CMPS-224 Homework 2: Assemblers, Linkers & Loaders - Creating the Process Image

Refer to as needed:

- CS Units of Measure
- MIPS32 ISA
- MIPS quick guide

All questions except #17 are taken from Appendix A.1 - A.5. Read A.1-A.5 completely before starting homework. Do not worry about references in the Appendix to chapters in the text, we will cover that material as needed.

1. How does assembly language differ from machine language? hint

Assembly language is human-readable whereas machine language is pure binary. Assembly is a symbolic representation of the binary encoding of machine language. Machine language is ready to be executed by the processor - assembly must be translated into machine by the assembler.

2. Code written in a high-level language (C/Java/Python) can, in theory, be ported to a different architecture without modifying the source. An assembly language program, on the other hand, is specific to a certain architecture; i.e., a MIPS assembly program will only assemble on a MIPS architecture, x86 assembly is for Intel architectures, etc. Why is assembly not portable unless you use a cross-assembler? hint

The processor of a machine defines the machine architecture. Each processor type has its own unique instruction set architecture (ISA). The assembly language for an architecture is based on the instruction set for that processor. Unless you use a cross-assembler or a simulator - you cannot assemble and execute MIPS code, for example, on an x86 machine.

3. Define the terms assembler, macro, and object file. hint

An assembler takes an assembly language source and produces object code (machine code); a macro is a set of frequently used instructions (pseudo-instructions) that are replaced at by the assembler to make programming easier, an object file is machine code that cannot be executed yet because it needs the linker to resolve addresses.

4. What is the difference between a compiler and an assembler? Compare GNU C compiler and GNU gas assembler under Linux as a specific example. hint

The GNU compiler translates the high-level C source code into assembly (there are some intermediate steps but not necessary to know now). The GNU assembler translates the assembly code into a binary object file of machine code. The object file is not executable yet - it needs to be linked with system libraries to produce a machine code executable. When you call gcc all three steps are performed for you.

5. What are the disadvantages to writing in assembly language versus writing in a high-level language? hint

Assembly source is not portable. Coding in assembly language will generally require more lines of code (LOC) than coding the same task in a high-level language. LOC generally relates to more errors and less productivity. Assembly is less readable and writable than high-level languages - again error prone and less productivity. Modern compilers can produce better assembly code than most humans.

6. Define the terms external label, local label, and forward reference in the context of assembly language. How do these concepts differ from a high-level language context? hint

An external label is a reference (address) to an object outside the assembly source file. A local label is a reference to something within the same file. A forward reference is a label that is used before it is defined. Assembly language allows forward references but high-level languages do not. Local and external labels are similar to local and global variables in high-level languages.

7. How does an assembler handle forward references? hint

An assembler handles forward references by making two passes of the source -the first pass breaks up the source into lexemes and the second pass produces the machine code.

8. List two important differences between assembly language and a high-level language such as C in terms of code writability. hint

High-level languages provide rich syntax, data types, and control structures. Assembly has 3 data types: characters, integers and floats and the primary control structure is the goto. (Assembly and high-level languages do share basic arithmetic, boolean and relational operators.)

9. List the primary sections of an object file under Unix. hint

header, text segment, data segment, relocation information, symbol table, debugging information.

10. What is an assembler directive? Give an example of one and explain what it does. hint

An assembler directive is an instruction for the assembler only and will not be translated into a machine instruction. An example is

```
.data
```

which simply tells the assembler that what comes next belongs in the data segment of the object file.

11. What is a pseudoinstruction? hint

A pseudo-instruction (aka macro) is an instruction provided by the assembler that is not part of the hardware instruction set. The pseudo-instruction is two or more instructions from the ISA (instruction set architecture). The purpose of pseudo-instructions is simply to make coding easier. examples are move and li.

12. List the duties of a linker. What is the name of the Unix linker? hint

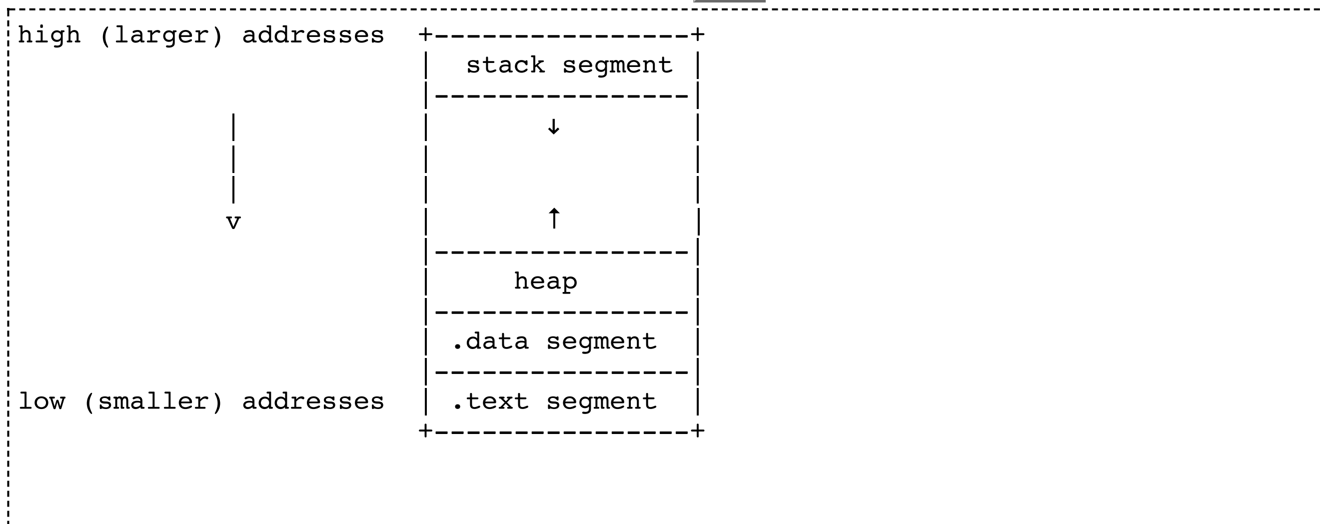
The Unix linker is named ld. The linker joins multiple object files together, resolves addresses across multiple object files, relocates absolute addresses, and searches library path to find libraries used by the program.

13. Describe the steps by which an executable is loaded into memory under Unix? (This is also typical of most OSs.) hint

The Linux kernel 1) reads the header to get executable size; 2) creates an address space for .text, .data, stack and heap segments; 3) maps instructions to .text segment; 4) copies cmdline args for main onto stack frame

for main; 5) initializes registers and stack pointer; 6) calls a startup routine to copy args to registers and call first instruction in main function; 7) startup routine passes control back to OS after program exits.

14. Draw a picture of how a program is loaded in memory (AppenA shows MIPS which is fairly typical). Include the stack, the heap, the text segment and the data segment. hint



15. Where does the MIPS global pointer register ($gp) point and what is $gp used for? hint

$gp points to the static data segment. A load and store instruction can use the signed 16-bit offset field to access the first 64 KB of the static data segment in a single instruction. $gp makes addressing the static data segment faster than the heap. Global variables are stored in this area for that reason.

16. Why is the maximum size of a program's stack and the size of each stack frame not necessarily known at compile time? hint

The stack is used for procedure calls. The order and frequency of procedure calls often depends on user input, which is not known until program execution. Additionally, the size of variables local to a procedure is often determined at runtime.

17. Little-endian (little-end in first) and big-endian (big-end in first) refer to the order in which chunks (one byte or two bytes) of a word are loaded into memory. Little refers to the "little end" (smallest value) of a number; i.e., the "little end" of 347 decimal is 7. The "big end" of 342 decimal is 3 (since the value of that digit is 300). In little-endian the least significant chunk is loaded into lowest memory address. Consider a 2-byte unsigned integer with binary and hex representations shown here:

```
            1100_0111 1111_0101  0xB7_F5
```

Assuming a machine with an 8-bit data bus (memory is loaded one byte at a time), the little end is the least significant btye (F5) and the big end is the most significant btye (B7). Show the order of bytes in both little-endian and big-endian. hint

Little-endian order loads the bytes into contiguous memory like this:

```
            1111_0101 1100_0111   0xF5_B7
```

with smaller memory address corresponding to the little end of the number. In Big-endian the bytes are loaded in the opposite order like this: 0xB7_F5. Why? Big-endian order means that smaller memory addresses correspond to the most significant byte of the number. The number would thus be loaded as B7 F5. x86 machines are little-endian and IBM 360 and IBM POWER PC are big-endian machines. MIPS machines are bi-endian and can switch between endianness. NOTE: a 16-bit data bus means the word is loaded in 16-bit chunks (two bytes at a time).

18. Write MIPS code for a subroutine **get_int** that reads an integer from the keyboard and echos it to the display and stores the result into $v0. hint

Please write this code yourself.

19. Write MIPS code for a subroutine **to_neg** that takes the value in $a0 and converts it to negative two's compliment. Assume that the value in $a0 is positive (don't do any input error checking). Store the result into $v0. hint

Please write this code yourself.

20. How do each of these commands modify the program counter?

```
a. jr
b. jal
c. j
```

hint

Use any sources you need to answer this, then write your answer in your own words.