Ceng 334 – Introduction to Operating Systems – Spring 2014

Programming Assignment 1

Pipe Graph

due date: 6th of April

1 Introduction

In this homework you will implement a pipe graph which is similar to pipeline of Unix-like systems. A pipeline is a set of processes chained to each other sequentially by their standard streams such that *stdout* (standard output stream) of one process feeds *stdin* (standard input stream) of the next process. It is called pipeline because streams of the processes connect to each other via pipes. A pipe is a unidirectional data channel which can be used for interprocess communication. A pipeline can be created using '|' delimeter on a Unix Shell such that:

```
     <command1> | <command2> | ... | <commandN>
```

where each <command> contains an executable name with command line arguments. When a pipeline created all processes run in parallel (at the same time). Unix Shell does not accept new commands until all processes in the pipeline terminate unless pipeline is executed in background.

Example:

```
$ ps aux | grep isikligil | cat -n
$ echo we got hope | wc
```

The pipe graph you will implement differs from regular pipelines in terms of connection between processes. In pipe graph, processes are not chained sequentially but in a graph structure. Graph structure will be provided from stdin and your program will execute given commands after establishing necessary connections between them. You will be familiar to necessary Unix system calls at the end of this assignment.

2 Task

Your task in the scope of this assignment is developing a program which behaves like Unix Shell to execute pipe graph. In other words, your program will read commands and graph structure from stdin and execute related executables after establishing communication network between them using pipes. You will be given such an input:

Input:

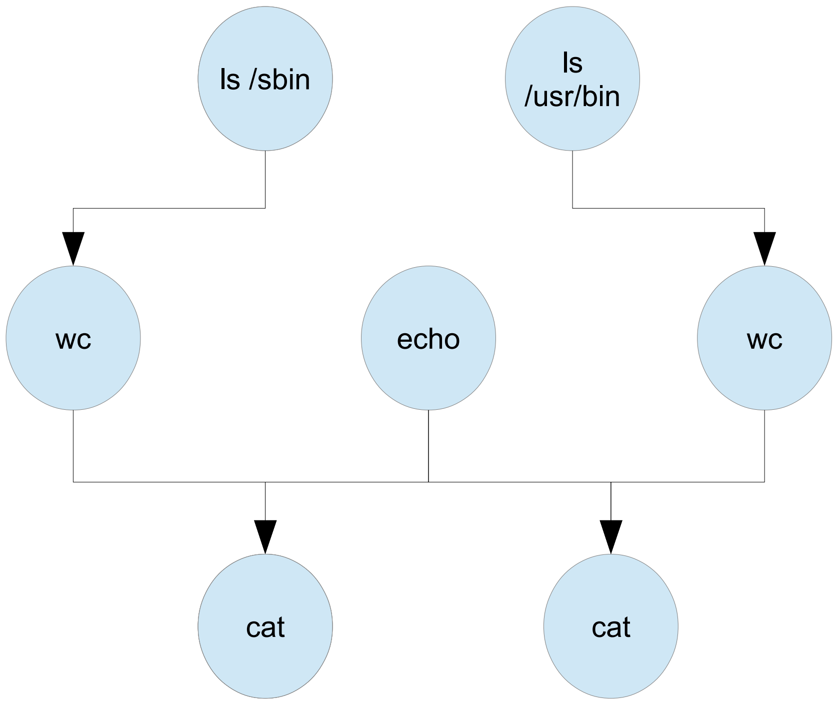
```
n
<program_0> [arguments] : [successors]
<program_1> [arguments] : [successors]
...
<program_(n-1)> [arguments] : [successors]
```

where *n* is the number of processes to be executed. *<program_i>* is the name of the executables. Tokens given between [ ] are optional. *[arguments]* are the command line arguments to be passed to the executables. Arguments will separated by space (' ') character. *[successors]* show the processes that stdout of the current process will be connected to. Successors will be given as a set of integers separated by space character. Each integer points a process with its process number. Process numbers start from 0 and goes to (n-1) where n process exists. Mind that there will be a space character before and after delimiter (':').

You can find a sample input and the corresponding pipe graph below:

Example:

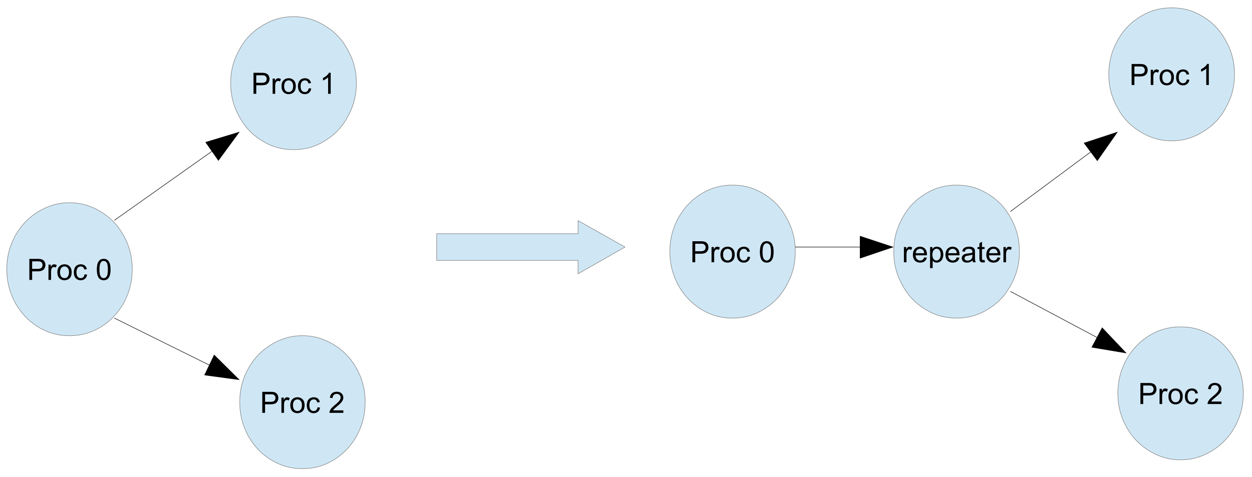
```
7
ls /sbin : 3
ls /usr/bin : 4
echo -n number_of_words : 5 6
wc -w : 5
wc -w : 6
cat :
cat :
```



For this task you should:

- read stdin for commands and pipe graph structure.
- create child process for each command *(hint: fork).*
- create pipes between child processes and redirect standard stream *(hint: pipe, dup2).*
- create a child process and replace it's process image with the given command for each command *(hint: 'exec'system call family)*. Mind that when you use *exec* system call family to replace a process into another, file descriptors of old process remain open. So, all changes you did on standard streams will be preserved in the new process.
- wait for all child processes to terminate before terminating the parent process *(hint: wait)*.

As you can realize some processes in the pipe graph may have more than one successors or predecessors. You do not need to handle the more than one predecessor case, because a pipe can be written by multiple processes at the same time. But, when a process has more than one successors you should handle the situation to be able to split the output. Best way to do that is to create a repeater process which reads output and then send it to related processes. If you will use this strategy you should establish connections between processes in that way.



Some executables like *'cat'* or *'grep'* read from stdin until they see EOF. An input pipe is sent a EOF character when file descriptor of the write hand of the pipe is not open in any process. So, you should close unnecessary file descriptors in parent and all child processes.

3 Specifications

- Executable names may be given as full path or not. If Unix Shell executes the command properly, your program should also do.
- All processes should be run in parallel. Therefore, order of the outputs will be nondeterministic. In other words, every execution of the same pipe graph may generate different outputs. Output order will be unnecessary during evaluation.
- Your program should terminate after all child processes terminate. It should not leave any zombie processes *(hint: wait system call)*.
- Any process can be successor of any process except itself. For example, 0th process can be successor of the 3rd process.
- Processes which do not have any successor will use stdout. Processes which are not successor of any process will use stdin.
- You can use C or C++ for the implementation.
- All inputs will be proper. Non-existing executables will not be given. Provided pipe graph will not cause a deadlock. Circular connections will not exist.
- Evaluation will be done using black box technique. So, your programs must not print any unnecessary character and outputs of the processes must not be modified.
- You are responsible to follow course page on newsgroup (COW) for any clarification and update. Please ask your questions on newsgroup instead of sending emails.

- Evaluation will be done on inek machines. So, test your implementation on inek machines before submitting.
- Submission will be done through COW.
- Everything you submit must be your own work. Any work used from third party sources will be counted as cheating.

4 Submission

You will submit a tar archive file named hw1.tar.gz which includes your source code and a makefile. Makefile should create an executable named *'hw1'*. Your impelementations will be compiled using the command chain:

```
$ tar xf hw1.tar.gz
$ make
```

Mind that makefile should be on the root of your archive file. Executable ('hw1') should be created at the same directory with the makefile (ie. root directory).

Kolay gelsin.