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 `nano` - a mini case study in linking

1.1 Observing The Build Process

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.

This semester, we will use the text editor Nano 6.4, which you can download from https://www.nano-editor.org/dist/v6/nano-6.4.tar.gz. Extract the tarball and follow the instructions in its README, omitting the `make install` step (which in its default configuration would require write access to the `/usr/local` directory).

After compiling it, the subdirectory `src` contains a number of object files that are linked together into the nano executable.

For part 1.1, answer the following questions:

1. The program is built using a Makefile that issues compilation and linking commands using the gcc compiler driver. Identify the line that links the program and copy it to your answer.
2. Which static library is `nano` linked with?
3. Which dynamic libraries is `nano` linked with?
4. Which 3 symbols occupy the largest amount of space in the binary? (Hint: investigate the `-size-sort` option to `nm`.)
5. Which global (non-static) constant occupies the largest amount of space in the binary, and on which line in which source file does its definition start?
6. Use the `size` command (with no flags) to find out how big the text, data, and bss section in the resulting executable are and provide those values here.
7. Use the `strip` command to remove debugging symbols from the `nano` binary. By what percentage was the size of the executable reduced?

1.2 Best Practices

In the lectures, we had identified a number of best practices when it comes to separate compilation and linking. In this part of this exercise, we ask that you examine nano to see whether its developers followed those practices or not.

For part 1.2, answer the following questions:

1. First, it is good practice to keep symbols that aren't used outside one compilation unit local (i.e., using the `static` keyword in C. Generate a list of all global symbols

that are not local but that are used in only one file. Include only symbols defined in files that are located inside the `src` directory in your analysis.

Submit a file `nano-globals.txt` with a list of these symbols, with each symbol on its separate line.

Explain how you obtained your answer. If you used a program or script to obtain the answer (which we recommend you do) include the program or script. The script will not be autograded.

2. Sometimes, developers may have reasons for keeping such symbols global - for instance, if the source code is compiled with a different configuration, such as perhaps for a different OS, these symbols may be used. Pick a sample of 2 symbols from your list and determine whether the developers would have had a reason to keep this symbol global (despite being used in only a single compilation unit).

3. A second bad practice is to include multiple definitions of weak common symbols, or letting a strong symbol definition override a weak symbol definition, thus relying on the linker's legacy resolution rules. Does nano's build process include object files that provide multiple definitions for any symbol that has is defined as a common symbol?

   How did you arrive at your answer?

2 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..;
```

The linker does not check if the types of a strongly global symbol coincides with the type of a weak global symbol of the same name.

In this problem, you are asked to reproduce a type-related bug in which one file (`file1.c`) assumes that the type of a global variable is a character array of length 8, whereas another file (`file2.c`) actually defines this symbol strongly as an initialized double.

Change the double constant in file2.c such that the program outputs |cs_3214_|

```
$ gcc -o file file1.c file2.c
$ ./file
cs_3214_
$
```

Note: you may not change the type of a in file2.c, i.e., it must stay a double.

Submit file2.c.

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. Languages such as Rust use LTO to be able to perform optimizations across the different source files that are part of a crate.

In this part of the exercise, you will be looking at how LTO works in a current compiler (gcc 8.5.0).

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

```
// declare externally defined function
extern long math(long a, int n);

int
main() {
    return math(3, 4);
}
```

```
#include <stdlib.h>

extern long math(long a, int n)
{
    long p = a;
    while (n--)
        p *= a;
    return (p - 1) / (a - 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 `math` function.

   List all labels defined in this function.

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

4. Now compile these programs without LTO like so:

   ```
   gcc -O3 lto1.c lto2.c -o ltonormal
   ```

   Use `objdump -d ltonormal` to look at the main function, and reproduce the assembly code here.

   Explain in your own words what optimization(s) are performed when LTO is enabled and why!