Middle East Technical University | Department of Computer Engineering

CENG 334

Introduction to Operating Systems

Spring 2021-2022

Homework 1 - Bundle Shell

Due date: 17 04 2022, Sunday, 23:59

1 Overview

In this homework you are going to implement a special shell, called *bshell*, which supports the concurrent execution of a bundle of executable together with pipeline and file redirection support.
Keywords: *unix, shell*

2 Background

A pipeline is a sequence of processes chained to each other such that standard output (stdout) of each process feeds stdin of the next process via pipes, a unidirectional inter-process communication support.

A pipeline can be created using '|' delimiter 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 concurrently and the shell blocks until all processes in the pipeline terminate.

The following is a sample command that a regular shell supports:

```
ps aux | grep python | wc -l
```

where the list of all user processes containing the python are counted.

File redirection works similarly to pipelines by redirecting the standard input and/or output of a process to a file. A redirection for standard input and output can be created using '<' and '>' delimiters respectively as:

```
(command) < (input_file_name) > (output_file_name)
```

where the input or output is redirected to the respective files.

The redirection can also be combined with pipelining such as:

```
echo "Hello" > hello.txt
cat < config.txt
cat < manifest.txt | grep "Ship" > shipmanifest.txt
```

3 Process Bundle Shell

The *bshell* differs from a regular one by allowing its users to create process bundles that contain one or more commands. After creation, when a bundle is executed, all of its processes in the bundle execute concurrently. The bshell should support the creation and execution of process bundles instead of single processes in a regular shell and support pipeline and file redirection operations on the bundles.

However, the process bundles output can be redirected to files or other process bundles standard input like how normal processes' I/O is redirected. When process bundles' input is redirected from an another process bundle, you may need to replicate the input so that every process in the bundle receive all of the input. This input can come from another process bundle via pipeline or from an input file. In case of an input file, you need to find a way to make the file fully available to every process in the bundle. Output redirection on the other hand is straightforward since multiple outputs can be redirected to a single pipe. The input and output a process in a bundle can not be redirected individually (in a way different than the other processes in the bundle).

3.1 Execution

If there is no redirection or pipelining, you need to fork the necessary number of child processes (see `fork()`) and execute the command(s) (see `exec` family of functions).

Else, you need to consider three different scenarios. For the single process scenario:

1. **Output redirection to a file:** Fork the process and redirect its output to that file(see `dup2()`). Afterwards execute the command as depicted in Figure 1.

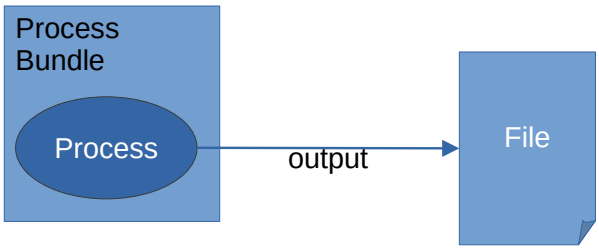


Figure 1: Single Process Output Redirection

2. **Input redirection from a file:** Fork the process and redirect its input to that file. Afterwards execute the command as depicted in Figure 2.

3. **Pipeline:** Create set of pipes for each connection in the pipeline (see `pipe()`). Note that if there are N processes in the pipeline, there should be $N-1$ sets of pipes. After creating the pipes, fork the processes in the pipeline and redirect their output and input to the pipes on each connection. Finally they all should execute the commands. A basic schema of a pipeline between two processors is given in Figure 3.

For process bundles that have more than one process, the execution is more complicated. There are multiple ways of doing it but the general idea is given below:

1. **Output redirection to a file:** Open the file in the append mode. Fork the processes and redirect their output to that file. Afterwards execute the commands. A basic schema can be seen in Figure 4.

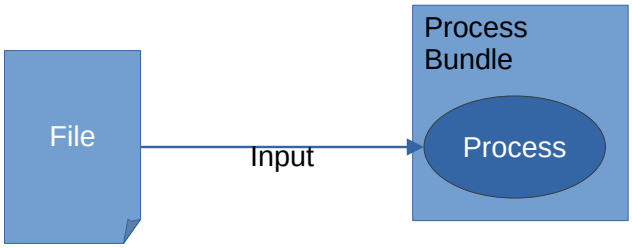


Figure 2: Single Process Input Redirection

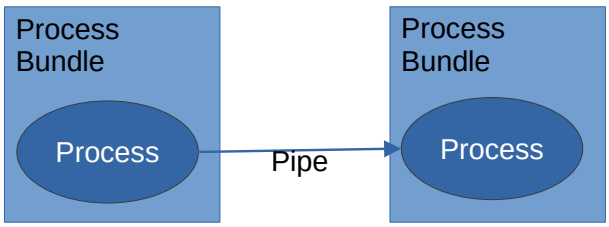


Figure 3: Single Process Pipe

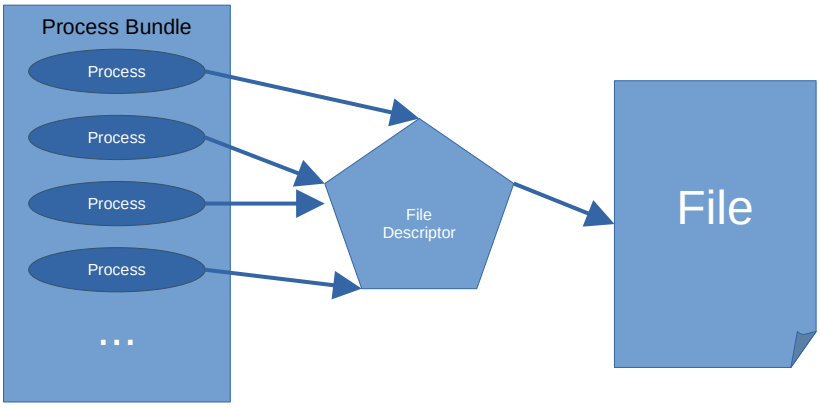


Figure 4: Multi Process Output Redirection

2. **Input redirection from a file:** Fork all of the processes inside the bundle. Redirect the input of the processes to that file. Afterwards, execute the commands as depicted in Figure 5.

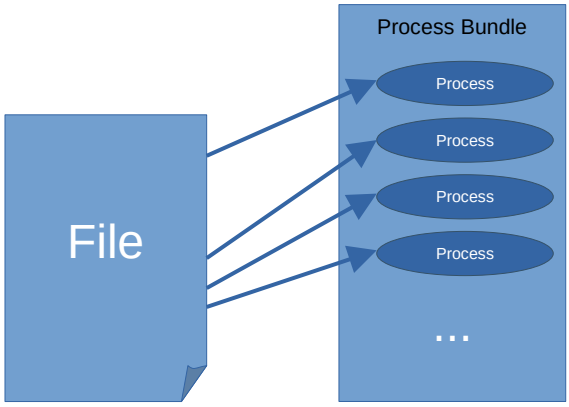


Figure 5: Multi Process Input Redirection

3. **Pipeline:** A repeater process is necessary for this case. A repeater process replicates the inputs it receives to multiple channels (or pipes). Create necessary pipes required for the communication between predecessor bundle's processes and the repeater. Fork the predecessor bundle's processes and redirect their output. Similarly, create the necessary pipes for the communication between the repeater and successor bundle's processes. Fork the successor bundle's processes and redirect their inputs. Finally all predecessor and successor bundles' processes should execute the commands. These actions should be repeated for every connection in the pipeline. Please note that the repeaters should work as long as there are active predecessor processes connected to them. A basic schema of a pipeline between two processor bundles is given in Figure 6.

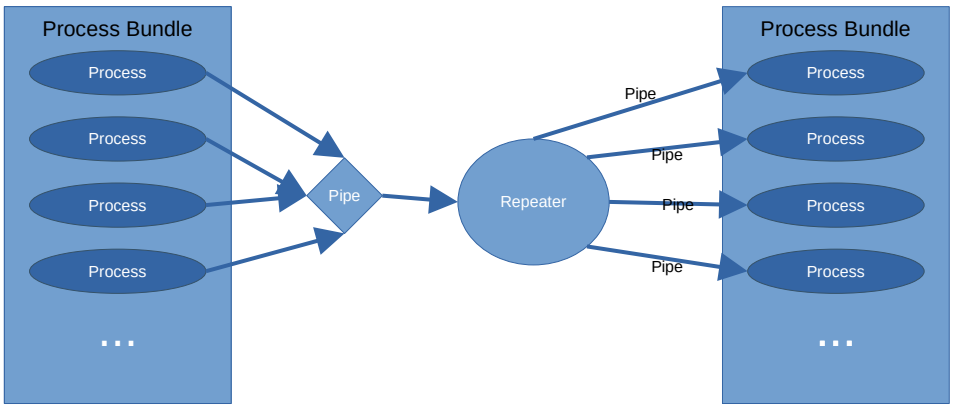


Figure 6: Multi-Process Pipe

Before accepting new input, the bshell should wait until all of the processes are finished and reap them (see `wait` family of functions). There should not be any zombie processes.
Note: Inside a pipeline, first bundle's input and the last bundle's output can be redirected to a file. In that case combine the redirection directions with a pipeline. Similarly single process bundles and multiple process bundles can be mixed. Use the necessary parts from each explanation.

4 The Bundle Shell

Your task is to implement the *bshell* which supports only process bundles, pipes and file redirection. It should not allow regular execution of processes. More specifically, the bshell should read one of the three commands from the standard input, create the necessary processes inside the bundles using regular commands and execute the process bundles after receiving the execute order. The bshell must store the commands and parameters of the processes that is inside each bundle until execution. There will be no redirection inside the process bundles and each process will be given in a single line. Each process bundle will have a name and should be executable using that name. Once a process bundle is executed, their details can be deleted. The bshell should wait the termination of every process inside the bundle(s) it is executing before accepting new commands. It should be possible to reuse bundle names after execution, although your test cases will not include such scenarios. To create a process bundle, `pbc` command will be used in the following manner:

```
pbc (process bundle name)
```

After receiving this input, the bshell should accept regular commands together with their arguments at every new line until it sees the stopping command `pbs`. There should be at least one command inside each process bundle. The format:

```
(command) [(arguments)]
[(command) [(arguments)]]
[(command) [(arguments)]]
...
pbs
```

The symbol [ ] indicate that the inside is optional. The stored bundle is now executable. It should be able to be executed directly using its name:

```
(process bundle name)
```

After receiving the name of the bundle, the bshell should execute every command inside the bundle at the same time and wait for their completion before accepting new commands. Finally the command `quit` should terminate the program. It should quit even if there are bundles that are not yet executed.
Example:

```
pbc pb1
ls -l
cat /etc/group
pbs
pb1
```

The bshell should use the same shell symbols for pipeline and redirection as follows:

```
(process bundle name) < (input file name)
```

Example:

```
pbc pb1
cat
pbs
pb1 < /etc/group
```

The format for output redirection as follows:

```
(process bundle name) > (output file name)
```

Example:

```
pbc pb1
ls -l
cat /etc/group
pbs
pb1 > pb1_output.txt
```

As you can see from the example, there can be multiple processes writing to the same file. This may sometimes cause one process to override another. To prevent this from happening, the file should be opened with an append flag to ensure that write operations are atomic and do not override one another. Obviously, the outputs of the processes may be mixed and there may be previous contents of the file. Mixing of the output is not a problem and you will only be tested with empty or nonexistent files during grading.
Of course these redirections can be used together:

```
pbc pb1
cat
pbs
pb1 < /etc/group > pb1_output.txt
```

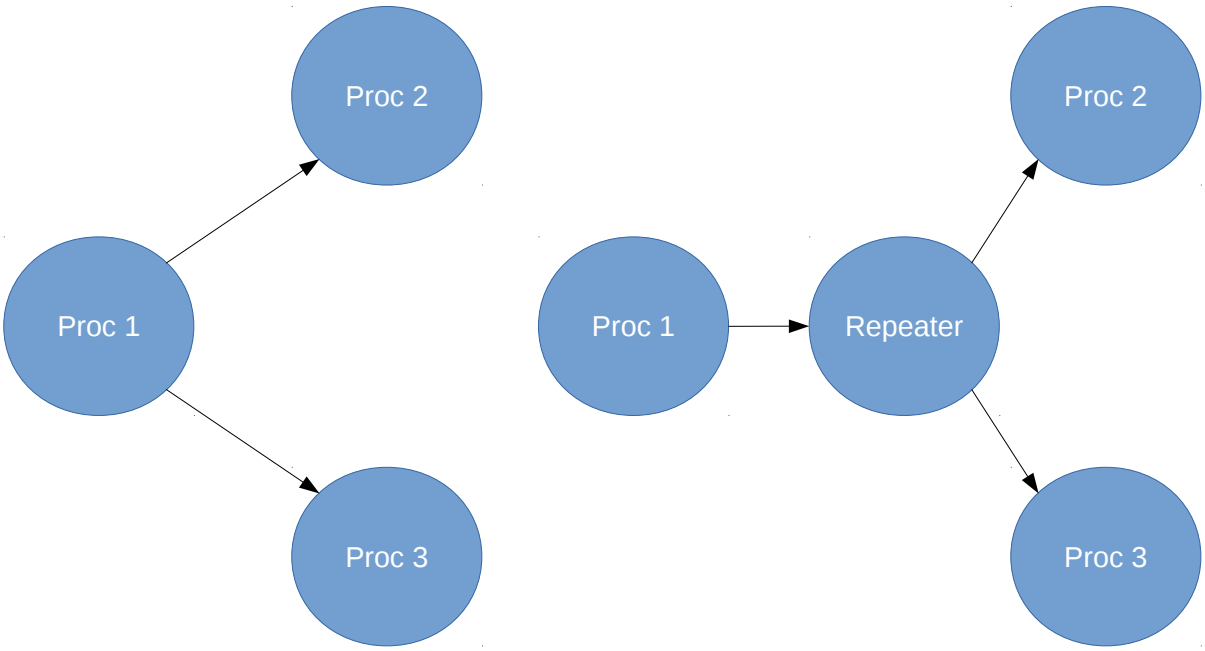
Finally, the bshell should support pipeline operations using the format (Please consider the next two lines to be a single line. It is written in two lines to fit the page):

```
(process bundle name) [< (input file name)] | [(process bundle name) |]
(process bundle name) [> (output file name)]
```

As you can see from the format, there can be many process bundles chained together. Moreover, the input of the first process bundle and the output of the last process bundle can be redirected to a file. When a pipeline is executed, all of the process bundles should be executed at the same time.
Example:

```
pbc pb1
ls -l
pbs
pbc pb2
tr /a-z/ /A-Z/
pbs
pb1 | pb2
```

Clearly, you can realize that some processes inside a bundle may have more than one successors or predecessors when they are executed in a pipeline. 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 and wait system call family, fork*

5 Input

The bshell should behave like a UNIX shell application. Therefore, it needs to wait for input continuously until it receives the `quit` command. There can be multiple number of bundle declarations and executions. These can be repeated as many times as the user wants. If we assume for every repeat, there is N number of declarations and M number of executions, in general, inputs can be written in this format:

```
<process bundle declaration_1>
<process bundle declaration_2>
...
<process bundle declaration_N>
<process bundle execution_1>
wait results
<process bundle execution_2>
wait results
...
<process bundle execution_M>
wait results
```

```
...
<<repeated as many times as wanted>>
...
quit
```

The N and M values can change for every repeat. Each process bundle declaration is formatted as:

```
pbc (process bundle name)
(command) [(arguments)]
[(command) [(arguments)]]
[(command) [(arguments)]]
...
pbs
```

Example:

```
pbc example_bundle
ls -al
ps aux
cat /etc/legal
pbs
```

Execution line of process bundle format (consider the following, a single line):

```
    (process bundle name) [< (input file name)] | [(process bundle name) |]
(process bundle name) [> (output file name)]
```

After the declaration of the bundles, bundles can be executed as:

```
bundle0
bundle1 < input1.txt | bundle2 | bundle3 | bundle 4
bundle5 < input2.txt > output1.txt
bundle6 > output2.txt < input3.txt
bundle7 > output3.txt
bundle8 < input4.txt
bundle9 | bundle10
bundle11 | bundle12 | bundle13 > output4.txt
bundle14 | bundle15 | bundle16
...
```

Important remarks:

- We will provide a parser for you. It will take a single line of input and return the necessary information as an object. Details on how it works, will be on provided as comments. The only information it will need, is one line of string and whether there is currently a process bundle declaration going on.
- Process bundle declarations cannot be interrupted before it is finished.
- Each process bundle can only be executed once.
- A bundle cannot create a pipe to itself.
- After execution, bundle names should be reusable. Although it is most likely not going to be tested during grading.

- Commands are given in a single line. The parameters, redirection and pipe symbols, file names and process bundle names are separated by a single space character.
- Executable names may be given as full path or not. The bshell must be able to execute both.
- All inputs will be proper. Non-existing executables will not be given. The `pbc`, `pbs`, and `quit` commands will not be used as a bundle name or commands inside a bundle.

6 Specifications

- The bshell must terminate when the user gives the *quit* command. Otherwise your homework will not be graded.
- All processes in the process bundles that are being executed should run in parallel. Therefore, order of the outputs will be non-deterministic. In other words, every execution of the exact same bundles may generate different outputs. Output order will not be important during evaluation.
- The bshell should reap all its child processes. It should not leave any zombie processes.
- When a bundle or multiple bundles are being executed, the bshell must wait for the termination of all the child processes before accepting new commands.
- If a process bundle's output is not redirected to a file or to another bundle (via pipeline), it should use stdout as its output. Similarly if its input is not redirected to a file or to another bundle (via pipeline), it should use stdin as its input.
- 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.

7 Regulations

- **Programming Language:** Your program should be coded in C or C++. Your submission will be compiled with `gcc` or `g++` on department lab machines. Make sure that your code compiles successfully.
- **Late Submission:** Late submission is allowed but with a penalty of $5 * day * day$.
- **Cheating:** Everything you submit must be your own work. Any work used from third party sources will be considered as cheating and disciplinary action will be taken under or "zero tolerance" policy.
- **Newsgroup:** You must follow the ODTUClass for discussions and possible updates on a daily basis.
- **Grading:** This homework will be graded out of 100. It will make up 10% of your total grade.

8 Submission

Submission will be done via ODTUClass. 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
```