Operating Systems - 1: CS3510

Programming Assignment 1: Multi-Process
Computation of Execution Time

Assignment Report

Sagar Jain - CS17BTECH11034

November 16, 2018

Program Design

The following points describe the code design in detail.

1. The file Assign1Src-CS17BTECH11034.cpp begins by including the necessay libraries for input-output, few system calls and working with strings.

2. ***main*** has the arguments,

   (a) ***argc*** i.e. the number of command line arguments.

   (b) ***argv*** i.e. an array of string pointers.

3. In *main*, we start off by initilizing the following,

   (a) ***SIZE*** i.e. the size of the shared memory we would like to create.

   (b) ***name*** i.e. the name of the shared memory object to be created or opened.

   (c) ***fd*** i.e. a nonnegative value, the file descriptor of the shared memory object created.

   (d) ***ptr*** i.e. a pointer to the mapped shared area.

   (e) ***tv*** i.e. a timeval struct to store the time value returned from ***gettimeofday***.

   (f) ***pid*** to store the pid when a process is forked.

   (g) ***arguments*** is filled with appropriate values to be entered into ***execvp*** (NULL ending array of strings).

4. We then call ***shm_open*** to create a new shared memory region. We use the arguments ***O_CREAT / O_RDWR*** to create a new shared memory object if one doesn't already exist and open the object for read-write access.On success, ***shm_open*** returns a nonnegative file descriptor and we store it in ***fd***.

5. We then check for errors in creating of the shared memory.

6. We then truncate the file to the required size in bytes, using ***ftruncate***.

7. Then using ***mmap*** we store the value of the pointer to the mapped shared area in ***ptr***. We use the argument ***PROT_READ / PROT_WRITE*** to allow both read and write access. We use ***MAP_SHARED*** since we want to share the mapping.

8. We now use ***fork*** to fork a new process. Using the value of pid we make sure there are no errors in forking.

9. Then using the value of ***pid***, we control the child and parent process seperately.

10. The child notes the values of the current time using ***gettimeofday*** and stores it into the pointer using ***sprintf***.

11. The child executes the command given using command line arguments by putting them as arguments into ***execvp*** which replaces the current process image with that of the command given. The following are the points to note in the usage of ***execvp***,

    (a) I have used the command ***execvp*** instead of ***execlp*** so a user can run any command as long as it is present in the ***PATH*** of the system not restricted to any directory.

12. I have used wait at the beggining of the parent code (pid > 0), so that it starts running only and as soon as the child process is done.

13. The parent calls ***gettimeofday*** immidiately to record the time when the child process is done.

14. Then, using ***stringstream*** the parent converts the time value of the beginning of the child process (stored as a string in ptr) to a long intger.

15. The parent then subtracts the beginning and ending times to get the time the process took to complete and prints it into ***STDOUT***.

Output Analysis

The following is the analysis of the outputs using various commands,

1. The ***ls*** command,

   (a) Using this command in a folder with 10 files has given values between **0.002** and **0.004** seconds depending on the number of processes running in the background and other parameters. The following are the statistics for the same.

   (b) Usint *ls / -lR* , the total files printed out were **26140** taking a total of **6.68192s**.

Number of files in Folder	Number of Test Cases	Mean Time Taken	Variance
10	10	0.00245901	2.70561E-6
15	10	0.00249992	2.94336E-6

2. The ***echo*** command,

   (a) Using the word ***test*** as an argument for ***echo*** the following stats are observed. Mean: ***0.001934***, Variance: ***3.86830E-6***

   (b) using a ***10000*** letter string the time taken on average is ***0.004282***.

3. On running the command ***nano test*** and letting it run for 20 seconds according to a stopwatch, the recieved output is $20 \pm 0.1s$

4. On running the command ***top*** and letting it run for 20 seconds according to a stopwatch, the recieved output is $20 \pm 0.1s$

5. The ***mkdir*** and ***rm*** commands have the following statistics.

Command	Number of Test Cases	Mean Time Taken	Variance
mkdir	10	0.00481166	6.3473E-7
rm -r (3 levels deap)	10	0.00381166	5.5273E-7

Points To Note

1. The time taken for various commands shows variation based on the number of background processes running. At certain instances the time taken was even found to be 2X when their were browser, text-editors, etc. open in the background.

2. Commands requiring writing data seem to take more time than those which just read data (***touch*** vs ***cat***).

3. Running the same command in succession shows improvement in the time taken on some occassions, this could be in part due to the instruction getting stored in the cache when used repeatedly.

4. Since the values for commands like **ls** and **cat** are in order of 10^{-3} their is chance of errors getting introduced, so a statistical analysis of their execution time is necessary.

5. The output being recieved may be higher than the actual time taken due to the following reasons,

   (a) The parent process records the time only when the child has terminated and the wait command detects this termination, so their is some time getting added between the end of the command and the parent process recording the time. Using the following code for parent

   ```
   gettimeofday(&tv, NULL);
   sprintf(ptr,"%ld",tv.tv_sec*1000000 + tv.tv_usec);
   ```

   the time comes out to be an average of 0.000214s. This could be the magnitude of the error in the times recorded.

   (b) The command ***gettimeofday*** itself has some running time, this gets added when we use it to record the time.