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. Linking Minibash

In this part of the exercise, you are asked to make small changes to the mini-bash starter code to induce linker errors and changes to the executable. To that end, you should clone a fresh copy of the starter code with

```
git clone git@git.cs.vt.edu:cs3214-staff/minibash.git
cd minibash
(cd tommyds; make)
(cd tree-sitter; make)
cd src
make
```

The 3 parts are independent and you should undo any changes you made for one part before continuing to the next.

1. After changing a total of 3 lines [1] in 2 files, minibash rebuilds but the linking step of the build fails with:[2]

   ```
   ld: signal_support.o:/..../minibash/src/./utils.h:18: multiple definition
   of 'variableId'; minibash.o:/.../src/./utils.h:18: first defined here
   ld: utils.o:/..../src/./utils.h:18: multiple definition of 'variableId';
   minibash.o:/..../src/./utils.h:18: first defined here
   clang: error: linker command failed with exit code 1 (use -v to see invocation)
   make: *** [Makefile:29: minibash] Error 1
   ```

   What lines were changed? Provide the output as a patch (which you can produce via `git diff`.)

2. (Undo any changes, perhaps with `git checkout .`, before the next step.) After changing one line and adding one line to the original code, the project builds without errors, but the following `nm` command shows this:

   ```
   $ nm minibash | grep ts_extract_single_node_char
   0000000000431818 T ts_extract_single_node_char
   ```

   What lines were changed and how? Provide a possible set of changes as a patch.

3. (Reset the project again first.) After changing a single line in the `Makefile`, and then running `make -B`, the project build fails with

   ```
   ....../bin/ld: minibash.o: in function 'main':
   /..../src/minibash.c:309:(.text+0x2c):
       undefined reference to 'tommy_hashdyn_init'
   ....../bin/ld: /.../src/minibash.c:383:(.text+0x606):
       undefined reference to 'tommy_hashdyn_foreach'
   ....../bin/ld: /.../src/minibash.c:384:(.text+0x615):
       undefined reference to 'tommy_hashdyn_done'
   clang: error: linker command failed with exit code 1 (use -v to see invocation)
   ```

[1] This is not counting any empty/whitespace lines. Also, in your reproduction, the line numbers do not need to match.

[2] I introduced newlines for readability.

```
make: *** [Makefile:29: minibash] Error 1
```

What line was changed and how? Provide the output as a patch.

Don't forget to revert to the original starter code for each part!

2. Baking Pie

From past courses (CS 2505, CS 2506) you are familiar with threats that can affect vulnerable applications that contain buffer overflows. Some of the exploits that targeted such applications made assumptions about the way in which they were built and run.

One particular assumption relates to the virtual addresses at which functions (text) or data can be found, which traditionally has been static. Recently, some OS have instead adopted position-independent executables as a default where the locations of functions and global variables is randomized from run to run.

Write a program that can determine if it was built as a position-independent executable or not.

For instance, if built as

```
gcc -no-pie ispie.c -o no.pie
```

it should output

```
$ ./no.pie
built with -no-pie
```

But if built like so:

```
$ gcc -pie -fPIE ispie.c -o pie
```

it should output

```
$ ./pie
built with -pie
```

Call your program `ispie.c`.

Your program should not write to or read any external files. Your program may spawn child processes.

Hint: Non-pie executable's global variables will be always at the same virtual address from run to run whereas these addresses will vary from run to run when an executable is linked as PIE.

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 (clang 20.1.8).

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

```
double evaluate(double a, double b, double c, double x);
```

```
#include <stdio.h>
#include "lto.h"

int
main()
{
    double s = evaluate(1, 2, 1, -4);
    return (int)(s);
}
```

```
#include <stdlib.h>

// some math function
#include "lto.h"

double evaluate(double a, double b, double c, double x)
{
    return a * x * x + b * x + c;
}
```

Compile and build the two files using the following commands:

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

Then answer the following questions:

1. 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)!

2. Now compile these programs without LTO like so:

   ```
   clang -O3 lto1.c lto2.c -o nolto
   ```

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

   Explain in your own words what the compiler and linker did when LTO was enabled and how this was possible using LTO but not when LTO was not being used.

3. What is the output of

   ```
   ./lto; echo $?
   ```

   and why?

4. Building Redis

Redis is a popular in-memory data store that is widely used in industry as a cache or database. It was created by Salvatore Sanfilippo, better known as antirez.

In this part of the exercise you will look at how redis is compiled and built in order to observe how compilers and linkers are used in a larger software project.

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

1. Download and extract the source code of Redis 8.2.1.
2. Change into the directory into which you've extracted the source code and build it using the command

   ```
   make -j V=1 |& tee LOG
   ```

   The make program will run a number of commands to configure, compile, and link multiple executables that are part of Redis. The tee program will receive the standard output and standard error streams and write them to the file `LOG` (in addition to writing them to the console), which will come in handy for the rest of this section.

   It will say that it's a good idea to run make test, but you do not need to do that here.

3. Find the command that makes the `redis-server` executable in the LOG. Look for an invocation of `cc` that includes the flag `-o redis-server`.

   Reproduce the command in your answer.

4. List the 6 libraries that are statically linked with the `redis-server` executable.
5. How much space does the `redis-server` executable take up on disk? (Use `ls -l` to find out.)
6. The command `size` (without any arguments) gives you an estimate of how much memory is needed if the executable were fully loaded into memory, broken down

by text/code, data, and bss. Run `size` on `redis-server`. What fraction of the executable is taken up by code and data? Answer in percent, rounded to the nearest percent.

7. The command `strip` strips an executable of those parts that are not necessary to run it. Run `strip` on the `redis-server` executable and measure its new size with `ls -l`.

   What percentage of the stripped executable is taken up by program code and data? Give your answer as a percentage of the size of the stripped executable, rounding to the nearest tenth of a percent.

8. Last, but not least, do not forget to remove the source directory from your rlogin file space. It takes up about 228MB of space.