Introduction

The purpose of this project is to measure and analyze the time to startup a process. This will be done by using the C programming language to two programs that we will go into more depth shortly. The objectives for this project are to gain a thorough understanding of process creation and launch, file system calls and time functions.

As part of this project, there are two programs that we had to develop. The first program will be referred to as the “app” and the second program will be referred to as the “timer” henceforth.

1. Description of Programs

App

For the purpose of this project, the App will be the program that is launched by the timer have its startup time measured. The app is a program that executes for several minutes while completing the following tasks:

- Before anything begins, record the time when the application started.
- Create a File
- Write 10 records to the file where each record consists of 120 randomly generated characters.
- Read one randomly selected record back from the file and compare it to what was written originally.
- Delete the file before terminating the program.

In order to have the program run for several minutes, we must have the above procedure done repeatedly. This is accomplished by surround these operations in for loop, which we will discuss later on. Because a loop was needed, we had to adhere to the following constraints:

- All records on each loop iteration must be unique and cannot be the same as previous iterations.
- The records must be written to the same file, where previous records should be removed. (This meaning that at the end of the loop, there should be only 10 records).

In order to gain a better understanding of what the program is doing, we will look at the pseudo code and description of the implementation on the following page.

Pseudo Code for App

```
main procedure {

// get the system time once app starts
gettimeofday()
assign the value returned from the function to a double variable
create a int variable to store the pid

// create a file pointer to read and write to the file
File  *filePointer

// create array to hold the file name
char fileName[]

get the pid so we can use it for a unique file name

//open the file for reading and writing
file_pointer = fopen(fileName, "w+")

// call the fileWriter() function to open the file for reading and writing
fileWriter(file_pointer);

close the file pointer
remove the file
}
```

Main Procedure Description:

The main function is kept short so I could modularize the app. As we can see above, We start off by calling the gettimeofday function and create a double variable to store the value from gettimeofday(). A int variable is instantiated as well to store the pid for purposes of testing and creating a unique file name. Two char arrays are created to store the pid as a string and to store the filename. The functions snprintf, strcpy and strcpy are called to copy the pid into a string, copy the pid into the filename array and concatenate the the filename with the pid respectively. This allows for unique file names so we don't write to the same file. We then proceed to open the file for reading and writing using the "w+" parameter. If the file has been sucessfully be opened or created, we call the fileWriter function which is described on the next page.

```
fileWriter procedure (File *file_pointer) {
//create a char array for the 120 characters
char sequence []

//create a record holder array
char sequenceHolder[]

// run the loop for
for integer i; i  <  loop time; i++
        // seed the rand function by adding i to it to make it unique
        call srand() and add i to it
        call rand() function

        // iterate to create 10 records
        for integer j; j < record length; j++

                // iterate to sequence length to create 120 random characters
                for integer k; k < sequence length; k++
                        put each randomly generated character into sequence array

                store the sequence into sequenceHolder to use forcomparison for later
                write the record record to the file
                // rewind to beginning of the file so we can compare
                rewind(file_pointer)

                // call the function to compare the file to what we have written
                compareFile(sequenceHolder, file_pointer)
}
```

fileWriter Procedure Description:

The purpose of the fileWriter function is for us to generate a sequence of 120 random characters and write 10 distinct records of these sequences to a file. This is done repeatedly by defining the variable LOOP_TIME which specifies the number of times we will read and write to the file. We start the function by creating a char array to store the sequence of 120 random characters. Next, we create a char array (sequenceHolder) to store the records that we will write to the file. We iterate through a loop for the length that LOOP_TIME has specified with two nested loops where the first iterates to create the 10 records following a loop to write the 120 characters to that record. Once the writing has finished, we store the records in sequenceHolder for comparison later. Last, we rewind the file pointer to the beigning of the file for comparsion and call the compareFile function.

```
procedure  compareFile( char sequenceHolder[][], File *file_pointer) {

// create count variable to keep track of records we write
int count = 0

// to select a random file from the 10 records we stored
int randFile = (rand() % record length)

// create an array to store the file
char fileInput[][]

for integer i = 0; i < record length(10 records); i++
            get a record from the file
            compare the record from the file to the randomly selected record we chose

//rewind the file pointer so we can begin the over writing the file with new records
rewind(file_pointer)
}
```

<u>compareFile Procedure Description</u>

The compareFile function selects one random record that we have written to the file and read the records back from the file to compare them. The function starts by creating a count variable to keep track of the number of records not matched. This is important because if the count reaches 10 then this means we have no matches and there has been an error in the program. A int variable called randFile is created for us to assign a value created by calling the rand() function for integers between 1-10 for the random selection of a record to compare. Whichever value we get, we use as the index for the stored records to use for comparison. A char array called fileInput is then created to store the file record and we read through the file using a for loop to compare the read records. Once this has finished, we rewind the file pointer using the rewind() function and return to the main method.

<u>Timer</u>

The purpose of the timer is to measure the amount of time it takes to launch the app. The timer does this by utilizing the fork() and execve() system calls. The timer needs to launch the app twice, where both instances of the app are running at the same time. The timer achieves this by performing the tasks in the following order:

- Get the system time
- Call fork()
- Call execve() to start the app
- Call fork() once more

- Call execve() once more

This is a simplistic view at what the timer is actually doing so to gain a better understanding, we will take a look at the pseudo code for it.

Pseudo Code for Timer

```
main procedure() {
create structs for the time.
create  2 pidt_t  variables to store the pids

// call the get time of day function to record the time before we call fork()
gettimeofday()

// call the fork function to create a child process
pid = fork()

// if it's equal to 0 we are in the child process
if the pid == 0
    then call execve to launch the app

else
  //call the get time of day function to record the second start time  before the second fork()
  gettimeofday()

// the parent would reach this point and call fork
  fork()

// we are in the child process for the second fork
if pid2 == 0
  then call execve to launch the second app

else
// call the wait function to wait for both of the apps to finish
wait()

}
```

Timer Description:

The timer is a simple program that measures the startup times to launch the app and for utilizing the fork() and execve(). The most important parts of this function is the creation of the struct timeval to keep track of the times and the ordered nesting of the fork() function calls. The timer begins by creating the timeval variables and variables to store the pids for testing. Right before

the first fork() is called, we record the time. When fork() is called, the current process we are in becomes the parent and a new process is created and is known as the child.
The child runs as the timer would only this time it calls execve because it meets the conditional that the pid == 0. The execve quits the child process and starts the app. During this, the original parent reaches the else statement and records the second time and calls fork once more. Because fork() starts the process from after where it is called, execve is called once more and the child is destroyed while starting a second instance of the app. The parent continues to last else statement where the wait() function is called and the timer waits for both instances of the app to finished before exiting.

2. List of Functions and descriptions

The Following functions were used in the app:

- **gettimeofday(struct, NULL)**
  - This function is important because it allows us to obtain the time before we call fork() and to obtain the time when the app starts. This is what we will use for our measurements. The parameters for this function is a timeval struct and NULL.
- **fork()**
  - This function takes no arguments but creates an exact copy of the process it is being called from, this exact copy of it is known as the child. We will use this function to have 2 copies of timer running the app to measure performance.

- **execve(app, NULL, NULL)**
  - This function takes three arguments but for the purpose of this project, we will only use one, which is the name of the binary executable. When this function is called, the program that is calling this is completely terminated and the passed application is run. This function will be called after fork().
- **rand()**
  - This function is used to generate the random sequence of characters for each record as well as for the random selection of a record to be read back. The issue with this function however, is that it is pseudo random. This meaning that eventually, there will be duplicate “unique” randomizations. We also use rand to make sure we have unique files for each copy of the application running.
- **srand(time(NULL))**
  - To eliminate the repeating of duplicate record, we will use srand() to seed the rand() function, srand takes an argument to seed and in this case, we will use the time + i to ensure every iteration is unique for writing records where i is our current iteration.

- **fclose(FILE *file_pointer)**
    - This function is used to close the file we are reading and writing from at the end of the program. This ensure that there is no incoming and outgoing stream from and to the file.
- **remove(filename)**
    - This function is used to delete the file that we are writing and reading from. This is used to follow our specifications that the file should not exist after the program exits.
- **strcpy(const char *dest char *src)**
    - This function was used to copy the process id (pid) to the file char array to help create unique files for each instance of the app running.
- **strcat(const char * dest, char *src)**
    - This function was used to concatenate the file name into the filename char array with the pid so we could create a unique file for each instance of the app running.
- **snprintf(char *buffer, size_t count, const char* format_string)**
    - snprintf is used to read the pid and store it as a char in the char array. (This must be done because pid is returned as an int.)
- **printf()**
    - We will use this function to print out the times recorded, as well as any other vital information to be recorded for the purpose of this project.

- **rewind(File *filepointer)**
    - rewind is used to go back to the beginning of the file. This is useful because we have to write to a file, then read from it as well. We also need to overwrite the records in the file on each iteration.
- **fgets(char *str, int n, File *stream)**
    - fgets() is an important function because this is what we use to read the records back from the file.
- **fputs(const char *str, File *stream)**
    - This function was used to put the records into the file as a string.
- **wait(int *status)**
    - wait is useful because it makes sure that once we have started both applications then we must wait until they finish. This makes sure that the files are removed and the programs execute as designed.

3. Testing and Results

The main objective of our testing is to measure the startup times for running 2 instances of the app. However, before we can test this, We must make sure that the program itself functions as required. For the app, it must be able to properly open a file, write randomly generated records to it then read and compare those records to what we have written. The timer must start the app using fork() and execve() and properly record the time taken to do so. Therefore, the first step was making sure that the file was being created and that there were 10 records of random characters being written to it. In order to test this, I ran the program several times while checking that 120 characters were generated and that each record was unique. Next, care was needed to make sure that the file was properly read from and compared. So i compared the values that were stored and was was written to the file. By using printf() statements and GDB (debugger) I was able to confirm the values.

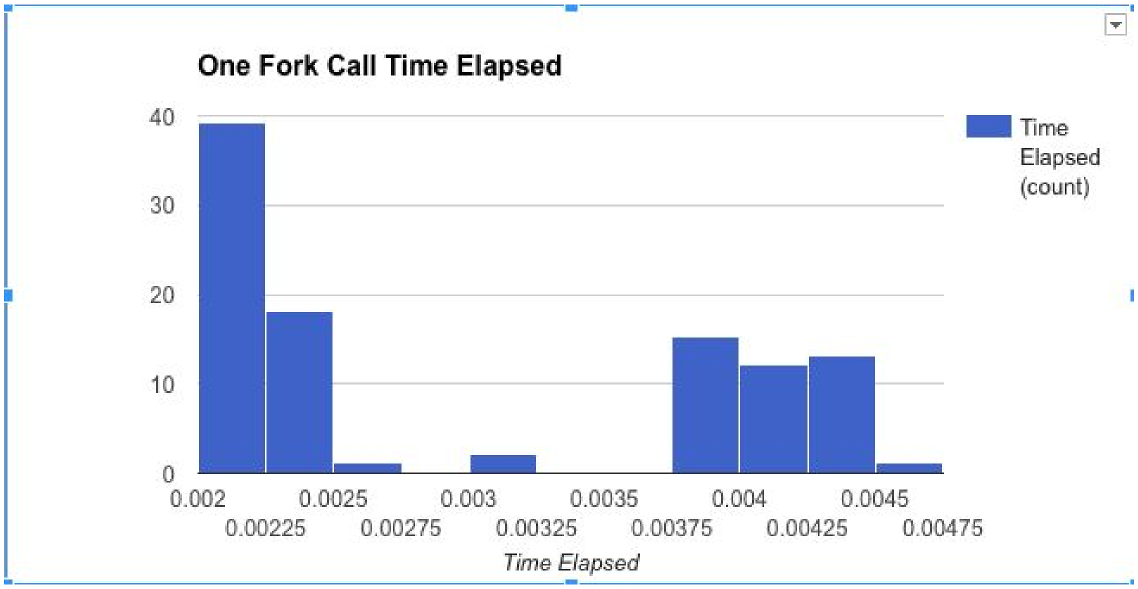
The Timer was tested by using printf() statements where I printed the pid's as well as used the GDB debugger to step from the program to make sure it the fork() and execve() functions worked as expected. I have left print statements within the timer and app to confirm the start times as well as the process ids for confirmation when running the program.

Errors found

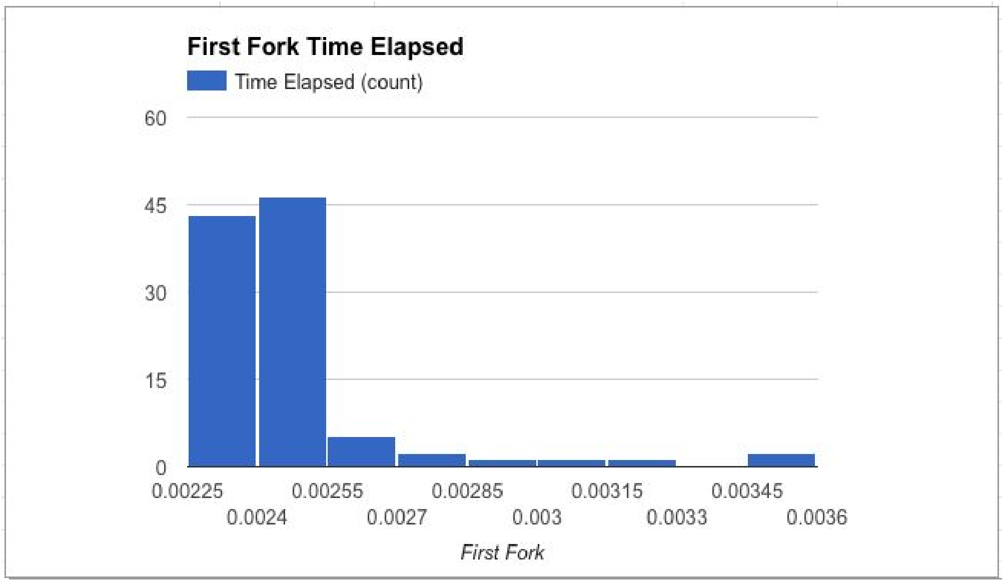
When testing, there was an error with comparing the read records to the records that were stored in the array in the compareFile() function. After careful testing, it was determined that the fgets() function was reading in a new line character (\n) alongside each record. Because of this, when comparing the file, the tests were showing that there was no matches. I solved this by placing the \0 character in place of the new line at the end of the array. Once this was done, the program began finding matches as expected.

Results

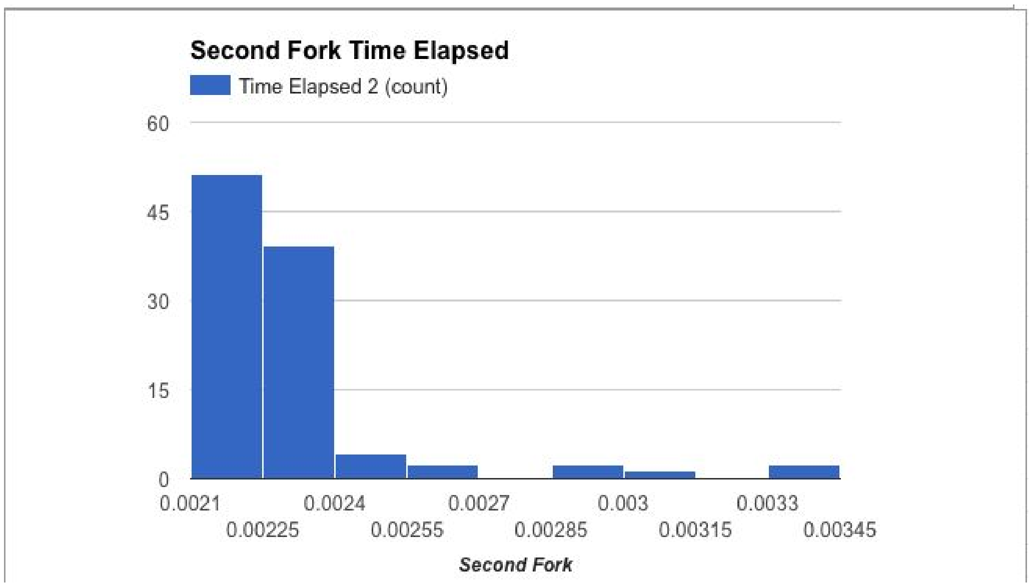
To get an accurate measure, the program was executed 100 times twice. The first 100 times were with only one call to fork to run the app once. The Second 100 times consisted of two fork calls and two instances of the app running at the same time.All times were recorded and stored in the accompanying excel sheet. We were specifically looking for the time it took the timer to launch the app once and for the second part of the test, launch a second instance of the app while the first one was running. First we will look at the times for only one instance of the app being started on the next page.



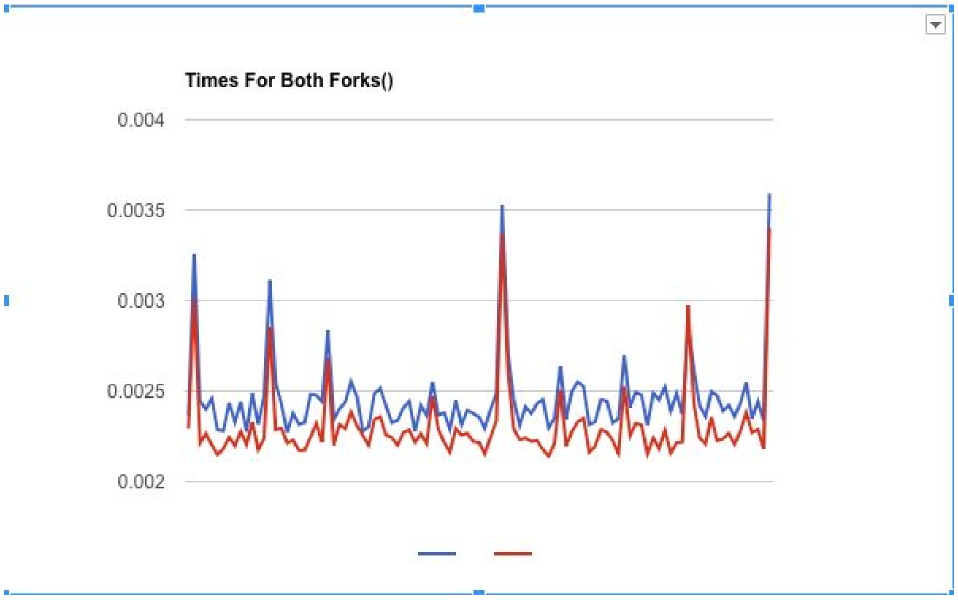
Looking at the above histogram, we can see that the majority of the app launches were done within 0.002 and 0.00225 microseconds. The mean is 0.003 microseconds with a standard deviation of 0.00093 microseconds and a 95% confidence interval of 0.0000059 microseconds. There were a few recorded times of 0.003 microseconds and a clustering of times from 0.00375 - 0.00475 microseconds. Next, we will take a look at the times for fork() being called twice where app was launched twice (the second instance of the app launching while the first was still running) below.



By looking at the histogram on the above , we can see that the majority of the instances ranged from 0.00225 to 0.00255 microseconds. For the first instance of the app, the mean is 0.00246 microseconds with a standard deviation of 0.000218 microseconds and a 95% confidence interval of 0.00000137 micro seconds. This information can be viewed on the excel sheet. We are in particular, interested in the time it takes for the second instance of the app to be launched. We look at the times in the histogram below:



By reviewing the above data, we can see that the second instance of starting the app ranged from 0.0021 microseconds to 0.0024 microseconds. The mean is 0.0023 microseconds with a standard deviation of 0.000213 microseconds and a 95% confidence interval of 0.000001338 microseconds.
To gain a better sense of the difference of times, here is a chart with times for both fork calls shown below:



As we can see on the previous page, The blue line corresponds to the first fork call and the red line corresponds to the second fork call. This seems to indicate that the second instance of app launched faster than the first.

Conclusion

After looking over the results, I have determined that the second instance of the app started the fastest. With only one fork call, the mean was 0.003 microseconds. With two fork calls being made, the first call had a mean of 0.00246 microseconds (which is less than the single fork call) and the second instance of the app launched with a mean of 0.0023 microseconds. I am not sure of why the single fork was the slowest and as a result, further testing will be required. However, I believe that the second launching of the app is faster because the program is already running and the code for starting the app has already been used, it’s possible that temporal and/or spatial locality aids in the speed of the second process launching the app. Overall, the application is launched faster on the second process than the first.