Middle East Technical University | Department of Computer Engineering

CENG 334

Introduction to Operating Systems

Spring 2015-2016

Homework 1 - Pipe Graph Shell

Due date: 27 03 2016, Sunday, 23:55

1 Objective

In this homework you are going to implement a pipe graph shell which is similar to how shells that support pipeline work in Unix-like systems. A pipeline is a set of processes chained to each other sequentially by their standard streams such that standard output (stdout) of one process feeds standard input (stdin) 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 is 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. To be able to provide a graph like structure every pipe is given a unique id; inputs and outputs of the processes are redirected with special symbols defined in the next chapter.
Keywords: *unix, pipe, graph, shell*

2 Program

Your task is to implement a program which behaves like a unix shell that supports only foreground processes and pipes. More specifically, your program will read commands from the standard input, create the necessary pipe graph and execute the commands after establishing communications between them using pipes if all the pipes are connected. You must store the commands, parameters and pipe ids until the pipe graph is ready to execute. When the pipes inside the graph is connected, you should execute it immediately and wait the termination of every processes before accepting new commands. You can assume that the testcases will not include multiple independent pipe graphs, there will be only one at

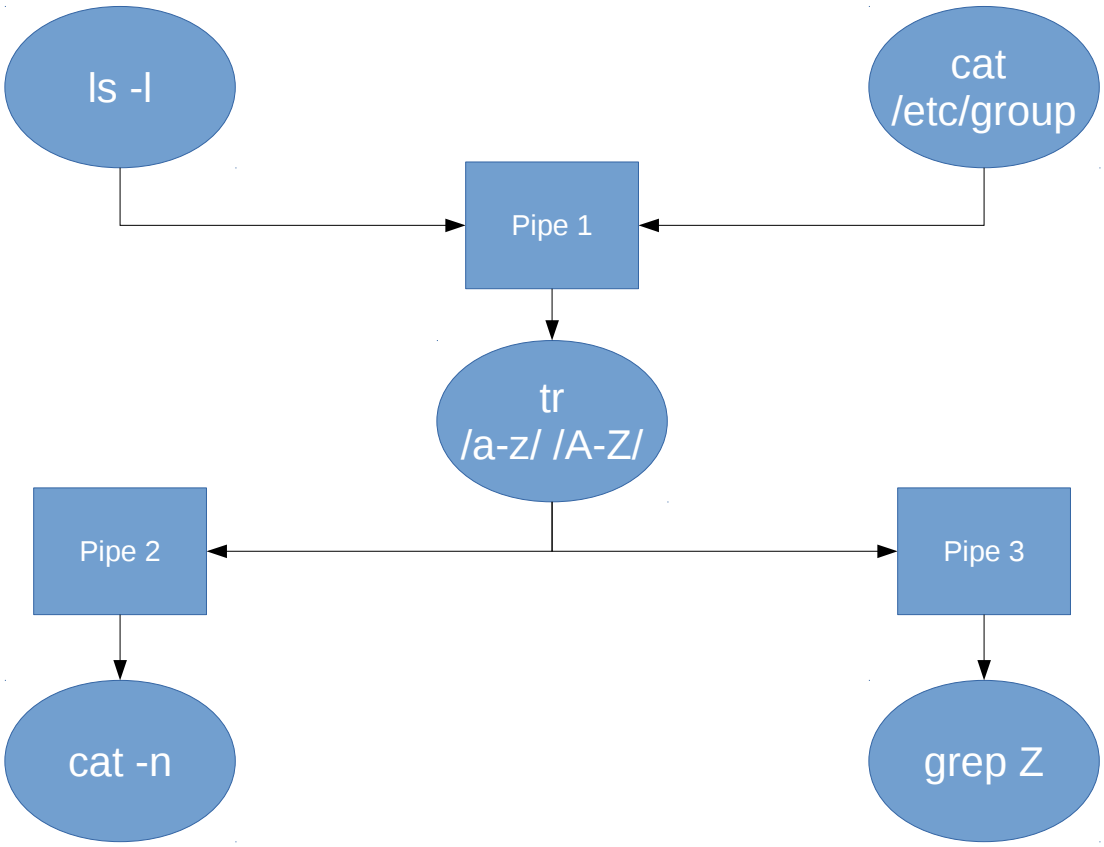
any given time. At every line only single instruction is given. Input and output pipes are given using two special symbols followed by the pipe ids. For the input pipe '<|' symbol is used and it is followed by a single positive integer since it is possible for multiple processes to write to a single pipe and input of a process must be single pipe. For the output pipe '>|' symbol is used and it is followed by multiple pipe ids separated by spaces.
Input:

```
<command> [<arguments>] [<| <pipe_id>] [>| <pipe_ids>]
```

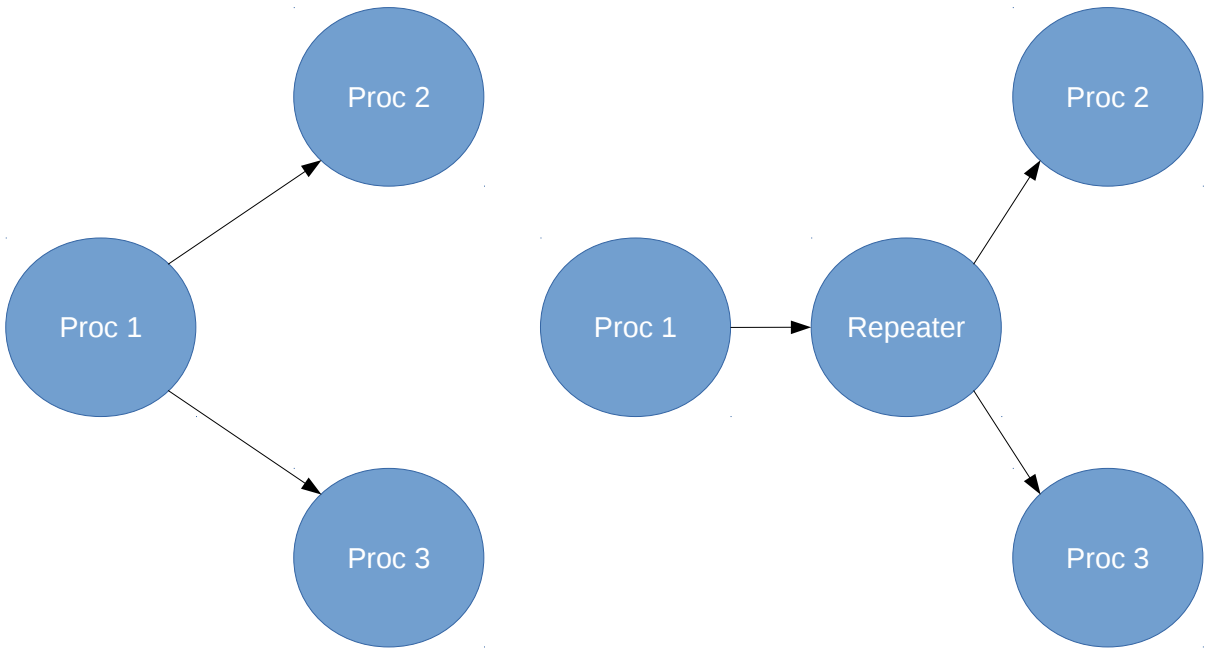
The symbol [ ] indicate that the inside is optional. If there is no defined input or output pipes and if it is necessary, the processes should use standard streams. For the simplicity you can assume there will be no testcases where one of the processes to be executed require standard input for its input. For a graph to be executable, every output pipe id must be matched with a corresponding input pipe id. If the input pipe is defined first, then even single output pipe is enough for a match.
Example:

```
ls -l >| 1
cat /etc/group >| 1
tr /a-z/ /A-Z/ <| 1 >| 2 3
cat -n <| 2
grep Z <| 3
```

As you can see from this example all of the three pipes are matched. Pipe 1 is written from "ls -l" and "cat /etc/group" commands and read from "tr /a-z/ /A-Z/" command. Pipe 2 is written by "tr /a-z/ /A-Z/" command and read from "cat -n" command. Lastly, Pipe 3 is written by "tr /a-z/ /A-Z/" command and read from "grep Z" command. For a visual inspection:



Clearly, you can realize that some processes in the pipe graph may have more than one successors or predecessors. As I have mentioned before you do not need to handle the case where there are more than one predecessor, because a pipe can be written by multiple processes at the same time. However, when a process has more than one successors you should handle the situation to be able to deliver the output to every process. Best way to do that is to create a repeater process which reads output of the process and then send it to related processes with additional pipes. It would look like this (Even though they are still needed, pipes are not drawn in this figure.):



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.
Related UNIX system calls: *dup2, pipe, exec system call family, fork*

3 Specifications

- The program must terminate when the user gives the *quit* command
- Commands are given in a single line. The parameters, pipe symbols and pipe ids are separated by a single space character.
- Executable names may be given as full path or not. Your program must be able to execute both.
- All processes should 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 not important during evaluation.

- Your program should reap all its child processes. It should not leave any zombie processes (hint: wait system call).
- When a graph is executable, your program must wait for the termination of all the child processes before accepting new commands.
- Processes which do not have any successor will use stdout. Processes which are not successor of any process will use stdin.
- 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.

4 Regulations

- **Programming Language:** You must code your program in C or C++. Your submission will be compiled with `gcc` or `g++` on department lab machines. You are expected make sure your code compiles successfully.
- **Late Submission:** Late submission is allowed and penalty of $5 * day * day$ is applied for every late day.
- **Cheating:** Everything you submit must be your own work. Any work used from third party sources will be counted as cheating. In case of cheating, the university regulations will be applied.
- **Newsgroup:** You must follow the newsgroup (news.ceng.metu.edu.tr) for discussions and possible updates on a daily basis.
- **Grading:** This homework will be graded out of 100. It will make up 12% of your total grade.

5 Submission

Submission will be done via COW. Create a tar.gz file named `hw1.tar.gz` that contains all your source code files along with a makefile. The tar file should not contain any directories! The make should create an executable called hw1. Your code should be able to be executed using the following command sequence.

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