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. Checking Best Practice in Linking

To better understand how the choices you make as a C programmer affect the robustness of larger programs written in C, and to also understand the information provided by symbol tables, this exercise includes the creation of a small link checker that looks for certain violations of best practice.

You should write a program in a high-level language of your choice, which will be invoked with multiple arguments denoting filenames. In addition, your program must accept the `-l` command line switch, which is also followed by a filename, but which may be given multiple times.

For example, a possible invocation might be

```
link-checker.py -l list.nm -l signal_support.nm \
    -l termstate_management.nm \
    -l shell-ast.nm -l shell-grammar.nm \
    utils.nm cush.nm
```

Your program should collect its command line arguments and sort the filenames by whether they are prefixed with `-l` or not. We refer to those that are as *libraries*, and those that are not as *core* modules.

Each filename is guaranteed to end in `.nm` and it reflects the output of running `nm -S` on an object module, e.g., `nm -S cush.o > cush.nm`. The `-S` switch includes the size of a symbol in the output (which is the size of the range this symbol spans in the object file - equivalent to the size of a variable or the size taken up by a function's instructions).

Your link checker should read the symbol tables in each library and core module and perform the following checks, in this order:

- Ignore all symbols that start with 2 underscores (__)
- Verify that each common symbol is defined in exactly one module. If you encounter a second definition of a common symbol, output:

  ```
  Common symbol {symbol} multiply defined, first in {module1} and
  now in {module2}
  ```

  If you encounter a third definition or more, repeat this warning.
- Verify that no common symbol is dominated by a strong global symbol. If you encounter this situation, output:

  ```
  Weak {symbol} overridden, defined in {module1} and overridden
  as {type} in {module2}
  ```

  where 'type' refers to the one-letter type nm provides.
- Check if there are any static functions that are defined in multiple modules with the same name and the same size. This could reflect functions defined in header files

that the compiler failed to inline. Output

```
Static function {symbol} of size {size} appears in both {module1}
and {module2}, check for inlining
```

If the functions appears to be defined in multiple files, repeat the error accordingly.

- Check that each global symbol that is defined in a *core* module is referenced outside the module in which it is defined. For each such symbol, output:

  ```
  'Global symbol {g} defined in {module} is not referenced by
  any other file, should be static'
  ```

  Do not perform this check for global symbols defined in *library* modules.

- For each library module, determine the longest prefix shared by all exported symbols. If this prefix is empty, output:

  ```
  Global symbols exported by library {module} do not
  share common prefix
  ```

  Otherwise, output the prefix you found:

  ```
  Global symbols exported by library {module} share
  common prefix {prefix}
  ```

For the files in the cush starter code, the checker should output:

```
Global symbols exported by library list share common prefix list_
Global symbols exported by library signal_support share common prefix signal_
Global symbols exported by library termstate_management share common prefix termstate_
Global symbols exported by library shell-ast share common prefix ast_
Global symbols exported by library shell-grammar do not share common prefix
```

We recommend that you run the checker on your final cush project as well.

Include your code in a single file in your submission named `linkchecker.py`, `linkchecker.java`, `linkchecker.cc`, or `linkchecker.yourlanguage` depending on your implementation language. If using Python, include a shebang line to indicate the version.

- 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.

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 a typical build process. Your answers will be specific to the version of the GCC tool chain installed on rlogin this semester.

1. Download the source code of Node.js 14.15.5 Read and follow the build instructions in BUILDING.md (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. Thanks to your engineering fees, `make -j64` is ok to use an rlogin machine.

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

2. Find the `-o` flag, which specifies the directory in which the build process will leave the 'node' executable post linking. Copy and paste the full path name appearing after `-o` here:

3. Name the *first* 2 libraries that are linked statically with the node executable, based on their order in the link command line!

4. Name 2 libraries that are linked dynamically (hint: use ldd)!

5. List the name of the $4\,000^{\text{th}}$ strong global text symbol found in the 'node' executable, when considering them in alphabetical order. Hint: use a combination of `nm`, `grep`, `head`, and `tail`.

6. How many global BSS symbols occur in the node executable?

7. What percentage of the node executable consists of machine code? Hint: use `size(1)` without flags to find the size of the proper segment, use `ls -l` to find the size of the executable!

8. Last, but not least, do not forget to remove the source and build directory from your rlogin file space. It takes up about 1.2GB of space.

3. Link Time Optimization

Traditional separate compilation and linking has an important drawback: since the intermediate representation created by the compiler is no longer available at link time, potential interprocedural optimizations cannot be performed. For instance, the linker cannot inline functions or replace calls to functions that produce constant results with their values.

Link Time Optimization (LTO) overcomes this drawback by preserving the compiler's intermediate representation and passing it along to the linker which can then perform whole-program optimization across modules.

In this part of the exercise, you will be looking at how LTO works in a current compiler (gcc 8.3.1).

Create or copy the following files `lto1.c` and `lto2.c`:

```
// declare externally defined function
extern long factorial(long n);

int
main() {
    return factorial(8);
}
```

```
long factorial(long n)
{
    if (n <= 1)
        return 1;
    return n * factorial(n-1);
}
```

Compile and build the two files using the following commands:

```
gcc -O3 -flto -c lto1.c lto2.c
gcc -O3 -flto lto1.o lto2.o -o lto
```

Then answer the following questions:

1. Try running `objdump -d lto2.o` to look at the object file created by the compiler. Can you find machine code in the object file?

2. Now run

   ```
   gcc -c -O3 -flto -fdump-tree-gimple lto2.c
   ```

   You should find a file `lto2.c.004t.gimple` that shows the intermediate representation the compiler created (and which is provided in a binary serialized format).

   This file contains an assembly-like representation of the `factorial` function. Copy all return statements contained in this intermediate representation here.

3. Use `objdump -d` to find the code for the `main()` in the final executable. Copy and paste the body of main (the disassembled machine code)!

   Explain in your own words what optimization the compiler performed.