Parallel Program Running
Practical 5- SP

Due 9pm Friday 21th April
This practical is worth 25% of your coursework mark.

1 Submission

Submit your program as a zip file containing your source, and the output of your program under `stacscheck`. Submit a short (maximum three pages) report covering your design, implementation and testing, and any problems encountered and lessons learned as PDF.

You should write a `Makefile`, with a `clean` rule, and a rule to build the program in each section.

Warning: This practical involves creating processes, creating and (possibly) deleting files. Be careful to back up your work.

You should test your program using `stacscheck`. Name the directory your practical is in 'Practical-SP'. The tests can be run by:
`stacscheck /cs/studres/CS2002/Practicals/Practical-SP/stacscheck`

2 Introduction

In this practical, you will write a program which runs a series of commands from a file. You will perform some simple 'shell' redirections, and also run commands in parallel.

Example – Given the input file:

```
./prog1 1 2 3
./prog2 1 "2 3"
./prog3 1 2 3 > output.txt
```

Your program will:

- Run the program `./prog1` with 3 arguments, `1`, `2` and `3`.
- Run the program `./prog2` with 2 arguments, `1` and `2 3`
- Run the program `./prog3` with 3 arguments, `1`, `2` and `3`, and send the output of the program to a file called `output.txt`.

A complete description of how your program should parse it's input is given in the next part.

3 Part 1 - Parsing Shell

Standard unix shell is a very complicated language with many strange rules. Here are the pieces you should parse for this practical. You may assume any input line will have no more than 1024 bytes. Note there are many strings this will parse differently from `bash`.

3.0.1 Splitting

Firstly, scan string, splitting at each space. As an exception, if you see `"`, ignore any spaces until the next `"`.

Examples:

- `a b c` splits to `["a","b","c"]`
- `"a b" c` splits to `["a b","c"]`
- `"a b " c` splits to `["a b ","c"]`
- `"a b "c d` splits to `["a b c","d"]`
- `c"a t"d e"f"g` splits to `["ca td", "efg"]`

3.0.2 Handling tokenized string

Now your string is split into a series of substrings:

- The first item is the program to run.
- If any token is `>`, then the next item is a file to write the output of the program into. Do not pass the `>` or the next argument to the program (there will be at most one `>`).
- If any token is `<`, then the next item is a file to read the input of the program from. Do not pass the `>` or the next argument to the program (there will be at most one `<`).

All remaining items are the command line arguments.

You should write a program called `shellparse` which reads a single line from standard in, and prints out an English description of the command to be run. This should be of the following form (all on one line, sentences separated by full stops):

1. `Run program`.
2. The program name in quotes.
3. If there is one argument, then `with argument` followed by the argument in quotes.

4. If there is more than one argument, then `with arguments` followed by the arguments in quote separated by `and`.

5. If there is a redirection to a file, then `Write the output into the file` followed by the filename in quotes.

6. If there is a redirection from a file, then `Read the input from the file` followed by the filename in quotes.

Examples:

- `./prog`
  Run program `"./prog"`.

- `./prog cat dog`
  Run program `"./prog"` with arguments `"cat"` and `"dog"`.

- `./prog cat > fish dog`
  Run program `"./prog"` with arguments `"cat"` and `"dog"`. Write the output into the file `"fish"`.

- `./prog cat < cow dog`
  Run program `"./prog"` with arguments `"cat"` and `"dog"`. Read the input from the file `"cow"`.

- `"./prog 2" x`
  Run program `"./prog 2"` with argument `"x"`.

- `./prog a b "c d" > e f`
  Run program `"./prog"` with arguments `"a"` and `"b"` and `"c d"` and `"f"`. Write the output into the file `"e"`.

- `./prog "c d"g > e f`
  Run program `"./prog"` with arguments `"c dg"` and `"f"`. Write the output into the file `"e"`.

You may implement other shell features (consider `|` for joining processes). Any other features you implement should be discussed in your report.

4 Part 2 - Executing Commands

Based on your parsing from Part 1, write a program called `runcmds`, which reads a series of commands from stdin, and executes each one in turn. Wait until each command is finished before starting the next one. Any commands which does not redirect output should just print to the screen. Close stdin for any program which is does not redirect input from a file.

Use the following error messages, replacing `filename` with the file that failed. After printing an error, continue executing future lines.

- If trying to open a file for reading fails: `Read failed: filename`.
- If trying to open a file for writing fails: `Write failed: filename`.
- If trying to execute a command fails: `Execute failed: filename`.

HINTS

To control stdin and stdout, look at the `dup2` function.

5 Part 3 - Parallelisation

Extend your program from part 2 to a new program `runparallelcmds`, which runs the commands in parallel. Your program `runparallelcmds` should accept a single optional argument of the form `runparallelcmds -j cpus`, where `cpus` is the number of processes to run in parallel. You should put the output of all programs which output to the screen in the same order as they originally ran (so the output is exactly the same as Part 2, just programs run in parallel). Programs which try to read from stdin will "share" input from the keyboard – this is fine.

HINTS

Consider putting output of programs into a temporary file when they should output to the screen, and then read these files after the programs are finished.

While stacscheck will make sure your parallelisation works correctly, it will not be able to check if your code is running multiple instances in parallel (so a correct answer for Part 2 which interprets the `-j` option will pass Part 3). To test it, consider using programs which call `sleep`, and then use the program `time` to check how long your program takes to run.

6 Extensions

Implement a larger selection of the "bash shell", for example `|` (piping), `;` (multiple commands) , or `( )` (subshells).

7 Policies and Guidelines

If the credit weighting and due date are different from those on MMS, the information on MMS is to be taken as definitive. If you detect a discrepancy please inform the responsible lecturer and level co-ordinator.

7.1 Marking

See the standard mark descriptors in the School Student Handbook: `http://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/feedback.html#Mark_Descriptors`. As a guideline, completing part 1 will achieve a mark of 11, parts 1 and 2 a mark of 14 and parts 1,2 and 3 a mark of 16.

7.2 Lateness penalty

The standard penalty for late submission applies (Scheme B: 1 mark per 8 hour period, or part thereof): `http://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/assessment.html#lateness-penalties`

7.3 Good academic practice

The University policy on Good Academic Practice applies: `https://www.st-andrews.ac.uk/students/rules/academicpractice/`