Operating Systems

REPORT - Assignment 1

CS17BTECH11050 - SHRAIYSH VAISHAY

Goal

A multi-process program that determines the amount of time necessary to run a command from the command line.

Functions used

A list of new functions used apart from normal `printf()`, `scanf()`, `strcat()` etc...

Function Name	Purpose	Prototype
`fork`	Create a child process	`pid_t fork(void);`

`shm_open`, `shm_unlink`	Create/open or unlink POSIX shared memory objects	`int shm_open(const char *name, int oflag, mode_t mode);`
		
		`int shm_unlink(const char *name);`

`ftruncate`	Truncate a file to a specified length	`int ftruncate(int fd, off_t length);`
`mmap`	Map files or devices into memory	`void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);`
`gettimeofday`	Get time	`int gettimeofday(struct timeval *tv, struct timezone *tz);`
`execvp`	Execute a file - Duplicating the actions of shell while searching for it if file name does not have a slash character	`int execvp(const char *file, char *const argv[]);`
`wait`	Wait for process to change state	`pid_t wait(int *status);`

Strategy

We fork a child process that will execute the specified command. However, before the child executes the command, we will record a timestamp of the current time (`start`). The parent

process will wait for the child process to terminate. Once the child terminates, the parent will record the current timestamp for the ending time(`end`). The difference between the starting and ending times represents the elapsed time to execute the command.

Explaining the working of code

- Two processes are executed during the lifetime of this program. We will call them parent and child processes, where parent refers to the process created right after the application starts executing. The child process is a process created by the parent process.

- When the application starts, only parent process is in running state. The following snippet runs — in parent process.

```
char *command[argc];
for (int i = 1; i < argc; ++i) {
    command[i-1] = malloc(sizeof(argv[i]));
    strcpy(command[i-1], argv[i]);
}
command[argc-1] = NULL;

struct timeval *start, end;
int status, fd;
```

  The job of these commands is to store the command in a `command` array with it's last entry as `NULL`. It also declares a pointer to `start` and struct `end`, which are going to hold the start and end times of the process respectively.

- The command `pid_t pid = fork();` runs on the parent process and creates a child process, such that it has all variables that the parent has with same values, except the variable `pid` itself.
  - ❏ `pid = 0` for child process
  - ❏ `pid > 0` for parent process - pid stores the actual process id (which the OS uses to refer to the child process) of child process
  - ❏ `pid < 0` for any process implies that there has been an error while creating the process and it should be terminated.

  It should be observed that the rest of the instructions are also copied to the child process. The two processes run independently now.

- The following snippet runs only for child process

```
fd = shm_open("TimeCalculationSharedMem", O_CREAT | O_RDWR, 0666);
ftruncate(fd, sizeof(struct timeval));
struct timeval *start = (struct timeval *)mmap(0, sizeof(struct
```

```
timeval),
    PROT_WRITE, MAP_SHARED, fd, 0);
gettimeofday(start, NULL);
execvp(command[0], command);
```

`shm_open("TimeCalculationSharedMem", O_CREAT | O_RDWR, 0666)` creates a shared memory region and returns an integer which holds the *File Descriptor* for it. The area of shared memory can be accessed with this integer.

`ftruncate(fd, sizeof(struct timeval));` truncates the length of the memory area to the size of `sizeof(struct timeval)` — which we are going to use to store the timeval object.

The variable `start` is a pointer for an area in shared memory that will hold the current time — mapped using `mmap` with writing permissions as symbolized by `PROT_WRITE`. The next line writes the current time object in the shared memory, which it addresses using `ptr`. Right after the current time is stored, command is set to execute. Upon execution of the line `execvp(command[0], command);`, the instructions in this(child) process are replaced by the some other instructions(given by the OS) to execute the required command (with it's arguments in the array). The child process has to terminate as soon as the system call terminates.

- The following snippet runs only for the parent process - after creation of child process

```
wait(&status);
gettimeofday(&end, NULL);
if(status == 0) {
    // Correct execution of child!
    fd = shm_open("OS", O_RDONLY, 0666);
    start = (struct timeval *) mmap(0, sizeof(struct timeval),
PROT_READ, MAP_SHARED, fd, 0);
    printf("Elapsed time: %lf\n", (end.tv_sec - start->tv_sec) +
(end.tv_usec - start->tv_usec)/1000000.0);
}
shm_unlink("OS");
```

The variable `status` stores the exit status of child process and `end` stores the time of child process termination. Once the child process terminates, it is no longer linked to the child process. The shared memory region is opened for the parent with `RD_ONLY` - Read Only access. Then it reads the address to the command execution start time, that was written by the child process in the shared memory. The time taken by the process, as calculated by the program is given by,

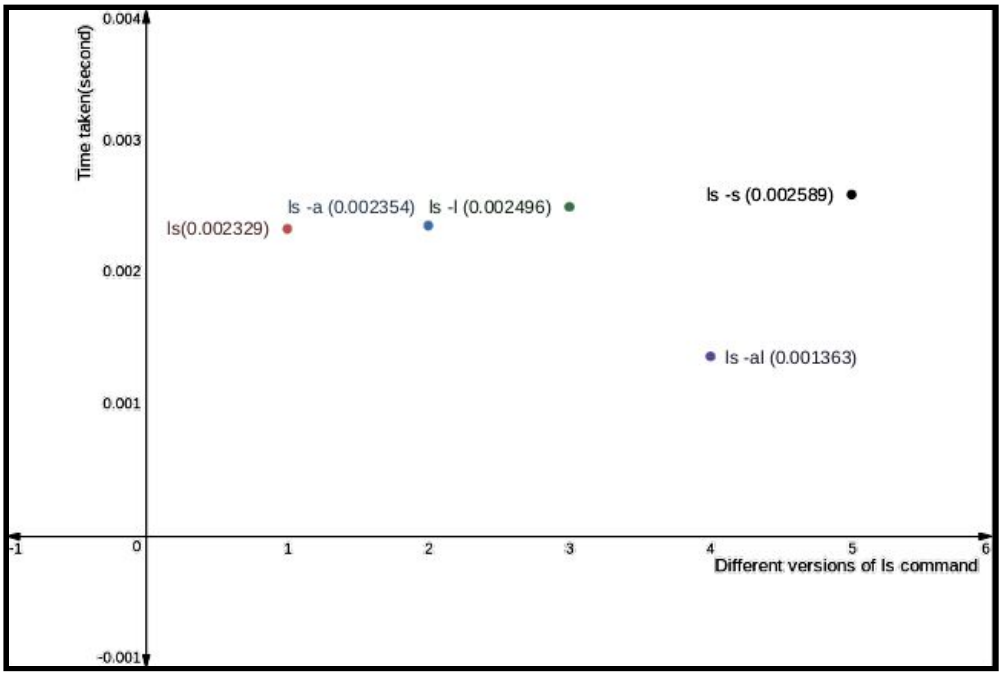
$$T_{net} = (end_{sec} - start_{sec} + \frac{(end_{usec} - start_{usec})}{10^6}) \, seconds$$

The value of elapsed time(T_{net}) is then displayed on the screen and the shared memory region is unlinked from the main process.

Analysis

A few observations with the code

- The parent process need READ-ONLY access and child process needs read and write access to the shared memory.
- The statement `gettimeofday(start, NULL);` is assumed to take zero time, which is not true! So, in this program we are calculating time taken by the command plus the time taken to write the current time in memory. This is a minute error, which could not be avoided.
- Time taken by different versions of the ls command



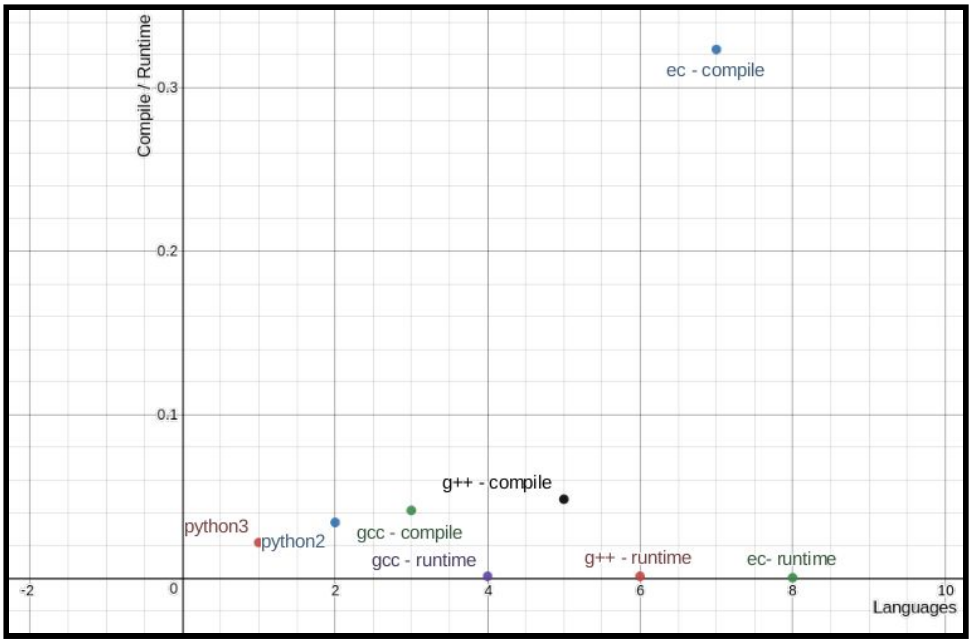
- Overhead for writing time using `gettimeofday()` can be calculated using the following procedure:
  - ❏ Remove the unlinking command from the program.
  - ❏ Now, we run the following command

```
$ ./a.out ./a.out ls
```

This writes the start time of outer process in the shared memory. Then the inner process writes it's start time in the shared memory, which overwrites the previous value. Now, the inner process terminates with the time 0.002509 seconds. As soon as the inner process terminates, it logs the time elapsed(including time taken by `gettimeofday()`) and stops. Then the outer process terminates and logs the total time taken (0.002936) from the start of inner process to logging the time for the

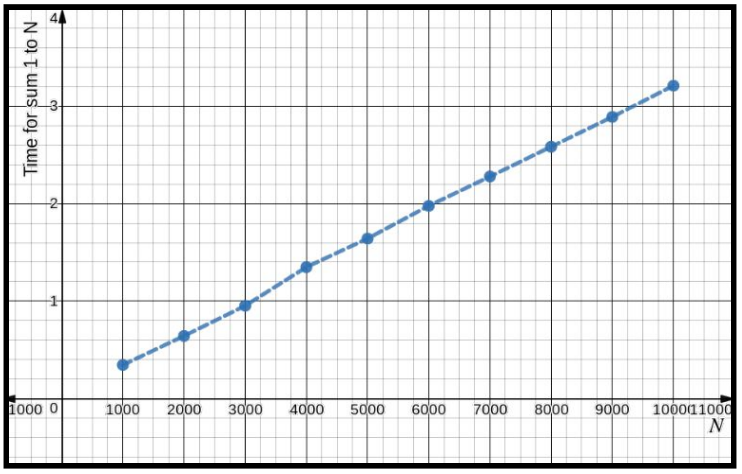
second time. The difference in these values gives the overhead. The average value was found to be 0.00043 seconds.

- A program to find sum of numbers from 1 to 999
  Python takes much longer than c or c++ to run a file, which is expected. Also, eiffel takes long to compile, as it is a contract based language and the binary file runs fast.



- Linearity in repeated execution of same statements.

  The following graph illustrates that the time taken increases linearly with the number of computations (in bash). This is in agreement with our intuition.



*Scripting was used for running the command many times with linearly increasing computations.

- Such behaviour is not observed if general system call commands are executed repeatedly such as —

```
touch hello.txt
echo "Hello world!" >> hello.txt
rm -rf hello.txt
```

A reason for such behaviour could be operating system level optimisations and the time is observed to be nearly a constant.