Ex7 - Pipes, Processes and Forking

COP4600

Slides by: Ernesto A. Mujica

Basic Pipes

- Pipes are a mechanism for interprocess communication, usually by chaining together their output streams.
- A basic pipe can be utilized in a unix terminal by using the “ | “ character. (Shift + Backslash)
- Ex: ls | grep “phrase” will search the standard output of ls for the text “phrase” using grep.

Basic Pipes (cont.)

- You will be provided a output binary part1.o that prints a random amount of successful operations:

```
reptilian@localhost:~/Ex6$ ./part1.o
Operation successful.
Operation successful.
Operation successful.
Operation successful.
Operation successful.
Operation successful.
Operation successful.
Operation successful.
Operation successful.
Operation successful.
Operation successful.
Operation failed.
reptilian@localhost:~/Ex6$
```

- For part 1, you will be writing a C++ program that takes in standard input (using cin or getline or another similar structure) and prints out on which line the operation failed. It will be run as follows:

```
reptilian@localhost:~/ex7$ ./part1.o | ./a.out
Program failed on operation 8
reptilian@localhost:~/ex7$ ./part1.o | ./a.out
Program failed on operation 20
```

Named Pipes

- Named pipes are files that can be thought of as temporary “queues” to hold information piped into them. You will be creating a named pipe in part 2 using the following command:

  mkfifo ex7_Pipe

- Since named pipes are files, you can pipe into them using the redirection operator “>”:
  ./part1.o > ./ex7_Pipe
- Your part 2 program (named lastname_part2.cpp) will read from ex7_Pipe using file reading functions (fstream, POSIX functions, etc.) and again, print where the program failed in the following format:

```
reptilian@localhost:~/ex7$ ./a.out
Program failed on operation 5
reptilian@localhost:~/ex7$ ./a.out
Program failed on operation 61
```

Processes & Process Forking

- A process is a program in execution. Your source code is just a program, and your source code once compiled into an executable binary is still a program, just in machine code. When your binary is ran, then it is categorized as a process.
- Processes can fork to create child processes. At the point at which fork() is called, both the parent and child processes will run all lines of code beneath the fork.
- Processes are forked using the fork() system call. It returns an integer that indicates the identity of the current process. The value will be 0 if it is a child process.

```
 GNU nano 4.8                    example.cpp
#include <iostream>
#include <sys/types.h>
#include <unistd.h>

int main() {
        int pid = fork();
        std::cout << "Hello, my process ID is: " << pid << "\n";
        return 0;
}
```

```
ern@DesktopPC:Ex7$ ./forktest
Hello, my process ID is: 107
Hello, my process ID is: 0
ern@DesktopPC:Ex7$
```

Processes & Process Forking (cont.)

- There may be instances where you specifically want child processes to only run a specific section of code unique to the child process.
- The simplest way to do this is to differentiate the child from the parent by its returned ID, and exit when the child is finished with a return statement.

```
 GNU nano 4.8                    example.cpp
#include <iostream>
#include <sys/types.h>
#include <unistd.h>

int main() {
        int pid = fork();

        if(pid == 0){
                std::cout << "Hello, I am the child process!\n";
                return 0;
        }

        std::cout << "Hello, I am the parent process!\n";
        return 0;
}
```

```
ern@DesktopPC:Ex7$ g++ -o ./forktest example.cpp
ern@DesktopPC:Ex7$ ./forktest
Hello, I am the parent process!
Hello, I am the child process!
```

Processes & Process Forking (cont.)

- Note that if you are forking multiple children (like you need to in part 3), you must make sure to guard the fork calls correctly like shown before.
- For example, let's say you wanted to spawn 3 child processes, for a total of 4 running processes total (3 children + 1 parent).

```
int pid1 = fork();
int pid2 = fork();
int pid3 = fork();
```

- The code above will result in 8 total processes, not 4!

Interprocess Communication with Pipes

- An important note about forked processes is that they **do not** share the same memory space. All memory from the parent process is copied and mapped elsewhere for the child process to use.
- Meaning, the child process cannot modify data from the parent process directly. As shown below:

```
  GNU nano 4.8                    example.cpp
#include <iostream>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>

int main() {
        int pid = fork();
        int testInt = 10;
        if(pid == 0){
                std::cout << "Hello, I am the child process!\n";
                testInt*=2;
                std::cout << "New testInt: " << testInt << "\n";
                return 0;
        }
        sleep(1); //Waiting for child process to complete...
        std::cout << "Hello, I am the parent process!\n";
        std::cout << "Printing testInt: " << testInt << "\n";
        return 0;
}
```

```
ern@DesktopPC:Ex7$ ./forktest
Hello, I am the child process!
New testInt: 20
Hello, I am the parent process!
Printing testInt: 10
ern@DesktopPC:Ex7$
```

Interprocess Communication with Pipes (cont.)

- So how do we share information between parent and child processes (or even child to child processes)?
- Pipes! Specifically, the pipe() syscall. The pipe system call takes in a 2 integer array as a parameter, and turns them both into file descriptors, meaning they can be used with the read() and write() POSIX calls to pass data around.
- Index 0 of the array is the read end, index 1 is the write end. Reading from pipes will **block** the process performing the reading until there is data written to the pipe.
- It important to note that pipes are **unidirectional.** This means that to pass data back and forth between two processes requires **two** pipes. One from process 1 to process 2, and another from process 2 to process 1.
- Good practice is to call close() on any pipes or pipe ends you are not using (or are done using). This is to prevent accidentally reading/writing from the wrong end of a pipe, leading to very difficult to debug code.

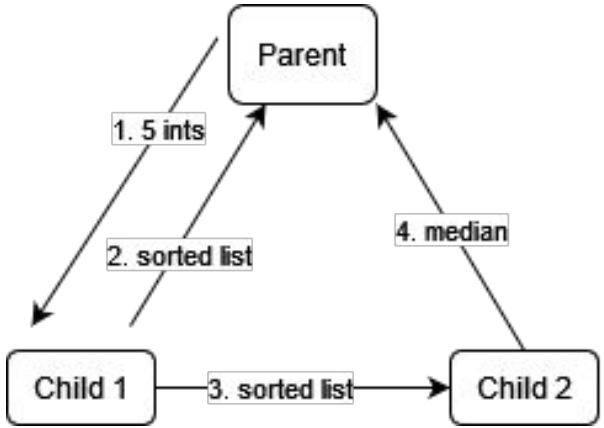
Interprocess Communication with Pipes (cont.)

```
  GNU nano 4.8                                  example.cpp
#include <iostream>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>

int main() {

        int childToParentPipe[2];
        pipe(childToParentPipe);

        int pid = fork();
        if(pid == 0){ //Child Code
                close(childToParentPipe[0]); //Closing read end
                std::cout << "Hello, I am the child process!\n";
                int testInt = 50;
                std::cout << "Sending data to parent...\n";
                write(childToParentPipe[1], &testInt, sizeof(int));
                return 0;
        }

        //Parent Code
        close(childToParentPipe[1]); //Closing write end
        int receiveInt = 0;
        read(childToParentPipe[0], &receiveInt, sizeof(int));
        std::cout << "Hello, I am the parent process!\n";
        std::cout << "Printing int from child: " << receiveInt << "\n";
        return 0;
}
```

```
ern@DesktopPC:Ex7$ ./forktest
Hello, I am the child process!
Sending data to parent...
Hello, I am the parent process!
Printing int from child: 50
```

Interprocess Communication with Pipes (cont.)

- In part 3 of this exercise you will be creating 4 pipes between the parent process, the first child process, and the second child process.
- 1. One from the parent to child 1, to send the 5 ints taken as command line args.
- 2. One from child 1 to parent, to send the sorted list of the 5 ints
- 3. One from child 1 to child 2, to send the sorted list of the 5 ints
- 4. One from child 2 to parent, to send the median of the list.



Ex7 - Pipes

- Your program for part 3 will be name “lastname_part3.cpp”.
- It will be compiled with the following command:
  g++ -o part3.o lastname_part3.cpp
- Your program should work with any 5 integers we provide. We will only test it with 5 integers, and it will be ran as follows (with different numbers):
  ./program.out 42 15 8 16 23
- Your output should look like:

```
Sorted list of ints: 8  15  16  23  42
Median: 16
```

- There should be 1 space after the colon, and 2 spaces between each number.

- (Go over pdf.)