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 Cush

In this part of the exercise, you are asked to make small changes to the cush 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/cs3214-cush.git
cd cs3214-cush
(cd posix_spawn; 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 2 lines [1] in 2 files, cush rebuilds but the build fails with:[2]

   ```
   cc -Wall -Werror -Wmissing-prototypes -I../posix_spawn -g -O2
       -fsanitize=undefined -o cush -L../posix_spawn cush.o shell-grammar.o
       list.o shell-ast.o termstate_management.o utils.o signal_support.o
       -lspawn -ll -lreadline
   /usr/bin/ld:
       termstate_management.o:/.../cs3214-cush/src/termstate_management.h:6:
       multiple definition of ‘shell_pgrp’;
       cush.o:/.../cs3214-cush/src/termstate_management.h:6: first defined here
   collect2: error: ld returned 1 exit status
   make: *** [Makefile:27: cush] Error 1
   ```

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

2. After a single line change from the original code, the project builds without errors, but the following `nm` command shows this:

   ```
   $ nm cush | grep job_list
   000000000041efc0 B job_list
   ```

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

3. After changing a single keyword on a single line, the project build fails with

   ```
   cc -Wall -Werror -Wmissing-prototypes -I../posix_spawn -g -O2
       -fsanitize=undefined -o cush -L../posix_spawn cush.o
       shell-grammar.o list.o shell-ast.o termstate_management.o utils.o
       signal_support.o -lspawn -ll -lreadline
   /usr/bin/ld: signal_support.o: in function ‘signal_is_blocked’:
   /.../cs3214-cush/src/signal_support.c:23:
       undefined reference to ‘mask’
   /usr/bin/ld: /.../cs3214-cush/src/signal_support.c:26:
       undefined reference to ‘mask’
   /usr/bin/ld: /.../cs3214-cush/src/signal_support.c:26:
       undefined reference to ‘mask’
   ```

[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.

```
collect2: error: ld returned 1 exit status
make: *** [Makefile:27: cush] Error 1
```

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

Remember 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 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.

For instance, a hypothetical program `pie.c` could be built either as a regular executable like so:

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

Or as a position-independent example like so:

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

When running the `./no.pie` version, the output will always be:

```
$ ./no.pie
0x404000
```

But when running the `./pie` version, the output will vary randomly, perhaps like so:

```
$ ./pie
0x55622bfb7000
$ ./pie
0x55d426b68000
$ ./pie
0x55a3bab1e000
```

Write `pie.c`! Your solution should calculate its output based on the address of a global (static or non-static) variable or function. It should be specific to our current version of gcc on rlogin.

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 11.5.0).

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(3, 1, -2, 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:

```
gcc -O3 -flto -c lto1.c lto2.c
gcc -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:

   ```
   gcc -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 7.4.2.
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`. Now `cd` into the `src` directory and rerun this command with the `time` builtin of bash like so

   ```
   $ time cc -O3 -flto=auto .... make sure to copy and paste the
   entire line without introducing any line breaks
   ```

   Then, repeat this command, but change `-flto=auto` to just `-flto`.

   Write down how much real time each invocation took, and how much combined user and system time it took as reported by the `time` builtin.

   Given what you learned about LTO in the previous question, explain the reason for the difference between the two runs; in particular, explain the large amount of CPU time spent.

4. List the 5 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`. Roughly what fraction of the executable is taken up by code?
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 size with `ls -l`. Then run `size` on the stripped executable. Roughly what fraction of the stripped executable is taken up by the programmer-provided initial values of global variables that must be stored as part of the executable?
8. One of the best practices we had discussed was to ensure that global functions and variables that are used only within a single C file are declared with the `static` keyword. This way they will not leak into the global namespace and encapsulation is preserved.

   List 3 *variables* (not functions) in the Redis code that could have been made static but weren't (yet).

   Hint:

   You can obtain .o files with symbol tables for analysis by recompiling without LTO. To that end, do

   ```
   make clean
   rm src/.make-settings
   make -j OPTIMIZATION=-O0
   ```

   then you should be able to examine the symbol tables of the object modules in the source directory with nm.

   List the names of these 3 variables as they appear in the symbol table. (Think about how you can test that your answers is, in fact, a global variable that could have been made static!)
9. Last, but not least, do not forget to remove the source directory from your rlogin file space. It takes up about 200MB of space.