Coding Assignment 1 - Report

Aniruthan R
ME17BTECH11007

Contents

Chapter 1

Overview

The aim of this project is to develop a multi-process program, written in C, involving IPC to find the time needed to execute an instruction from the command line.

To run the command line instruction, the image of the current process must be completely replaced by that of the command. Therefore, the timer cannot be implemented as a single process. To run the command line instruction, a child processes is created and replaced with the image of the issued command. This way, the execution time can be computed in the parent process.

The implementation involves the use of the following:

1. **Process Creation:** `fork()` and `exec()` system calls are used to run the command line instruction. The procedure is discussed in Chapter 2.

2. **Inter Process Communication (IPC):** Shared Memory IPC is used to pass the time stamp just before the command line instruction is executed, which is used in computing the elapsed time. The procedure is discussed in Chapter 3.

Chapter 2

Process Creation

2.1 What is a Child Process?

In multitasking Operating System, one process can create an other process. Here, the process that creates another process is referred to as the Parent Process and the created process is referred to as the Child Process.

2.2 POSIX API for Process Creation

The `fork()` system call creates an identical copy of the current process. Both processes continue to execute exactly after the `fork()` call.

The `exec()` collection of system calls in UNIX/Linux operating systems cause the running process to be completely replaced by the program passed as an argument to the function. As a new process is not created, the process identifier (PID) does not change, but the data, heap and stack of the original process are replaced by those of the new process.

The `wait()` system call blocks until child process exits or terminates.

Chapter 3

IPC - Shared Memory

3.1 What is shared memory?

Inter Process Communication through shared memory is a concept where two or more process have access to a common memory location. The processes communicate by reading and/or writing to this shared memory location.

> **Note:** It is important to note that read-write synchronisation must be implemented by the processes using shared memory communication. It is not provided by the OS.

3.2 POSIX API for Shared Memory Communication

POSIX shared memory is organized using memory-mapped files, which associate the region of shared memory with a file. The following steps are involved in creating a shared memory region.

A process must first create a shared-memory object using the `shm_open()` system call:

```
int shm_open(const char *name, int oflag, mode_t mode);
```

`const char *name`: Points to the name to be assigned to the created shared memory object.

`int oflag`: Specifies the file status flags and file access modes to be used with the open file description.

`mode_t mode`: Specifies the values to be set for the permission bits of the created shared memory object.

A successful call to `shm_open()` returns an integer file descriptor for the shared-memory object.

Once the object is established, the `ftruncate()` function is used to configure the size of the object in bytes.

```
int ftruncate(int fildes, off_t length);
```

`int fildes`: Is the file descriptor for the file to be truncated.

`ofF_t length`: Is the new file size.

Finally, the `mmap()` function establishes a memory-mapped file containing the shared-memory object. It returns a pointer to the memory-mapped file that is used for accessing the shared-memory object.

```
void *mmap(void *addr, size_t length, int prot, int
flags, int fd, off_t offset);
```

`void *addr`: Is the desired starting address of the memory mapped region. A page-aligned address is chosen if `NULL` is passed.

`size_t length`: Is the number of bytes to map.

`int prot`: Is the Desired protection of the memory mapped region. This parameter is specified as a bitwise inclusive OR of one or more of `PROT_NONE`, `PROT_READ`, `PROT_WRITE`, and `PROT_EXEC`.

`int flags`: Specifies attributes of the mapped region as the results of a bitwise inclusive OR of any combination of `MAP_FILE`, `MAP_ANON` (or `MAP_ANONYMOUS`), `MAP_VARIABLE`, `MAP_FIXED`, `MAP_SHARED`, or `MAP_PRIVATE`.

`int fd`: Is the file descriptor specifying the file that is to be mapped. This parameter should be -1 if mapping anonymous memory.

`off_t offset`: Offset from where file should be mapped. This parameter has no meaning if mapping anonymous memory.

Chapter 4

Implementation in C

4.1 Required Header Files

```c
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/time.h>
#include <sys/mman.h>
#include <sys/wait.h>
```

Listing 4.1: Headers

1. **stdio.h**: Included for the definition of `printf()`, used to print text to the console.
2. **fcntl.h**: Included for the definition of macros like `O_CREAT` and `O_RDWR`.
3. **unistd.h**: Included for the definition of `fork()` and `exec()` functions.
4. **sys/time.h**: Included for the definition of `struct timeval` and `gettimeofday()`.
5. **sys/mman.h**: Included for the definition of shm related functions, mentioned in Section 3.2.
6. **sys/wait.h**: Included for the definition of `wait()`, used in the parent process.

4.2 Common instruction and fork

In the common instructions, variables are declared to store data common to both the child and the parent processes and the shared memory space is set up.

```
double* ptr; // shared memory pointer
struct timeval current_time; // to store the current time
pid_t pid; // to store the child process id
int SIZE, FD, i;
```

Listing 4.2: Declaration

`ptr` is a pointer that points to the shared memory region (opened in Listing 4.3). `current_time` is a variable of type `struct timeval` that is used to obtain the current time using `gettimeofday()` in both the processes. `pid` is used to store the returned value from the `fork()` call. `FD` is used to store the file descriptor returned by `shm_open()`, `SIZE` is used to store the size of the shared memory space, and `i` is used as an iterator in the child process.

```
SIZE = sizeof(double); // size of the shared memory space
FD = shm_open("start_time", O_CREAT | O_RDWR, 0666); // opening the shared memory
ftruncate(FD, SIZE); // resizing the shared memory to the required size
ptr = (double*) mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, FD, 0); //
    mapping the shared memory locaiton to the pointer
pid = fork();
```

Listing 4.3: Initialization

`SIZE` is initialised to be `sizeof(double)`, which is usually 8 bytes. `FD` is initialized as the descriptor returned by `shm_open()`. This sets up a shared memory region that both the processes can use to communicate. The `O_CREAT` flag specifies that the file must be created if it doesn't exist, and `O_RDWR` specifies that the file is opened for read and write operations. `ftruncate()` is used to resize the shared region to `SIZE` length, which is `sizeof(double)` in this case. Finally, the mapping is created and stored in `ptr`. `MAP_SHARED` is used to set up the region as a shared region. The other parameters are as explained in Section 3.2.

This is followed by a `fork()` call. This is where the child process is created. The processes can be distinguished by the value of `pid`, returned from `fork()`. It's values is `-1` is case of a fork failure, `0` in case of the child process and the ID of the child process in case of the parent process.

After the child and parent process definition `shm_unlink()` is called to unlink the shared memory space.

4.3 The Child process

```c
if (argc > 1) { // if a command is passed
    // prepare the argv list to pass
    for (int i = 1; i < argc; i++) {
        argv[i-1] = argv[i];
    }
    argv[argc - 1] = NULL;
    // write cur time into shared memory space
    gettimeofday(&current_time, NULL); // get the current time
    *ptr = current_time.tv_sec + 1e-6 * current_time.tv_usec;
    //  replace the child process with the command line instruction
    if (execvp(argv[0], argv) == -1) {
        printf("Error running your command.\n"); // in case an error occurs while replacing
    }
}
else { // no command is passed as argument
    printf("No command to execute\n");
}
```

Listing 4.4: The Child Process

The child process can be identified by the `pid` returned by `fork()`. For the child process `pid = 0`. First, The first entry of `argv` will be the executable fine name of the C code. This should be truncated before passing it to `execvp()`. This can be done with a simple `for` loop which overwrites the unneccesary argument. The final argument in the list of arguments must be explicitly stored as `NULL` before passing it to `execvp()`, as that is how `execvp` will understand that the list of arguments is over.

Now, the current time is obtained using `gettimeofday()`. The `struct timeval` has two members: `tv_sec`, which stores the current time in seconds, and `tv_usec` which stores the current time in micro second. It is converted to a double and is stored in the shared memory space pointed by `ptr`.

Now that the start time is safely saved in the shared memory space, the child process can be completely replace with the command line instruction. To do this `exec()` family of functions can be used. Here, `execvp()` is used to run the command.

Appropriate errors are handled if the command is not executed successfully, or if no command is passed.

4.4 The Parent process

```
wait(NULL); // wait for the child to complete
gettimeofday(&current_time, NULL); // get the current time
printf("\n\nElapsed Time: %lf second(s)\n", current_time.tv_sec + 1e-6 *
    current_time.tv_usec - *ptr); // logging the time
```

Listing 4.5: The Parent Process

The parent process can be identified by the `pid` returned by `fork()`. For the parent process `pid > 0`. First, the parent process waits for it's child to complete by calling `wait()`. After successful completion of the child process, the current time is obtained once again as done in the child process. The start time is read from the shared memory using `ptr`. The difference between these two values is printed as the elapsed time.

4.5 Sample I/O

4.5.1 Compiling the code

The code is compiled using GCC as follows:

```
gcc Asgn1-me17btech11007.c -o time -lrt
```

`-lrt` is specified to link the definition of `shm_open()` and `shm_unlink()`, as mentioned in the Linux Manual[2].

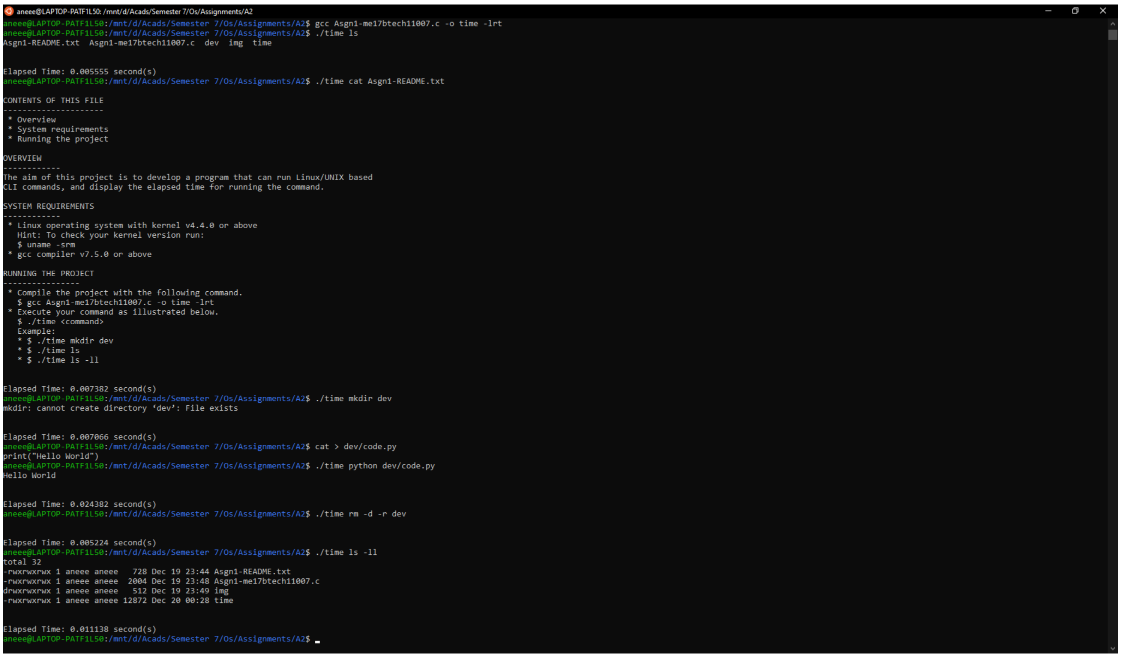
4.5.2 Examples

The first command passed is `ls`, which lists the contests of the directory, and takes `0.005555` seconds to execute.

The next command is `cat Asgn1-README.txt`, which lists out the contents of Asgn1-README.txt and takes `0.007382` seconds to execute.

The next command is `mkdir dev`, which creates a directory named dev, and takes `0.007066` seconds to execute.

The next command is issued to create a python file `code.py` for testing.



```
aneee@LAPTOP-PATF1L50: /mnt/d/Acads/Semester 7/Os/Assignments/A2
aneee@LAPTOP-PATF1L50:/mnt/d/Acads/Semester 7/Os/Assignments/A2$ gcc Asgn1-me17btech11007.c -o time -lrt
aneee@LAPTOP-PATF1L50:/mnt/d/Acads/Semester 7/Os/Assignments/A2$ ./time ls
Asgn1-README.txt  Asgn1-me17btech11007.c  dev  img  time


Elapsed Time: 0.005555 second(s)
aneee@LAPTOP-PATF1L50:/mnt/d/Acads/Semester 7/Os/Assignments/A2$ ./time cat Asgn1-README.txt

CONTENTS OF THIS FILE
---------------------
 * Overview
 * System requirements
 * Running the project

OVERVIEW
------------
The aim of this project is to develop a program that can run Linux/UNIX based
CLI commands, and display the elapsed time for running the command.

SYSTEM REQUIREMENTS
------------
 * Linux operating system with kernel v4.4.0 or above
   Hint: To check your kernel version run:
   $ uname -srm
 * gcc compiler v7.5.0 or above

RUNNING THE PROJECT
----------------
 * Compile the project with the following command.
   $ gcc Asgn1-me17btech11007.c -o time -lrt
 * Execute your command as illustrated below.
   $ ./time <command>
   Example:
   * $ ./time mkdir dev
   * $ ./time ls
   * $ ./time ls -ll


Elapsed Time: 0.007382 second(s)
aneee@LAPTOP-PATF1L50:/mnt/d/Acads/Semester 7/Os/Assignments/A2$ ./time mkdir dev
mkdir: cannot create directory ‘dev’: File exists


Elapsed Time: 0.007066 second(s)
aneee@LAPTOP-PATF1L50:/mnt/d/Acads/Semester 7/Os/Assignments/A2$ cat > dev/code.py
print("Hello World")
aneee@LAPTOP-PATF1L50:/mnt/d/Acads/Semester 7/Os/Assignments/A2$ ./time python dev/code.py
Hello World


Elapsed Time: 0.024382 second(s)
aneee@LAPTOP-PATF1L50:/mnt/d/Acads/Semester 7/Os/Assignments/A2$ ./time rm -d -r dev


Elapsed Time: 0.005224 second(s)
aneee@LAPTOP-PATF1L50:/mnt/d/Acads/Semester 7/Os/Assignments/A2$ ./time ls -ll
total 32
-rwxrwxrwx 1 aneee aneee   728 Dec 19 23:44 Asgn1-README.txt
-rwxrwxrwx 1 aneee aneee  2004 Dec 19 23:48 Asgn1-me17btech11007.c
drwxrwxrwx 1 aneee aneee   512 Dec 19 23:49 img
-rwxrwxrwx 1 aneee aneee 12872 Dec 20 00:28 time


Elapsed Time: 0.011138 second(s)
aneee@LAPTOP-PATF1L50:/mnt/d/Acads/Semester 7/Os/Assignments/A2$
```

Figure 4.1: Sample I/O.

The next command is `python dev/code.py`, which executes the previously created python file, and takes `0.024382` seconds to execute. The output of this python code is `Hello World` as expected.

The next command is `rm -d -r dev`, which deletes the directory named dev and recursivelt deletes all the contents of dev. It takes`0.005224` seconds to execute.

The next command is `ls -ll`, which lists the contests of the current directory with some additional information. It takes`0.011138` seconds to execute. It can now be verified that the previous command deleted the directory dev and it's contents.

Chapter 5

References

1. Operating System Concepts - *Abraham Silberschatz, Peter Baer Galvin, Greg Gagne*

2. Linux Manual - *https://linux.die.net/man/*