Part A:

1. Compile the program prog.c using gcc -O2 prog.c -o prog-dyn and run prog-dyn. What does it do?
   1. Prints out Hello World!
2. Statically compile and optimize prog.c by running gcc -O2 -static prog.c -o prog-static. How does the size compare with prog?
   1. hello is 16k vs 880k for hello-static
3. Run ldd on the static and dynamic versions of the program. How does the output compare? Why?
   1. Dynamic version printed out the shared object dependencies. Static version didn’t because it wasn’t a dynamic file since its statically built and doesn’t use shared libraries.
4. See what system calls prog-static produces by running strace -o syscalls-static.log ./prog-static. Do the same for prog-dyn. Which version generates more system calls? **Note: system calls are saved in the log file syscalls-static.log. Feel free to save them in a different file.**
   1. Dynamic version creates more system calls most likely because it uses the systems shared libraries so it would need to interact with the kernel more.
5. See what library calls prog-static produces by running ltrace -x '\*' -o library-static.log ./prog-static. Do the same for prog-dyn. Which version generates more library calls? (Note: you will have to run sudo apt install ltrace to enable the command.)
   1. Dynamic has more library calls and static has no library calls. This is because ltrace traces calls to external or dynamic libraries. Since static are using internal calls ltrace wont be able to trace them.
6. Compile the program dynamically but with "lazy" linking: gcc -O2 -z lazy prog.c -o prog-dynlazy. Run ltrace -o library-lazy.log ./prog-dynlazy. How does the output of this compare to that of the previous ltrace?
   1. basically no external calls made compared to previous
7. Use the command ls -l to see the metadata associated with prog.c and prog-dyn, and prog-static. Who owns these files? What group are they in? Do you notice any pattern with the permissions (rwx) associated with each file?
   1. Student owns all files and all are in student group.
   2. Dynamic and static files give everyone permission to execute. Normal c file doesn’t

Part B:

Do the following with hello.c and syscall-hello.c, as before.

Assembly Code Syntax notes:

* The last letter of many instructions refers to the size of the operand.
  + callq means call a function using a "quad" value (64 bits).
* A dollar sign preceding a value means that it is a literal value
* A percent sign means it is a register.
  + If a register is in parentheses, then it is being used as a "pointer"
    - It contains an address, so the CPU goes to that address and interacts with the memory there.
  + If there is a number before the parentheses, it is an offset to the register's value.

1. Using the nm command, see what symbols are defined in prog-static and prog-dyn. Which defines more symbols?
   * Static file defines more symbols than dynamic file does
   * why? This is because static needs to define their symbols/ vars to use them while dynamic files have their symobls/vars defined elsewhere in other libraries
2. Run the command gcc -c -O2 prog.c to produce an object file. What file was produced? What symbols does it define?
   * -C compiles/assemble .c file into an object file hello.o
   * Using nm it defines r, T, and U symbols , r is for read only data section, T is for text code section, U is undefined
     1. man nm hello.o reads object file and shows the symbols defined.
3. Look at the assembly code of the program by running gcc -S -O2 prog.c. What file was produced? Identify the following in the assembly code (if present):
   * A function call (call)
     1. Present – handles the passing the return address for you, function ca;;
   * A return from a function (ret)
     1. present – handles using that address to return back to where you called the function from
   * Registers being saved onto the stack (push)
   * Registers being retrieved from the stack (pop)
   * Subtraction (sub)
     1. Present subq
   * A system call (syscall)
     1. A source file was made with -S flag/option with assembly code
     2. Use nl command to read the source file
4. Disassemble the object file using objdump -d. How does this disassembly compare with the output from gcc -S?
   * Much less assembly code compared to source file
5. Examine the headers of object file, dynamically linked executable, and the statically linked executable using objdump -h
   * Examined, 8 ,26 ,27 lines. Whatre we supposed to notice?
6. Examine the contents of object file, dynamically linked executable, and the statically linked executable using objdump -s
   * Examined , object file has short contents, dynamic has more, but static has so much it took a couple seconds to load everything
7. Re-run all of the previous gcc commands adding the "-v" flag. What is all of that output?
   * Verbose output of the file displays the full directopry paths used to search for header files and libraies, the predefined preprocessor suymbols, and the object files and libraries used for linking

Part C:

Compile and run [3000memview.c](https://homeostasis.scs.carleton.ca/~soma/os-2019f/code/3000memview.c), then consider the following questions.

1. Why are the addresses inconsistent between runs?
   1. The controller creates variables based on available memory locations in RAM so its constantly being accessed
2. Roughly where does the stack seem to be? The heap? Code? Global variables?
   1. Labelled in code <https://stackoverflow.com/questions/14588767/where-in-memory-are-my-variables-stored-in-c>
3. Observe how the heap grows (i.e. the value of sbrk changes) in response to malloc calls. Would you expect the heap to ever run into the stack? Why or why not?
   1. Heap shouldn’t run into stack, malloc() would notice the situation and return null and fail the memory allocation……most of the time. Usually just returns out of memory exception or stack exception. Or invalid/bad memory exception, or program just crashes.
4. Change each malloc call to allocate more than 128K. What happens to the values of sbrk? Why? (Hint: use strace)
   1. Supposed
5. Add more code and data to the program, and add more printf's to see where things are. Are things where you expect them to be?