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. 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 18.18.0.

   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 on any rlogin machine.

   During the make process, identify the link command that produces the `node` executable and then 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`.

3. Find the location of the static library `libuvwasi.a` which is given in the link command line. How many .o files does this static library contain?

4. Which libraries are linked dynamically with the `node` executable (hint: use ldd and ignore `linux-vdso.so.1` and `/lib64/ld-linux-x86-64.so.2`)?

5. List the name of the 4 000th 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 much space does the `node` executable take up on disk? (Use `ls -l` to find out.)

7. The command `size` (without any arguments) gives you an estimate of how much memory is needed if the executable were fully loaded into memory. Run `size`. Roughly what fraction of the executable that is stored on disk would be loaded into memory?

8. The command `strip` strips an executable of those parts that are not necessary to run it. Run `strip` on the executable.

   After stripping the executable, did the stored size of the executable on disk get larger, smaller, or stay the same?

9. After stripping the executable in the previous step, did the amount of memory to run the executable get larger, smaller, or stay the same?

10. Almost half of the text size comes from a single .o module, `out/Release/-obj.target/node/gen/node_snapshot.o`. Use nm on this file and pipe the output through the `c++filt` program. Compare the output of this command to the node.js API documentation at https://nodejs.org/dist/latest-v18.x/docs/api/.

    Research what code and data is contained in this "snapshot" file and how it benefits Node.js.

11. Last, but not least, do not forget to remove the source and build directory from your rlogin file space. It takes up about 1.4GB of space.

2. 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 3 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/utils.h:11:
    multiple definition of ‘job_list’;
    cush.o:/..../cs3214-cush/src/utils.h:11: first defined here

/usr/bin/ld: utils.o:/..../cs3214-cush/src/utils.h:11:
    multiple definition of ‘job_list’;
    cush.o:/..../cs3214-cush/src/utils.h:11: first defined here

/usr/bin/ld: signal_support.o:/..../cs3214-cush/src/utils.h:11:
    multiple definition of ‘job_list’;
    cush.o:/..../cs3214-cush/src/utils.h:11: 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`.)

[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.

2. After a single line change, the project builds but the following `nm` command shows this:

   ```
   $ nm cush | grep job_list
   000000000041ffc0 B job_list
   ```

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

3. After a single line change, 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: cush.o: in function ‘main’:
   /..../cs3214-cush/src/cush.c:272: undefined reference to ‘job_list’
   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!

3. 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.

Consider the following program `pie.c`:

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

int data = 42;
int bss;

int
main()
{
    int stack;
    printf("my heap  is at %p\n", malloc(4));
    printf("my stack is at %p\n", &stack);
    printf("my text  is at %p\n", main);
    printf("my data  is at %p\n", &data);
```

```
    printf("my bss   is at %p\n", &bss);
}
```

Compile and build two executables using the following commands:

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

Run `./no.pie` two or more times, and run `./pie` two or more times.

1. What does "PIE" stand for, and how did it change the environment when a program built using this facility was loaded and run?
2. Which specific kind of attack technique is made more difficult when `-fpie` is used?

4. 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 11.4.1).

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

```
// declare externally defined function
// (this is not best practice - this declaration should be
//  in a header file)
extern long math(long a);
#include <stdio.h>

int
main() {
    long h = math(3025);
    printf("%d\n", h);
}
```

```
#include <stdlib.h>

// some not necessarily correct math function
extern long math(long a)
```

```
{
    long l = 0;
    long h = a;
    for (int i = 0; i < 18; i++) {
        long m = l + (h - l) / 2;
        if (a > m * m)
            l = m + 1;
        else
            h = m;
    }
    return l;
}
```

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.