CPS1012 — Operating Systems & Systems Programming 1

Coursework

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

June 27, 2022

Contents

List of Figures

Listings

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	June 27, 2022
Student's full name	**Study-unit code**	**Date of submission**

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

Student's signature

2 Structure & Design

2.1 Structure

The coursework was broken up into four tasks.

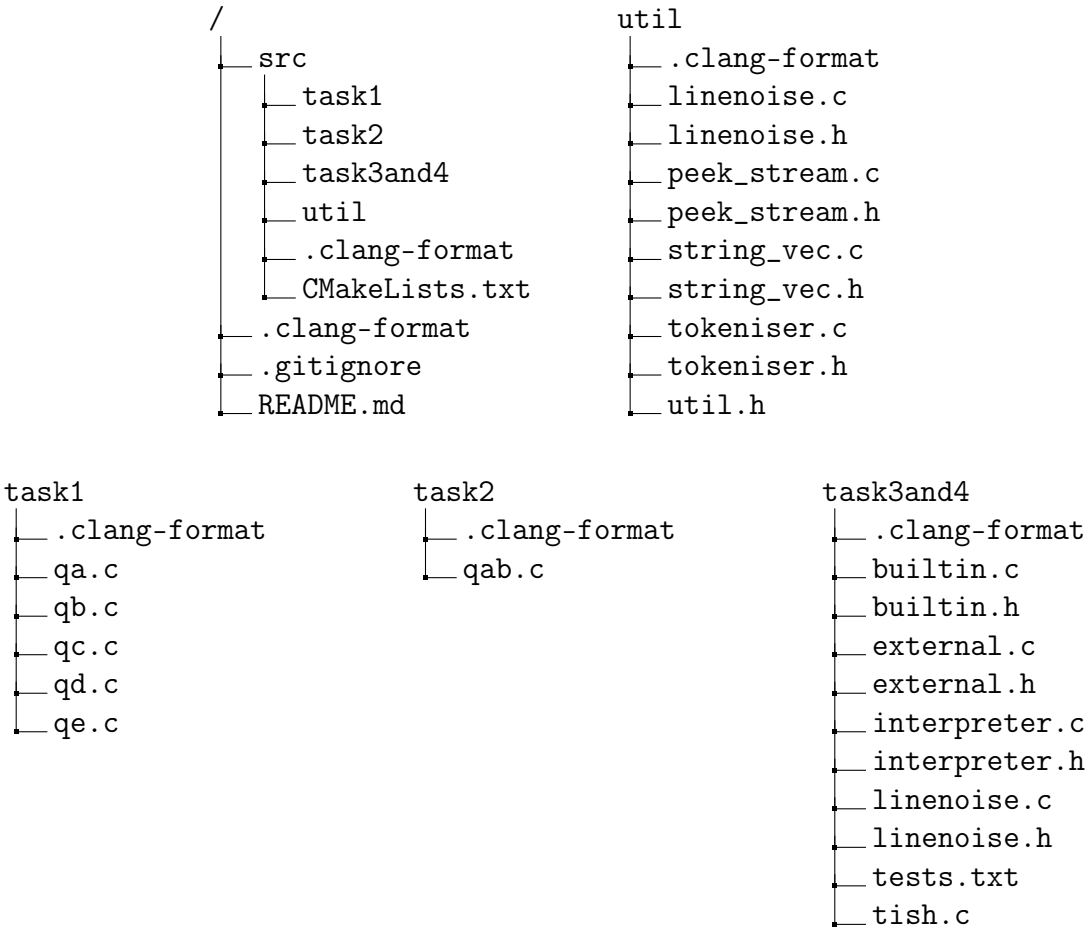
All the questions of *Task 1* have been answered separately, each having there own `.c` file.

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 illustrations. Where necessary supplementary explanation will be provided and code snippets will be inserted.

2.2.1 Design of Task 1

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

```
if ((pid = fork()) == -1) {
  return -1;
} else if (pid > 0) {  // Parent
  if (waitpid(pid, &status, 0) == -1) {
    return -1;
  }
} else {  // Child
  if (execvp(args[0], args) == -1) {
    return -1;
  }
}
```

Listing 1: An implementation 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.

Figure 2: An illustration of the pattern used to connect two processes through a pipe.

The pattern described in 2 is used to allow two processes to communicate with each other. This communication is facilitated by a pipe. Pipes are an Inter-Process Communication (IPC) mechanism provided by the OS.

```
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 2: Configuration of a pipe after `fork()` and before `execvp()`.

Note: Configuration of the pipes is done after `fork()` and before `execvp()` (or any other variant of `exec()`).

Figure 3: An illustration of the start and middle patterns used to construct an arbitrary pipeline.

Figure 4: An illustration of the end pattern used to construct an arbitrary pipeline.

For question (c), a new variant was created called `execute_pipeline(char **)`. As input, it takes in a `NULL`-terminated array of programs (including there arguments).

The function constructs an arbitrary pipeline which allows every program to communicate with the next program. This is achieved through the patterns illustrated in figures 3 and 4.

```
pid_t execute_pipeline(char **pipeline[]) {
  int current_fd[2];
  int previous_fd[2];
  pid_t pid = -1;

  for (size_t i = 0; pipeline[i] != NULL; i++) {
    // Create a new pipe only if the current process is not
    // the last.
    if (pipeline[i + 1] != NULL) {
      if (pipe(current_fd) == -1)
        return -1;
    }

    if ((pid = fork()) == -1)
      return -1;

    if (pid == 0) {  // Child
      // Connect the read-end of previous pipe only if the
      // current process is not the last.
      if (i > 0) {
        // Disconnect write-end of previous pipe.
        close(previous_fd[WR]);
        // Connect read-end of previous pipe to stdin.
        if (dup2(previous_fd[RD], STDIN_FILENO) == -1)
          return -1;

        close(previous_fd[RD]);
      }

      // Connect the write-end to a pipe only if the current
      // process is not the last.
      if (pipeline[i + 1] != NULL) {
        // Disconnect read-end of current pipe.
        close(current_fd[RD]);
        // Connect write-end of current pipe to stdout.
        if (dup2(current_fd[WR], STDOUT_FILENO) == -1)
          return -1;
```

```
        close(current_fd[WR]);
      }

      if (execvp(*pipeline[i], pipeline[i]) == -1)
        return -1;
    }

    // Only disconnect parent from previous pipe if there
    // is more than 1 pipe i.e. more than 1 process.
    if (i > 0) {
      close(previous_fd[RD]);
      close(previous_fd[WR]);
    }

    previous_fd[RD] = current_fd[RD];
    previous_fd[WR] = current_fd[WR];
  }

  return pid;
}
```

Listing 3: The implementation of the described mechanism for constructing a pipeline.

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

For the actual implementation the middle pattern is augmented with `if`-statements to cater for the start and end patterns.

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.

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

Listing 4: `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).

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

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.

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

Listing 5: 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 6: Redirecting pipeline output.

For question (e), a simple addition of three extra parameters and two `if`-statements in the start and end stages 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

```
typedef int (*builtin_t)(char **);

typedef struct {
  char *name;
  builtin_t func;
} builtin_command_t;

builtin_command_t builtins[] = {{"exit", &sh_exit},
                                {"cd", &sh_cd},
                                {"cwd", &sh_cwd},
                                {"ver", &sh_ver}};
```

Listing 7: Structure to support builtin commands.

For question (a), 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.

```
  for (size_t i = 0; i < get_num_of_builtins(); i++) {
    if (!strcmp(args[0], builtins[i].name)) {
      return builtins[i].func(args);
    }
  }
```

Listing 8: `for`-loop to check if the command is a builtin.

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`.

Note: In *Task 3* and *Task 4* `sh_exit()` was changed. These changes were made to ensure proper termination (i.e. adequate relinquishing of allocated resources).

```
#define AUTHOR "Juan Scerri"
#define VERSION "0.1"
#define MESSAGE                                          \
  "Async was successful. You can glitch into the " \
  "Backrooms."
```

Listing 9: The `#define`s for `sh_ver()`.

`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 `#define`s.

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

char cwd[PATH_MAX];
```

Listing 10: The declaration of the global `cwd`.

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

Listing 11: 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

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

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

The main function contains a simple prompt which allows for user input. Commands where 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 where being report. 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

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 12: Comment indicating the usage of `#define DEBUG`.

```
      prevTokType = tok.type;

    // The first token must always be a command or a
```

Listing 13: 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.

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.