SAINT MARY'S UNIVERSITY SINCE 1802 | FACULTY OF SCIENCE

CSCI 3431: Operating System
Winter 2022
Project 1-Inter Process Communication using Pipes
Date out: January 26, 2022
Due on: Feb 22, 2022 (11:59pm)

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/textbooks/papers. As the solutions to some of these problems are already available in the web, it is 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 MS Class Teams before 11:59 pm on the due date.
- Name your file following this convention: CSCI3431-<*Lastname*>.

- During the evaluation, the students are expected to download the zipped file from the MS Class Teams and show the results on either their laptop or the lab machine using screen sharing. 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**. If you plan to work in a group, you must inform the instructor about the team details latest by Feb 1. Team members should divide the tasks meaningfully. For this project, one possible task division would be the following. *One team member can write the code for parent process and the other can write the code for child process*. During the evaluation, each team member will be asked to explain your role and contribution to the project.

Background:

We discussed about process creation, management, and inter-process communication in Lectures 5 and 6. In many applications, there is clearly a need for the cooperating processes to communicate with each other by sending messages or other control information. In this project, we will experiment with a very important mechanism of inter-process communication used in UNIX, called pipe. A pipe is essentially a buffer created via a pipe(fd) system call where fd[2] is an integer array that holds the file descriptors pointing to the two ends of the pipe. One descriptor fd[1] is used for writing, and the other fd[0] for reading. When a process creates a pipe, and later calls a *fork()* system call, then the parent and child processes both hold a copy of the file descriptors associated with the pipe which when used diligently facilitates communication between with each other, using the pipe.

Before starting your experiment, you are strongly recommended to practice the following two programs that demonstrate the use of pipe() and fork() system calls in IPC.

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";
void 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";
void 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 this project, 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 is given below:

Communication Codes:

Parents and Children communicate using codes. Codes are simply integers which are assigned predefined values. In this algorithm, we will use five 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 semantics of each command is explained in the Child and Parent algorithms below).

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 to 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 choose 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.

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.

Sample Run:

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
```

Grading Scheme:

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

- For different inputs

Viva & Demo	20
Report	15

References:

1. 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
2. Chapter 3, Operating Systems Concept, Silbershatz et al.
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