by 100
⋮	⋮	⋮
0x188	0x03e0 0008	
0x18C		Epilogue…

Dynamic Linking

	f.merl	Epilogue
0x18C		relocation entry
0x190		relocation addr (a)
0x194		relocation entry
0x198		relocation addr (pr)
0x19C		relocation entry
0x1A0		relocation addr (b)
0x1A4		Ext Symbol Definition
0x1A8		ESD address (pr)
0x1AC		length of symbol
0x1B0		ASCII p
0x1B4		ASCII r