Lecture #27

Separate Compilation

- Would like to have in our program the line:
  JSR Sqrt
  where "Sqrt" is a label in a *different* program!
  - Aside 1: what does your current assembler do with such a thing?
  - Aside 2: Sqrt must still be on the same page as the JSR instruction!
- We extend our language and provide a (typical) mechanism for resolving this...

New Pseudo Op: EXT

1. .EXT *symbol_name*
   - "external"
   - indicates that *symbol_name* is defined in a different program
   - legal to use, but assembler can't fill it in

```
Prog  .ORIG
      .EXT      Sqrt
      JSR       Sqrt
      TRAP x25
      .END
```

New Pseudo Op: ENT

2. .ENT *symbol_name*
   - *symbol_name* is defined in *this* program
   - it is a *global* symbol
   - i.e., may be referenced in other programs

- These pseudo ops change the "scope" of a symbol
  - local (default) -----> global (with .ENT)
- Q: why not make global the default, or make them all global?

Example: Two Programs

```
Main  .ORIG
      .EXT  Sqrt
        . . .
      JSR Sqrt
        . . .
      .END
```

```
Subr    .ORIG
        .ENT Sqrt
        . . .
Sqrt    ST R1,Tmp1
        . . .
        RET
Tmp1   .BLKW #1
        .END
```

- These two program can now be:
  - independently written
  - independently assembled into 2 object files
- How does the *linkage* between these object files get resolved?

So now back to loaders…

Binary Symbolic Subroutine Loaders (BSS)

- One of the first relocating loaders
  - 1956: IBM, GE, UNIVAC
- Allows multiple program segments ("control sections" in your textbook)
  - different languages
  - different times

  } Separate compilation!
- Let's examine each of the "tasks" in turn…

Tasks for BSS Loader

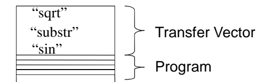
- Allocation
  - assembler calculates each segment length
  - loader adds them all up
  - load location obtained from OS
- Relocation
  - assembler flags words for relocation (bit masks)
  - loader makes modifications

Tasks for BSS Loader II

- Linking
  - by loader (with help from assembler)
  - a restricted form
  - uses a "transfer vector"
- Loading
  - by loader

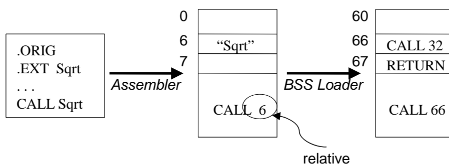
Transfer Vector

- Contains 1 entry / external symbol used by this program segment
- Assembler sets aside room at the beginning of the object file for TV
- Assembler places symbolic representation of referenced external symbols in TV



Transfer Vector II

- All calls to external symbols are replaced with calls to appropriate locations in TV
- Loader replaces the entries in TV with calls to the appropriate location



Example

```
Prog   .ORIG
       .EXT   Sqrt, Print
Soln   .BLKW  #1
main   LD     R1,=#16
       JSR    Sqrt
       ST     R2,Soln
       LD     R1,Soln
       JSR    Print
       TRAP   x25
       .END   main
```

```
000A  5555
      2211
      4800
      340A
      220A
      4805
      F025
0011  0010
```

```
0000  3C04
      48A5
      2C04
      D000
      0000
```

```
0005  3C09
      48C0
      2C09
      D000
      0000
```

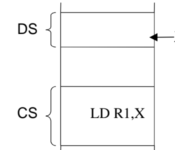
```
BProg 00000012
V0000Sqrt
V0005Print
T000B2211M9
T000C4800M9
T000D340AM9
T000E220AM9
T000F4805M9
T0010F025
T00110010
E000B
```

Disadvantages of Transfer Vector

- Overhead
  - time (extra call instruction)
  - space (for transfer vector in object file)
- Works for subroutine calls, but what about for sharing *data*?
  - e.g., LD R1,XValue
  - this cannot be replaced with a call to TV, or even a load from the TV

Data Sharing with BSS Loaders

- Permit 1 common (shared) “data segment”
- All external data is in this one data segment

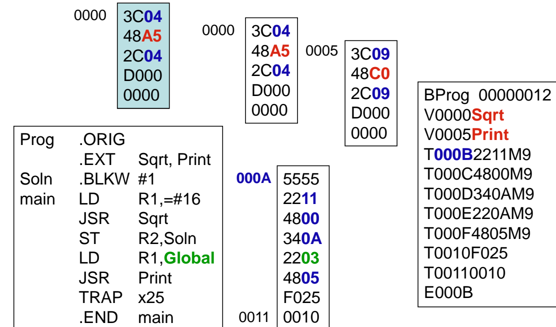


- Assembler replaces X with its (relative) address in the data segment

Impact on Relocation

- What does this mean for relocating the program?
- There are now 2 different kinds of “relative”!
  - relative to the LL of data segment, and relative to the LL of the control segment
- Assembler must distinguish between them
  - extend relocation information
  - e.g., use 2 bits per word
    - 00 - absolute
    - 01 - relative (to CS load location)
    - 10 - relative (to DS load location)

Example



Lecture #28

Direct Linking Loaders

Introduction

- General linking/loading strategy
- Very common in modern systems
- And used in Lab #3!
- Advantages:
  - separate assembly
  - multiple control *and* data segments
  - lower time overhead (in program execution)
  - lower space overhead (in program's object file)

Assembler Responsibilities

1. Header information
   - length of segment
   - execution start address
2. List of entry points
   - those defined in this segment
   - gives their (relative) value
3. List of external references
   - used in this segment
   - defined outside of segment

Assembler Responsibilities II

4. Relocation information
   - modification records
5. Machine code
   - text records

- There are some new things here, which suggests defining some new record types…

Entry Record

- Similar to BSS loader
- List and define all the entry points
- Possible format:

  <*Flag*> <*symbol_name*> <*value*>
- Examples:
  - ESqrt 0008   (since symbols are <= 6 characters)
  - ESqrt=0008
- Idea: provide information to loader

For Lab #3

- We'll adopt the following conventions:
  - the program name is always (implicitly) an entry point
  - entry point symbols must be relative

    (if you wish to handle absolute too, that's up to you)

External Record

- Can be combined with text and modification records if you wish
- Examples:
  - .FILL x3202 T301F3202
  - ST R1,Num T301F3202M9
  - ST R1,Enum T301F3200X9Enum
- Format:

  T <*addr*> <*machine_code*> X <*symbol_name*>

External Records II

- Such a record tells the loader to:
  - find CS that defines that (external) symbol
  - find the value of that symbol within that CS (i.e., look at the corresponding entry record!)
  - add the low 9 bits of the value to the text record
  - add the load location of the CS *that defines the symbol* to the text record
    - this last step is just like the usual "relocation operation" of relative symbols, but using the LL of segment that defines the symbol

Lecture #29

Direct Linking Loaders: Algorithm and Data Structures

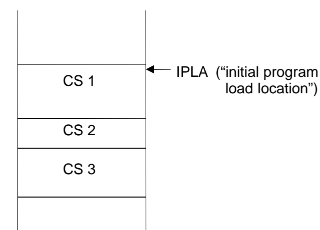
Algorithm & Data Structures

- Problem is similar to assembler:
  - resolve symbols (with forward references)
  - use these values
- Solution is similar too!
  - use _______________
- Pass #1: find definition of all external symbols
- Pass #2: relocate, link, load

Pass #1

- Q. What does the assembler tell the loader about each ENT symbol?
- A.
- So, to determine the *actual* symbol value, loader must calculate:
  _______________ + _______________
- For lab #3, we can load each segment into one contiguous block of memory

Loaded Memory



Pass #1: Pseudocode

```
calculate total size (all segments)
get IPLA (from OS)
for each segment do:
    calculate PLA
    add entry symbols to "external symbol
                                table" (EST)
    if symbol already present
        flag an error
rof
```

Example

Main	.ORIG	
	.EXT	Put
	.ENT	Num
	JMP	Put
Num	.FILL	#7
	.END	

Lib	.ORIG	
	.EXT	Num
	.ENT	Put
Put	LD	R0,Num
	TRAP	x31
	TRAP	x25
	.END	

External Symbol Table

- For our example:

  Name Value R/A

- (assume: IPLA = ________________)
- Note: for lab #3, can restrict external symbols to be relative only

Pass #2: Pseudocode

```
set IPLA
for each CS
    save PLA of this CS
    for each text record in CS
        calculate memory location
        relocate record:  absolute -
                          relative  -
                          external  -
        load the word
    rof
rof
transfer control to "start" of first segment
```

- Recommended exercise: link and load our example (assuming IPLA of ________)

Checking External Symbols

- Q. When do we check whether an external symbol is actually defined?
- A.
- Option #1: during / after pass 1
  - keep a list of symbols seen in external records
  - after pass 1, check this list against EST

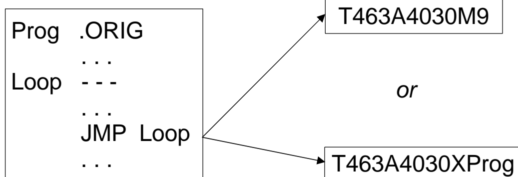
Checking External Symbols II

- Option #2: during pass 2
  - we *know*:
    - external symbols used ⊆ EXT symbols declared
  - read in the records generated from EXT declarations
  - check symbols in these against the EST
- Option #3: during pass 2
  - when each text record is processed, report an error if the symbol is not in the EST

Unifying "X" and "M"

- Don't really need 2 separate mechanisms!
- Recall meanings of
  - T _ _ _ _ _ _ M
  - T _ _ _ _ _ _ XSym
    - "Sym" is in EST
    - *add* this value of "Sym" to address field
- Recall that *segment names* always in EST
- This suggests that X can be seen as a more general form of M!

Replacing "M" with "X"



Linking with Libraries

- Common func's often defined in *libraries*
- Library linking can be made implicit:
  - after pass 1, may still have "unresolved external symbols" in EST
  - if so, search libraries for matching definitions and load them (after pass 1)
  - still some unresolved externals? Then error
- Typically, user-specified libraries searched first, then standard ones (automatically)

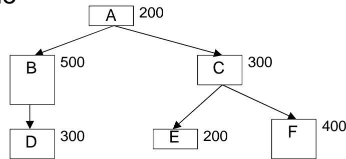
Loader Refinements and Optimizations

Problem: Space

- Consider a program that calls "sqrt", "rnd", and "substr"
- Each defined in its own (large) library
- So, linked and loaded program is *huge*
- Solutions (for saving memory):
  - virtual memory and paging
  - dynamic loading
  - dynamic linking

Dynamic Loading

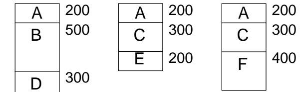
- Observe: program does 1 thing at a time
  - ∴ don't need all segments present simultaneously
- Example



Total Size = 1.9 Mb

Dynamic Loading - Overlays

- B/D never together with C/E/F
- Define an "overlay structure" for how segments can be swapped in and out
- 3 scenarios:



Total Size = 1000 700 900

- Only 1Mb needed (length of longest path)
- Trade-off: memory space & performance

Dynamic Linking

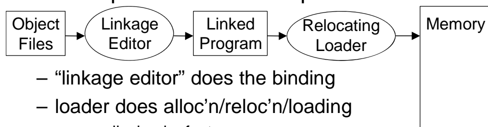
- Instead of branching directly to an external symbol, program issues a call request to OS
  - subroutine name is parameter for request
- OS responsibilities
  - keep table of loaded libraries
    - loads new library if needed
    - manages swapping of libraries as appropriate
  - transfer control to appropriate subroutine
  - return control to original program

Dynamic Linking II

- "Binding": the association of an actual address (x6E5E) with a symbolic name ("Sqrt")
- Dynamic linking delays binding from *load* time to *execution* time ("late binding")
- Advantages:
  - many programs can share 1 loaded library
  - library can be recompiled on-the-fly
  - library only loaded if actually used

Problem: Time

- Every time we want to execute a program, must re-link, relocate and re-load
  - costly if object code hasn't changed
- Idea: separate these two operations



  - "linkage editor" does the binding
  - loader does alloc'n/reloc'n/loading
    - small, simple, fast