SAINT MARY'S UNIVERSITY SINCE 1802 | FACULTY OF SCIENCE

CSCI 3431: Operating System
Fall 2018
Assignment 1-Inter Process Communication

Date out: Sept 16, 2018
Due on: October 6, 2018 (11:45pm)

Instructions:

- Final submission should include the code which is compilable and runnable, a 2 page report describing the approach (including the pseudo-code), results and discussion, any innovative features added, reasons for failure (if any) and References (important).
- Cite in the report if you have adapted your algorithm from any web resources/text books/papers. As the solutions to some of these problems are already available in the web, it's perfectly fine to refer to them and understand the context and their approach. However, I strongly encourage you to write your own code from the scratch. That is the only way you can practice and get more insights into the problem and the OS in general.
- Final submissions should be a zip file (code and report) and to be uploaded to the Moodle before 11:45pm on the due date.
- Name your file following this convention: CSCI3431-Assignment1- <*StudentID*>, where *StudentID* is your Banner ID number starting with A.
- During the evaluation, the students are expected to download the zipped file from the Moodle and show the results on either their laptop or the lab machine. The evaluation will be done in the following two/three recitations or office hours.
- **You may work in groups of two, if you wish to.**

Background:

We discussed in the lectures about process creation and management. The processes as created via a fork () call, share their working space with the creating, or parent, process. Nevertheless, there is indeed a need for communication between real, independent, processes, which do not share working space, such as interactive email messages. In this project, we will experiment with a very important mechanism of inter-process communication tool used in UNIX, called pipe.

A pipe is essentially a buffer that has two file descriptors associated with it, and is created via a pipe(1) system call. One descriptor [1] is used for writing, and the other [0] for reading. When a process creates a pipe, and later makes a fork() call, then the parent and child processes both hold a copy of the file descriptors associated with the pipe and they may communicate with each other, using the pipe.

Example1-Parent child communication:

Below is a simple example involving the pipe mechanism, where the parent process and the child process sends information to each other.

```
#include <stdio.h>
#include <stdlib.h>
char string1[] = "Hello from parent";
char string2[] = "Hi from child";
main()
{
	char buf[1024];
	int i, fds1[2], fds2[2];
	pipe(fds1);
	pipe(fds2);

	if (fork()>0){//Parent process starts
		for (i=0; i<3; i++) {
		//write parent message into fds1
		close(fds1[0]);
		write(fds1[1], string1, sizeof(string1));
		//read child message from fds2
		close(fds2[1]);
		read(fds2[0],buf,sizeof(string2));
		printf("parent read %s\n", buf);
		}
		exit(0);
	}
	else {
		for (i=0; i<3; i++) {
		//write child message into fds2
		close(fds2[0]);
		write(fds2[1], string2, sizeof(string2));
		//read parent message from fds1
		close(fds1[1]);
		read(fds1[0], buf, sizeof(string1));
		printf("child read %s\n", buf);
		}
	}
}
```

```
Output:
parent read Hi from child
child read Hello from parent
child read Hello from parent
parent read Hi from child
parent read Hi from child
child read Hello from parent
```

Example2-Parent to multi-child communication:

```
#include <stdio.h>
#include <stdlib.h>
char string0[] = "Hello, this is the parent process";
char string1[] = "Hi, this is the child 1";
char string2[] = "Hi, this is the child 2";
main(){
        char buf[1024];
        int i, fds0[2], fds1[2], fds2[2];
        pipe(fds0); //pipe used by the parent process
        pipe(fds1); //pipe used by the child 1 process
        pipe(fds2); //pipe used by the child 2 process
        //The first child process is created
        if(fork()==0) {
                close(fds0[1]);
                //read from the parent
                read(fds0[0], buf, sizeof(string0));
                printf("child 1 reads: %s\n", buf);
                //write child message to parent via its pipe
                close(fds1[0]);
                write(fds1[1], string1, sizeof(string1));
                exit(0);
                }
                //The second child process
                else if(fork()==0) {
                sleep(1);
                close(fds0[1]);
                //Get something from the parent process
                read(fds0[0], buf, sizeof(string0));
                printf("child 2 reads: %s\n", buf);
                //write child message into fds2
                close(fds2[0]);
                write(fds2[1], string2, sizeof(string2));
                exit(0);
                }
```

```
        else {//Parent process starts
        //write parent message into fds0
        close(fds0[0]);
        write(fds0[1], string0, sizeof(string0));
        //read child 1 message from its associated pipe
        close(fds1[1]);
        read(fds1[0],buf,sizeof(string1));
        printf("parent reads from Child 1: %s\n", buf);
        //write something into fds0 again to child 2
        close(fds0[0]);
        write(fds0[1], string0, sizeof(string0));
        //read child 2 message from its associated pipe
        close(fds2[1]);
        read(fds2[0],buf,sizeof(string2));
        printf("parent reads from Child 2: %s\n", buf);
        exit(0);
        }
}
```

```
Output:
jiju@os:~$ ./multi_process_pipe
child 1 reads: Hello, this is the parent process
parent reads from Child 1: Hi, this is the child 1
child 2 reads: Hello, this is the parent process
parent reads from Child 2: Hi, this is the child 2
```

Your Task:

In assignment 1, your task is to implement distributed median finding algorithm using pipes. The parent process spawns K (5) identical child processes along with 2*K (10) pipes - two for each *parent-child* pair (one sends messages from *parent*→*child*, the other sends messages from *child*→*parent*). Each child reads an array of 5 integers. The numbers are read from 5 files (one for each child process). The files are named as "*input_1.txt*", "*input_2.txt*", … , "*input_5.txt*". Various stages of the algorithm (reproduced from the assignment[1]) is given below:

Communication Codes:

Parents and Children communicate in the form of codes. Codes are simply integers which are assigned predefined values. In this algorithm we will use five kinds of commands – each represented by a unique code: REQUEST (*#define REQUEST 100*), PIVOT (*#define PIVOT 200*), LARGE (*#define LARGE 300*), SMALL (*#define SMALL 400*) and READY (*#define READY 500*).

These queries are sent along the respective *parent→child* or *child→parent* pipes. i.e., if a parent needs to send the first type of command to a particular child, it simply sends the integer 100 along their *parent→child* pipe. Since the child already knows that this integer corresponds to the command type REQUEST, it then goes on to behave in the required fashion to fulfil that command (the behavior of each command is explained in the algorithm).

Reading Input:

The parent allots ids (1,2,...5) to the children and communicates it via the *parent→child* pipe. The child receives the id and gets the input from the corresponding data file. The numbers are **distinct** and lie **between 1 and 100** (inclusive). Each file contains sorted numbers. Your task is to find the median of the 25 random numbers (*n=25*).

Child Task:

The child waits upon the *parent→child* pipe to receive its id *i*.

- It then reads an array of 5 integers from its corresponding file (*input_i.txt*).
- Upon doing so, it sends the code READY along the *child→parent* pipe.
- It then enters a while loop (broken by a user defined signal - which is sent by the parent to terminate the child process).
- In each iteration it waits on the *parent→child* pipe to respond according the codes it gets.
- If it receives the command REQUEST from parent:
    - If its array is empty, write -1 on the child->parent pipe
    - Else chose a random element from its array and write it to the *child→parent* pipe
- If it receives the command PIVOT from parent:
    - It waits to read another integer (and store it as pivot).
    - It then writes the number of integers greater than pivot on the *child→parent* pipe. If it has an empty array, the number would be 0.
- If it receives the command SMALL from parent:
    - It deletes the elements smaller than the pivot and updates the array.
- If it receives the command LARGE from parent:
    - It deletes the elements larger than the pivot and updates the array

Parent Task:

The parent forks five child processes along with their respective pipes. It goes on to *exec* the child program in each of the children.

- It allots ids 1-5 to each of the children and sends the same along the *parent→child* pipe.
- The parent waits on all the *child→parent* pipes until it receives the code READY from all child processes. This ensures that the algorithm initiates only after all child processes have read the input completely.
- The parent instantiates k = n/2 (we find the kth smallest element in the array – to find median, we require k=n/2).
- The parent selects a random child and queries it for a random element.
- The parent sends the command REQUEST to a random child.
- It then reads the response from the child along the corresponding *child→parent* pipe. If the response is -1, it repeats the same again. If not, it continues.
- The first non-negative value forms our *pivot element.*
- The parent subsequently broadcasts this *pivot element* to all its child processes. To do the same, it first writes the code PIVOT along each of the *parent→child* pipes and subsequently writes the value of the *pivot element*.
- It then reads the response from each child. This represents the number of elements *larger* than the pivot in that child.
- It sums up the total from all its children, call it *m.* if *m* = k, there are n/2 elements larger than *pivot* in the data set. Thus, *pivot* is the median. (Make sure you handle even values correctly).
- If m>k, it sends the command SMALL to all its children which signifies that the children should drop all elements smaller than the pivot element. (Since the median would lie on the right)
- if m<k, it sends the command LARGE to all its children which signifies that the children should drop all elements larger than the pivot element. (Since the median would lie on the left). It also updates k = k - m. (Find out why?)
- It then repeats the REQUEST until it finds the median.
- Once the median is found, the parent reports it and sends a user-defined signal to all its children - and the child processes exit after handling the signal

Please use print commands appropriately to display the status of the algorithm.

Sample output:

Sample Input:

input_0.txt- 1 2 3 4 5

input_1.txt- 6 7 8 9 10

input_2.txt- 11 12 13 14 15

onput_3.txt- 16 17 18 19 20

input_4.txt- 21 22 23 24 25

Sample Output

--- Child 1 sends READY

--- Child 5 sends READY

--- Child 3 sends READY

--- Child 4 sends READY

--- Child 2 sends READY

--- Parent READY

--- Parent sends REQUEST to Child 3

--- Child 3 sends 13 to parent

--- Parent broadcasts pivot 13 to all children

--- Child 1 receives pivot and replies 0

--- Child 1 receives pivot and replies 0

--- Child 1 receives pivot and replies 2

--- Child 1 receives pivot and replies 5

--- Child 1 receives pivot and replies 5

--- Parent: m=0+0+2+5+5=12. 12 = 25/2. Median found!

--- Parent sends kill signals to all children

--- Child 1 terminates

--- Child 2 terminates

--- Child 3 terminates

--- Child 4 terminates

--- Child 5 terminates

Suggestions

- Start early!! If you are not comfortable with the language/concepts, it may take you a bit longer to implement.

- Backup your work frequently. It's possible (and most likely) you go try a new feature and your program crashes!

- Document your work properly

Grading Scheme:

Program compiles, runs	25
Shows the expected output	30
Code quality	15
Report	15
Viva	15

References:

1. http://cse.iitkgp.ac.in/~bivasm/os_notes/Assignment2(b).pdf
2. *Distributed Selection Algorithm to find median*: http://www.quora.com/What-is-thedistributed-algorithm-to-determine-the-median-of-arrays-of-integers-located-on-differentcomputers
3. *Piping in C*: http://www.cs.cf.ac.uk/Dave/C/node23.html
4. *Interprocess communication*: http://www.advancedlinuxprogramming.com/alpfolder/alp-ch05-ipc.pdf