ECE4730J
Advanced Embedded System

LAB 2: PROGRAM PROFILING ON LINUX

October 11, 2021

Name	Student ID
Yihua Liu	518021910998
Shuocheng Chen	517021911139
Yiming Ju	518370910059

Lab 1 Task 3. (Optional)

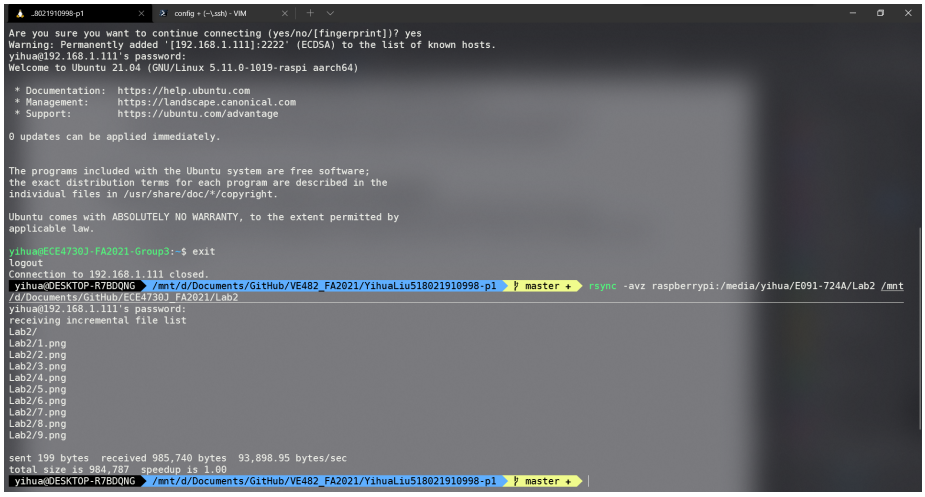


Figure 1. Setup the remote access to your Raspberry Pi.

1. No submission for the first exercise.
2.

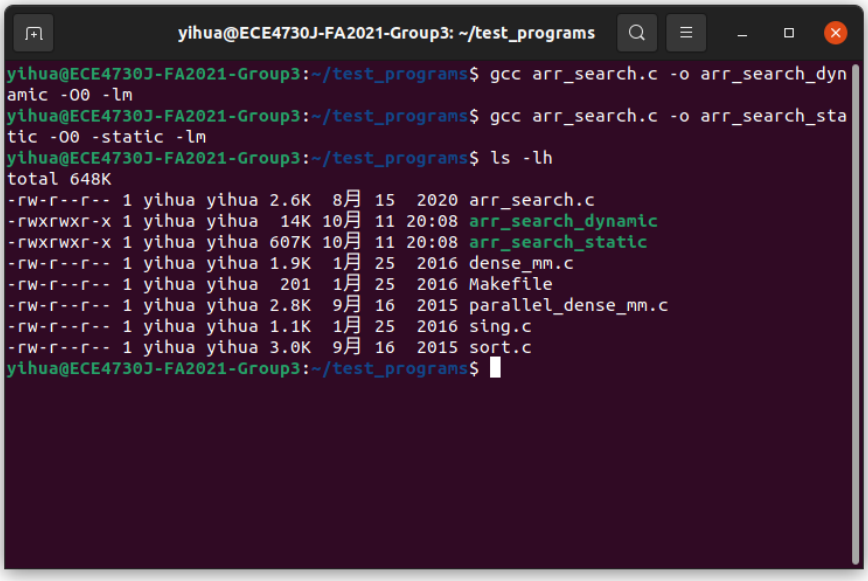


Figure 2. The list of the size of each program.

program	size
arr_search_static	607K
arr_search_dynamic	14K

Table 1. The list of the size of each program.

The statically compiled program is much larger than the dynamically compiled program because the statically compiled program includes all the library code in the executable file, while the dynamically compiled program places calls to the code of shared libraries during runtime.

3.

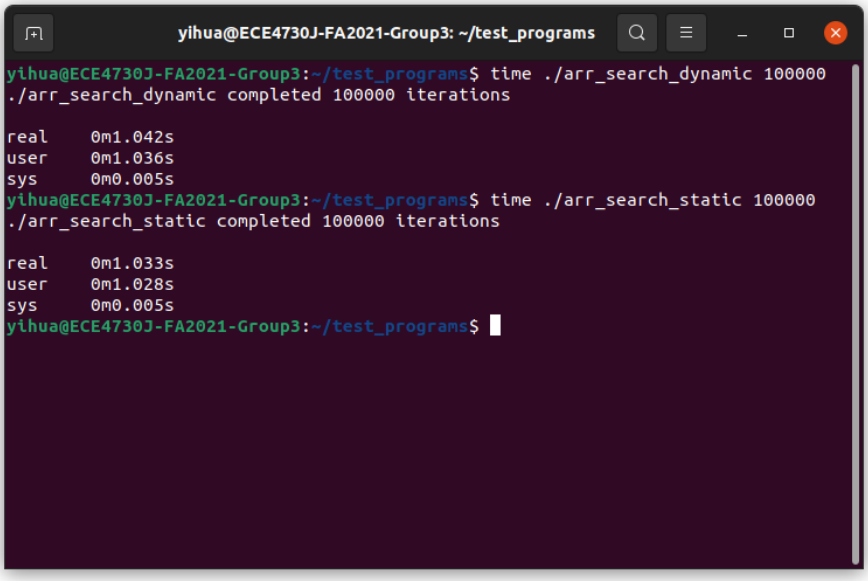


Figure 3. `time` command.

The number of iterations used: 100000.

Note that there should be enough time interval between two measures of time, otherwise the measurement will be inaccurate.

The "real" execution time for the arr_search_dynamic binary: 1.042s.

The "real" execution time for the arr_search_static binary: 1.033s

The statically compiled program is slightly faster than the dynamically compiled program because the statically compiled program can directly execute the library code without calling to shared library. However, the difference is very small. If you repeat many times, you will find that sometimes they almost have the same execution time.

4.

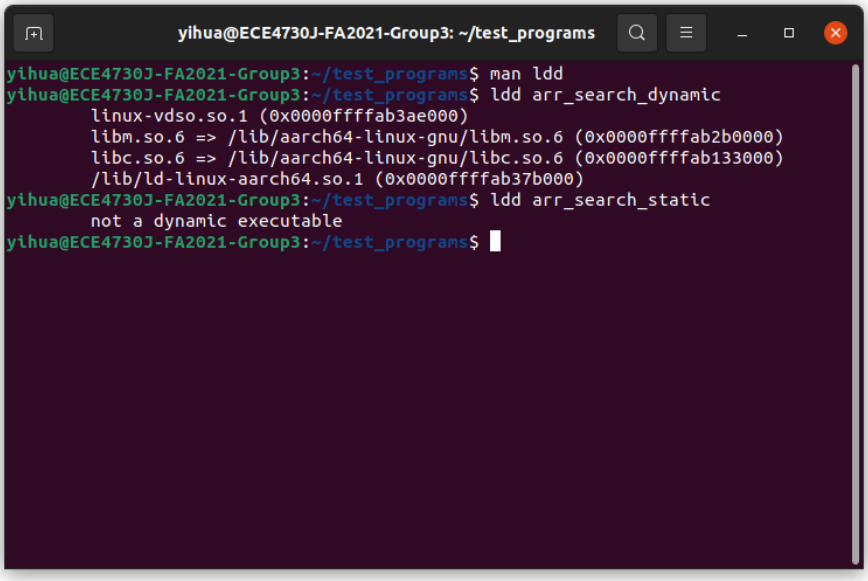


Figure 4. `ldd` command.

They are different because `ldd` is used to print shared object dependencies. The statically compiled program does not require shared objects (shared libraries), so it prompts "not a dynamic executable". On the contrary, the dynamically compiled program requires shared objects (shared libraries), so `ldd` prints the shared objects (shared libraries) it used:
linux-vdso.so.1
libm.so.6 =>/lib/aarch64-linux-gnu/libm.so.6
libc.so.6 =>/lib/aarch64-linux-gnu/libc.so.6
/lib/ld-linux-aarch64.so.1

5.

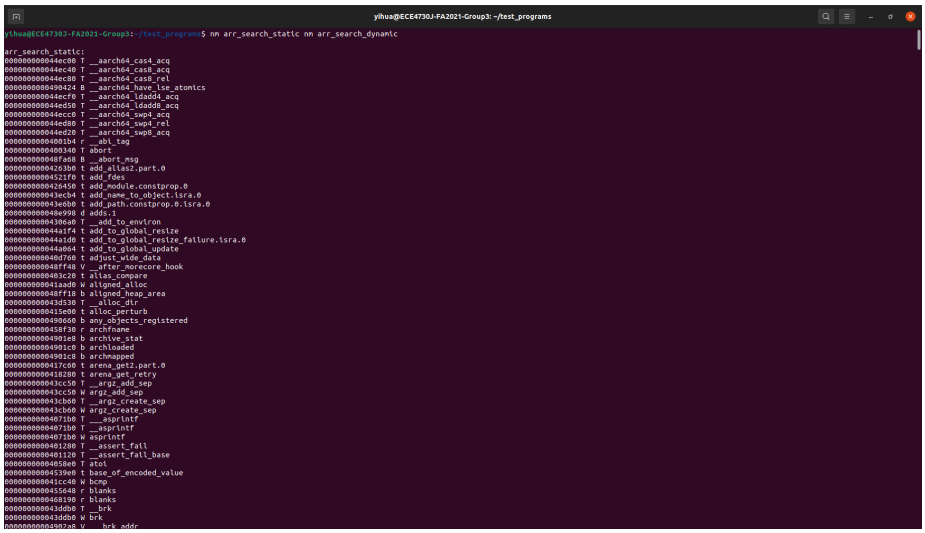


Figure 5. `nm` command (static).

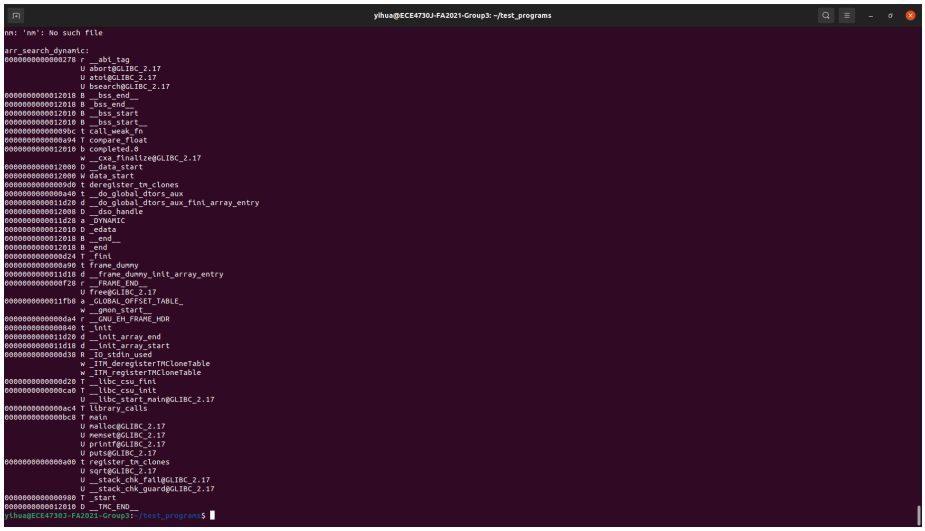
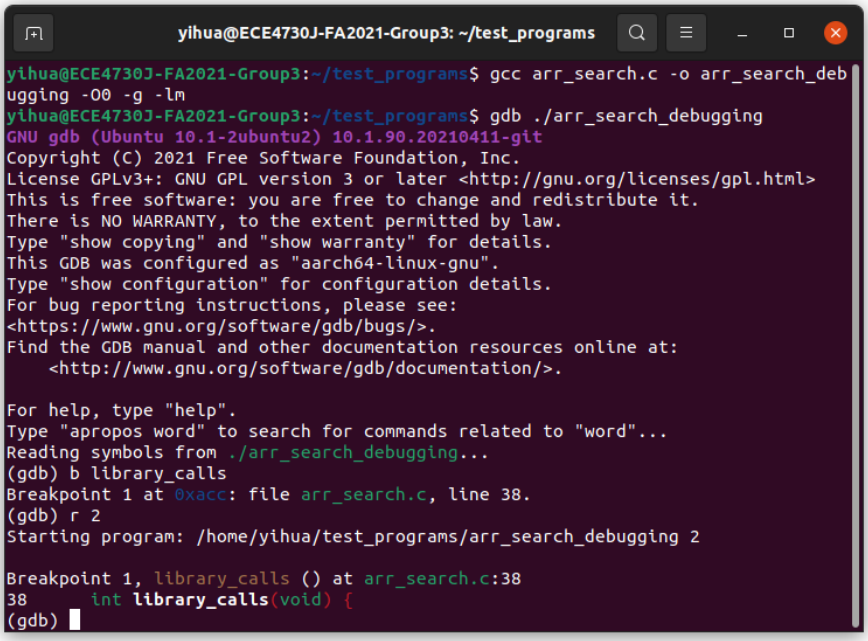


Figure 6. `nm` command (dynamic).

`nm` command is used to list symbols from object files. The statically compiled program has a lot of symbols, and all the symbols of library code are listed, such as `malloc` of `stdlib.h` and `strchr` of `string.h`, while the dynamically compiled program has relatively less symbols.

6.



```
yihua@ECE4730J-FA2021-Group3: ~/test_programs
yihua@ECE4730J-FA2021-Group3:~/test_programs$ gcc arr_search.c -o arr_search_deb
ugging -O0 -g -lm
yihua@ECE4730J-FA2021-Group3:~/test_programs$ gdb ./arr_search_debugging
GNU gdb (Ubuntu 10.1-2ubuntu2) 10.1.90.20210411-git
Copyright (C) 2021 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "aarch64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./arr_search_debugging...
(gdb) b library_calls
Breakpoint 1 at 0xacc: file arr_search.c, line 38.
(gdb) r 2
Starting program: /home/yihua/test_programs/arr_search_debugging 2

Breakpoint 1, library_calls () at arr_search.c:38
38      int library_calls(void) {
(gdb)
```

Figure 7. Compiling and loading programs by `gdb`.

Using `-g` option to compile our program by `gcc` we can use `gdb` to debug.

7. `b` command adds a breakpoint, where `library_calls` is the function. `r` commands runs the program, and 2 is the number of iterations.

8.

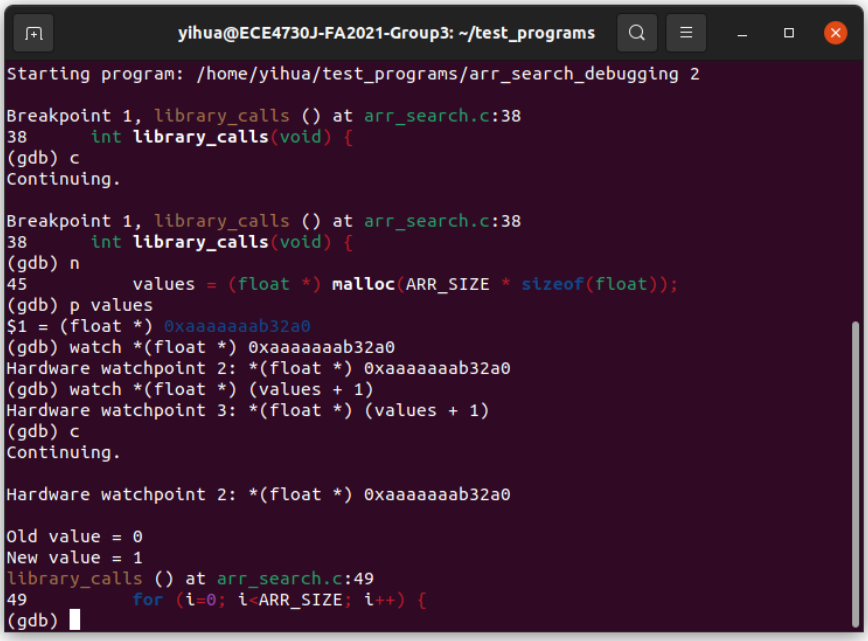


Figure 8. Executing the program in `gdb`.

`c` command continues to the next breakpoint, and we can see that the program stops again at the function `library_calls`, which indicates that we are executing the second time.

9. We can see that the address of the variable `values` is 0xaaaaaaab32a0.

10.

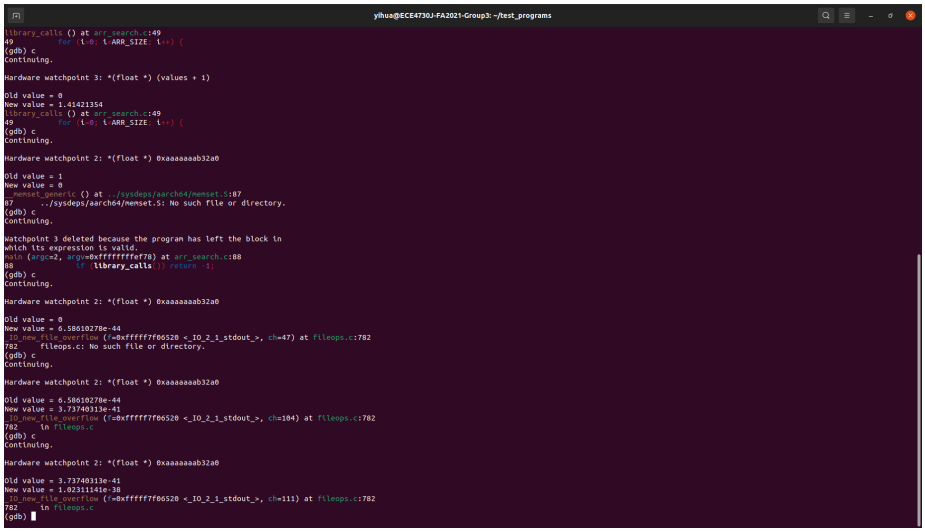


Figure 9. Continue execution until the program execution exits.

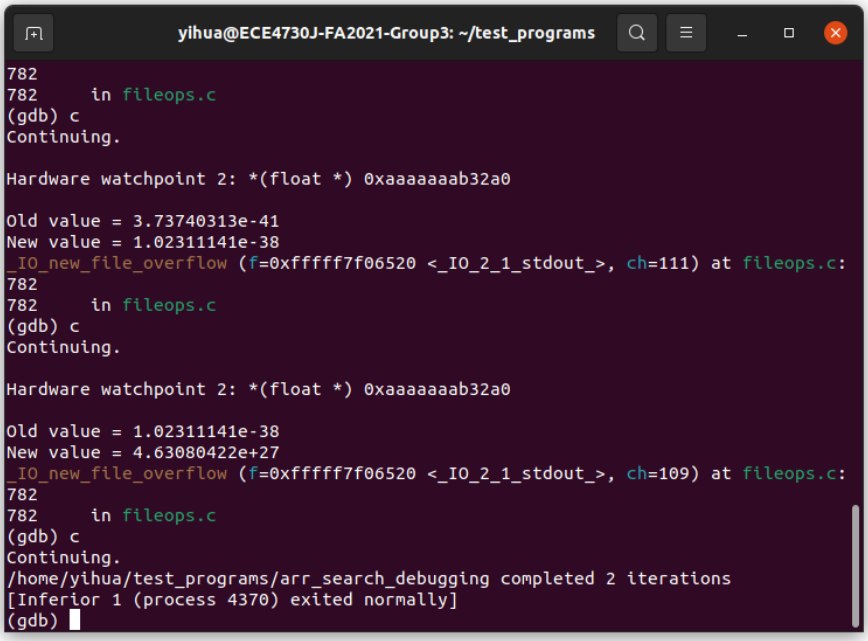


Figure 10. Continue execution until the program execution exits.

Values of `values[0]`: $0 -> 1 -> 0 -> 6.58610278e-44 -> 3.73740313e-41 -> 1.02311141e-38 -> 4.63080422e+27$

Values of `values[1]`: $0 -> 1.41421354$