HW 3: The UPS Shell

You will create a full interactive shell, to help the user navigate a computer's file system. Your source code will be called `upssh.c`. When it is executed, it will query the user with its prompt, that should look like this:

```
UPS> user types commands here
```

The user may type anything at all, and your program must respond in a rational way.

You must implement the following commands yourself:

- `exit`: The shell immediately terminates.
- `cd`: The shell will change the current working directory to the one specified. It may be . for the current directory, or .. for the parent directory, or a compound name with several slashes in it.
- `history`: Prints the most recent user inputs. The number of inputs to print should be stored in a constant called `HISTORY_LEN`, which will be 100. If there have been fewer than 100 inputs, it should print all of them with no empty spaces.
- `mypwd`: Prints the current working directory.

On any other input, the shell will attempt to find and execute the program. If the program name begins with a slash, it will assume that it is an absolute path and try to find the file as such. Otherwise, if it contains slashes in other locations, it will assume that it is relative to the current directory. If it has no slashes at all, it will retrieve the system path (see `getenv()` below) and append the program's name to each folder in the path, one by one, in the order given. If it finds the file but does not have permission to read or execute it, it will stop looking and print an appropriate error message. Finally, it will also print an appropriate error message if it cannot find the file at all.

Command-line arguments are separated from the program by one or more spaces. You do not need to worry about files with spaces in their names.

If the user's command contains `&`s, you must handle it in a special way. If the command contains internal `&`s, they separate tasks that must be done simulaneously. For example:

```
UPS> ls -l&pwd
-rw-rw-r--  1 user user 83227 Jan 29 19:23  file1.txt
/home/user
-rw-rw-r--  1 user user  3547 Feb  5 12:32  file2.txt
-rw-rw-r--  1 user user 75345 Feb  8 21:01  file3.txt
UPS>
```

Each command is a separate process, and their outputs will probably interleave. You will have no control over the order in which they do. In addition, if the full command ends with an `&`, your shell must query the user for the next command before the processes are finished. (If it ends with some other character, you must wait for all programs to finish before delivering another prompt.) This may result in confusing output, such as this:

```
UPS> ls -l&
-rw-rw-r--  1 user user 83227 Jan 29 19:23  file1.txtUPS>
-rw-rw-r--  1 user user  3547 Feb  5 12:32  file2.txt
-rw-rw-r--  1 user user 75345 Feb  8 21:01  file3.txt
```

The following functions will be of use. Most of them are defined in `unistd.h`.

- `int access(const char *path, int amode)`: Tests a file for allowed actions. The second argument may be `R_OK` (tests read permission), `W_OK` (write permission), `X_OK` (execution permission), or `F_OK` (file existence). (It may also be a bitwise `|` of these values to test several conditions.)

- `int execv(const char *path, char *const argv[])`: Immediately discards this program and replaces it with a different one. The first argument is the program to run, and the second is an array of its arguments. (In the array, the first element must be the program name itself, and the final element must be `NULL`.) Usually used in conjunction with `fork()`.

- `pid_t fork(void)`: Immediately duplicates your program. *Be careful, because doing this in a loop can make your system unstable as your program instances will grow exponentially.* Returns negative on failure, 0 if this process is the duplicate, or a positive number (the child's ID) if the process is the parent.

- `char *getcwd(char *buffer, size_t size)`: Returns the current working directory. The arguments are the string to save it into, and the string's maximum size (to avoid buffer overflows).

- `char *getenv(const char *name)`: Gets the value of an environment variable, such as `PATH`. In this case, it will return a :-delimited list of all the directories on the path.

- `pid_t wait(int *wstatus)`: Puts the current process to sleep, until it is woken up by one of its children terminating. The argument points to an int that will be modified depending on how this function executes, but you will not need to inspect it. It returns the ID of the child that terminated. (Defined in `sys/wait.h`.)