

CPS1012 — Operating Systems & Systems Programming 1

Coursework

Juan Scerri

B.Sc. (Hons)(Melit.) Computing Science and Mathematics (First Year)

April 23, 2022

Contents

1	Plagiarism Declaration	3
2	Structure & Design	4
2.1	Structure	4
2.2	Design	5
2.2.1	Design of Task 1	5
2.2.2	Design of Task 2	10
2.2.3	Design of Task 3 and 4	11
3	Testing Methodology	15
3.1	Testing for Task 1	15
3.2	Testing for Task 2	16
3.3	Testing for Task 3	16
4	Issues & Bugs	18
4.1	Issues	18
4.1.1	Design Issues	18
4.1.2	Functionality Issues	18
4.2	Bugs	18

Listings

1	Configuration after <code>fork()</code> and before <code>execvp()</code>	5
2	<code>waitpid()</code> being called in the parent for every forked process.	8
3	Redirecting pipeline input.	9
4	Redirecting pipeline output.	9
5	The declaration of the global <code>cwd</code>	10
6	Initialising <code>cwd</code>	10
7	Comment indicating the usage of <code>#define DEBUG</code>	17
8	Usage of the <code>DEBUG</code> definition for debugging.	17

1 Plagiarism Declaration

Plagiarism is defined as “*the unacknowledged use, as one’s own, of work of another person, whether or not such work has been published, and as may be further elaborated in Faculty or University guidelines*” (University Assessment Regulations, 2009, Regulation 39 (b)(i), University of Malta).

I, the undersigned, declare that the report submitted is my work, except where acknowledged and referenced. I understand that the penalties for committing a breach of the regulations include loss of marks; cancellation of examination results; enforced suspension of studies; or expulsion from the degree programme.

Work submitted without this signed declaration will not be corrected, and will be given zero marks.

Juan Scerri

CPS1012

April 23, 2022

Student’s full name

Study-unit code

Date of submission

Title of submitted work: Operating Systems & Systems Programming 1 Coursework

Student’s signature

A handwritten signature in black ink, appearing to read 'J. Scerri', is written over a horizontal line.

2 Structure & Design

2.1 Structure

The coursework was broken up into four tasks.

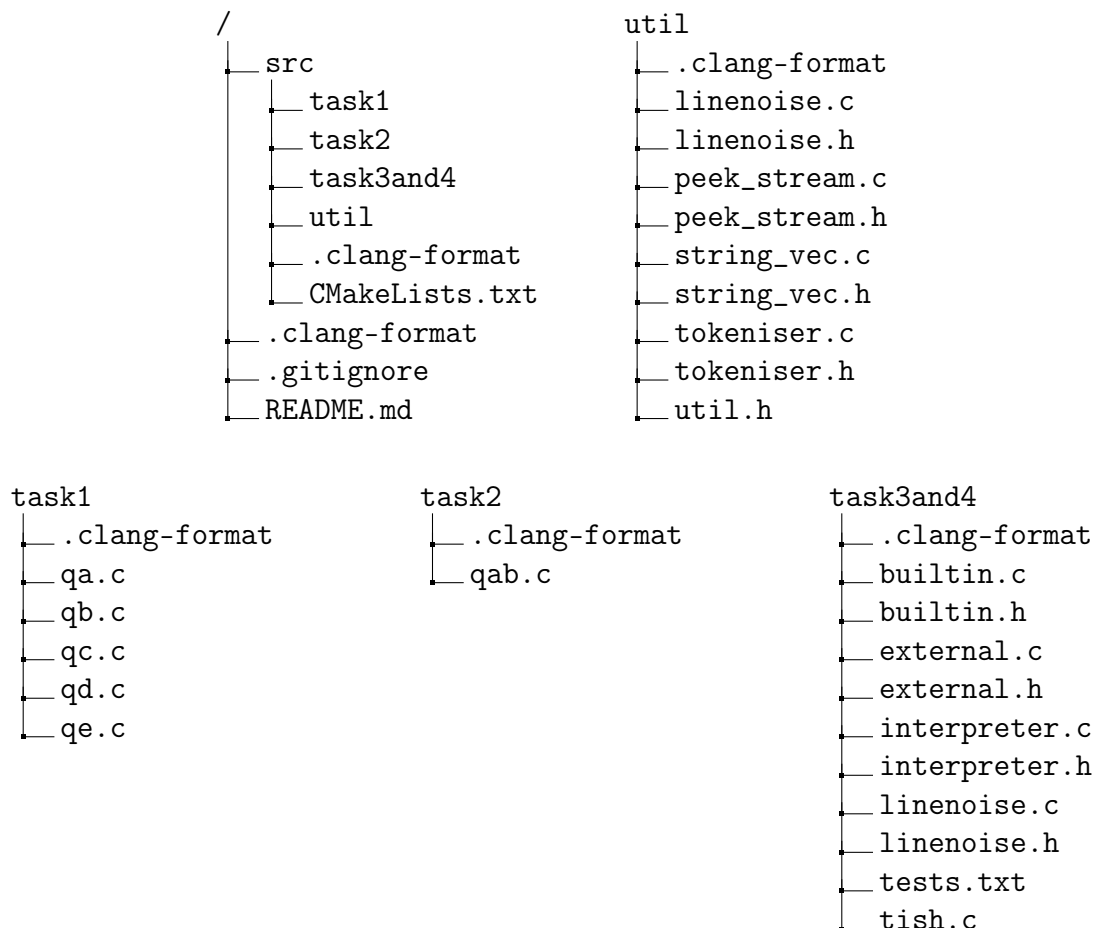
All the questions of *Task 1* have been answered separately, each having there own `.c` file.

The two questions of *Task 2* where answered in one `.c` file.

Task 3 and *Task 4* where done simultaneously to avoid code repetition and reduce complexity of development. This is because it is easier to develop an interpreter where all the constraints are known.

Naturally, some of the code developed from *Task 1* and *Task 2* was copied over to the directory of *Task 3* and *Task 4*. Specifically, the relevant code from *Task 1* was placed in `external.c` and the relevant code from *Task 2* was placed in `builtin.c`.

Note: There is an extra directory called `util` which contains older versions of code which can be found in `interpreter.c`. The header files within this directory are used in *Task 1* and *Task 2* to allow for easier testing.



2.2 Design

In this section the majority of the implemented algorithms and design will be explained through diagrams/images. Where necessary supplementary explanation will be provided and codes snippets will be inserted.

2.2.1 Design of Task 1

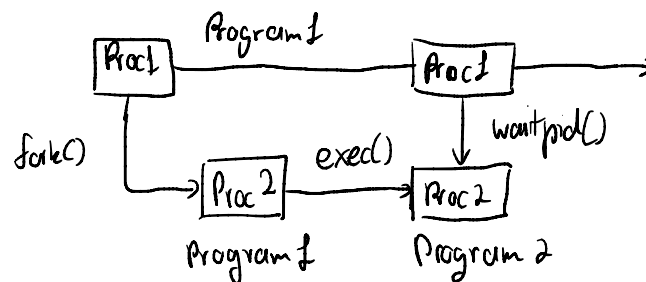


Figure 1: A drawing of the fork-exec-wait pattern.

For question (a), the requested algorithm implements a simple `fork-exec-wait` pattern. This pattern is used to allow processes to load other programs.

Nevertheless, this might not always be enough. If two or more processes need to communicate with each other, they need to make use of Inter-Process Communication (IPC) mechanisms. For question (b), two processes will be allowed to communicate together through a pipe an OS provided IPC mechanism.

Note: Configuration of the pipes is done after `fork()` and before `exec()`.

```

if ((pid = fork()) == -1) {
    return -1;
} else if (pid == 0) { // Left Child
    close(closefd);
    if (dup2(oldfd, newfd) == -1)
        return -1;

    close(oldfd);
    if (execvp(args[0], args) == -1)
        return -1;
}
  
```

Listing 1: Configuration after `fork()` and before `execvp()`.

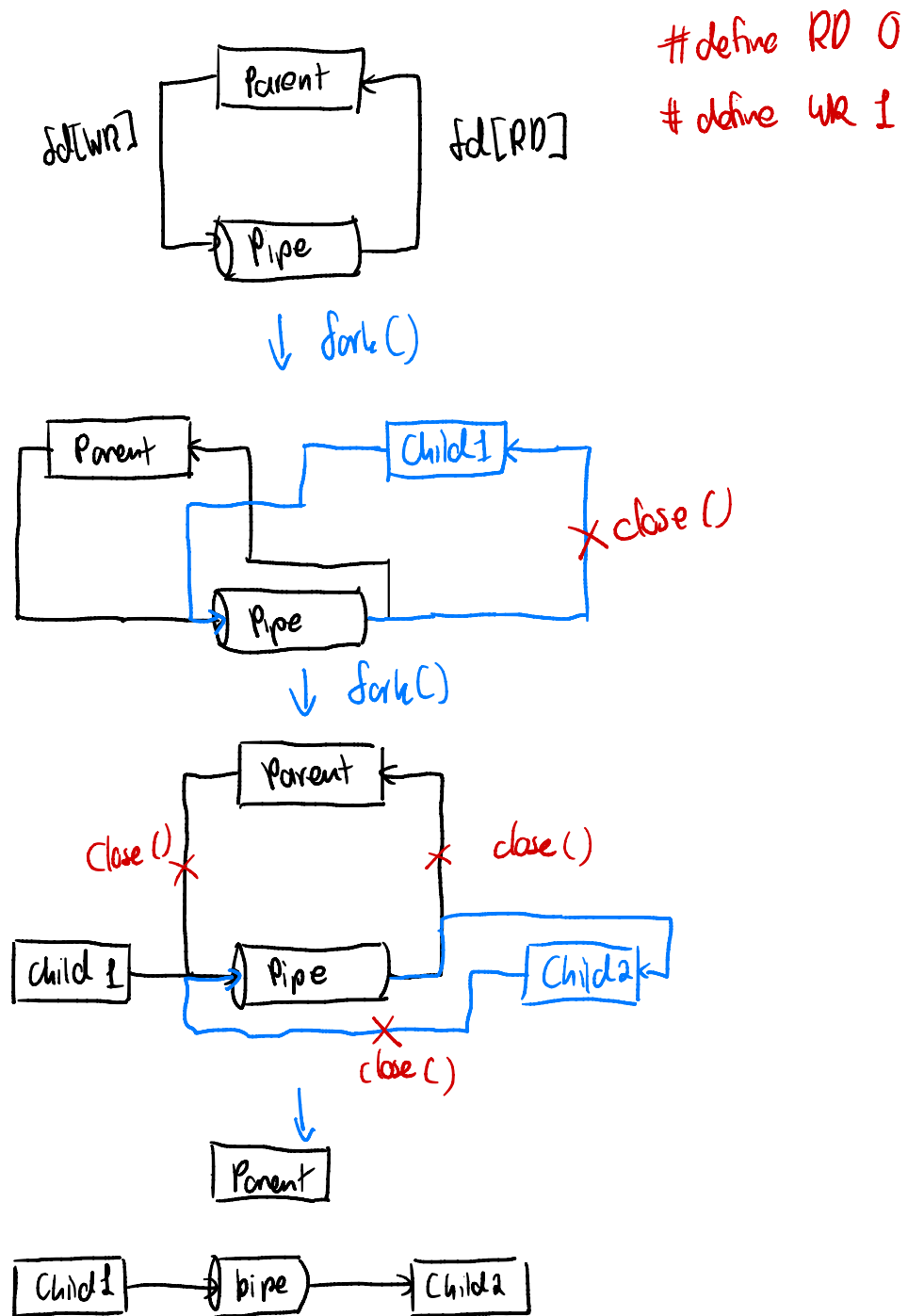


Figure 2: A drawing of the procedure used to connect two processes through a pipe.

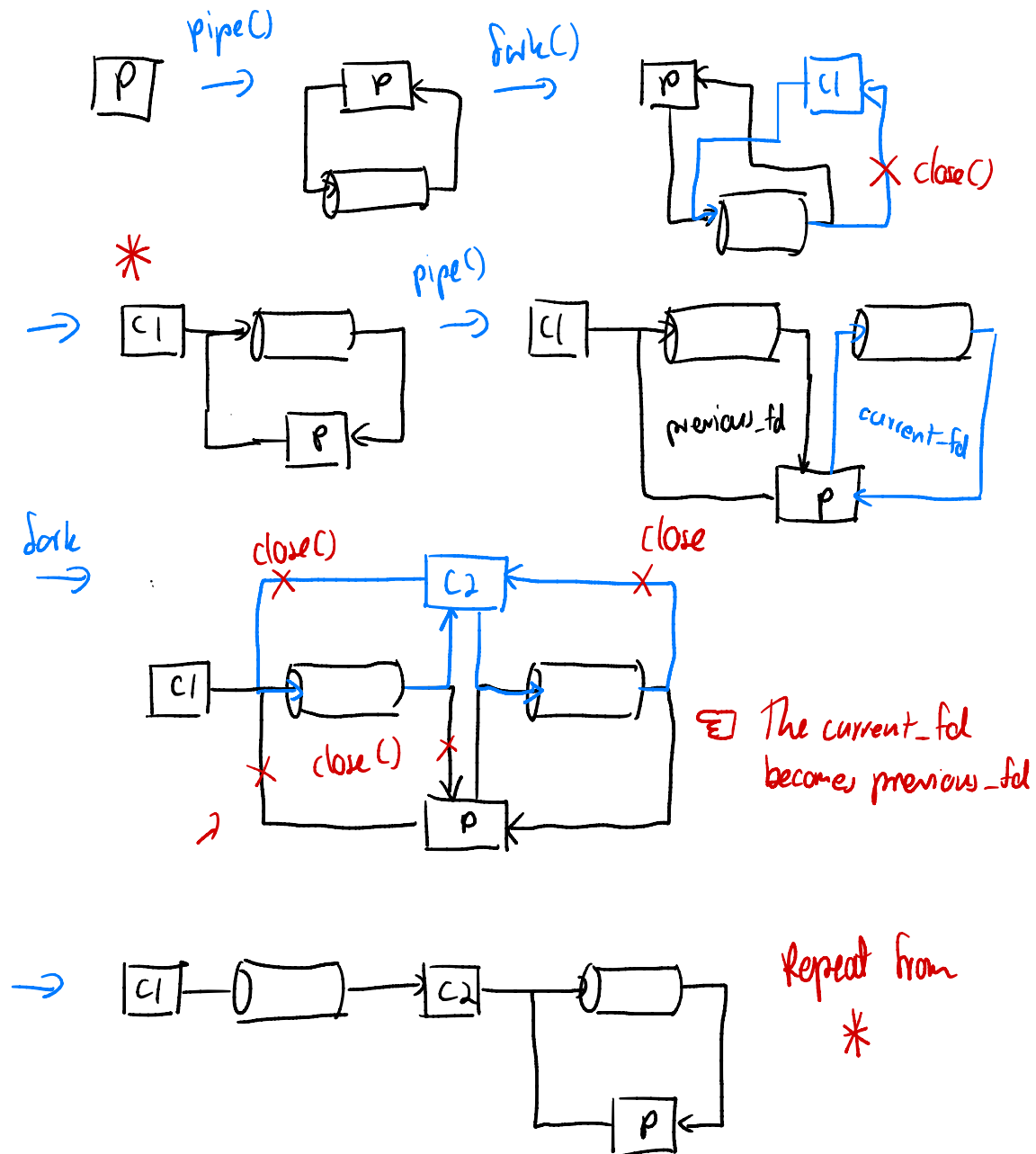


Figure 3: A drawing of the start procedure and intermediate procedure used to construct a pipeline.

For question (c), a new variant was created called `execute_pipeline()`. As input, it takes in a NULL-terminated array of programs (of type `char **`). Then, it constructs a pipeline which allows every program to communicate with the next program in the array. This is done through the procedure described in figure 3.

Note: There is no need to know the length of the pipeline. This is because the only four file descriptors which the parent has awareness of at any one point in time during the construction of the pipeline, are the file descriptors associated with the previous and current pipe.

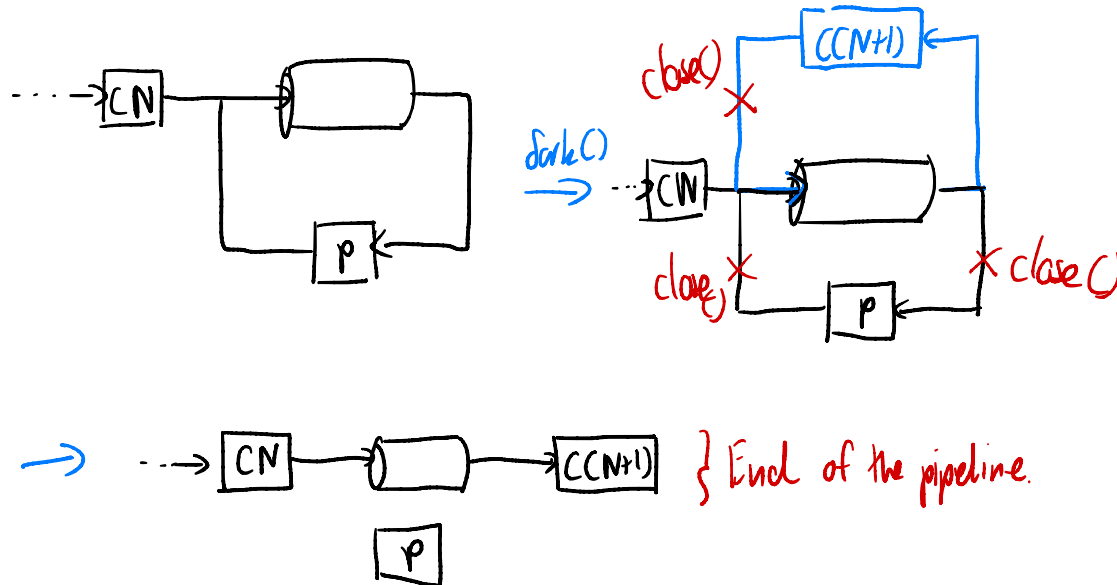


Figure 4: A drawing of the end procedure used to construct a pipeline.

In particular there are three distinct unique procedures which the algorithm has to perform. These are labelled as start, intermediate (described in figure 3) and end (described in figure 4). In particular the intermediate step is repeated until the end is reached.

```
// NOTE: This is the code we'd want to have if we do not
// want to have zombies during a pipe.
if (waitpid(pid, &status, options) == -1)
    return -1;
}

// NOTE: This results in zombies which are not reaped by
// the parent.
/* if (waitpid(pid, &status, options) == -1) */
/* return -1; */
```

Listing 2: `waitpid()` being called in the parent for every forked process.

For question (d), an `options` parameter was added to the previous function and a call to `waitpid()` is made for every forked process (see the above listing).

The function blocks for every process to ensure no zombie processes are present. Figure 5 demonstrates a version of the question where the function blocks only for the last process.

```

==17746== still reachable: 0 bytes in 0 blocks
==17746== suppressed: 0 bytes in 0 blocks
==17746== Rerun with --leak-check=full to see details of leaked memory
==17746== For lists of detected and suppressed errors, rerun with: -s
==17746== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
os-coursework/build on main [x!?] via Δ v3.22.2
> ./q1d
(Enter to Stop) > ls -al
(Enter to Stop) > figlet
(Enter to Stop) > sleep 60
(Enter to Stop) >
[
252M S 0.0 3.6 0:00.00 /usr/lib64/firefox/firefox
5228 S 0.0 0.6 0:00.15 /usr/lib64/firefox/firefox -contentproc -childID 5 -i
1248 S 0.0 0.0 0:00.00 /usr/bin/xsel --nodetach -i -b
5356 S 0.0 0.0 0:00.00 systemd-userwork
5456 S 0.0 0.0 0:00.00 systemd-userwork
5456 S 0.0 0.0 0:00.00 systemd-userwork
252M S 0.0 3.6 0:00.00 /usr/lib64/firefox/firefox
872 S 0.0 0.0 0:00.00 ./q1d
0 Z 0.0 0.0 0:00.00 ls
0 Z 0.0 0.0 0:00.02 figlet
784 S 0.0 0.0 0:00.00 sleep 60
5228 S 0.0 0.6 0:00.00 /usr/lib64/firefox/firefox -contentproc -childID 5 -i
F1Help F2Setup F3Search F4Filter F5Tree F6SortBy F7Nice F8Nice F9Kill F10Quit

```

Figure 5: A screenshot of zombie processes (in `htop`) present because the function waits only for the last process.

```

// If this is the first process.
if (i == 0) {
    if (infile != NULL && redirect_input(infile) == -1)
        return -1;
}

```

Listing 3: Redirecting pipeline input.

```

// If this is the last process.
if (pipeline[i + 1] == NULL) {
    if (outfile != NULL &&
        redirect_output(
            outfile, append ? O_APPEND : O_CREAT) == -1)
        return -1;
}

```

Listing 4: Redirecting pipeline output.

For question (e), a simple addition of three extra parameters and two if-statements in the start and end steps suffice to redirect the input and the output of the pipe. The third parameter is a flag used to choose between appending to a file or truncating a file.

2.2.2 Design of Task 2

For question (a), the structure of what needs to be implemented is laid out clearly in the question. Specifically, the creation of a structure for holding function pointers and function names together.

Then a linear search is performed every time before trying to execute an external process to check if the specified command is a builtin.

For question (b) a number of builtin functions had to be implemented.

`sh_exit()` is the simplest of the functions simply calling the `exit()` function from `stdlib.h`.

`sh_ver()` prints a formatted string to `stdout` containing the author's name, version and a small message. The author's name, version and message are all `#defines`.

```
#ifdef __linux__
#include <linux/limits.h>
#else
#define PATH_MAX 4096
#endif

char cwd[PATH_MAX];
```

Listing 5: The declaration of the global `cwd`.

```
char *line;
char **args;
if (getcwd(cwd, PATH_MAX) == NULL) {
    perror("getcwd");
}
```

Listing 6: Initialising `cwd`.

`sh_cwd()` prints the contents of the global variable `cwd` to `stdout`. `cwd` is a character array with a length of `PATH_MAX`, which is a definition found in `linux/limits.h` on Linux. If the OS is not Linux, the limit will be set to 4096 characters. Moreover, the global has to be initialised at the start of the shell.

`sh_cd()` uses `chdir()` to change the current directory of the shell to the first argument provided to `cd` or to the home directory of the current user if no argument is provided. Naturally, `sh_cd()` will have to update `cwd` every time since the directory might change.

2.2.3 Design of Task 3 and 4

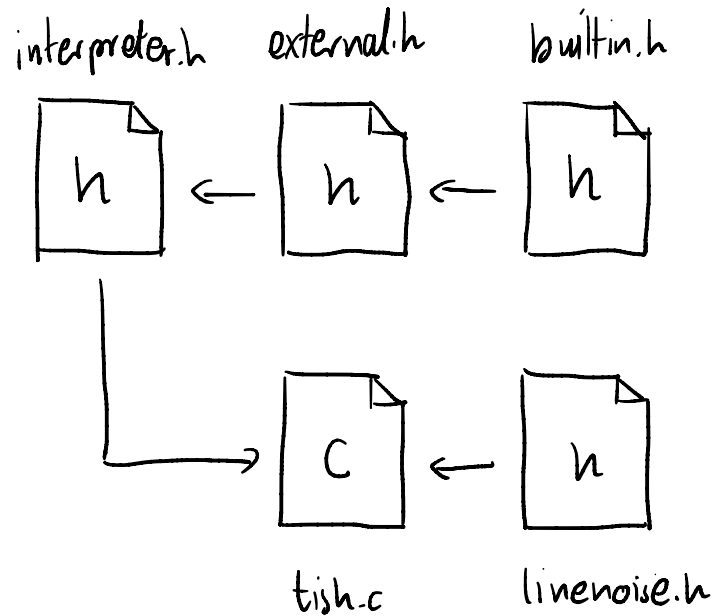


Figure 6: A drawing of how the files used for *Task 3* and *Task 4* are linked (through `#include`).

The design of these tasks have been merged into one. Particularly, there are five main `.c` files. `external.c` contains the functions written for *Task 1* and `builtin.c` contains the functions written for *Task 2*. `interpreter.c` is the file where all the new functionality is present. Specifically, it contains, the tokeniser, syntactic checker and translator. Moreover, it also contains a simple `execute()` function which interacts with the components in `external.c` and `builtin.c` through the header files.

`linenoise.c` is the suggest line editing library, which allows for the easy creation of prompts and command line user interfaces. The repository of the original author can be found at <https://github.com/antirez/linenoise>.

`tish.c` is the source file which brings all of these components together to create a functional shell named ‘Tiny Shell’ (`tish`).

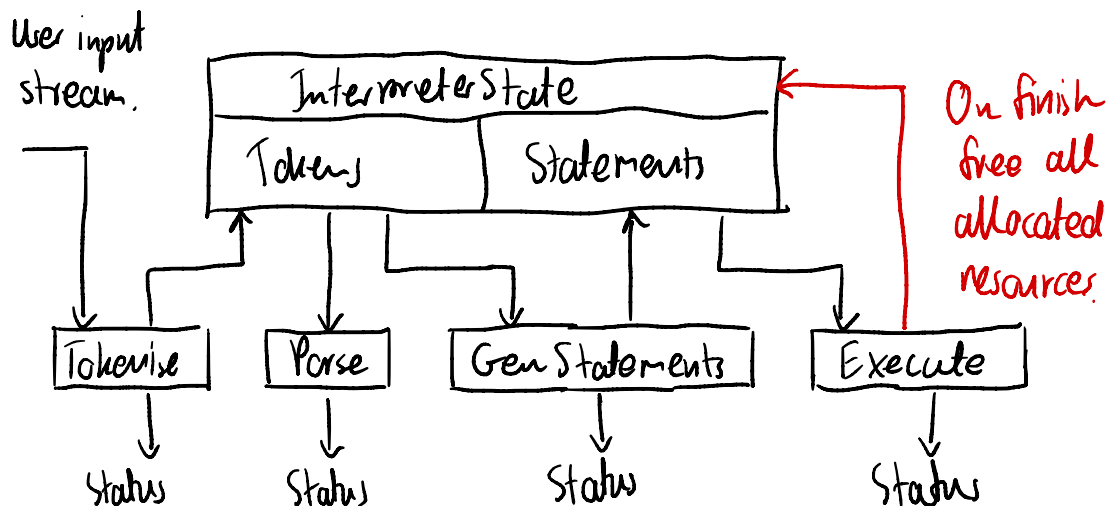


Figure 7: A drawing of how the interpreter is structured and how it processes user input lines.

In the interpreter there is one static global structure which is used throughout all of the components of the interpreter. These components/functions are the following: `tokenise()`, `parse()`, `genStatements()` and `execute()`. There is one exposed function in the file `interpret()` and it is called in `tish.c`.

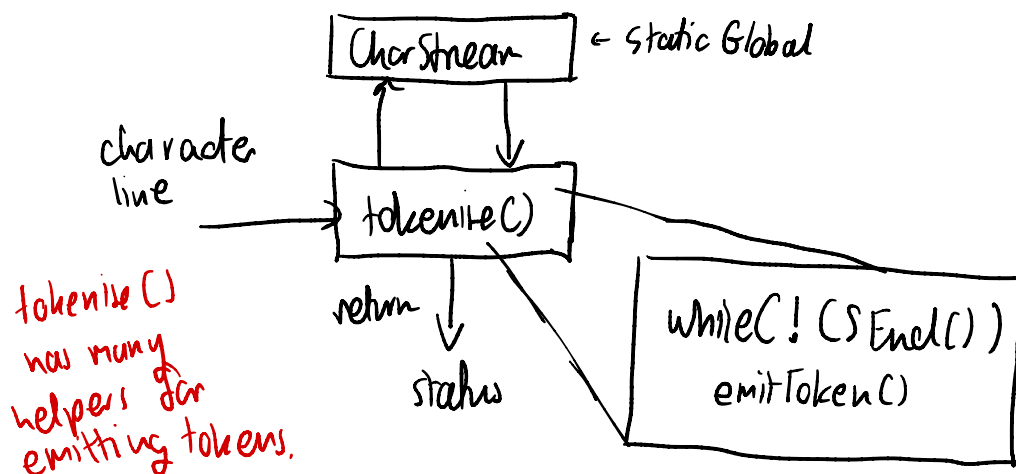


Figure 8: This is a simple description of how the tokeniser works.

The `tokenise()` function makes use of a static global called `CharStream`. This allows the tokeniser to traverse a character array without losing state across different function calls. This makes the code more readable and manageable.

The most important functions in the tokeniser are the emitters. They basically emit a token which can later be used to check if the user input is syntactically correct. Emitted tokens are stored

in the interpreter state. Escaping of characters is dealt with in two emitters: `emitString()` and `emitQuotedString()`.

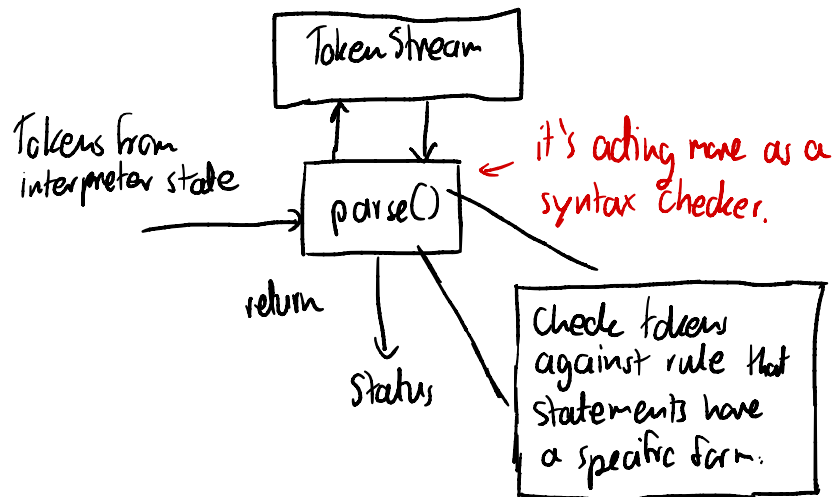


Figure 9: this is a simple description of how the parser works.

`parse()` initialises the `TokenStream` and checks if syntax of the tokens is correct. The general form of a statement is:

```

statement := cmd1 | cmd2 | ... | cmdN < infile {> / >>} outfile
cmd := prg arg1 arg2 ... argN
prg := str
arg := str

```

If a collection of tokens cannot fit into the above production rule, the parser will print an error and clean any allocated resources.

Note: Many of the characters are optional for example there is no need for input or/and output redirection. Nevertheless, if for example a `>` is present it must be followed by the `infile`. In the function there are many similar rules.

After checking that the tokens are syntactically correct, other functions specifically `genStatements()` can assume a lot about the structure of the tokens. The `genStatements()` function does two main things. It counts the number of commands in a pipeline and the number of strings per command.

Having this information allows the function to allocate the exact amount of memory required for the statements which are then populated accordingly.

Note: The process of counting, allocating and populating are all done simultaneously throughout the code. This is because it is more convenient for certain tasks to be done when the information is directly available.

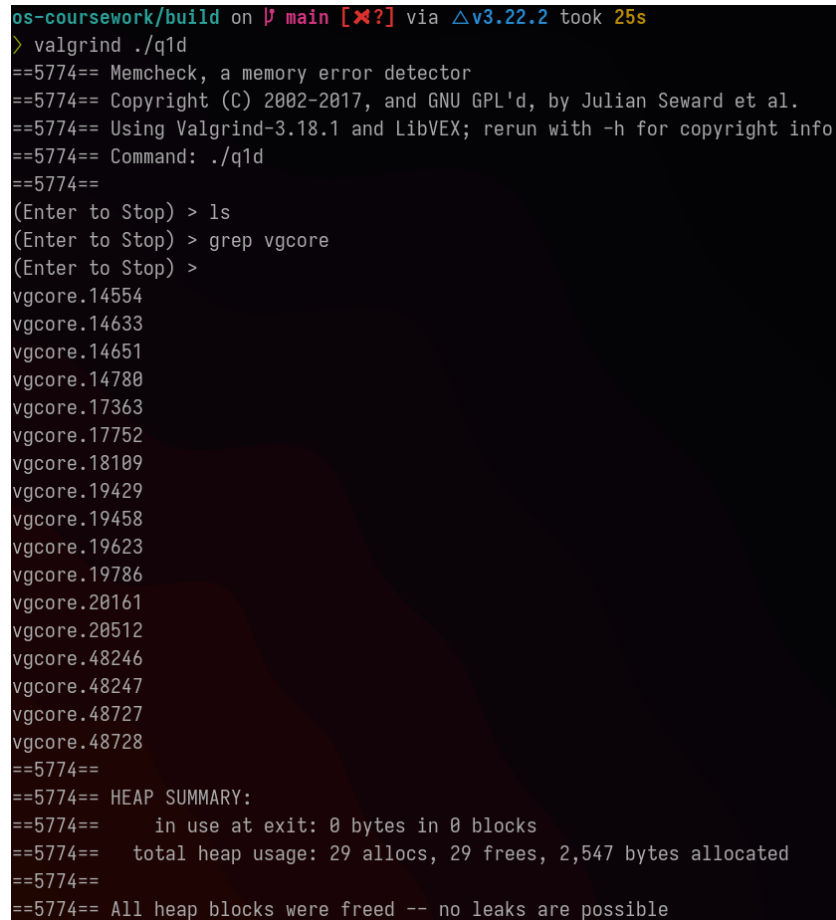
The `execute()` function has two parts. The first part checks if the command is a builtin. If the command is a builtin `execute()` will call the respective builtin function. Otherwise, it will try to execute an external program.

Note: There are two scenarios where -2, which is defined as `EXIT_SHELL`, is returned. First, when the `exit()` builtin is called or second, when a forked process fails. The latter case is special because it avoids the possibility of having two concurrent shell processes where the forked shell spins because its `stdin` is bound to a pipe.

The final function is `interpret()` and it collects all of these functions together into one function which can be called in `tish.c`. Specifically, in `tish.c` there is an event loop which waits for user input and then passes the user input to `interpret()` and the process repeats until the user decides to exit.

3 Testing Methodology

3.1 Testing for Task 1



```
os-coursework/build on ' main [X?] via v3.22.2 took 25s
> valgrind ./q1d
==5774== Memcheck, a memory error detector
==5774== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==5774== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==5774== Command: ./q1d
==5774==
(Enter to Stop) > ls
(Enter to Stop) > grep vgcore
(Enter to Stop) >
vgcore.14554
vgcore.14633
vgcore.14651
vgcore.14780
vgcore.17363
vgcore.17752
vgcore.18109
vgcore.19429
vgcore.19458
vgcore.19623
vgcore.19786
vgcore.20161
vgcore.20512
vgcore.48246
vgcore.48247
vgcore.48727
vgcore.48728
==5774==
==5774== HEAP SUMMARY:
==5774==   in use at exit: 0 bytes in 0 blocks
==5774== total heap usage: 29 allocs, 29 frees, 2,547 bytes allocated
==5774==
==5774== All heap blocks were freed -- no leaks are possible
```

Figure 10: A screenshot of the `q1d` executable running under `valgrind`.

For testing, all functions developed in *Task 1* were isolated in their own file, with a simple main function.

The main function contains a simple prompt which allows for user input. Commands were entered and the output was noted and compared to the expected behaviour.

Moreover, `valgrind` was used to make sure that no memory was being leaked and no errors were being reported. Moreover, all compiler warnings were enabled (see the `CMakeLists.txt` file).

Note: To properly test for blocking, a command like `sleep 10` was used (see figure 5). Moreover, testing of the answer for question (e) was done with hard coded file names (see functions `test1()` and `test2()` in `src/task1/qe.c`).

3.2 Testing for Task 2

```

==13108== Command: ./q2ab
==13108==
> cd ..
> ls
build      report.aux  report.blg      report.log  report.run.xml  r
images     report.bbl  report.fdb_latexmk  report.lol  report.synctex.gz  s
README.md  report.bcf  report.flx      report.pdf  report.tex
> cd src
> cwd
/home/juan/Desktop/os-coursework/src
> ver
Author: Juan Scerri
Version: 0.1
Message: Async was successful. You can glitch into the Backrooms.
> exit
==13108==
==13108== HEAP SUMMARY:
==13108==    in use at exit: 26 bytes in 3 blocks
==13108== total heap usage: 56 allocs, 53 frees, 3,232 bytes allocated
==13108==
==13108== LEAK SUMMARY:
==13108==    definitely lost: 0 bytes in 0 blocks
==13108==    indirectly lost: 0 bytes in 0 blocks
==13108==    possibly lost: 0 bytes in 0 blocks
==13108==    still reachable: 26 bytes in 3 blocks
==13108==    suppressed: 0 bytes in 0 blocks
==13108== Rerun with --leak-check=full to see details of leaked memory
==13108==
==13108== For lists of detected and suppressed errors, rerun with: -s
==13108== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
os-coursework/build on p main [✖?] via v3.22.2 took 10s
>

```

Figure 11: A screenshot of the q2ab executable running under valgrind.

Testing for *Task 2* was performed using a similar method to *Task 1*.

Note: The implementation of `exit` in this task leaks memory because it is using `exit()` with freeing any potentially allocated data. This has been addressed in the actual shell.

3.3 Testing for Task 3

Testing for *Task 3* is a bit more complicated because it requires a lot of debug output to ensure that tokenisation, parsing and translation into statements are being done correctly.

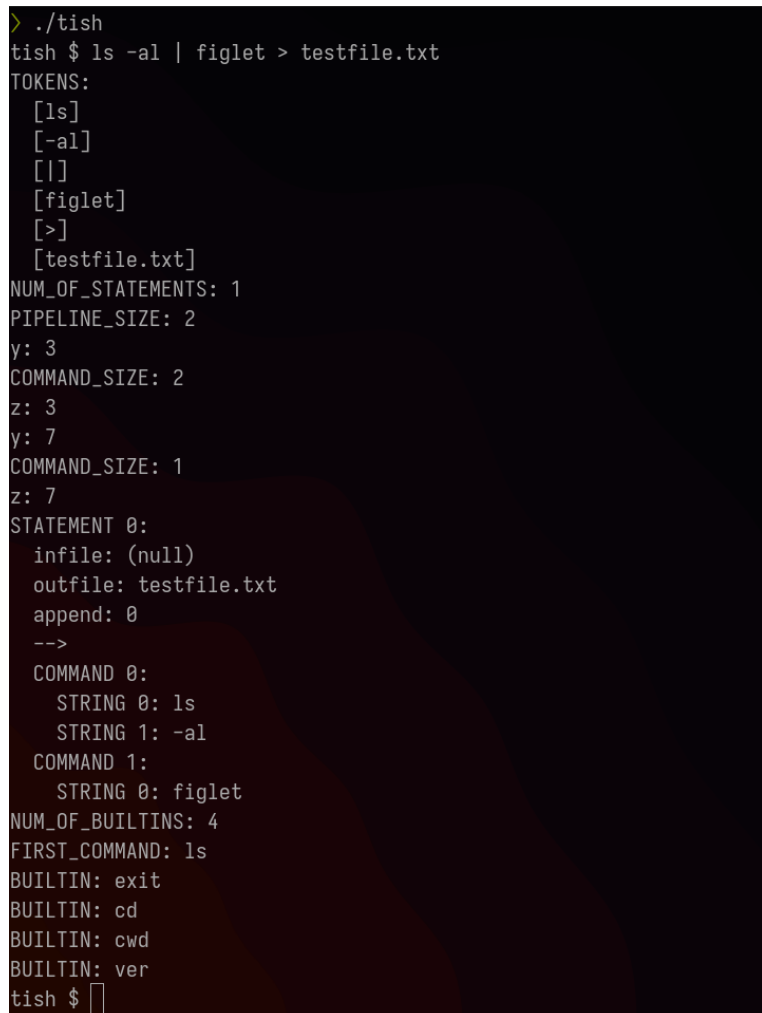
```
// Define DEBUG to get debugging information from stderr.  
// You can use redirection to output the errors into a file.  
/* #define DEBUG */
```

Listing 7: Comment indicating the usage of `#define DEBUG`.

```
#ifdef DEBUG  
    printTokens();  
#endif
```

Listing 8: Usage of the `DEBUG` definition for debugging.

To do this, a number of functions calls (`fprintf(stderr, ...)`) are littered throughout the code. They can be activated by defining `DEBUG` in the `interpreter.c` source file and recompiling.



```
> ./tish  
tish $ ls -al | figlet > testfile.txt  
TOKENS:  
[ls]  
[-al]  
[|]  
[figlet]  
[>]  
[testfile.txt]  
NUM_OF_STATEMENTS: 1  
PIPELINE_SIZE: 2  
y: 3  
COMMAND_SIZE: 2  
z: 3  
y: 7  
COMMAND_SIZE: 1  
z: 7  
STATEMENT 0:  
  infile: (null)  
  outfile: testfile.txt  
  append: 0  
  -->  
COMMAND 0:  
  STRING 0: ls  
  STRING 1: -al  
COMMAND 1:  
  STRING 0: figlet  
NUM_OF_BUILTINS: 4  
FIRST_COMMAND: ls  
BUILTIN: exit  
BUILTIN: cd  
BUILTIN: cwd  
BUILTIN: ver  
tish $
```

Figure 12: A screenshot of the `tish` executable running with `DEBUG` defined.

Note: A list of test input can be found at `src/task3and4/test.txt`. There are ten test inputs. They are all identified as being either valid or invalid.

4 Issues & Bugs

4.1 Issues

There are a number off issues both with regards to the design of the code and the functionality of the shell.

4.1.1 Design Issues

- The way in which the shell exits could be improved through the use of an exit handler.
- A revamped interpreter, generating an actual Abstract Syntax Tree (AST) could help both readability and versatility. Along with allowing the shell to conform to POSIX-like behaviour. For example the behaviour of `bash` and `tish` are very different for the following command `figlet | sed "s/ */g" < infile.txt`.
- The current implementation of `tish`, especially the interpreter relies heavily on heap-allocated memory. It would be beneficial if heap-allocations are reduced both for performance and code readability, as often times control flow is obscured because of constant validation and error checking.

4.1.2 Functionality Issues

- As already mentioned the current shell does not conform to a `bash`-like syntax. So, improving compatibility with already existing shells would be beneficial.
- Currently, `Ctrl-C` causes the whole shell to exit even whilst running a program like `figlet` in `stdin`. This should be changed to simply kill the current blocking program in the shell. This could either be done by registering a signal handler for `SIGINT` or by entering into raw mode so the bytes can be handled directly.

4.2 Bugs

From the testing I have done, I have not found any bugs or unexpected behaviour in the final binary i.e. `tish`.

As I mentioned throughout the document there were some minor issues (e.g. memory leaks, zombies etc.) which where present in the earlier versions of the code.