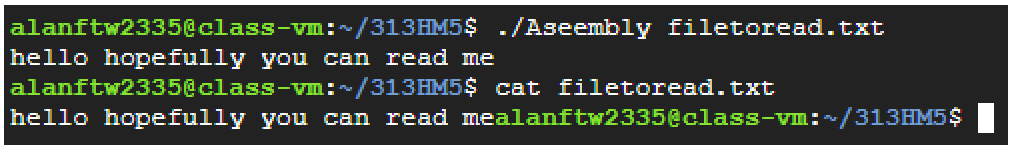
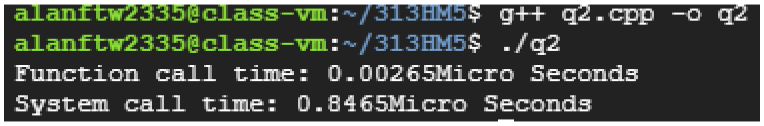
Question 1:
Function Output:



```
alanftw2335@class-vm:~/313HM5$ ./Aseembly filetoread.txt
hello hopefully you can read me
alanftw2335@class-vm:~/313HM5$ cat filetoread.txt
hello hopefully you can read mealanftw2335@class-vm:~/313HM5$
```

Question 2:
Function Output on Google VM:

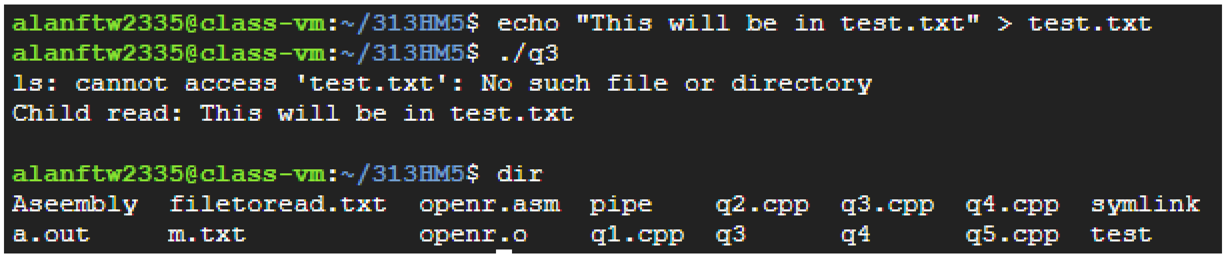


```
alanftw2335@class-vm:~/313HM5$ g++ q2.cpp -o q2
alanftw2335@class-vm:~/313HM5$ ./q2
Function call time: 0.00265Micro Seconds
System call time: 0.8465Micro Seconds
```

Response:
In the above, I compiled the average of each call over 20000 iterations. And timed the execution of each call using the clock() function from the ctime library, subtracting the end time from the start time and dividing the result by CLOCKS_PER_SECOND to translate into readable seconds.

From the results, it's easy to see that function calls happen at a much faster rate than system calls in C++. This is mainly because the function calls invoke user-defined code in easily accessible memory. In contrast, system calls require a transition of control to the kernel to request the required services from the operating system, which increases overhead between transfer times.

Question 3:
Program Output:



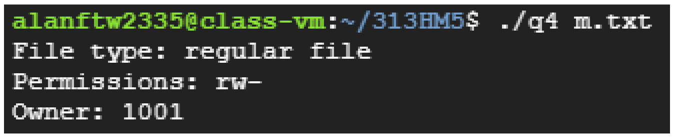
```
alanftw2335@class-vm:~/313HM5$ echo "This will be in test.txt" > test.txt
alanftw2335@class-vm:~/313HM5$ ./q3
ls: cannot access 'test.txt': No such file or directory
Child read: This will be in test.txt

alanftw2335@class-vm:~/313HM5$ dir
Aseembly  filetoread.txt  openr.asm  pipe    q2.cpp  q3.cpp  q4.cpp  symlink
a.out     m.txt           openr.o    q1.cpp  q3      q4      q5.cpp  test
```

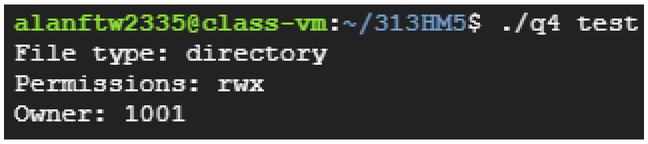
Response:
In the above example, the child is still able to access the content of the file as it still exists in the file descriptor data structure that the operating system maintains to keep track of all open files in a process. When calling unlink to remove a file from the directory, the file name is removed from the file system but not the file data itself. Allowing the child to read the data still as it knows where it's located.

Question 4:

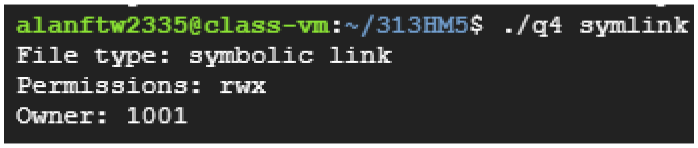
Function Output:



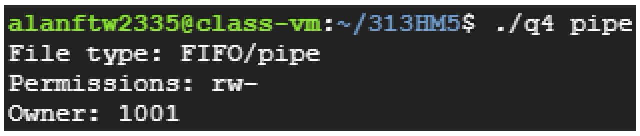
```
alanftw2335@class-vm:~/313HM5$ ./q4 m.txt
File type: regular file
Permissions: rw-
Owner: 1001
```



```
alanftw2335@class-vm:~/313HM5$ ./q4 test
File type: directory
Permissions: rwx
Owner: 1001
```



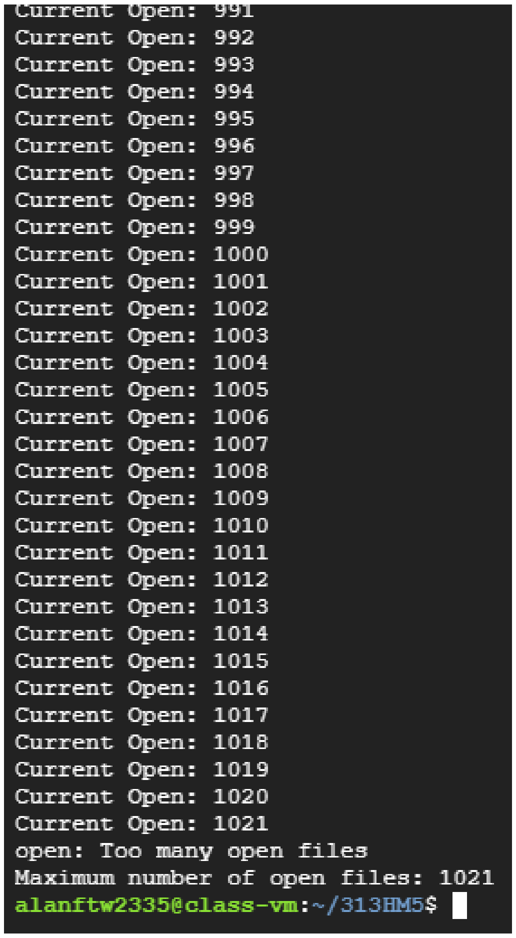
```
alanftw2335@class-vm:~/313HM5$ ./q4 symlink
File type: symbolic link
Permissions: rwx
Owner: 1001
```



```
alanftw2335@class-vm:~/313HM5$ ./q4 pipe
File type: FIFO/pipe
Permissions: rw-
Owner: 1001
```

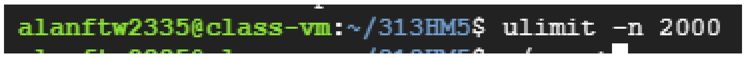
Question 5:

Program Output:

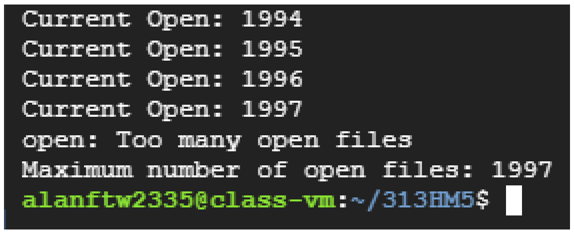


```
Current Open: 991
Current Open: 992
Current Open: 993
Current Open: 994
Current Open: 995
Current Open: 996
Current Open: 997
Current Open: 998
Current Open: 999
Current Open: 1000
Current Open: 1001
Current Open: 1002
Current Open: 1003
Current Open: 1004
Current Open: 1005
Current Open: 1006
Current Open: 1007
Current Open: 1008
Current Open: 1009
Current Open: 1010
Current Open: 1011
Current Open: 1012
Current Open: 1013
Current Open: 1014
Current Open: 1015
Current Open: 1016
Current Open: 1017
Current Open: 1018
Current Open: 1019
Current Open: 1020
Current Open: 1021
open: Too many open files
Maximum number of open files: 1021
alanftw2335@class-vm:~/313HM5$
```

Response:
To test that the open system call failed with this error we can run the open command inside a loop until it returns -1 which means that the file cannot be opened. Given that there would be no other context for this command to fail at 1021 invocations we know that this error is occurring because of the EMFILE error limit. To change and test this limit we can run the “ulimit -n” command which will change the file limit on our machine. For example:



```
alanftw2335@class-vm:~/313HM5$ ulimit -n 2000
```



```
Current Open: 1994
Current Open: 1995
Current Open: 1996
Current Open: 1997
open: Too many open files
Maximum number of open files: 1997
alanftw2335@class-vm:~/313HM5$
```