Assignment #6

Procedures and basic driver core

Assignment #6 requires the coding of a simple linker for Mic1 assembly modules

This will include the following steps:

1. Your program must process an arbitrary number of object files on its command line that have been prepared by the **masm_mrd** tool as described in the URL:
   http://www.cs.uml.edu/~bill/cs305/Assignment_6_help_dir/example_linker_use.txt
2. Each command line object file must be opened and read, resulting in any machine instructions found to be written to an intermediate file that you must create in the /tmp directory, just as the masm assembler does.
3. The assembler generates these lines into its intermediate file, but you will just copy them from the object file to the intermediate file
4. Your main concern here is to adjust the line numbers from their relative values in the object file to an absolute (relocated) number in the intermediate file.

5. You must take each relative line number found in an object file, increment it to a new absolute location if needed, the first object file lines need no adjustment.
6. These steps require a file to be created in the /tmp directory to collect the code lines found among the command line object files. The file is created using a random name (so no collisions) and then unlinked
7. You will need 2 FILE * objects declared as globals, one to hold the descriptor for the intermediate file and one to use repeatedly to open, process and close each object file:

```
FILE  *p1, *p2;
```

See the next slide for code to create and unlink the /tmp file needed

I suggest that you use the following code to create the /tmp file with a unique name so you do not have collision problems with students who create the file and fail to unlink the filename:

```
struct timeval randtime;
char temp_file[20];

   gettimeofday (&randtime, (struct timezone *)0);
   srandom((unsigned int)randtime.tv_usec);
   sprintf(temp_file, "/tmp/bill%d", (unsigned int)random()%10000);

   p1 = fopen(temp_file, "w+");
   unlink(temp_file);
```

8. The second file descriptor is used to repeatedly open each object file:

```
for(i=start; i<argc; ++i){
    if((p2 = fopen(argv[i], "r")) == NULL){
       printf("ERROR: cannot open file %s\n", argv[i]);
       exit(6);
    }
```

9. This is the beginning of the for loop used to read each object files from the command line
10. You will be scanning a line from an object file that must have 2 or 3 fields. Every line has a relative line number and some form of instruction (either a complete instruction (cooked) or an instruction that have a u as its first character, signifying that this instruction is not complete (un-cooked).
11. An instruction fields with a first character of u means that there is a third field on the file line in the form of a label: name which must also be scanned

12. Whether you scan 2 fields or 3, the line must be written to the intermediate file with any needed line number adjustment

```
while(fscanf(p2,"%d %s", &pc, instruction) != EOF){
                    if(pc == 4096)break; // code done, now symbols
                    new_pc = pc + pc_offset;
                    symbol[0] = '\0';
                    if(instruction[0] == 'U'){
                      fscanf(p2, "%s", symbol);
                    }
                    fprintf(p1, "  %d  %s  %s\n", new_pc,
                    instruction, symbol);
```

13. When we scan a line whose line number is 4096 (out of range), we can leave the above while loop and enter a new while loop to process the symbols found in the object file
14. Each symbol in the object file includes the label: name and is relative location in the current object file

15. We must read each symbol in the current object file, adjust its relative line number to an absolute line number and add it to a new symbol table that we will construct to collect all symbols from all object files
16. We scan 2 fields for each symbol, updating our symbol table, and continuing until we hit the EOF of the object file (add_symbol() functionality is in masm code in update_sym_table())

```
while(fscanf(p2,"%s %d",symbol, &line_number) != EOF){
            add_symbol(symbol, line_number+pc_offset);
}
          pc_offset = new_pc + 1;
          fclose(p2);
```

17. Now, we need to open p2 again on the next object file from the command line and continue back through steps 1-17
18. When we have processed all object files, we should have a full file on p1 with all the adjusted instruction formats and a full symbol table with all the adjusted symbol formats

19. Now we need to rewind the file connected to p1 so we can start to perform final code generation to the output.
20. The code to do this is in the masm sources in the function generate_code()
21. This code reads each line back from the p1 file by first scanning 2 fields and then a third if the instruction field has the u character at the beginning.
22. If the instruction is cooked, then its 16 bit characters are written to the executable output destination (usually a .exe file)
23. If the instruction is un-cooked, then we must scan the third field from the line to get the dependent label: name and search the symbol table for the value of that label:
24. All of our un-cooked instructions are 4 bit op-code, 12 bit address instructions. The 12 bit address field in the instruction (which is zero for an uncooked instruction) now needs to be set to the label: value obtained from the symbol table in order to cook the 16 bit characters in the instruction and emit it to the output destination.

25. When a line is cooked and ready to be emitted into the output destination, the generate_code() function must make sure that there are no gaps in the line numbers between a preceding instruction and this instruction. If there are missing line numbers (because of a .LOC), then the generate_code() function must emit a 16 bit line of all 1s to fill the missing space:
26. The label: value needed to cook an instruction is found here by the get_sym_val() function which is coded in the masm sources.
27. In addition to performing a full link, your linker must look for a command line option of -o and if found, then it must emit a single object file (just as masm does) that combines all the object file instructions and the adjusted symbol table. This is pretty much what's already in place after pass1 and before code generation. You just have to dump the content of p1 , write a line of 4096 x and then dump the content of the new symbol table, to look like any other single object file.
28. The source code in the ~bill/cs305/MasmSrc directory should be very helpful for coding this linker project

```
line_number = old_pc = 0;  // part of generate_code()
rewind(p1);

sprintf(linbuf,"%5d:  ", line_number);

while(fscanf(p1,"%d %s", &pc, instruction) != EOF){
if((diff = pc - old_pc ) > 1){
  for(j=1; j<diff; j++){
        sprintf(linbuf,"%5d:  ", line_number++);
        printf("%s1111111111111111\n",(linum ? linbuf: "\0"));
  }
}
sprintf(linbuf,"%5d:  ", line_number++);
old_pc = pc;

 if(instruction[0] == 'U'){
   fscanf(p1, "%s", symbol);
   if((sym_val = get_sym_val(symbol)) == -1){
        fprintf(stderr, "no symbol in symbol table: %s\n", symbol);
        exit(27);
   }
```