Table of Contents

Introduction 2

Algorithm 3

Implementation Challenges 4

Compiling and Executing Program 5

Output Sample 6

Output Explanation 11

Conclusion 13

Works Cited 14

Vagner Machado

CSC 340 Project 1

Professor John Svadlenka

15 February 2020

# Introduction

Project one has as objective exploring the use of system calls that allow for exchange of information between parent and child processes. To implement the requirements system calls such as pipe(), fork(), execlp(), wait(), exit() and gettimeofday() were used. Those system calls are supported in POSIX standard, which covers Linux, Unix and MacOS systems. This document explains the project’s requirements, research sources and implementation challenges while providing instructions to successfully compile and run the program.

Broadly, the requirements state that one is to write a program in C that uses a the above system calls to time how long a child process takes to execute a command passed as parameter into the program. During the program execution, the console displays the start and end time for the execution in seconds and milliseconds, as well as the command executed, the command’s outputs and time elapsed. Even though those are reasonably straightforward instructions, starting to code without much thinking can be counterproductive in the long run. Therefore, as any task related to computer science requires, one must do the proper planning and research before tackling any programming. For that, I studied Chapter 3 of *Operating System Concepts* ([Silberschatz](http://www.cs.yale.edu/homes/avi) et al. 2018) and carefully analyzed and digested the concepts and code examples. Additionally, I used as reference data and examples found on the web at *cplusplus.com* and the book *The C Programming Language* (Kernighan and Ritchie 2015). Upon studying these sources and requirements, I felt ready to design the following step-by-step algorithm.    
 Algorithm

0 - Validate user input, if error found print message and exit.

1 - Create a pipe.

2 - Parent forks a new process.

2.1 - Parent waits.

3 - Child process:

3.1 – Records start time.

3.2 - Displays start seconds and milliseconds.

3.3 - Writes seconds and milliseconds into pipe.

3.4 - Executes function passed as parameter by the   
 user, which prints output to console.

3.5 - Prints information if execlp() fails and exit   
 the program.

4 - Parent records current time or exits if child did not run.

4.1 - Parent displays its end time for seconds and   
 milliseconds.

4.2 - Parent displays the start time seconds and   
 milliseconds received from child process.

4.3 - Time elapsed is calculated and displayed on the   
 console.

5 - Successful run achieved.

Note that more elaborate and code specific comments can be found in the time.c file   
containing the source code for the project.

# **I**mplementation Challenges

While the mentioned sources informative and provided all resources needed to write this project, I encountered two situations while coding that caused bugs not warned by either the Eclipse or the Xcode compiler. Unfortunately, this lack of warning caused program debugging to be somewhat time consuming. The first situation occurred when I, inadvertently, called fork() before calling the pipe() function. This caused a *SIGPIPE* error*,* which occurs when a process attempts to write to a pipe that has its read end closed (Stack Overflow). I could not find a clear answer on the mechanics of a fork() before pipe(), but once I inverted those calls, as shown in examples from Chapter 3, my code worked. To the best of my knowledge, the order of function calls interferes in what variables are passed from parent to child process upon a fork() call and that by calling fork() before pipe(), the child and parent are no longer referring a single pipe and it is plausible that the child was attempting to write to a pipe not properly inherited. The second implementation bug was relatively faster to find. Inside the child process, a call to execlp() occurs to execute the parameter passed by the user into the program. I was not aware that the lines located in the child block after that function call are only run if execlp() fails to execute. Unaware of this fact, I had placed some print statements after that function call, and initially could not understand why they where not being executed on successful runs. Having the first implementation bug still fresh in my mind, I referred to the examples in the Chapter 3 and noticed the execlp() function was the last line in the block. A quick search on Stack Overflow revealed that I was not the only one out there with the same question and soon after learned that lines below execlp() are only reached case that function fails to execute. A simple rearranging of the code within the child block sufficed to fix this issue. These bugs illustrate how important it is to fully understand the usage, effects, and proper execution order of system call when programming.

# Compiling and Executing Program

In addition to the broad requirements cited earlier in this document, a specific set of five test cases was provided. The implemented application must support proper execution of these parameters passed as command line argument: ls, ps, whoami, hostname and date, though one parameter only per run. As it is common on command line application to provide a help command, this application supports a --help parameter that displays valid parameters, compilation and execution guidelines on the console. If the user provides an invalid parameter or if there is not a parameter provided to execute, the application prints a similarly informative message about the error and valid parameters.

This application was compiled and run using gcc, the compiler for most Linux systems, in a Macintosh computer following these steps. First, one must open the terminal and navigate to the folder in which the time.c is located. Then one runs the command gcc –c time.c in the terminal, which causes the program to compile, but not link, the file. A successful compilation generates an object file with a.o extension. In this case, time.o is created by the compilation. Then, one must link the object file created in the previous step into an executable file. I prefer to give the executable file the same name as the source and object files. To accomplish this task, the user must enter into the terminal gcc –o time.exe time.o. If this step is successful, one can see the file time.exe was created in the same directory as time.o and time.c. The last step is to run the executable file and provide the desired parameter to execute by entering ./time.exe and hit enter on terminal. Below are the step-by-step compilation, linking and execution of the program, as well as sample runs for help, and by passing valid and invalid parameters. For ease of reading the commands associated with the above tasks can be seen in bold.

# Output Sample

Vagners-MBP:Project1 Eclipse vagner$ ls

time.c

**Vagners-MBP:Project1 Eclipse vagner$ gcc -c time.c**

Vagners-MBP:Project1 Eclipse vagner$ ls

time.c time.o

**Vagners-MBP:Project1 Eclipse vagner$ gcc -o time.exe time.o**

Vagners-MBP:Project1 Eclipse vagner$ ls

time.c time.exe time.o

**Vagners-MBP:Project1 Eclipse vagner$ ./time.exe ls**

Name of command: ls

Start Time Seconds in Child: 1581879513

Start Time Milliseconds in Child: 531552

Command execution output:

time.c time.exe time.o

End Time Seconds set in Parent: 1581879513

End Time Milliseconds set in Parent: 535105

Start Time Seconds from Child in Parent: 1581879513

Start Time Milliseconds from Child in Parent: 531552

Time Elapsed: 3553 milliseconds

**Vagners-MBP:Project1 Eclipse vagner$ ./time.exe ps**

Name of command: ps

Start Time Seconds in Child: 1581879522

Start Time Milliseconds in Child: 496792

Command execution output:

PID TTY TIME CMD

750 ttys000 0:00.27 -bash

811 ttys000 0:00.00 ./time.exe ps

End Time Seconds set in Parent: 1581879522

End Time Milliseconds set in Parent: 533211

Start Time Seconds from Child in Parent: 1581879522

Start Time Milliseconds from Child in Parent: 496792

Time Elapsed: 36419 milliseconds

**Vagners-MBP:Project1 Eclipse vagner$ ./time.exe whoami**

Name of command: whoami

Start Time Seconds in Child: 1581879535

Start Time Milliseconds in Child: 892060

Command execution output:

vagner

End Time Seconds set in Parent: 1581879535

End Time Milliseconds set in Parent: 917940

Start Time Seconds from Child in Parent: 1581879535

Start Time Milliseconds from Child in Parent: 892060

Time Elapsed: 25880 milliseconds

**Vagners-MBP:Project1 Eclipse vagner$ ./time.exe hostname**

Name of command: hostname

Start Time Seconds in Child: 1581879548

Start Time Milliseconds in Child: 156976

Command execution output:

Vagners-MBP.fios-router.home

End Time Seconds set in Parent: 1581879548

End Time Milliseconds set in Parent: 177179

Start Time Seconds from Child in Parent: 1581879548

Start Time Milliseconds from Child in Parent: 156976

Time Elapsed: 20203 milliseconds

**Vagners-MBP:Project1 Eclipse vagner$ ./time.exe date**

Name of command: date

Start Time Seconds in Child: 1581879556

Start Time Milliseconds in Child: 900531

Command execution output:

Sun Feb 16 13:59:16 EST 2020

End Time Seconds set in Parent: 1581879556

End Time Milliseconds set in Parent: 921094

Start Time Seconds from Child in Parent: 1581879556

Start Time Milliseconds from Child in Parent: 900531

Time Elapsed: 20563 milliseconds

**Vagners-MBP:Project1 Eclipse vagner$ ./time.exe**

\*\*\* ERROR: <BLANK> is not a valid parameter for this program \*\*\*

The program expects one of the following parameters, run as ./<appName> <parameter>

ls

ps

whoami

hostname

date

EXAMPLE:

Step 1 - Compile the program: gcc -c time.c <ENTER>

Step 2 - Make executable with object file: gcc -o time.exe time.o <ENTER>

Step 3 - Run the program: ./time.exe <parameter> <ENTER>

**Vagners-MBP:Project1 Eclipse vagner$ ./time.exe wRoNgPaRaM**

\*\*\* ERROR: <wRoNgPaRaM> is not a valid parameter for this program \*\*\*

The program expects one of the following parameters, run as ./<appName> <parameter>

ls

ps

whoami

hostname

date

EXAMPLE:

Step 1 - Compile the program: gcc -c time.c <ENTER>

Step 2 - Make executable with object file: gcc -o time.exe time.o <ENTER>

Step 3 - Run the program: ./time.exe <parameter> <ENTER>

\*\*\* Error in child process, parent process is exited execution \*\*\*

**Vagners-MBP:Project1 Eclipse vagner$ ./time.exe --help**

\*\*\* Here are some helpful tips \*\*\*

The program expects one of the following parameters, run as ./<appName> <parameter>

ls

ps

whoami

hostname

date

EXAMPLE:

Step 1 - Compile the program: gcc -c time.c <ENTER>

Step 2 - Make executable with object file: gcc -o time.exe time.o <ENTER>

Step 3 - Run the program: ./time.exe <parameter> <ENTER>

**Vagners-MBP:Project1 Eclipse vagner$**

# Output Explanation

As previously mentioned, the application must support proper execution of these parameters passed as command line: ls, ps, whoami, hostname and date. The above sample runs show the proper steps for compilation, linking and executing of the required parameters, as well as proper output messages for invalid parameters and help request. This section provides a succinct explanation of the output for each of the supported parameters.

ls is a Linux command that lists all the contents of the present working directory. The execution of this command shows in the output above time.c, time.o and time.exe, which are the files contained in the directory where the first is the source code file, the second is the object file created during compilation of source file and the last one is the executable file created while linking the object file. ps is a Linux command used to check process status. The sample run for this command displays:

PID TTY TIME CMD

750 ttys000 0:00.27 -bash

811 ttys000 0:00.00 ./time.exe ps

This table provides the following information: PID column shows the process ID for the current processes in the system, TTY identifies the terminal type the user is logged into, TIME displays the amount of CPU time the process is running. The TTY value starts at zero and increments when more terminal windows are open (Stack Overflow). CMD lists, on the last column, the command that launched the process (Linfo). The whoami command output shows the username of the current user when the command is invoked, hence the output vagner. hostname,similarly, has an easy output to interpret. Vagners-MBP.fios-router.home can be broken down into two parts: the system’s host name (Vagners-MBP) and the system’s network information (fios-router.home). Lastly, date command output shows Sun Feb 16 13:59:16 EST 2020, a comprehensive date which includes week day, month, day, hour, minutes, seconds, time zone and year. It is worth mentioning that the last three outputs after date display information pertaining to missing, invalid, or help request parameters passed into the application. Those cases cause an output message to be printed onto the console explaining the application usage. This section provided a more detailed explanation of the output generated by the successful compilation, linking and execution of the program with the proper command line parameters.

# Conclusion

Using the fork() and pipe() amongst other system calls to program this assignment provided me with valuable insight on how a computer system works on a lower level. More specifically, I learned in more detail inter process communications between a parent and child process through the use of a pipe, and the unpredicted program flow caused by the wait() and execpl() system functions. After implementation and debugging, the analysis of the command line program output provided important information about POSIX functions supported in Linux and their usage.

# **Works Cited**

*Cplusplus.com*, “The C++ Resources Network.” V3.1, 2019, www.cplusplus.com/.

Kernighan, Brian W., and Dennis M. Ritchie. *The C Programming Language*. Pearson, 2015.

*Linfo.org*, The Linux Information Project, 20 May 2005, [www.linfo.org/ps.html](http://www.linfo.org/ps.html).

Silberschatz, Abraham, et al. *Operating System Concepts*. Ed. 10. Wiley, 2018.

*Stack Overflow*, “Where Developers Learn, Share, & Build Careers.” Stack Exchange Inc., 2020, stackoverflow.com/.