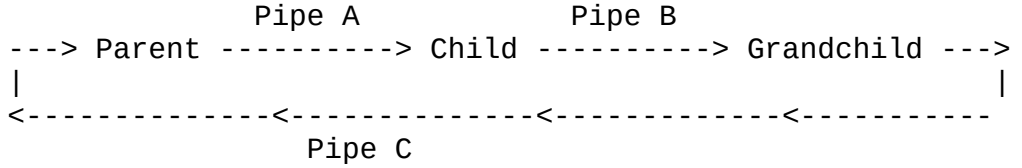
Assignment 2 -- Process Communication (100 points)

This assignment involves using the LINUX system function pipe() as well as fork().

We will have three processes which communicate with each other using pipes. The three processes will be parent, child and grandchild. Their communication will run in a ring like this:



Each process will read a number (in character format) from a pipe, convert it to an integer, do a bit of arithmetic, convert the result to character format (a string) and write it into the other pipe for the next process to read.

Write a program on the hopper system using C or C++ to implement this.

The following description assumes that the strings involved are C-style strings (character arrays ending in \0). It should also be possible to do this using the C++ string class. If you wish to do so, see the notes below.

Main program

1. Declare three pipe variables (which I will call Pipe A, Pipe B and Pipe C). Each is an array of two integers.

2. Call the pipe() function once for each pipe variable.

3. If any use of pipe() fails (returning -1), print an error message (such as "pipe #1 error") and exit with a status of -5.

4. Call the fork() function twice to create the three processes: once in the parent and then again in the child. Do not create more than 3 processes.

   If fork() fails (returning -1), print an error message (such as "fork #2 error") and exit with a status of -5.

5. At this point we have three processes, parent, child and grandchild. Each of these should call a function to do the rest of its work. We can call these PWork() for the parent, CWork() for the child and GWork() for the grandchild. Before calling its function, each process should close the appropriate ends of the pipes it will use. (The parent reads from Pipe C and writes to Pipe A, so it should close the write end of Pipe C and the read end of Pipe A, and likewise for the other two.)

   When the functions end, the processes should close the remaining ends of the pipes and then exit with a status of 0. (The child should use wait() to wait until the grandchild terminates, and the parent should do the same for the child.)

6. Return a value of 0. (This is not really needed as we are using exit(), but it's a good habit.)

What do the functions do?

1. Declare two char arrays, Buffer and Value, each of size at least 10. Initialize Value to "1". Also declare an integer M with the initial value 1.

   In PWork() (but not in the other two), start by writing Value to the appropriate pipe. Print a message announcing this is the parent and providing the value of Value. (This is the principal difference between PWork() and the others: it starts the processing.)

2. In a loop, while M is no more than 99999999 (that is eight 9s), do the following:

   (a) Read one byte at a time from the appropriate pipe and store them in Value, building a string. Stop when you find '\0'.

   (b) Make sure Value ends properly as a string with '\0'.

   (c) Convert the string Value to an integer value in M.

   (d) Compute M = 4 * M + 3.

   (e) Convert M to a string in Buffer.

   (f) Write Buffer to the appropriate pipe. Print a message identifying who is printing (parent, child or grandchild) and providing the value of Value.

Option to use the C++ string class

If you prefer to use the C++ string class, you will need to make a few changes to the functions:

--- The variables Buffer and Value should be declared as strings.

--- When you write a string to the pipe, the other process must be able to identify the end of the transmission. Therefore, use some character such as '@' as a delimiter.

--- When a process reads a string from the pipe, it should read one byte at a time until it finds the delimiter. Do not print the delimiter or use it in converting a string to an integer.

Notes

You should decide for yourself what arguments you need to pass to these functions.

As the three functions are so similar, you may be able to improve the design by making the overlapping part into a separate function.

You may want to read up on read() and write() and how to convert a string to an integer.

Please print your output without buffering. As in Assignment 1, the simplest way to do this is to use stderr instead of stdout.

Your program should use reasonable variable names and should be appropriately indented and well documented. You can find style guidelines on the web site of CSCI 241 courses.

You should have a makefile. The name of the executable file should be "Assign2".

When you are done, you need to submit your work on Blackboard. As in Assignment 1, you should create a tar file containing the two files involved: the program file and the makefile. To do this, you need the "tar" utility.

Do the following (replacing "Znumber" with your own Z-ID):

(a) Create a subdirectory named Znumber_A2_dir.
(b) Copy the three files into it.
(c) In the parent directory of Znumber_A2_dir, use this command:
    tar -cvf Znumber_A2.tar Znumber_A2_dir

Use an FTP program to retrieve the tar file and then submit it on Blackboard. The TA will move it to hopper, extract the files and run your makefile, as in:

```
tar -xvf Znumber_A2.tar
cd Znumber_A2_dir
make
Assign2
```

If your makefile does not run or your program does not compile and run, you will receive no credit.

Sample Output

This is an example of the output. If you run your program several time, you may find you have some of the lines in a different order.

```
Parent:       Value =  1
Child:        Value =  7
Grandchild:   Value =  31
Parent:       Value =  127
Child:        Value =  511
Grandchild:   Value =  2047
Parent:       Value =  8191
Child:        Value =  32767
Grandchild:   Value =  131071
Parent:       Value =  524287
Child:        Value =  2097151
Grandchild:   Value =  8388607
Parent:       Value =  33554431
Child:        Value =  134217727
Grandchild:   Value =  536870911
Parent:       Value =  2147483647
```