CS 307 Assignment 1

Yavuz Can Atalay 32445

November 2, 2025

1 Overview

This report elaborates on CS 307 course assignment 1, which is about a shell command line and its interpretation. Code written inside shell.c is a simple command-line shell similar to bash as it supports execution of certain commands, pipelines, I/O operations and a distinctive feature added, loop pipe. The program operates in an interactive mode, reading user commands and processing them using multiple processes and inter-process communication mechanisms such as pipes and file descriptors.

2 Multi-Process Design

This code heavily utilizes the `fork()` and `execvp()` system calls, both fundamental multi-process programming. Each command execution involves creating a child process using fork() system call and `execvp()` if a following process is necessary. Parent process waits for all children before it moves on with `wait()` or `waitpid()` system calls. This is the the fundamental approach followed in this assignment in order to complete the given task, SUShell.

3 Input Parsing

When the code is compiled successfully, it expects an input containing valid command blocks from user. After the input is taken, program parse it providing an array of arguments having properties of inputs which will be highly used during iteration phase.

Table demosntrated below is superficially demonstrating what exactly each variable refers to. Implementation of the code is constructed on these variables. Later on, I will be elaborating on them if the moment calls. Here is the table demonstrating each variable in the array:

Another important factor affecting the input parsing is the last character at the end of

Table 1: Description of `compiledCmd` struct members.

Type	Variable Name	Description
`CmdVec`	`before`	Commands before a loop pipeline.
`CmdVec`	`inLoop`	Commands within a loop pipeline (`|N`).
`CmdVec`	`after`	Commands after a loop pipeline.
`size_t`	`loopLen`	Number of loop iterations for `inLoop`.
`char *`	`inFile`	Input redirection filename (or `NULL`).
`char *`	`outFile`	Output redirection filename (or `NULL`).
`int`	`isQuit`	1 if ':q' was entered, 0 otherwise.

the input. If user explicitly provides backlash O, it may mislead the process. As a result, the code should also handle this possible issue.

```
static void rstrip_newline(char *s) {
    if(!s) return;
    size_t len = strlen(s);
    if (len && s[len-1] == '\n')
        s[len-1] = '\0';
}
```

4 Key Functions and Their Implementations

This section will highlight the core functions in the code. There are total of 5 functions. Certain ones are lower level than others and certain functions aren't called unless certain actions are needed. When necessary, they are called inside other core functions. These are the functions that will be elaborated on:

1. execute-pipeline-with-redirs-fd
2. capture-pipeline-output
3. run-looppipe-and-get-fd
4. execute-pipeline-with-redirs
5. execute-looppipe

Table 2: Overview of Core Shell Execution Functions

`Function Name`	**Key Role**	**Primary Responsibilities**	**Used When**
`execute_pipeline_with_redirs_fd()`	**Core pipeline executor.** Lowest-level function that performs actual process creation and pipe setup.	• Creates pipes between commands. • Forks child processes. • Uses `dup2()` for input/output redirection. • Executes commands via `execvp()`.	Whenever a normal or redirected pipeline must be executed.
`capture_pipeline_output()`	**Captures output** of a pipeline instead of displaying it.	• Creates a temporary pipe. • Calls `execute_pipeline_with_redirs_fd()` internally. • Reads data from the pipe to store output for later use.	When the shell needs to reuse a pipeline's output (e.g., looppipe).
`run_looppipe_and_get_fd()`	**Executes one iteration** of a loop-pipe sequence and returns its output descriptor.	• Runs the current pipeline iteration. • Retrieves its output via `capture_pipeline_output()`. • Passes resulting FD to next iteration.	Inside `execute_looppipe()` for iterative pipeline execution.
`execute_pipeline_with_redirs()`	**High-level wrapper** for pipelines with file redirection support.	• Parses and opens <, >, and » files. • Passes opened FDs to `execute_pipeline_with_redirs_fd()`. • Closes all temporary FDs after execution.	When command includes both a pipe and I/O redirections.
`execute_looppipe()`	**Top-level controller** that manages full loop-pipe execution.	• Parses loop count and passes previous output as input. • Repeatedly runs pipelines.	When a looppipe command is detected in user input.

5 Iteration Process

5.1 Single Command Execution (No Pipe)

Example:

`ls -l`

Process Hierarchy:

1. **Shell Process (Parent)**
   The shell reads user input, parses it, and detects that no pipe or redirection exists. It calls:

   `execute_pipeline_with_redirs_fd()`

   internally to execute the single command.

2. **Inside execute_pipeline_with_redirs_fd():**
   - The shell creates one child process via `fork()`.
   - The child executes the command using `execvp("ls", args)`.
   - The parent waits for the child with `waitpid()`.

Resulting Process Tree:

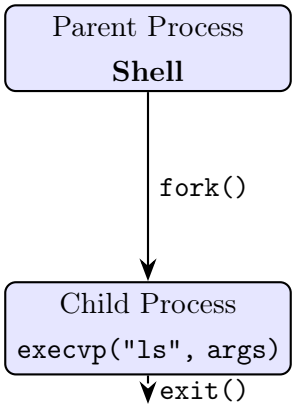


Figure 1: Single Command Execution Hierarchy

5.2 Two-Command Pipeline

Example:

`ls -l | grep txt`

Process Hierarchy:

1. The shell detects a pipe and calls:

   ```
   execute_pipeline_with_redirs();
   ```

   which internally invokes:

   ```
   execute_pipeline_with_redirs_fd(cmds, n, STDIN_FILENO, STDOUT_FILENO);
   ```

2. Inside `execute_pipeline_with_redirs_fd()`:
   - The shell creates a pipe using `pipe(fd)`.
   - It forks the first child (for `ls -l`):
     - Redirects `STDOUT` to `fd[1]` via `dup2()`.
     - Executes `execvp("ls", args)`.
   - It forks the second child (for `grep txt`):
     - Redirects `STDIN` to `fd[0]` via `dup2()`.
     - Executes `execvp("grep", args)`.
   - Parent closes both ends of the pipe and waits with `waitpid()`.

Resulting Process Tree:

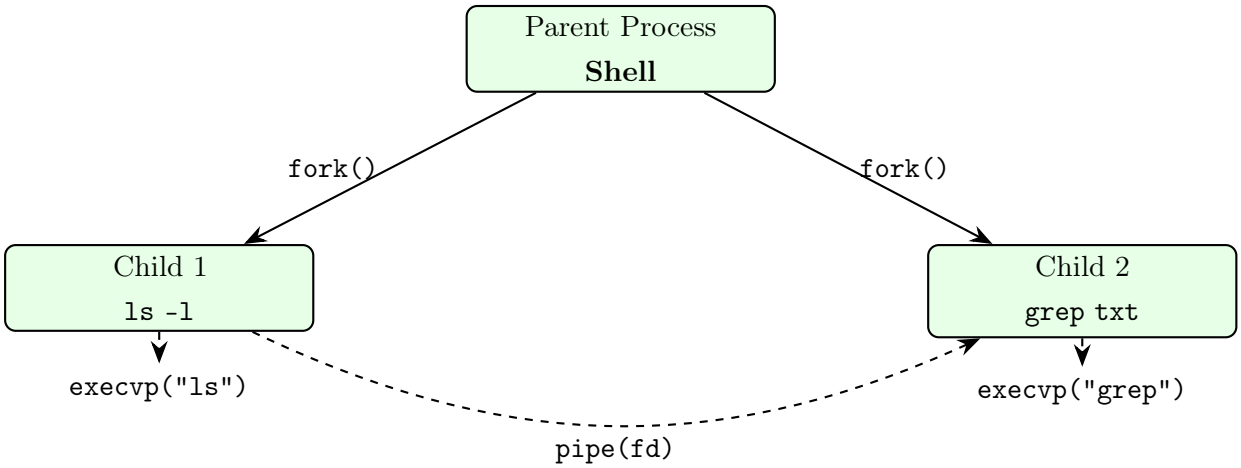


Figure 2: Two-Command Pipeline Process Hierarchy

5.3 Multi-Command Pipeline

Example:

```
cat file.txt | grep data | sort | uniq
```

Process Hierarchy:

- The shell forks four children (one per command) inside `execute_pipeline_with_redirs_fd()`.
- For each child:
  - `STDIN` and `STDOUT` are redirected using a series of `dup2()` calls.
  - Each child executes its command using `execvp()`.
- The shell closes all pipes and waits for every child to finish.

Conceptual Hierarchy Diagram:

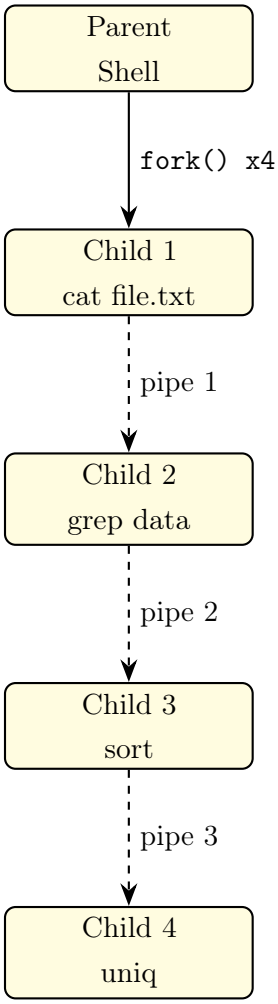


Figure 3: Multi-Command Pipeline Process Hierarchy

5.4 LoopPipe Execution (Pipeline Executed Repeatedly)

Example:

```
echo "abcd" |(rev)_2
```

Process Hierarchy:

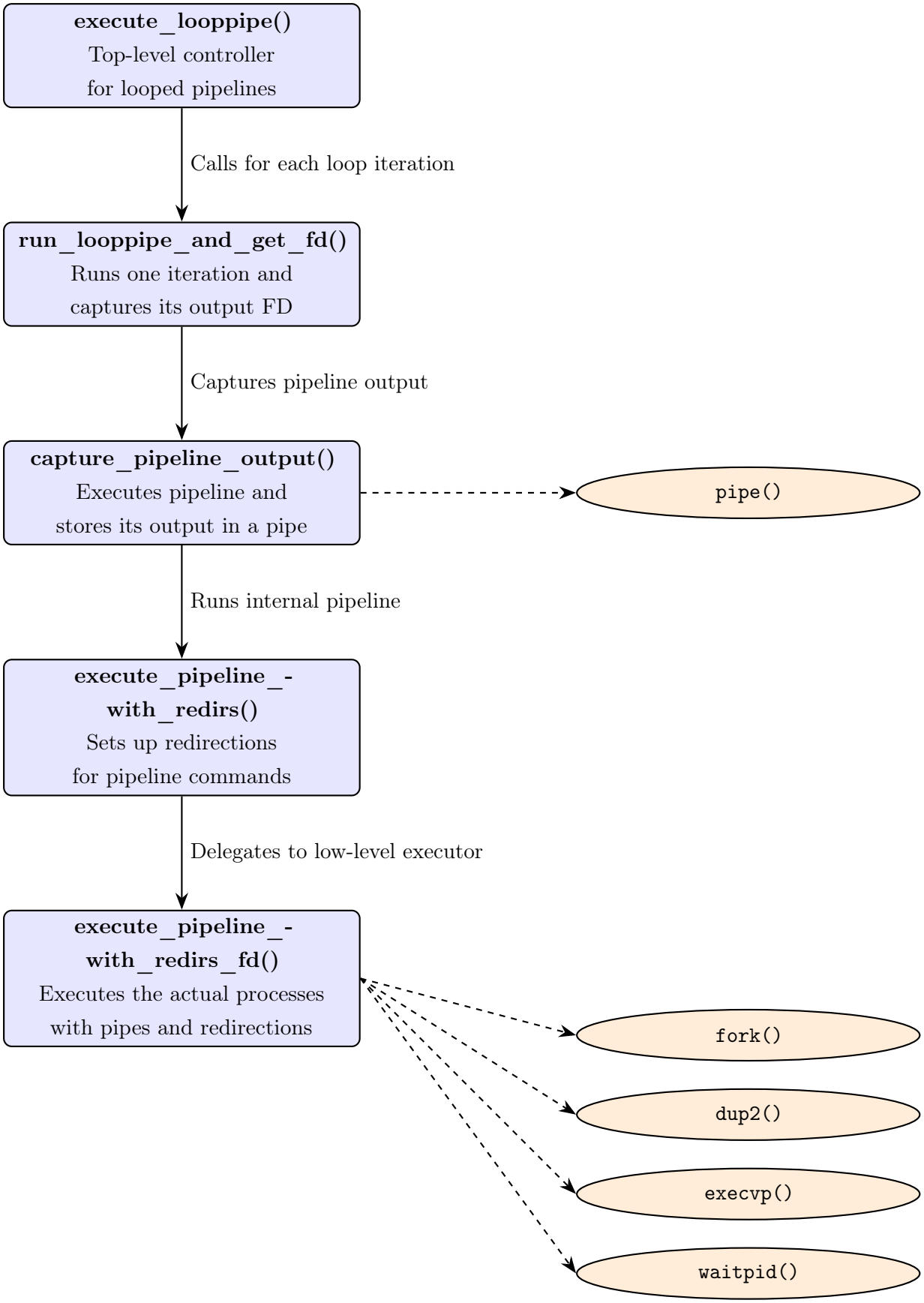
1. The shell detects loop syntax `|(...  )_N` and calls:

   ```
   execute_looppipe();
   ```

2. Inside `execute_looppipe()`:

   - Runs first iteration using `run_looppipe_and_get_fd()`, which calls `capture_pipeline_output()` internally.
   - Output is stored in a pipe and passed as input to the next iteration.
   - Each iteration reuses `execute_pipeline_with_redirs_fd()`.

Resulting Hierarchy Diagram:


execute_looppipe()
Top-level controller
for looped pipelines
Calls for each loop iteration
run_looppipe_and_get_fd()
Runs one iteration and
captures its output FD
Captures pipeline output
capture_pipeline_output()
Executes pipeline and
stores its output in a pipe
pipe()
Runs internal pipeline
execute_pipeline_-
with_redirs()
Sets up redirections
for pipeline commands
Delegates to low-level executor
execute_pipeline_-
with_redirs_fd()
Executes the actual processes
with pipes and redirections
fork()
dup2()
execvp()
waitpid()

6 Explicit Answers for Report Questions

6.1 Process Hierarchy and Command Creation Order

Example:

`A | B | C`

Creation Order:

1. Inside `execute_pipeline_with_redirs_fd()`, the shell iterates left to right over the commands.
2. For each command:
   - Creates a new pipe for the next stage using `pipe(fd)`.
   - Forks a child with `fork()`.
   - In the child:
     - Redirects `STDIN/STDOUT` as needed with `dup2()`.
     - Executes the command with `execvp()`.
   - The parent closes pipe ends it no longer needs.
3. After all forks, the parent waits for all children with `waitpid()`.

Concurrency: Each forked child process runs concurrently once created. The kernel manages synchronization automatically through pipe buffers: writers block when full, readers block when empty. All children execute in parallel to the extent allowed by CPU scheduling.

Figure: Process Hierarchy

```
Parent (Shell)
   |
   |-- fork --> [Child A: execvp(A)] --stdout--> p01 --read--> [Child B: execvp(B)] -
                 dup2(p01[1], STDOUT)                          dup2(p01[0], STDIN)   d

Parent closes p01[*], p12[*]; then waitpid(all).
```

—

6.2 Pipe Structure Between Two Processes

Between any adjacent pair (X | Y):

- Exactly one pipe is created using `pipe(fd)`.
- Left-side child (X):
  - `dup2(fd[1], STDOUT)`
  - `close(fd[0])`, `close(fd[1])`
- Right-side child (Y):
  - `dup2(fd[0], STDIN)`
  - `close(fd[0])`, `close(fd[1])`

Data Flow and Synchronization:

- Writes by process X appear as reads by process Y through the shared kernel pipe buffer.
- Only one pipe per edge (one direction: X→Y).
- The kernel enforces synchronization; closing unused pipe ends ensures EOF is detected correctly.

Figure:

```
[Child X: STDOUT -> fd[1]] ===(kernel pipe buffer)=== [Child Y: fd[0] -> STDIN]
```

—

6.3 Input Decision in LoopPipe Structures

LoopPipes are handled by:

```
execute_looppipe() → run_looppipe_and_get_fd()
                   → capture_pipeline_output()
                   → execute_pipeline_with_redirs_fd()
```

Input source logic:

1. If there is input redirection `< file`, that file descriptor becomes `fd_in`.
2. Else if the loopPipe is part of a larger pipeline, the upstream pipe's read end is used.
3. Otherwise (standalone), `STDIN_FILENO` is used.

For later iterations (_2, _3, ...):

- The output of the previous iteration becomes the input for the next iteration.
- This is achieved because `run_looppipe_and_get_fd()` returns the read-end of the captured output pipe.

Iteration Sequence:

```
iter1 fd_in:  <file  OR  upstream pipe  OR  STDIN
iter2..k fd_in:  previous_iteration_output_fd
```

Each iteration's children (pipeline processes) run concurrently, but iterations themselves are sequential inside `execute_looppipe()`.

—

6.4 Handling Piped Command After a LoopPipe

Example:

```
X | (Y)_3 | Z
```

Execution flow:

1. `execute_looppipe()` runs the looped block `(Y)_3`.
2. After the final iteration, the output is available as a readable FD (read end of the last iteration's captured output).
3. The next stage `Z` receives that FD as its `fd_in` when calling `execute_pipeline_with_redirs_fd()`.
4. In `Z`'s process: `dup2(looppipe_out_fd, STDIN)` ensures it reads the loopPipe output.

Conceptual View:

```
[Child X] --pX--> [LoopPipe (Y)_3]
      iter1: run_looppipe_and_get_fd() → fd_iter1_out
      iter2: fd_in = fd_iter1_out → fd_iter2_out
      iter3: fd_in = fd_iter2_out → fd_final_out
  (exposes fd_final_out as read end)
fd_final_out --pY--> [Child Z]
```

Note In the code, there is a bug about looppipe iteration which I couldn't figure out. So in one wa