Loaders

* operating system controls the programs that run on a processor
* to run a program, OS must load it from disk storage into main mem (RAM) and prep it for execution
  + OS uses another program called a loader to do this
* pseudocode for OS and loader are below:

Text

Description automatically generated

* + loader returns starting address of program that was just loaded
  + OS jumps to that address

Text

Description automatically generated

* + var N rep amount of mem the loader decides to allocate to the program
    - includes mem needed for instructions, stack, and heap
  + loader search for block of mem that has at least N contiguous bytes of mem to use for the program
    - α rep starting address of this block
  + loader also sets $30 to just past last word assigned to program because we assumed loader would initialize $30 to top of stack
* since loader isn’t loading first address at 0x00 but rather at α, .word directives will assemble to the wrong binary instruction because we replace the .word with the mem address of where the label was defined
  + mem address computed by assembler assumes that program was loaded at starting address 0x00
  + solution is to add α to the value at that location, but problem is that loader is going to load a stream of bits and we can’t distinguish which instructions were produced from the .word directive
* assembler should generate object code, which contains machine code for the assembly program plus additional info needed by the loader and linker
  + e.g. assembler encodes info of what locations contain instances of a .word used with a label

MERL Format

* we use object code format called MERL, which stands for MIPS Executable Relocatable Linkable
* assembler generates MERL output for MIPS assembly program in a binary format
* MERL format contains 3 sections:
  + header is exactly 3 words
    - word 1 is binary encoding of beq $0, $0, 2 (i.e. unconditional jump over next 2 words)
    - word 2 is address just past end of MERL file (i.e. indicates where MERL footer ends)
    - word 3 is address just past end of assembled code (i.e. indicates where MERL footer begins)
  + assembled code
  + footer that contains relocation entries, each with 2 words
    - first word is format specifier and has value 1 to indicate that it’s a relocation (REL) entry
    - second word is address of word that must be relocated (i.e. α should be added to word at address specified in second word of relocation entry)
* e.g. shown below is both non-relocatable assembled code and relocatable MERL output (also shown in assembly for easier reading)

Table

Description automatically generated

* + since we added MERL header before code segment, words appear 12 bytes ahead of what they used to be in the non-MERL version of the code
    - relocation entries use updated locations

Loader Relocation Algorithm

* algorithm for relocation:

Text, letter

Description automatically generated

* never hard-code memory addresses in MIPS assembly and always use labels when referring to locations
  + e.g.

Text

Description automatically generated with low confidence

* + - code loads address 12 into $2 and jumps to that address
    - if program doesn’t start at 0x00, it won’t execute properly because there’s no relocation entry for .word 12

Linkers

* linker assembles separate files first and then concatenates them into a combined program

Diagram

Description automatically generated

* + linker also takes care of externally defined labels and makes sure there’s only one header and footer in the final MERL file
* whenever assembler encounters label that can’t be resolved, it’ll create an External Symbol Reference (ESR) entry in MERL footer
  + only create ESR entries for labels that have been listed using the .import directive so that we’re not creating an ESR entry for an incorrect label
  + format code for ESR entry is 0x11
  + e.g.

Text

Description automatically generated with medium confidence

Text

Description automatically generated

* for files that will provide labels that are linked and used in another file, we need to indicate the presence of such labels
  + External Symbol Definitions (ESD) entries are created for all of these labels
  + programmer specifies which labels they want to be linkable by using the .export directive
  + ESD entries are only created for labels that also have the .export directive
  + format code for ESD entry is 0x05
* e.g.

Text

Description automatically generated with medium confidence

Text

Description automatically generated

* linking algorithm (m1.code refers to just code segment and m1.table refers to just table segment of m1 MERL file):
  + task 1: check for duplicate labels
    - confirm 2 MERL files don’t export the same symbols
  + task 2: combine code segments
    - order in which MERL files are provided to link matters (e.g. linking m1 and m2 in this order will result in m1.code appearing before m2.code in linked file’s code segment)

Diagram

Description automatically generated

* + task 3: relocate m2.table
    - update addresses in REL entries in m2’s table
    - relocation offset is equal to m1’s code segment minus 12 (size of header) since we’re not duplicating header from m2 in linked file
  + task 4: relocate m2’s code using REL entries
    - go through each REL entry in modified m2.table and update corresponding lines of m2.code by adding relocation offset computed in task 3 to each line
  + task 5: resolve imports for m1
    - for each ESR entry in m1.table, check if m2.table has corresponding ESD
    - update location of where ESR occurs in m1’s code with address of where exported label is defined
    - after resolving label, change ESR entry to REL entry because if label’s definition was to move due to relocation, address would also need to be relocated
  + task 6: resolve imports for m2
    - do the exact same as in task 5 but with m1 and m2 tables swapped
  + task 7: create table for linked MERL file
    - final table is concatenation of all ESD, REL, and any unresolved ESR entries from m1.table and m2.table
    - order in which table entries are stored don’t matter
  + task 8: compute header for linked MERL file
    - endCode is location of where code segment ends and it’s computed by adding 12 (size of header) to combined code segment concatenated in task 2
    - endModule is endCode plus size of table that was created for linked MERL file in task 7
  + task 9: output linked MERL file
    - output header which requires the MERL cookie (i.e. beq $0, $0, 2 or 0x10000002), endModule, and then endCode
    - output combined code produced in task 2
    - output table produced in task 7
* pseudocode for linking algorithm:

Text

Description automatically generated

Text

Description automatically generated