Week 7 Lab

Working with processes

Task 1. A simple command shell

Using the slides from Lecture 11 as a guide, write yourself a simple command shell. This shell should use `fork()` and `execv()` (or one of the other `exec...()` functions) to launch programs whose names the user types at the command line, and then call `waitpid()` to pause the parent shell until the child process terminates.

For the time being, it is okay if your shell only handles single-word command lines – it doesn't have to deal with commands that take parameters yet. For the argument array, you'll need to pass a `char*` array with the name of the command in the first slot and `NULL` in the second slot.

Use the `ps` command to make sure that your shell does not leave any zombie processes after execution.

Task 2. Complicated command lines

Add the ability to use command line parameters to your shell. To do this, you will need to split the input line and use it to construct an array of `char*`s. The function `strtok()` from the C string library may be helpful.

Task 3. More sophisticated process handling

To the shell you wrote in the previous Task, add the ability to run processes in the background. Your shell must scan each command line that the user types in for the character `'&'` – the `strchr()` function from the standard string library might help you here. If an `'&'` is found, it must be removed from the input and the child process must be run in the background using the `WNOHANG` flag in `waitpid()`.

Once that's done, you will need to modify your shell so that it tidies up any zombie children left in the process table whenever a command is run, and displays information about how these child processes terminated. Use `waitpid()` to wait without hanging for any child process, and if one was found, print its PID and its status (look in the `status` parameter to get this information).

If you're wondering how to display the information, background a process in `bash`, kill it (e.g. with `kill -9`), and notice the format of the information that `bash` displays next time you type a command. `bash` displays the job number, but you don't have to keep track of that – just printing the PID is sufficient.