Due: See website for due date.

What to submit: Upload a tar archive that contains a text file answers.txt with your answers for the questions not requiring code, as well as individual files for those that do, as listed below.

This exercise is intended to reinforce the content of the lectures related to linking using small examples.

As some answers are specific to our current environment, you must again do this exercise on our rlogin cluster.

Our verification system will reject your submission if any of the required files are not in your submission. If you want to submit for a partial credit, you still need to include all the above files.

1. Understanding Static Linking

To understand the process of static linking of multiple relocatable object files (.o) files into a single executable, we ask that you do a recreational programming exercise that simulates what a real linker does. Your program will be given the concatenated symbol tables of n .o files and should output the symbol table of the resulting relocated executable.

Consider the result of linking the following two .c files:

```
#include "code.h"
#include <stdio.h>
static double data[20];
static int temp = -2;

int
main()
{
    for (int i = 0; i < 20; i++)
        data[i] = i;
    temp = sum(data, 20);
    printf("sum = %d\n", temp);
    printf("inv = %d\n", invocationcount);
}
```

and

```
#include "code.h"
#include <math.h>
int invocationcount;
static int temp = -1;

int sum(double *data, int datalen)
{
    invocationcount++;
    int rc = 0;
    for (int i = 0; i < 20; i++) {
        temp = roundtoint(sin(data[i]) * 5 - cos(data[i]) * 10);
        rc += temp;
    }
    return rc;
}
```

If separately compiled (via `gcc -std=gnu11 -c code1.c code2.c`) two files `code1.o` and `code2.o` result. Using the command `nm -nS code1.o code2.o` one obtains the following output, which will be the input to your simulated linker:

```

code1.o:
```

```
                 U invocationcount
                 U printf
                 U sum
0000000000000000 00000000000000a0 b data
0000000000000000 0000000000000014 t roundtoint
0000000000000000 0000000000000004 d temp
0000000000000014 0000000000000078 T main

code2.o:
                 U cos
                 U sin
0000000000000000 0000000000000014 t roundtoint
0000000000000000 0000000000000004 d temp
0000000000000004 0000000000000004 C invocationcount
0000000000000014 00000000000000cc T sum
```

When linking this code (via `gcc code1.o code2.o -o code -lm` the resulting executable's symbol table will have the following entries[1]

Your program should simulate the linking process to compute this table:

```
                 U cos
                 U printf
                 U sin
0000000000400680 0000000000000014 t roundtoint
0000000000400694 0000000000000078 T main
000000000040070c 0000000000000014 t roundtoint
0000000000400720 00000000000000cc T sum
0000000000601044 0000000000000004 d temp
0000000000601048 0000000000000004 d temp
000000000060104c 00000000000000a0 b data
00000000006010ec 0000000000000004 B invocationcount
```

The detailed specifications are as follows:

- The input will consists of multiple lines. Your program must ignore empty lines in the input.
- A line with a alphanumerical filename followed by a colon (:) denotes the start of a new .o file.
- The remaining lines contain entries from the module's symbol table, in exactly the format output by `nm -nS`. Read the man page for `nm(1)` to understand this format.
- Only the following symbol types may occur in the input: `T t b C d D U`.

[1] The actual symbol table, which you can see with `nm -nS code`, has more entries which we ignore for the purposes of this exercise.

- Symbols should be laid out contiguously, introducing padding if necessary to ensure every symbol starts at an address that is a multiple of 4.
- Within each section, symbols that appear in the final executable should appear in the same order as they appear in the input.
- The first symbol in the text section should start at 0x400680.
- The first symbol in the data section should start at 0x601044. The bss section should follow right after the data section (with proper padding to achieve the required alignment.)
- If a symbol has multiple strong definitions, your program should output:

  `Symbol '%s' multiply defined`

  where `%s` should be replaced with the name of the symbol that is multiply defined. Your program must exit after the first error.
- The final symbol table should be output in following format: (1) first, all symbols that remain unresolved after the linking process, in alphabetical order. (In the real linker, these symbols would either lead to unresolved symbol errors, or they would be resolved dynamically). (2) all remaining symbols sorted by their numerical address.
- You may assume that local symbols are not defined multiple times within the same object file. (This would be flagged as an error by the compiler/assembler.)
- You do not need to handle libraries (e.g. modules that are included only if there are currently unresolved external references to symbols they define.) In other words, you must link all modules whose symbol tables are specified in the input.
- Your implementation should have an algorithmic complexity that is $O(n)$ where n is the number of symbols in the symbol tables of each file.
- Your program will be run multiple times on different inputs. It should read its input from stdin. Your program will be run under a suitable timeout.
- You may use any language that is available on the rlogin cluster so we can grade your submission. Contact techstaff@cs.vt.edu if you need a language not currently installed.

Below are two more sample I/O pairs:

Sample Input 2

```

code1.o:
0000000000000000 0000000000000004 C data
0000000000000000 0000000000000078 T main
```

```

code2.o:
0000000000000000 0000000000000004 D data

code3.o:
0000000000000000 0000000000000004 b data
```

Sample Output 2

```
0000000000400680 0000000000000078 T main
0000000000601044 0000000000000004 D data
0000000000601048 0000000000000004 b data
```

Include your code in a single file in your submission named `linker.py`, `linker.java`, `linker.cc`, or `linker.yourlanguage` depending on your implementation language.

2. Building Software

A common task is to use the compiler, linker, and surrounding build systems to build large pieces of software. In this part, you are asked to build a piece of software and observe the build process. Your answers will be specific to the tool chain installed on rlogin this semester.

1. Download the source code of Node.js 4.3.0. Read and follow the build instructions (note: Omit the 'make install' step unless you specified a directory to which you have write access as the installation directory (i.e., via the –prefix option to configure). The default installation destination directory is a system directory to which you do not have write access. Hint: lookup the meaning of the `-j` flag to the `make` command before running make.

   During the make process, identify the link command that produces the node executable and answer the next two questions.

2. Find the `-o` flag. In which directory does the build process leave the 'node' executable post linking?

3. List the `-l...` flags specified on the command line. You should find 3 libraries. List each library and look up, using online sources, what it is used for. Cite your sources.

4. How many symbols does the node executable define?

5. How many undefined symbols are left in the node executable (which will be resolved via dynamic linking)?

6. What is the size of the bss section of the node executable? Hint: use `size(1)`!
7. Linker map. When debugging linker problems, it is often useful to consult the linker map, which is a protocol left by the linker during the linking process. To obtain a linker map when building an executable, add `-Wl,-Map=linkermap` to the link command line. To that end, open the file node.gyp, identify where linker options are specified, and add this option. Hint: Look for other options that start with `-Wl,`! Then delete the node executable and run make to relink.

   Examine the resulting file `linkermap` and answer the following questions:

   (a) Which module (.o file) provides the symbol `ssl23_connect`?

   (b) Which module contained an external reference that first prompted the linker to include the object module `async.o`, contained in `libuv.a`, into the linking process?
8. Last, but not least, do not forget to remove the source and build directory from your rlogin space. It takes up about 276MB of space.

3. Type Confusion

Consider the following separately compiled files, file1.c and file2.c:

```
#include <stdio.h>

char a[8];

int
main()
{
    printf("%s\n", a);
}
```

```
double a = ..place a floating point number here..;
```

As discussed, the linker does not check if the types of a strongly defined symbol coincides with the type of a weak symbol of the same name. Change file2.c such that the program outputs `CS3214!`:

```
$ gcc -o file file1.c file2.c
$ ./file
CS3214!
$
```

4. Observing Dynamic Linking.

When a program starts, the dynamic linker ld-linux.so resolves the program's dynamic dependencies. On Linux, the function of the dynamic linker can be controlled using various environment variables, as described in ld.so(8). The shell script `ldd` displays an executable's shared library dependencies by performing a "dry-run" that resolves the executable's dynamic dependencies without actually running the program.

The environment variable LD_DEBUG provides information about what the dynamic linker is doing. Run the following program and answer the following questions

```
env LD_DEBUG=all stat .
```

1. Which is the first shared library that is needed to dynamically link the `stat` program?
2. Give the full path of the actual (.so) file being used!

5. Hiding Files.

A rootkit is a set of programs, libraries, and scripts designed to hide the fact that a system was compromised. Some rootkits use a system's dynamic loading facilities to hide their presence. For instance, an attacker may wish to hide the presence of certain files in a directory.

The LD_PRELOAD environment variable instructs the dynamic linker to load one or more libraries before loading the shared libraries referenced in an executable. Thus, if a symbol is defined in such a library, the program's reference will be resolved against the preloaded symbol, allowing it to intercept calls.

In this exercise, create a shared library "invisible.so" which, when added to LD_PRELOAD, will hide the existence of files whose name starts with the prefix invisible_.

Although a real rootkit would need to make sure that no call reveal the presence of this file, for the purposes of this exercise, you need to intercept only the `readdir` function that is used, for instance, by /bin/ls. Consult the manpage for readdir(3). The following file provides a skeleton you should fill in.

```
#include <string.h>
#include <stdio.h>

#define __USE_GNU 1
#define __USE_LARGEFILE64 1
#include <dirent.h>
#include <dlfcn.h>

/* Skip files with this prefix */
```

```
#define INVISIBLE "invisible_"

/* Define function pointer type that matches the signature of readdir */
typedef struct dirent * (*readdir_t)(DIR *dir);

/* Intercept readdir.  If the call would return a directory entry
 * whose name starts with INVISIBLE, return the next entry instead.
 * Otherwise, return the original directory entry.
 */
struct dirent *readdir(DIR *dir)
{
#include "readdirsolution.c"
}
```

Submit `readdirsolution.c`!

To obtain the value to which `readdir` would have resolved without interposition, use the `dlsym()` function, passing the special handle `RTLD_NEXT`.

To following sample session shows how `invisible.so` should work

```
$ mkdir readdir-test
$ touch readdir-test/a_file
$ ls readdir-test
a_file
$ touch readdir-test/invisible_file
$ ls readdir-test
a_file  invisible_file
$ env LD_PRELOAD=./invisible.so ls readdir-test
a_file
```

Submit `readdirsolution.c` and a script `makeinvisible.sh` that builds `invisible.so` from `readdir.c`.