ECES 338: Principles of Operating Systems

Assignment #6 Due: March 31, 1999

The **shell** is the term used for the **command line interpreter** of UNIX operating systems. That is, the shell is the program that reads and parses the user's input command line and then starts up the processes needed to carry out these command(s). Your task is to write a simplified shell for UNIX.

It is assumed that you are familiar with (a) UNIX commands such as cd, ls, date, ps, pwd, wc, etc., (b) UNIX utilities such as sort, etc., and (c) the basic functionality of the standard UNIX shell; namely, command execution in the background (i.e., the & as the command line terminator), pipes (i.e., the | symbol in the command line), and input/output redirection (i.e., the symbols < and > in the command line).

You will use the system calls fork, wait, exit, execvp, close, open, pipe, dup in your shell. You will also use a command line parser in your shell, which is available by anonymous ftp from erciyes.ces.cwru.edu. After ftp login proceeds, change your directory to 338, and download the file "parser.c" from the directory. You should include this parser code to your shell using the include directive. The parser code will parse a command line containing an arbitrary number of commands (separated by pipes), redirection of I/O, and the optional background flag. Your program will read a line of input and call the parser routine using the following statements:

```
char inline[81];
int ncmds;
.....
gets(inline);
ncmds=parsecmd(inline);
```

The parser will parse the command and set the entries of the global structure:

```
struct cmd {
        char *cmdname;          /* ptr to '\0' terminated string containing cmd */
                                /* NULL if there is no cmd */
        char *argv[10]          /* ptr's to up to 10 arguments for this cmd*/
                                /* if there are i args, argv[i] is NULL */
        char *infile;           /* NULL if no redirected input, else ptr */
        char *outfile;          /* to filename. Same for outfile. */
        int pipe;               /* ==YES means there IS a pipe after this command. */
} commands[MAXCMDS];            /* If there are i cmds on the input line */
                                /*then command[j].cmdname=NULL for all j ≥ i */
int background;                 /* ==YES means this cmd line should be executed. */
                                /* as a background command. */
```

When your shell does an execvp() system call to execute one of the commands on the command line, the command name and its arguments can be taken directly from the commands[] array. For example, the system call

```
execvp(commands[i].cmdname,&commands[i].argv[0])
```

will cause the image of the calling process to be overlayed with the executable image for the *ith* command on the command line.

The return code from parsecmd() is the number of commands contained in the command line. A negative return says that an error was encountered in parsing the command line. In this case, your shell should NOT attempt to execute any of the commands in the command line.

Using the above system calls and parsecmd(), you should write a shell which will

1. execute either a single command on a command line or two commands on a command line (separated by pipes). The commands may have arguments as well.
2. handle the arbitrary (but syntactically correct) use of redirection of input and/or output for the commands (together with the possible use of pipes).
3. implement the use of the background commands (i.e., the termination of a command by &), in which case your shell should initiate the named constants, but not wait for these commands to finish before giving its next prompt and possibly executing the next command.

Certain commands like **cd** are *not* executed by the shell by forking a child and execvp'ing. Instead, the shell itself makes the system call (in this case, **chdir**). Here are some examples of command line strings that your shell should be able to handle. Assume your shell prompt is #.

```
# date
# cc prog1.c
# date > file.txt
# ls -l > file.lst
# sort < file.lst | wc
# ls -l | sort -r
# sort file.lst | wc > count.txt
```

Start out simple and incrementally add to the functionality of your shell. Make sure that you check the return code for every system call that you make and print out an error message if an error or unexpected return code is encountered.

Once the parent process (your shell) has created a pipe and forked the two child processes that will read and write from the pipe, make sure that the parent explicitly closes the file descriptors for its read and write access to the pipe. If you fail to do this, the child process reading from the pipe will never terminate. This is because the child reading from the pipe will never get the END-OF-FILE (end-of-pipe) condition (and hence never terminate) as long as at least one process (in this case, your shell program, by mistake) has an open write file descriptor for the pipe. The overall result will be a deadlock–your shell waiting for the child to terminate and the child waiting for the shell to close the pipe.

The ps command is useful for listing all processes associated with your session. If you see a lot of processes lying around in the Z (zombie) state, you probably have a bug in your shell. Also, make sure that you consider what happens when the execvp() for the child fails. Finally, be careful with your use of the wait() system call. The shell should wait for a command to complete, unless it is to run in the background.

Turn in your code and several runs illustrating the correctness of your code. Also, you should comment your code extensively.