CSCB09 2024 Summer – Assignment 3
Due: July 28, 11:59 PM
This assignment is worth 10% of the course grade.

In this assignment, you will practice using Unix system calls for running programs and file redirection.

As usual, you should aim for reasonably efficient algorithms and reasonably well-organized, well-factored, comprehensible code.

Code correctness (mostly auto-marking) is worth 90% of the marks; code quality is worth 10%.

# Build Your Own Shell

In this assignment, you will implement a toy shell that interprets (runs) a small subset of shell commands. Overview: 3 cases: (1) echo, (2) fork-exec a program, (3) list of commands. Moreover, in each case, at each level, there may be stdout redirection. Thus there can be an example equivalent to:

```
{
  echo A
  { ls -l ; echo B > f1 ; cat f1 ; } > f2
  echo C
}
```

So every time you redirect, you must expect to restore later, and there can be many nested levels.

Commands are represented by `struct cmd` in byos.h. The interpreter function you will implement is

```
int interp(const struct cmd *c)
```

Its behaviour and return value are as follows.

## Command Semantics (Behaviour)

Firstly, each command may optionally specify stdout redirection to a pathname. In this case, the file should be opened for write-only, truncated if it already exists, created if not. (You are encouraged to use `open` or `creat` with octal number 0666 for the mode argument, rather than `fopen`.) If opening fails, the command should not be run, and the return value must be 1 (analogous to real shells). If opening succeeds, then perform stdout redirection, *taking care to save* the original stdout so you can restore later.

And then, the command is one of 3 types:

- Echo: For simplicity, there is only one C string in `struct echo_d` to print to stdout; if newlines are intended, they are already in the string. You may assume that writing succeeds.

  You are encouraged to use `write` rather than `printf` or `puts`—since we are doing stdout redirection left right and centre, bookkeeping data of `stdio.h` functions may become terribly invalid.

  The return value of this case is 0. (Optional: You may return 1 if you detect that writing fails, analogous to real shells.)

- Forx: This is the fork-exec case. `struct forx_d` has the program pathname and the command line arguments, in a format ready for straight passing to a suitable exec syscall (which is not `execlp`). Please also choose one that can search `PATH` so that the pathname can be simply `"ls"` for example.

  You may assume that `fork` succeeds.

  In the child, if `exec` fails, you may print an error message of your choice to stderr. Then the child must exit (why?), and the exit status must be 127 (analogous to real shells).

  The parent must use `wait` to wait for the child to terminate. Then the return value must be:

  - If the child exits, the return value is the child's exit status.
  - If the child is killed by signal, the return value is 128+signal (analogous to real shells).
  - You may ignore the other 2 cases (STOP and CONT).

- List: Multiple commands in an array to run sequentially; wait for one to finish before running the next. (If you use a recursive call, this is trivial.) The return value must be the return value from running the last command; if $n = 0$, so there is no command, the return value must be 0.

After the command is done, restore the original stdout (if redirection was done).

## Resource Leakage

There are a few resource limits and leakages to watch out for.

Forking: Only the Forx case should cause forking. There will be test cases under tight limits so unnecessary forking will cause failure.

File descriptors: Under intensive file redirections and execs, it is easy to be unaware that temporary or obsolete FDs should be closed ASAP. Some sources come from:

- When redirecting stdout to an opened file.
- When restoring a previously saved original stdout.
- Before/Upon exec, tons of saved original stdouts should not be kept. (Hint: "close on exec" is extremely handy, see `fcntl()` and its `FD_CLOEXEC` flag.)

There will be large test cases (long lists, deep nestings) run under tight FDT limits to catch FD leaks.

End of questions.