***List of Files***

|  |  |
| --- | --- |
| Filename | Purpose |
| Builtin.c | Source code for built-in functions for shell |
| Builtin.h | Header file for builtin.c |
| Builtin.o | Object file for linking to Unix\_Shell |
| Command.c | Source code for command breakdown for shell |
| Command.h | Header file for command.c |
| Command.o | Object file for linking to Unix\_Shell |
| Token.c | Source code to break up a string into tokens for shell |
| Token.h | Header file for token.c |
| Token.o | Object file for linking to Unix\_Shell |
| Unix\_Shell.c | Source code for the Unix shell |
| Unix\_Shell.o | Object file for Unix\_Shell |
| Unix\_Shell | Executable for Unix shell |
| Makefile | Make file to assemble compile and create executable for Unix shell |

***The project title and a brief description of the project***

Design and implement a simple UNIX shell program using the grammar specified in the [later part](https://www.it.murdoch.edu.au/~S900432D/oli1l1hsu_x3X64dg72kf7Th973yihbkys9M10in0x/assignments/a2/index.shtml#syntax) of this section. Please allow for at least 100 commands in a command line and at least 1000 arguments in each command.

***Self-diagnosis and evaluation***

Functional features:

* Reconfigurable shell prompt: the user can change the prompt from ‘%’ to a string entered after the built-in command ‘prompt’ [e.g. % prompt john$]
* Built-in command pwd: prints the current directory of the shell process.
* Directory walk: the user can change the directory location using ‘cd’ and if no path is stated will direct to the home directory.
* Wildcard characters: the shell allows wild card characters such as ‘\*’ and ‘?’ using the C function ‘glob’
* Shell pipeline: processes can be pipelined into one another using the ‘|’ indicator. Output from one process is inputted into the next.
* Background job execution: processes can be executed in the background using the ‘&’ indicator
* Sequential job execution: processes can be executed in sequential order (one after the other) with the ‘;’ indicator
* The shell environment is inherited from the parent process
* The shell has the build in command ‘exit’ which exits the shell.
* The shell cannot be terminated by the following inputs “CTRL-C”, “CTRL-\”, or “CTRL-Z”

Non-functional features:

* Standard input and output redirection: although input/output redirection works majority of the time there are instances where the commands output unintended error messages, or the redirect will not work at all.

***Discussion of your solution***

Firstly, the program set a signal mask so that SIGINT, SIGQUIT and SIGTSTP are ignored. After this, it enters a loop which only exits once a command execution returns a status of 1 (exit). In the while loop if first gets the user input from the command line. Once a string is collected it allocates memory to \*\*token. The function tokenise() is called. The tokenise() function separates the string using strtok() with “ \t\n” as the eliminators. These tokens are placed into the token array.

Next, the token is separated into commands using the function separateCommands(). SeparateCommands fills into the command data structure with the index of the first token in the token array, the index of the last token, the separator which can be “|”, “&” or “;”, the standard input and standard output file names. It finally fills in the argv array with the tokens related to the command excluding I/O redirects.

To find the first and last index of tokens it scans each token looking for a separator. If a separator is found the previous token becomes the last index. To find the I/O redirect files it similarly scans each token until it finds a “>” or “<” identifier. Once found the next token becomes the stdin/stdout file.

To pipe between processes the program first counts all pipes within the string and creates a pipe array with the size two times the count. Once the pipes are created it assigns the input and output accordingly using the following function:

        int m = 0;

        for (int i=0; i<num; i++) {

            int in = -1;

            int out = -1;

            if (i != 0) { //input

                if (strcmp(cmds[i-1].sep, pipeSep) == 0) {

                    in = pipefds[m]; // 0

                    m+=2;

                }

            }

            if (strcmp(cmds[i].sep, pipeSep) == 0) { //output

                out = pipefds[m+1]; // 1

            }

            status = execute\_command(in, out, &cmds[i]);

        }

Once assigned the correct pipes it called the execute\_command() function. This function first checks to see if there is a built-in command. If there is it executes it using a function pointer:

    for (int i=0; i<getBuiltinCount(); i++) {

        if (strcmp(cmd->argv[0], builtin\_cmd[i]) == 0) {

            return (\*exec\_builtin[i])(cmd);

        }

    }

If there is no built-in function it will fork the current environment. The child process which checks for I/O redirects and dups the standard input file descriptor and standard output file descriptor, respectively. It will then dup any pipes to the std input and std output. Once complete it will execute the command using the exevp call:

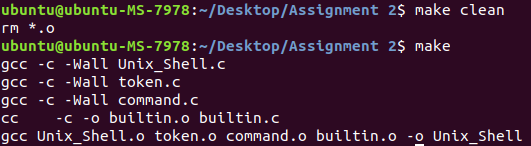
execvp(cmd->argv[0], cmd->argv);

The parent process will wait for the program to finish if it is not a background process and close the input and output pipes. It will then check for a signal for SIGCHLD which calls the killChild() function. This function kills the child so that zombie process’ are reclaimed.

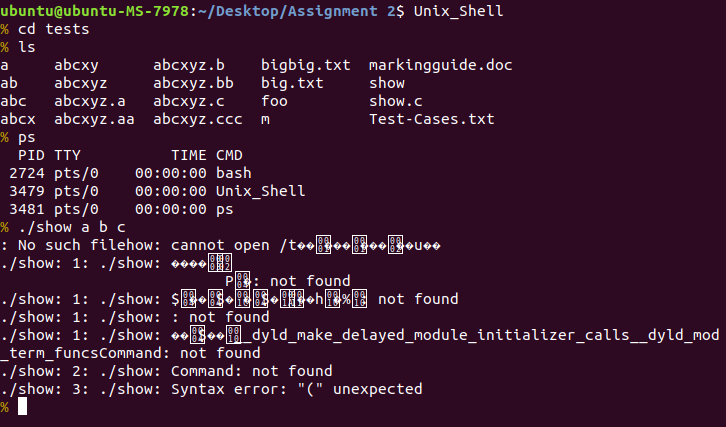
Overall the solution is pretty strong at breaking up a string into its commands however it still contains limitations. It cannot completely handle input and output redirection with pipes which is due to the duplication of file descriptors. It also has to contain a space into between each eliminator which could be improved so it’s not needed.

***Test evidence***

**#1 Compilation:** program compiled with -Wall flag without any warnings or errors



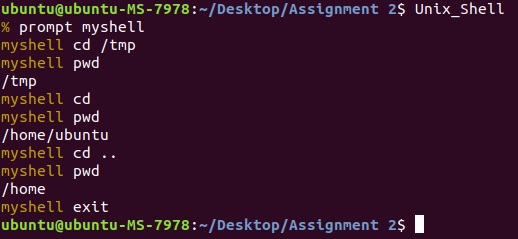
**#2 Basics:** program completed all processes except for “./show a b c”. However, when compared to the Ubuntu shell it also cannot display the file so it is unsure if this is intended behaviour or not.



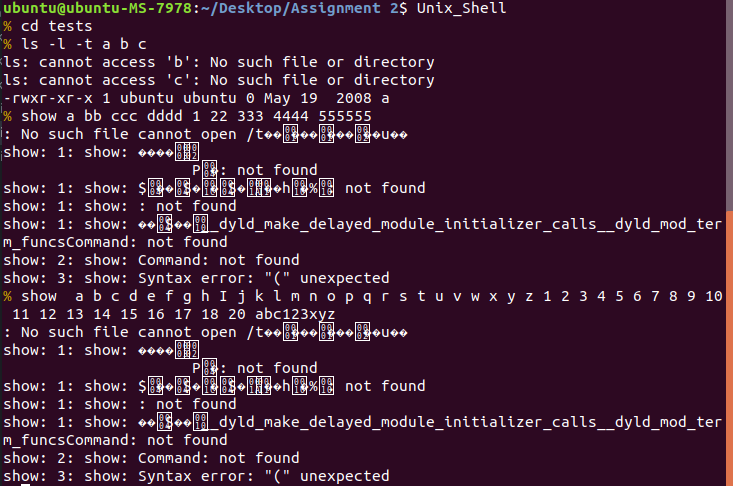
Ubuntu shell version



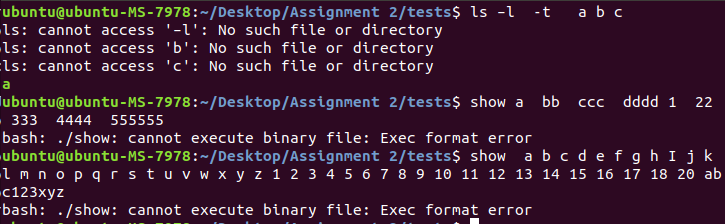
**#3 Built-in commands:** all commands executed correctly



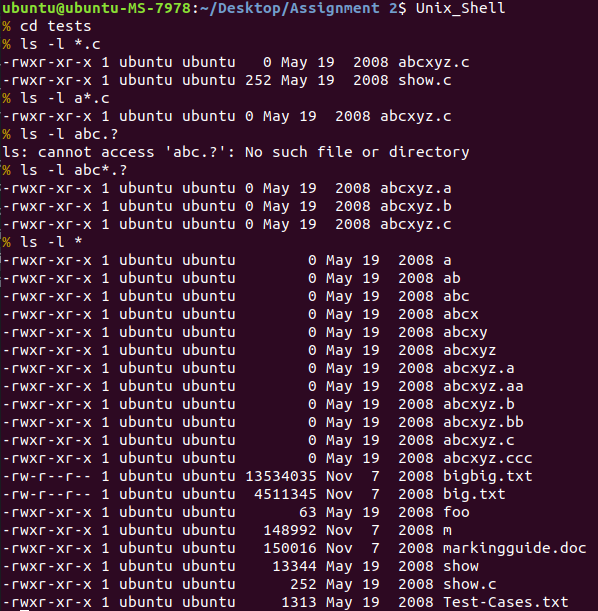
**#4 Long command:** all commands executed however the “./show” command did not which is also not able to be completed in the Ubuntu terminal, so it is unclear if this is intended behaviour.



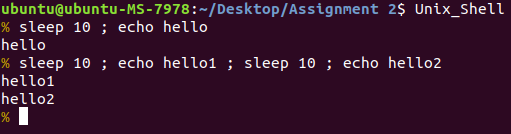
Ubuntu shell version:



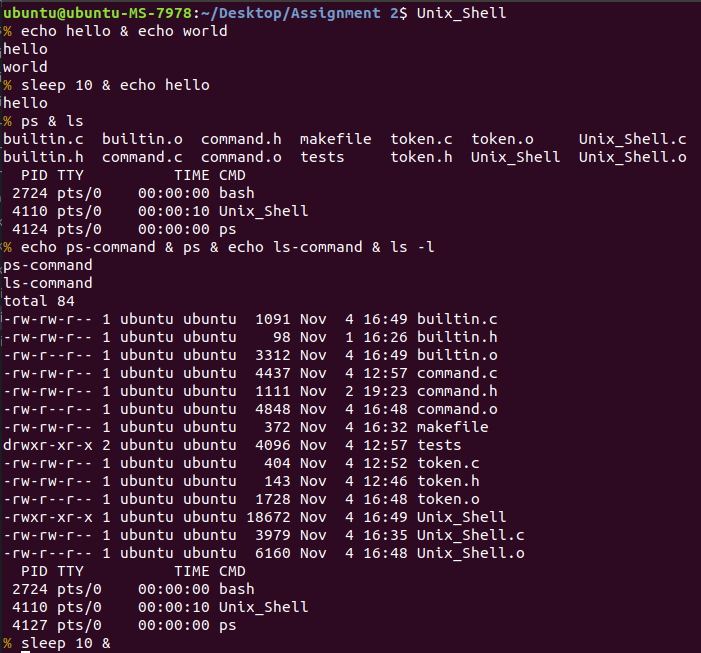
**#5 Wildcards:** all wildcard command executed correctly.



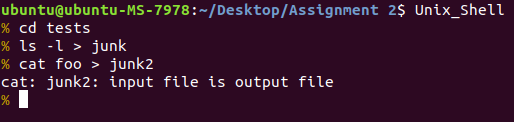
**#6 Sequential execution ‘;’:** all commands were executed sequentially.



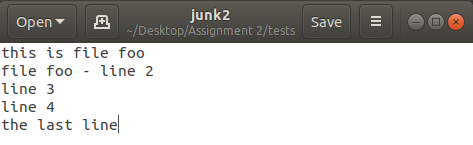
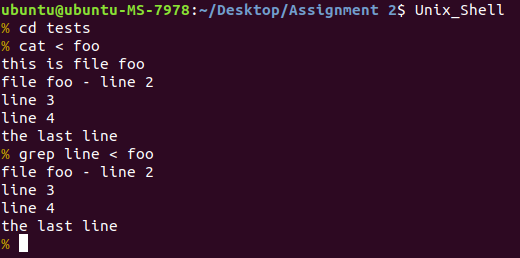
**#7 Concurrent execution:** all commands executed correctly

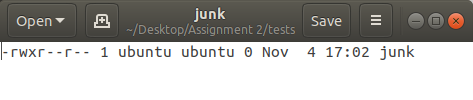


**#8 Standard input redirection ‘<’:** input redirected worked correctly except for the last command. It showed an error however the command still completed.

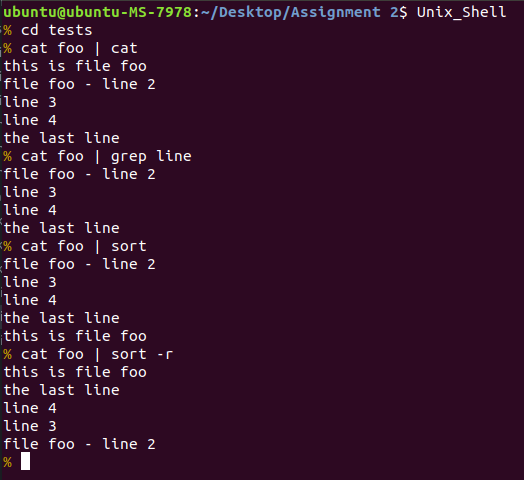


**#9 Standard output redirection ‘>’:** all command worked as intended.

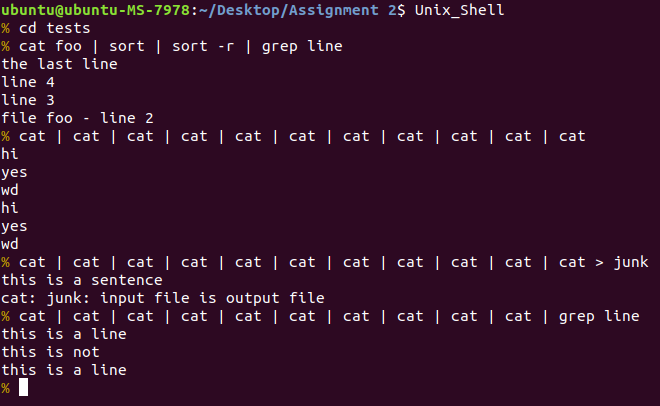




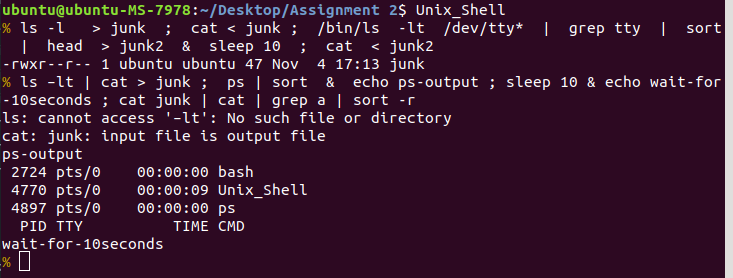
**#10 Simple shell pipeline:** all pipeline commands worked as intended.



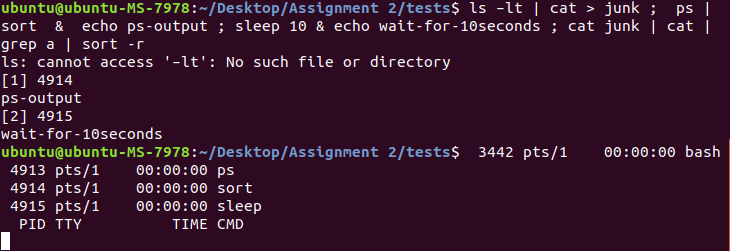
**#11 Long shell pipeline:** all commands executed correctly except for the line with output redirect. The output was not redirected to the file junk.

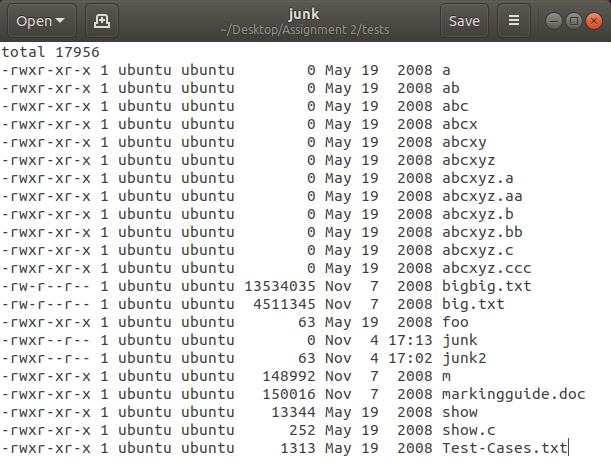


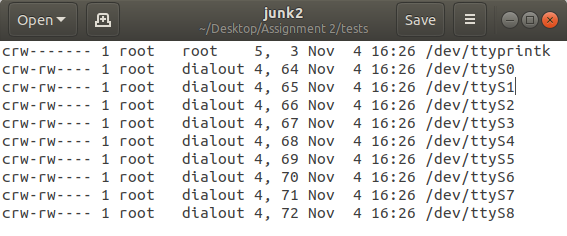
**#12 combinations:** all commands worked as intended. It states that ls -lt is not a command which is verified by the Ubuntu shell.



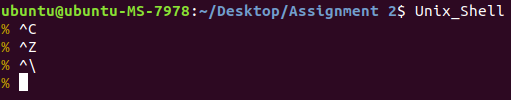
Ubuntu shell version:



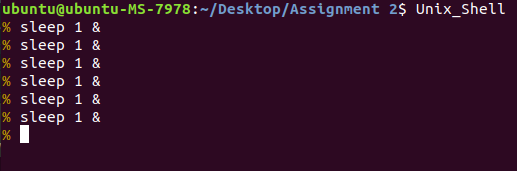




**#13 Ignore Ctrl-C, Ctrl-\ and Ctrl-Z:** all required signals were ignored

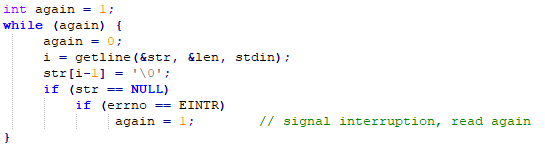


**#14 Claim zombie processes:** zombie processes were claimed and checked using the command “ps aux | grep “defunct”





**#15 Handline slow system calls:** slow system calls were handled by the following function.



***Source code listing***

Builtin.c

#include "command.h"

#include <stdio.h>

#include <string.h>

#include <unistd.h>

#include <stdlib.h>

char \*builtin\_cmd[] = {

    "prompt",

    "pwd",

    "cd",

    "exit"

};

int getBuiltinCount() {

    return sizeof(builtin\_cmd) / sizeof(char \*);

}

///change prompt

int prompt\_builtin(Command \*cmd)

{

    extern char \*prompt;

    strcpy(prompt, cmd->argv[1]);;

    return 0;

}

int pwd\_builtin(Command \*cmd)

{

    char cwd[256];

    if (getcwd(cwd, sizeof(cwd)) != NULL) {

               printf("%s\n", cwd);

    } else {

       perror("pwd err");

       return 1;

    }

    return 0;

}

int cd\_builtin(Command \*cmd)

{

    if (cmd->argv[1] == NULL) {

        if (chdir(getenv("HOME")) != 0) {

            perror("cd directory err");

        }

    }

    else

    {

        if (chdir(cmd->argv[1]) != 0) { //change directory

            perror("cd directory err");

        }

    }

    return 0;

}

int exit\_builtin(Command \*cmd)

{

    return 1;

}

//builtin pointer array

int (\*exec\_builtin[]) (Command \*cmd) = {

    &prompt\_builtin,

    &pwd\_builtin,

    &cd\_builtin,

    &exit\_builtin

};

Builtin.h

extern char \*builtin\_cmd[];

int getBuiltinCount();

extern int (\*exec\_builtin[]) (Command \*cmd);

Command.c

#include <string.h>

#include <stdlib.h>

#include <stdio.h>

#include <glob.h>

#include "command.h"

void searchRedirection(char \*token[], Command \*cp)

{

    int i;

    for (i=cp->first; i<=cp->last; ++i) {

        if (strcmp(token[i], "<") == 0) {   // standard input redirection

             cp->stdin\_file = token[i+1];

             ++i;

        } else if (strcmp(token[i], ">") == 0) { // standard output redirection

             cp->stdout\_file = token[i+1];

             ++i;

        }

    }

}

int separator(char \*token)

{

    int i=0;

    char \*commandSeparators[] = {pipeSep, conSep, seqSep, NULL};

    while (commandSeparators[i] != NULL) {

        if (strcmp(commandSeparators[i], token) == 0) {

            return 1;

        }

        ++i;

    }

    return 0;

}

void fillCommandStructure(Command \*cp, int first, int last, char \*sep)

{

     cp->first = first;

     cp->last = last - 1;

     cp->sep = sep;

}

void buildCommandArgumentArray(char \*token[], Command \*cp)

{

    glob\_t globResult;

    int n = 0;

    for (int t=cp->first; t<=cp->last; ++t ) {

        if (strcmp(token[t], ">") != 0 && strcmp(token[t], "<") != 0) {

            glob(token[t], GLOB\_NOCHECK, 0, &globResult);

            n += globResult.gl\_pathc;

            globfree(&globResult);

        }

    }

    n += 1; //null terminated

     cp->argv = (char \*\*) realloc(cp->argv, sizeof(char \*) \* n);

     if (cp->argv == NULL) {

         perror("realloc");

         exit(1);

     }

     int i;

     int k = 0;

     for (i=cp->first; i<= cp->last; ++i ) {

         if (strcmp(token[i], ">") != 0 && strcmp(token[i], "<") != 0) {

             for (int n=0; n<globResult.gl\_pathc; n++) {

                 glob(token[i], GLOB\_NOCHECK, 0, &globResult);

                 cp->argv[k] = malloc(sizeof(char) \* (strlen(globResult.gl\_pathv[n])+1));

                 strcpy(cp->argv[k], globResult.gl\_pathv[n]);

                 globfree(&globResult);

                 ++k;

             }

         }

     }

     cp->argv[k] = NULL;

}

int separateCommands(char \*token[], Command command[])

{

     int i;

     int nTokens;

     i = 0;

     while (token[i] != NULL) ++i;

     nTokens = i;

     if (nTokens == 0)

          return 0;

     if (separator(token[0]))

          return -3;

     if (!separator(token[nTokens-1])) {

        token[nTokens] = seqSep;

        ++nTokens;

     }

     int first=0;   // points to the first tokens of a command

     int last;      // points to the last  tokens of a command

     char \*sep;     // command separator at the end of a command

     int c = 0;         // command index

     for (i=0; i<nTokens; ++i) {

         last = i;

         if (separator(token[i])) {

             sep = token[i];

             if (first==last)  // two consecutive separators

                 return -2;

             fillCommandStructure(&(command[c]), first, last, sep);

             ++c;

             first = i+1;

         }

     }

     // check the last token of the last command

     if (strcmp(token[last], pipeSep) == 0) { // last token is pipe separator

          return -4;

     }

     int nCommands = c;

     for (i=0; i<nCommands; ++i) {

         searchRedirection(token, &(command[i]));

         buildCommandArgumentArray(token, &(command[i]));

     }

     return nCommands;

}

Command.h

#define MAX\_NUM\_COMMANDS  1000

// command separators

#define pipeSep  "|"

#define conSep   "&"

#define seqSep   ";"

struct CommandStruct {

   int first;      // index to the first token in the array "token" of the command

   int last;       // index to the first token in the array "token" of the command

   char \*sep;      // the command separator that follows the command,  must be one of "|", "&", and ";"

   char \*\*argv;

   char \*stdin\_file;

   char \*stdout\_file;

};

typedef struct CommandStruct Command;  // command type

int separateCommands(char \*token[], Command command[]);

Token.c

#include <string.h>

#include "token.h"

int tokenise (char line[], char \*token[])

{

      char \*tk;

      int i=0;

      tk = strtok(line, tokenSeparators);

      token[i] = tk;

      for (i=1; tk != NULL; i++) {

          if (i>=MAX\_NUM\_TOKENS) {

              i = -1;

              break;

          }

          tk = strtok(NULL, tokenSeparators);

          token[i] = tk;

      }

      return i;

}

Token.h

#define MAX\_NUM\_TOKENS  1000

#define tokenSeparators " \t\n"    // characters that separate tokens

int tokenise (char line[], char \*token[]);

Unix\_Shell.c

#include "token.h"

#include "command.h"

#include "builtin.h"

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/wait.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <errno.h>

void killChild() {

    int count = 1;

    pid\_t pid;

    int status;

    while (count) {

        pid = waitpid(-1, &status, WNOHANG);

        if (pid < 0)

            count = 0;

    }

}

int execute\_command(int input, int output, Command \*cmd) {

    if (cmd->argv == NULL) {

        printf(" NO ARGS");

        return 0;

    }

    for (int i=0; i<getBuiltinCount(); i++) {

        if (strcmp(cmd->argv[0], builtin\_cmd[i]) == 0) {

            return (\*exec\_builtin[i])(cmd);

        }

    }

    pid\_t pid;

    int status;

    if ((pid = fork()) < 0) {

        perror("fork");

        exit(1);

    }

    else if (pid == 0) {

        if (cmd->stdin\_file != NULL) { //redirect input

            int fd0 = open(cmd->stdin\_file, O\_RDONLY, 0);

            dup2(fd0, STDIN\_FILENO);

            close(fd0);

        }

        if (cmd->stdout\_file != NULL) { //redirect ouput

            int fd1 = open(cmd->stdout\_file, O\_WRONLY|O\_CREAT, 0766);

            dup2(fd1, STDOUT\_FILENO);

            close(fd1);

        }

        if (input != -1)

            dup2 (input, 0);

        if (output != -1)

            dup2 (output, 1);

        execvp(cmd->argv[0], cmd->argv);

        exit(1);

    } else {

        if (strcmp(cmd->sep, conSep) != 0) {

            wait(&status);

        }

        if (input != -1)

            close (input);

        if (output != -1)

            close (output);

        signal(SIGCHLD, killChild);

        return 0;

    }

}

char \*prompt;

int main(void)

{

    sigset\_t sigs;

    sigemptyset(&sigs);

    sigaddset(&sigs, SIGINT);

    sigaddset(&sigs, SIGQUIT);

    sigaddset(&sigs, SIGTSTP);

    sigprocmask(SIG\_SETMASK, &sigs, NULL);

    if (!(prompt = malloc(256 \* sizeof(char))))

        return 1;

    strcpy(prompt, "%");

    char \*str;

    size\_t len = 256;

    size\_t i = 0;

    int status;

    do {

       printf("\033[0;33m");

       printf("%s ", prompt);

       printf("\033[0m");

       int again = 1;

       while (again) {

            again = 0;

            i = getline(&str, &len, stdin);

            str[i-1] = '\0';

            if (str == NULL)

                if (errno == EINTR)

                     again = 1;        // signal interruption, read again

       }

        char \*\*token = calloc(1000, sizeof(char\*));

        tokenise(str, token);

        Command cmds[MAX\_NUM\_COMMANDS];

        for (int i=0; i<MAX\_NUM\_COMMANDS; i++)

        {

            cmds[i].first = 0;

            cmds[i].last = 0;

            cmds[i].sep = NULL;

            cmds[i].argv = NULL;

            cmds[i].stdin\_file = NULL;

            cmds[i].stdout\_file = NULL;

        }

        int num = separateCommands(token, cmds);

        //get pipe count

        int pipeCount = 0;

        for (int x=0; x<num; x++) {

            if (strcmp(cmds[x].sep, pipeSep) == 0) {

                pipeCount += 1;

            }

        }

        //initialise pipes

        int pipefds[2\*pipeCount];

        for(int i = 0; i < pipeCount; i++ ){

            if( pipe(pipefds + i\*2) < 0 ){

            }

        }

        int m = 0;

        for (int i=0; i<num; i++) {

            int in = -1;

            int out = -1;

            if (i != 0) { //input

                if (strcmp(cmds[i-1].sep, pipeSep) == 0) {

                    in = pipefds[m]; // 0

                    m+=2;

                }

            }

            if (strcmp(cmds[i].sep, pipeSep) == 0) { //output

                out = pipefds[m+1]; // 1

            }

            status = execute\_command(in, out, &cmds[i]);

        }

    } while (!status);

    return(0);

}