Introduction

The goal of this lab is the development of a shell with given specifications. In each section we will describe a specification and our approach of implementing it.

Execution of Simple Commands

In order to be able to execute commands, the input received via the command line is parsed and then handled by `RunCommand()` function.

The general method is to fork the current process into a parent and child, the latter of which executes a binary, replacing its core image but keeping it's file descriptors, among other things.

The `execvp` system call is used for execution as it is PATH-aware, meaning that the absolute path to binaries need not be specified, if they are on the PATH. Additionally, a pointer to a list of arguments is taken as input, meaning that the arguments do not need to be known at compile time.

The parent process wait to reap the child processes, suspending its own process.

```
pid_t pid = fork();
if (pid == -1)
  exit(EXIT_FAILURE);

if (pid == 0){ // Child code
  char **execArgs = getArgs(..., cmd->pgm); // Acquire list of arg strings.
  execvp(execArgs[0], execArgs);
}
else if (pid > 0) // Parent code
  waitpid(pid, NULL, 0);
```

Executing a command in the shell yields the following output:

```
0w0~$ date
Thu Sep 24 23:48:13 CEST 2020
```

Background Processes

Enabling the execution of background processes is vital because they handle important tasks such as scheduling, system monitoring and user notification.

The signal handler (`sigHandler`) is taking care of `SIGCHLD` and is enabled in child processes on the condition that the current command is executed as a background processes (or multiple processes thereof). The parent process, in this case, does *not* `wait()` for these processes, but instead saves their individual PIDs in a global buffer. Upon terminating, child processes will send a `SIGCHLD` signal to the parent, invoking the signal handler. The handler will iterate through aforementioned global buffer and `waitpid()` for each specific process, but in a *non-blocking* fashion, using the `WNOHANG` option. This ensures that it iterates through the list, reaping those processes that have terminated, but not suspending on those that have not.

This is perhaps a suboptimal solution, but seemingly works in practice.

`top` shows a process executing while the shell is not suspended, and does not leave a zombie process behind.

Pipes

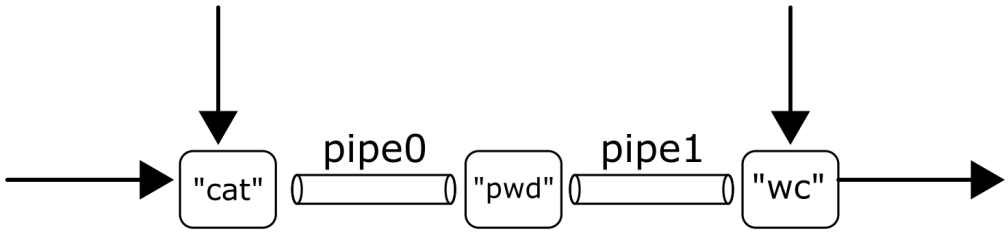

Figure 1: Pipes between individual processes. Standard stream or file descriptor redirections on first and last processes.

A pipe is created just before forking; once the next child process is about to be spawned, the output from the previous one is passed as argument to the next pipe. The input to the first child is the standard input (unless it has been redirected from a file), and the last child's output pipe is connected to the standard output, or the file it can be redirected to if desired.

Pipes are generally implemented by redirecting the file-descriptors normally associated with STDIN and STDOUT, the standard streams, to the read and write ends of a pipe, respectively.

This is illustrated in fig. 2.

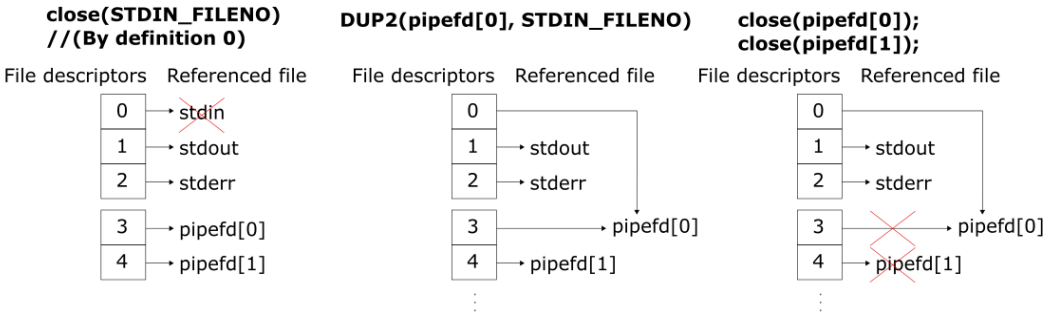

Figure 2: Pipe implementation using file-descriptor duplication and redirection.

```
/* Set up all pipes that we need */
 int pipefd[noOfChildren][2];
 int n;
 int pipeRet;
 for(n = 0; n < noOfChildren-1; n++){
   if(DEBUG)printf("Creating pipe %d\n", n);
   if (pipe(pipefd[n]) == -1){
     printf("%s\n", strerror(errno));
   }
 }
```

All required processes and pipes are created when a command is run, and parameters such as file-descriptors and redirections are set by iterating through the processes before `exec`ing them in turn.

Redirection of Standard Input and Output

Redirection was achieved by duplicating file-descriptors using `dup2`, taking advantage of the fact that programs usually read from STDIN and write to STDOUT. By mapping open files to these standard stream file-descriptors, redirection is implemented.

The general method is outlined in fig. 2.

`cd` and `exit` as Built-in Functions.

In order to use the commands `cd` to change the directory, and `exit` to exit the shell as built-in functions, we added the following lines to our code.

```
if(!strcmp(cmd.pgm->pgmlist[0], "exit")) exit(0);
if(!strcmp(cmd.pgm->pgmlist[0], "cd")) chdir(cmd.pgm->pgmlist[1]);
```

Changing the directory only works with an absolute path; `exit` terminates the shell.

Ctrl-C Functionality

The Ctrl-C key combination is supposed to end foreground processes, but nothing else: background processes should continue with their execution and the shell should not terminate.

To this end, a system call for the parent process to ignore `SIGINT` signals was implemented in the following way.

```
signal(SIGINT, SIG_IGN);
```

Whenever a foreground command is run, however, the signal handler for `SIGINT` is re-enabled in the child processes using

```
if (cmd->background == FALSE)
    signal(SIGINT, sigHandler);
```