#### COEN 177: Operating Systems

**Lab assignment 9: File Performance Measurement**

**Objectives**

##### To evaluate file system performance in the face of sequential I/O requests, mainly disk reads and writes.

##### To evaluate the impact of multiple competing threads attempting to read/write simultaneously.

##### 

##### **Guidelines**

The goal of this assignment is to gain experience with simple evaluation of the performance of the file system. Specifically, you will be testing the performance of the system under varying conditions. I/O performance can be affected by more than the volume of data being moved. For example, it can be affected by the size of the individual requests being made, whether the requests are reading or writing data, and by the degree of contention for access to the disk. It can also be affected dramatically by the pattern of data access (e.g., whether it is sequential or random), but we will only be looking at sequential access in this assignment.

##### **Test files**

1. Create files of random data with 100K, 1M, 10M, 100M, and 10G bytes. You may use “cat” and “head” commands **( i.e. $cat /dev/random | head -c <bytecount>).** /dev/random are special files that serve as pseudorandom number generator. “cat” is used to display the content and “head” is used to display the specified number of lines. So the result of “cat” is sent to the upstream end of PIPE and “head” receives these results and redirects the content of the specified bytes to a file. **10GB will especially take a long time to create,** plan to use 1-2 days to run the command to create it. If you run out memory for 10GB, try deleting your previous files and running tests on file5 at the end. If it still doesn’t fit with the rest of your memory, use the closest size to 10GB it is able to reach and run tests based on that.

You may write the following commands in .sh file.

#!/bin/bash

# cat /dev/random | head -c <bytecount>

cat /dev/random | head -c 100000 > file1.txt

cat /dev/random | head -c 1000000 > file2.txt

cat /dev/random | head -c 10000000 > file3.txt

cat /dev/random | head -c 100000000 > file4.txt

cat /dev/random | head -c 10000000000 > file5.txt

Check the size of the files with the command “ls -la”

##### **C programs**

1. Write a C program to read the files your created in Step 1 from beginning to end, then measure how long does your program take to read each file. You may pass the name of the file as a command line argument “argv[1]”. Use a buffer of size 10000 bytes for each read operation. You may use the following code snippet:

char buffer[10000];

FILE \*fp;

fp = fopen(argv[1], "rb");

while (fread(buffer, sizeof(buffer), 1, fp)){

}

fclose(fp);

The “time” command can be used to determine how long a given command/ program takes to run. It returns three values:

* real: total time from the moment you hit the Enter key until the moment the command is completed
* user: CPU time spent in user mode
* sys: CPU time spent in kernel mode

Use real value to measure the I/O performance. This can be implemented by reading the return real value of the command:

$time ./Step2 file1.txt //Step2 is your executable code

To measure time for all files, you may write the following commands in a .sh file

for file in file1.txt file2.txt file3.txt file4.txt file5.txt

do

echo "Step2 $file"

time ./Step2 $file

echo " "

done

1. Modify your program in Step 2 to measure the I/O performance for 100, 1,000, 10,000 and 100,000 bytes of buffer sizes. You may read the buffer sizes from the command line as “argv[2]”.

To measure the time for all files and for different buffer sizes, you may write the following commands in a .sh file

for file in file1.txt file2.txt file3.txt file4.txt file5.txt

do

for buffer in 100 1000 10000 100000

do

echo "Step3 $file $buffer"

time ./Step3 $file $buffer

echo " "

done

done

1. Modify your program in Step 3 so that a write operation is made to a newly created file for each read operation. In other words, you will now be measuring the I/O performance by timing your program that copies each file to a new file. So now you are testing the speed of sequential reads+writes for files of varying size, and using I/O operations of varying size.

You may use the same .sh file in Step 3.

1. Modify your program in Step 4 so that multiple copies of each file are made. This can be easily implemented in you program by creating concurrent multiple threads, each will read and copy a file. You may set the number of threads based on a value from the command line “argv[3]”.

Measure the I/O performance for 2, 8, 32, and 64 concurrent threads where each will read/write files of different sizes using buffers of different sizes as well. If you run out of memory running tests with file5.txt, please note in your final submission and submit the results of what you were able to test.

You may write a .sh file as follows:

for file in file1.txt file2.txt file3.txt file4.txt file5.txt

do

for buffer in 100 1000 10000 100000

do

for thread in 2 8 32 64

do

echo "Step5 $file $buffer $thread"

time ./Step5 $file $buffer $thread

echo " "

done

done

done

1. Use “make” command to compile all your c programs (Steps 2 – 5) in one command. To do so, create a Makefile and include the following statements:

all: Step2.c Step3.c Step4.c Step5.c

gcc -o step2 step2.c

gcc -o step3 step3.c

gcc -o step4 step4.c

gcc -o step5 step5.c -lpthread

clean:

rm -f \*.out step2 step3 step4 step5

**Requirements to complete the lab**

1. Write and discuss the measurements of I/O performance in Steps 2 – 5 using all the buffer sizes specified, and upload your report on Camino
2. Demo to TA the following steps
3. ls -la and show all your files:

\* 100000 (5 zeroes)

\* 1000000 (6 zeroes)

\* 10000000 (7 zeroes)

\* 100000000 (8 zeroes)

\* 10000000000 (10 zeros)

1. Run step2 in the lab instructions above on your biggest file
2. Run step3 in the lab instructions above on your biggest file using buffsize 100000:

\* Show your code and how you get buffsize

1. Run step4 on biggest file, buffsize 100000:

\* Show how you get buffsize

\* Show how you set the new name of the file

\* ls -la and show the file sizes of the original and the copy

\* Compare the new copied file with the original

1. Run step5 on the file size of 1M, using buffsize 100000, with 2 and 64 threads:

\* Show how you get buffsize, numthreads

\* Show how you generate the names of the new files

\* Show where and how you create your threads

\* Compare some of the new copied files against the original