Operating Systems–1
CS 3510 Autumn 2018
Programming Assignment 1:

Multi-Process Computation of Execution Time

INSTRUCTOR: DR. SATHYA PERI

Report By:
Vijay Tadikamalla
CS17BTECH11040

भारतीय प्रौद्योगिकी संस्थान हैदराबाद
Indian Institute of Technology Hyderabad

Date: 16/11/2018

Task

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

Approach

1. Fork a child process that will execute the specified command.
2. Record the timestamp of the current time in the child process.
3. Once the child process terminates, the parent will record the current timestamp for the ending time.
4. The difference between the starting and ending times represents the elapsed time to execute the command.

Implementation

Header files

1. **#include<stdio.h>** is a statement which tells the compiler to insert the contents of stdio at that particular place. stdio.h is the header file for standard input and output.
2. **#include <unistd.h>** defines miscellaneous symbolic constants and types, and declares miscellaneous functions.
3. **#include <fcntl.h>** defines the requests and arguments for use by the functions *fcntl*() and *open*().
4. **#include <sys/mman.h>** defines memory management declarations
5. **#include <sys/wait.h>** defines declarations for waiting
6. **#include <sys/time.h>** defines the timeval structure that includes the members like time_t, tv_sec, tv_usec

To perform the above mentioned task, we develop a c program that determines the amount of time necessary to run a command from the command line.

```
int main(int argc, char *argv[])
```

The commands from the command line are passed to our c programs as parameters to the main() function.

```
pid_t pid;
```

Now we create a Process Identifier variable pid. This helps us to identify the type of process by assigning each process a integer.

- If pid==0, then it is a child process
- Else if pid>0, then it is a parent process

```
// The size (in bytes) of shared memory object
const int SIZE = 128;
// name of the shared memory object
const char *name = "/OS1";
// shared memory file descriptor
int fd;
// pointer to shared memory object
char *ptr;
```

To achieve our goal, we need to establish a mode of communication between the child process and the parent process. So, we have to create a shared memory object with properties mentioned in the code snippet.

```
struct timeval current;
```

This struct consists of members like: tv_sec and t_usec

```
fd = shm_open(name,O_CREAT | O_RDWR,0666);
```

shm_open creates and opens a new, or opens an existing, POSIX shared memory object. A POSIX shared memory object is in effect a handle which can be used by unrelated processes to mmap.

Syntax

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

```
ftruncate(fd, SIZE);
```

ftruncate() configures the size of the shared memory object.

```
ptr= (char *)mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
```

mmap creates a new mapping in the virtual address space of the calling process. The starting address for the new mapping is specified in addr. The length argument specifies the length of the mapping (which must be greater than 0).

Syntax

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

```
pid = fork ();
```

fork() creates a new process by duplicating the calling process. The new process is referred to as the child process. The calling process is referred to as the parent process.

So, after fork() is called a child process is created.
We instruct the child process to execute the following commands by adding a if condition of **(pid==0)**

```
// This function is used to record the current timestamp.
gettimeofday(&current,NULL);
sprintf(ptr,"%ld %ld",current.tv_sec,current.tv_usec);
execvp(argv[1], argv+1);
```

We use the **gettimeofday()** function to record the current timestamp. This function is passed a pointer to a struct timeval object, which contains two members: tv_sec and t_usec. These represent the number of elapsed seconds and microseconds since January 1, 1970 (known as the UNIX EPOCH).
We now write the timestamp in the shared memory object using **sprintf().**

We use the **execvp()** system call to replace the process's memory space with a new program.
The **execv(), execvp(),** and **execvpe()** functions provide an array of pointers to null-terminated strings that represent the argument list available to the new program. The first argument, by convention, should point to the filename associated with the file being executed. The array of pointers must be terminated by a null pointer.

Now after executing the above mentioned program the child process will terminate and the parent will start executing.

```
wait(NULL);
```

To obtain the exact time of execution of the program, we make the parent process to wait for the child process to terminate by using the **wait()** command.

```
gettimeofday(&current,NULL);
long int temp1,temp2;
sscanf(ptr,"%ld %ld",&temp1,&temp2);
// Calculating the difference between the starting and the end time
double t=current.tv_sec+current.tv_usec*0.000001;
t=t-temp1-temp2*0.000001;
// Outputs the time elasped in seconds
printf("Elapsed time: %lf\n",t);
```

We again use the gettimeofday() function to obtain the current timestamp.
We read the timestamp written by the child process using the sscanf() function by providing it with the pointer to the shared memory object.
We can now easily calculate the difference between the timestamps and display the time taken by the process to complete.

```
shm_unlink (name);
```

We can now remove the shared memory object by using the above command.

Design Decision

1. Size of memory object
2. Using execvp instead of execlp to take multiple commands from command line

Output analysis

1. Time taken by commands is in the order of 0.01 secs
2. If no command is given then it display more time than actual