TDT4186 Operating Systems
Practical Exercise 3 - Multithreaded Server

Håvard R. Krogstie
Magnus Lykke Dahlø
Herman Håberg Seternes

April 19, 2022

3.1 Basic functionality

The basic functionality is handled by `control.c` and `tasks.c`. By calling `handle_command` we check the arguments and create a task with `create_task`. It then creates a new process with `fork` and execute the given command with arguments with `execvp(3)`. When running normal commands, the child process keeps the same `stdin` and `stdout` as its parent, and the parent waits for the child to finish using `waitpid(2)`, effectively letting the child take over the terminal.

3.2 Changing directories

We handle changing of directories in `handle_builtin`, which is a function that handles all built in commands. It first checks for matching commands and if found (will always be `ch` in this case) checks if the correct amount of arguments are given and changes to the given directory with `chdir(2)`. If `chdir` returns something other than 0 it prints an error message to the user. If `handle_builtin` runs, it will `continue` to a new iteration of our main loop, hereby skipping all code for handling normal commands.

3.3 IO redirection

We implemented this by letting the `create_task` function take an arbitrary file descriptor as input, and a `char*` as output. Normal execution uses `STDIN_FILENO` and `NULL`, respectively. To pass input from a file, the file is `open(2)`ed, passed as the input file descriptor, and inside the child overrides `STDIN_FILENO` using `dup2(2)`. Redirecting stdout to a file is the same deal, also handled by the child process. In no circumstance does the parent process need to manually move bytes between buffers and/or file descriptors, everything is OS based.

3.4 Background tasks

Background functionality was implemented through checking for an ampersand (`&`) as the last argument. If an ampersand was found, we would create the task as we did in **3.1**, but add it to our task list with `add_to_task_list` instead of waiting for the status of the task.

If a user ran a command as a background task and the task ends, the exit status of that task will be printed when the next prompt is shown, on the line above.

3.5 Background task status

We added a check in `handle_builtin` for the string `jobs`, to let a user print the background jobs currently running. If a user were to write in additional arguments after `jobs`, flush will return an error message to the user. See the following example of full job handling:

```
/home/havard/.../03-terminal: sleep 10 &
/home/havard/.../03-terminal: cat hilsen.txt
Hei
Heisann?
Exit status [cat hilsen.txt] = 0
/home/havard/.../03-terminal: sleep 2 &
/home/havard/.../03-terminal: jobs
3226: sleep 10 &
3237: sleep 2 &
/home/havard/.../03-terminal: <enter pressed on empty prompt>
```

```
Exit status [sleep 2 &] = 0
/home/havard/.../03-terminal: jobs
3226: sleep 10 &
Exit status [sleep 10 &] = 0
/home/havard/.../03-terminal:
```

3.6 Pipes

The parser is strict, only letting the first program in the pipe chain have < input, and only the last have > output. After removing these statements, the full list of commands are send to `handle_command` in `control.c`. The first command to run gets either `STDIN_FILENO` or a file as input. If any | tokens are encountered, the command up until that token is started using `handle_task`, but the `outputpath` is set to `"PIPE"`. This causes a pipe to be created using `pipe(2)`, and stdout of the new task is sent into its writing end (using `dup2`). Its reading end becomes the input file descriptor of the next command in the pipe chain. Only the last command can have an actual file as its output redirection.

Here is an example of output using multiple pipes and output to file, while reading from stdin:

```
/home/havard/.../03-terminal: cat | grep Hei | grep -v ! > hilsen.txt
Hei
Hallo
Heia!
Heisann?
Exit status [cat | grep Hei | grep -v ! > hilsen.txt] = 0
/home/havard/.../03-terminal: cat hilsen.txt
Hei
Heisann?
Exit status [cat hilsen.txt] = 0
```

A small sidenote is that the shell doesn't properly wait for all the commands in the chain, only the last one, leaving zombies around. This could be fixed, but would require a separate task list from the job list.