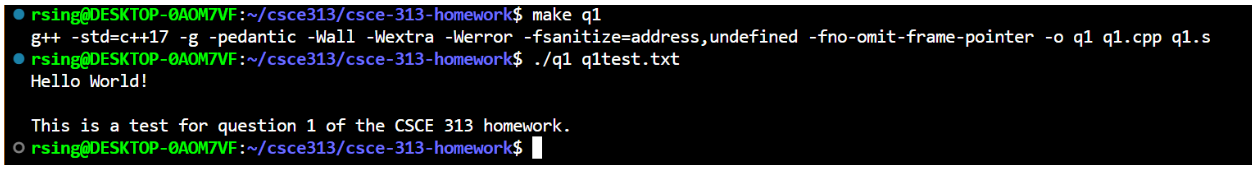
Rahul Singh
132002363
Section 510

Outputs and Written Answers for CSCE 313 Homework

(Note: I have created a Makefile and included it in the zip file for my submission for the grader's convenience. Just use make q<question#> as I have shown below in the screenshots of my output for each question)

Question 1:



```
rsing@DESKTOP-0AOM7VF:~/csce313/csce-313-homework$ make q1
g++ -std=c++17 -g -pedantic -Wall -Wextra -Werror -fsanitize=address,undefined -fno-omit-frame-pointer -o q1 q1.cpp q1.s
rsing@DESKTOP-0AOM7VF:~/csce313/csce-313-homework$ ./q1 q1test.txt
Hello World!

This is a test for question 1 of the CSCE 313 homework.
rsing@DESKTOP-0AOM7VF:~/csce313/csce-313-homework$
```

I created a file called q1test.txt which contains the text below for the purpose of opening a file for reading, and then copying the contents of the file to stdout. As you can see above from my output in the console, my openr() function worked correctly.

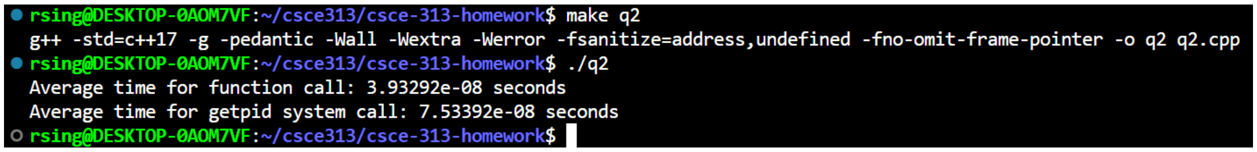
Contents of q1test.txt:

"Hello World!

This is a test for question 1 of the CSCE 313 homework."

Question 2:

My output(on WSL):



```
rsing@DESKTOP-0AOM7VF:~/csce313/csce-313-homework$ make q2
g++ -std=c++17 -g -pedantic -Wall -Wextra -Werror -fsanitize=address,undefined -fno-omit-frame-pointer -o q2 q2.cpp
rsing@DESKTOP-0AOM7VF:~/csce313/csce-313-homework$ ./q2
Average time for function call: 3.93292e-08 seconds
Average time for getpid system call: 7.53392e-08 seconds
rsing@DESKTOP-0AOM7VF:~/csce313/csce-313-homework$
```

Unix Machine	**Average Time per Function Call (seconds)**	**Average Time per getpid System Call (seconds)**
My WSL	3.93292e-08 seconds	7.53392e-08 seconds
First friend's GCP VM	7.17492e-08 seconds	8.58799e-08 seconds
Second friend's GCP VM	5.28116e-08 seconds	7.87846e-08 seconds

In this table, I have shown the average time per invocation for both function calls and getpid system calls on three different Unix-like machines (my windows subsystem for linux and two of my peers' GCP VMs). The values are in seconds and are based on a large number of invocations (one million invocations).

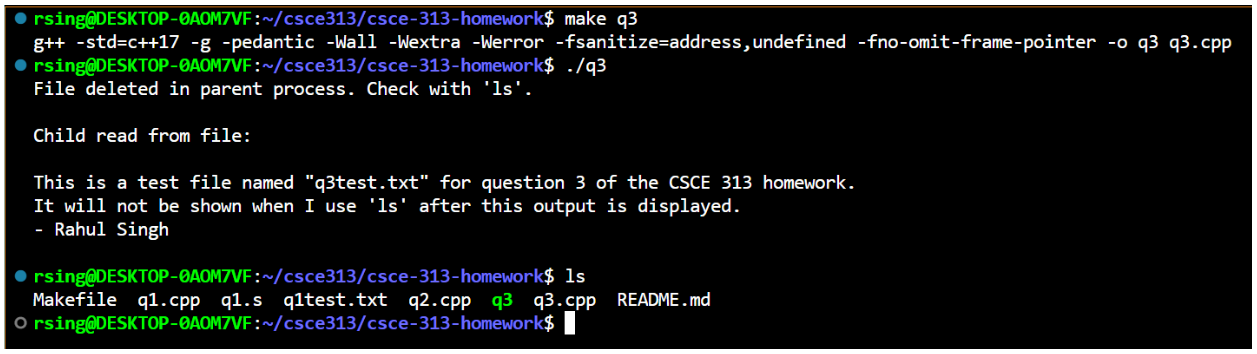
On all the Unix machines, function calls are faster than the getpid system call. The average time per function call is consistently lower than that of the system call. Between the three machines I ran my code on, after some calculations, the function call was an average of 2.53712e-08 seconds (or 25.3712 nanoseconds) faster than the system call.

The performance difference between function calls and system calls is relatively small. The exact difference may vary slightly from one machine to another, but in general, function calls are faster by a noticeable but not substantial margin.

This performance difference is expected because function calls typically involve less overhead compared to system calls. System calls often require transitioning from user mode to kernel mode, which introduces additional overhead.

The actual performance difference may vary based on the specific workload, the complexity of the function being called, and the efficiency of the underlying operating system's kernel.

Question 3:



```
● rsing@DESKTOP-0AOM7VF:~/csce313/csce-313-homework$ make q3
  g++ -std=c++17 -g -pedantic -Wall -Wextra -Werror -fsanitize=address,undefined -fno-omit-frame-pointer -o q3 q3.cpp
● rsing@DESKTOP-0AOM7VF:~/csce313/csce-313-homework$ ./q3
  File deleted in parent process. Check with 'ls'.

  Child read from file:

  This is a test file named "q3test.txt" for question 3 of the CSCE 313 homework.
  It will not be shown when I use 'ls' after this output is displayed.
  - Rahul Singh

● rsing@DESKTOP-0AOM7VF:~/csce313/csce-313-homework$ ls
  Makefile  q1.cpp  q1.s  q1test.txt  q2.cpp  q3  q3.cpp  README.md
○ rsing@DESKTOP-0AOM7VF:~/csce313/csce-313-homework$ 
```

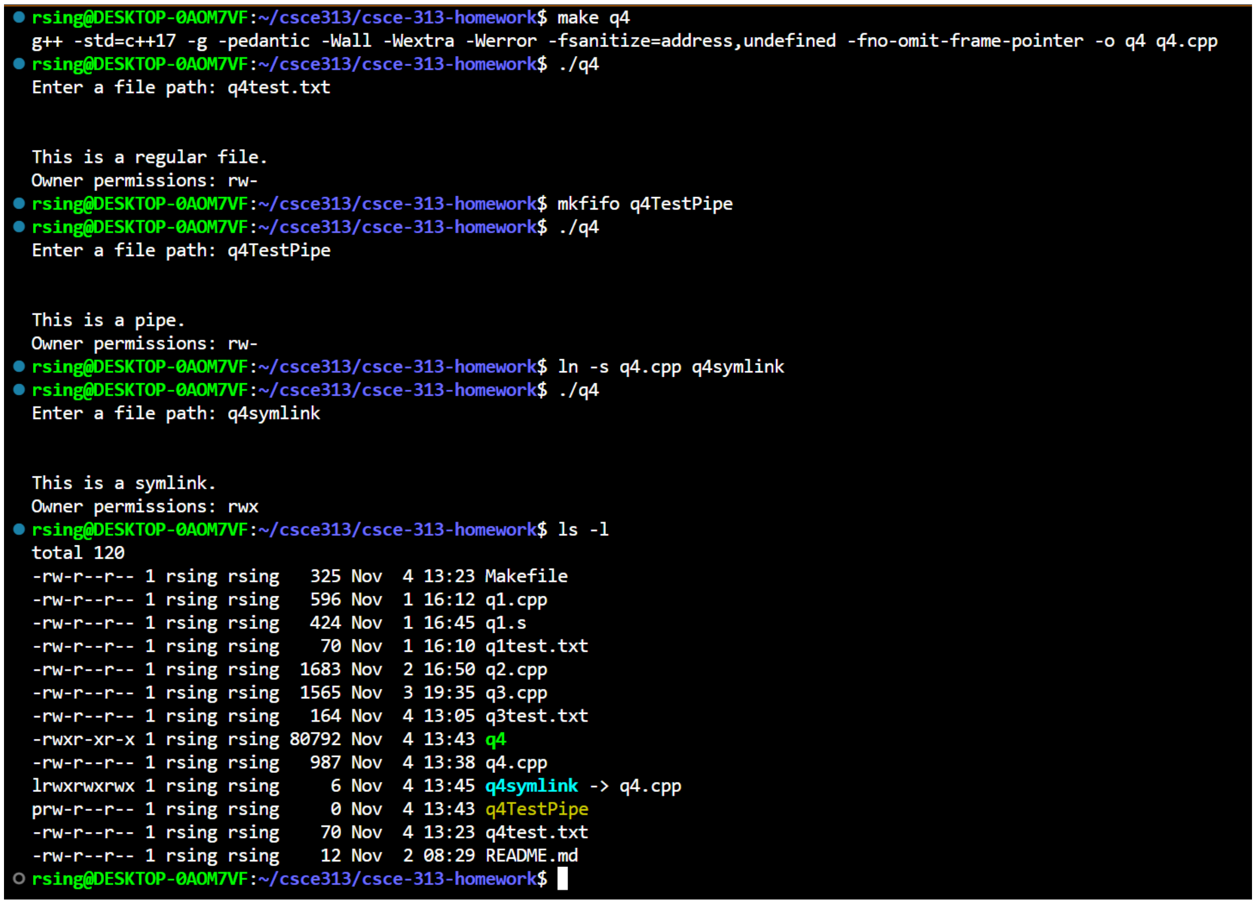
The program opens an existing file called “q3test.txt” for reading. I added some output to show when the parent deleted the file using the unlink system call. I also added some output to show when the child attempts to read from the file. As you can see in the screenshot of my terminal above, the output for child comes after the parent has already deleted the file.

This happens because the file's content is still in the pipe after the file is deleted. The pipe serves as a form of inter-process communication, allowing the child process to read the file's content from the pipe even after the file is deleted. This program demonstrates that even though the file is deleted by the parent process, the child process can still read the file's content.

Also shown in the screenshot above, I verified that file “q3test.txt” is no longer available immediately after running the program by using the ls command.

Contents of q3test.txt prior to being deleted:
“This is a test file named "q3test.txt" for question 3 of the CSCE 313 homework.
It will not be shown when I use 'ls' after this output is displayed.
- Rahul Singh”

Question 4:



```
● rsing@DESKTOP-0AOM7VF:~/csce313/csce-313-homework$ make q4
  g++ -std=c++17 -g -pedantic -Wall -Wextra -Werror -fsanitize=address,undefined -fno-omit-frame-pointer -o q4 q4.cpp
● rsing@DESKTOP-0AOM7VF:~/csce313/csce-313-homework$ ./q4
  Enter a file path: q4test.txt


  This is a regular file.
  Owner permissions: rw-
● rsing@DESKTOP-0AOM7VF:~/csce313/csce-313-homework$ mkfifo q4TestPipe
● rsing@DESKTOP-0AOM7VF:~/csce313/csce-313-homework$ ./q4
  Enter a file path: q4TestPipe


  This is a pipe.
  Owner permissions: rw-
● rsing@DESKTOP-0AOM7VF:~/csce313/csce-313-homework$ ln -s q4.cpp q4symlink
● rsing@DESKTOP-0AOM7VF:~/csce313/csce-313-homework$ ./q4
  Enter a file path: q4symlink


  This is a symlink.
  Owner permissions: rwx
● rsing@DESKTOP-0AOM7VF:~/csce313/csce-313-homework$ ls -l
  total 120
  -rw-r--r-- 1 rsing rsing   325 Nov  4 13:23 Makefile
  -rw-r--r-- 1 rsing rsing   596 Nov  1 16:12 q1.cpp
  -rw-r--r-- 1 rsing rsing   424 Nov  1 16:45 q1.s
  -rw-r--r-- 1 rsing rsing    70 Nov  1 16:10 q1test.txt
  -rw-r--r-- 1 rsing rsing  1683 Nov  2 16:50 q2.cpp
  -rw-r--r-- 1 rsing rsing  1565 Nov  3 19:35 q3.cpp
  -rw-r--r-- 1 rsing rsing   164 Nov  4 13:05 q3test.txt
  -rwxr-xr-x 1 rsing rsing 80792 Nov  4 13:43 q4
  -rw-r--r-- 1 rsing rsing   987 Nov  4 13:38 q4.cpp
  lrwxrwxrwx 1 rsing rsing     6 Nov  4 13:45 q4symlink -> q4.cpp
  prw-r--r-- 1 rsing rsing     0 Nov  4 13:43 q4TestPipe
  -rw-r--r-- 1 rsing rsing    70 Nov  4 13:23 q4test.txt
  -rw-r--r-- 1 rsing rsing    12 Nov  2 08:29 README.md
○ rsing@DESKTOP-0AOM7VF:~/csce313/csce-313-homework$ 
```

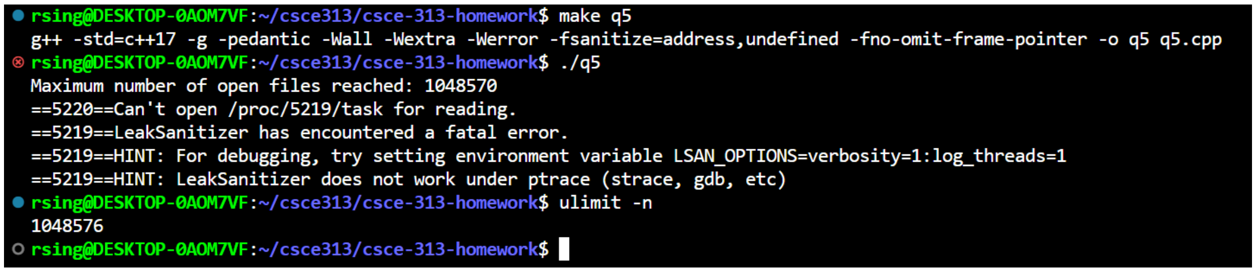
Inodes in the Unix file system are crucial because they store all the metadata about a file, except its name and actual data. This metadata includes information such as the file's size, its owner and group, its permissions, timestamps for creation, last modification, and last access, and the location of the file's data blocks.

My output above explained:

1. When I run the program with a text file called 'q4test.txt' as input, the output is "This is a regular file." and "Owner permissions: rw-". This means that 'q4test.txt' is a regular file and the owner has read and write permissions but not execute permissions.
2. When I create a named pipe 'q4TestPipe' using the 'mkfifo' command and run the program with 'q4TestPipe' as input, the output is "This is a pipe." and "Owner permissions: rw-". This means that 'q4TestPipe' is a pipe and the owner has read and write permissions but not execute permissions.
3. When I create a symbolic link 'q4symlink' to 'q4.cpp' using the 'ln -s' command and run the program with 'q4symlink' as input, the output is "This is a symlink." and "Owner permissions: rwx". This means that 'q4symlink' is a symbolic link and the owner has read, write, and execute permissions.

I then use ls -l to verify that the output I got is correctly displaying the owner's permission and whether the file path corresponds to a file/pipe/symlink.

Question 5:



```
● rsing@DESKTOP-0AOM7VF:~/csce313/csce-313-homework$ make q5
  g++ -std=c++17 -g -pedantic -Wall -Wextra -Werror -fsanitize=address,undefined -fno-omit-frame-pointer -o q5 q5.cpp
⊗ rsing@DESKTOP-0AOM7VF:~/csce313/csce-313-homework$ ./q5
  Maximum number of open files reached: 1048570
  ==5220==Can't open /proc/5219/task for reading.
  ==5219==LeakSanitizer has encountered a fatal error.
  ==5219==HINT: For debugging, try setting environment variable LSAN_OPTIONS=verbosity=1:log_threads=1
  ==5219==HINT: LeakSanitizer does not work under ptrace (strace, gdb, etc)
● rsing@DESKTOP-0AOM7VF:~/csce313/csce-313-homework$ ulimit -n
  1048576
○ rsing@DESKTOP-0AOM7VF:~/csce313/csce-313-homework$ █
```

(Note: I did my homework on WSL, not my GCP VM, so that might affect the number of files I am allowed to open with one process)

In the program I wrote, I'm continuously opening the same file in a loop until the system's limit for open files is reached (each time the open function is called, a new file descriptor is created, even if it's for the same file).

As you can see from the screenshot of my output above, my program crashes when I reach 1048570 files. I then check with ‘ulimit -n’ command to see the actual amount of open files I can have on my system which is 1048576 so they are roughly the same.

The reason why the maximum number of open files reached by my program (1048570) is less than the limit reported by ulimit -n (1048576) is because there are other open file descriptors that are not accounted for in my program. When a program is run, the operating system automatically opens several file descriptors for its own use. For example, by default, every C++ program has at least 3 file descriptors open when it starts: 0 for standard input (stdin), 1 for standard output (stdout), and 2 for standard error (stderr).