\_atoi.c

#include "shell.h"

/\*\*

\* interactive - returns true if shell is interactive mode

\* @info: struct address

\*

\* Return: 1 if interactive mode, 0 otherwise

\*/

int interactive(info\_t \*info)

{

return (isatty(STDIN\_FILENO) && info->readfd <= 2);

}

/\*\*

\* is\_delim - checks if character is a delimeter

\* @c: the char to check

\* @delim: the delimeter string

\* Return: 1 if true, 0 if false

\*/

int is\_delim(char c, char \*delim)

{

while (\*delim)

if (\*delim++ == c)

return (1);

return (0);

}

/\*\*

\* \_isalpha - checks for alphabetic character

\* @c: The character to input

\* Return: 1 if c is alphabetic, 0 otherwise

\*/

int \_isalpha(int c)

{

if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z'))

return (1);

else

return (0);

}

/\*\*

\* \_atoi - converts a string to an integer

\* @s: the string to be converted

\* Return: 0 if no numbers in string, converted number otherwise

\*/

int \_atoi(char \*s)

{

int i, sign = 1, flag = 0, output;

unsigned int result = 0;

for (i = 0; s[i] != '\0' && flag != 2; i++)

{

if (s[i] == '-')

sign \*= -1;

if (s[i] >= '0' && s[i] <= '9')

{

flag = 1;

result \*= 10;

result += (s[i] - '0');

}

else if (flag == 1)

flag = 2;

}

if (sign == -1)

output = -result;

else

output = result;

return (output);

}

==========================

builtin.c

#include "shell.h"

/\*\*

\* \_myexit - exits the shell

\* @info: Structure containing potential arguments. Used to maintain

\* constant function prototype.

\* Return: exits with a given exit status

\* (0) if info.argv[0] != "exit"

\*/

int \_myexit(info\_t \*info)

{

int exitcheck;

if (info->argv[1]) /\* If there is an exit arguement \*/

{

exitcheck = \_erratoi(info->argv[1]);

if (exitcheck == -1)

{

info->status = 2;

print\_error(info, "Illegal number: ");

\_eputs(info->argv[1]);

\_eputchar('\n');

return (1);

}

info->err\_num = \_erratoi(info->argv[1]);

return (-2);

}

info->err\_num = -1;

return (-2);

}

/\*\*

\* \_mycd - changes the current directory of the process

\* @info: Structure containing potential arguments. Used to maintain

\* constant function prototype.

\* Return: Always 0

\*/

int \_mycd(info\_t \*info)

{

char \*s, \*dir, buffer[1024];

int chdir\_ret;

s = getcwd(buffer, 1024);

if (!s)

\_puts("TODO: >>getcwd failure emsg here<<\n");

if (!info->argv[1])

{

dir = \_getenv(info, "HOME=");

if (!dir)

chdir\_ret = /\* TODO: what should this be? \*/

chdir((dir = \_getenv(info, "PWD=")) ? dir : "/");

else

chdir\_ret = chdir(dir);

}

else if (\_strcmp(info->argv[1], "-") == 0)

{

if (!\_getenv(info, "OLDPWD="))

{

\_puts(s);

\_putchar('\n');

return (1);

}

\_puts(\_getenv(info, "OLDPWD=")), \_putchar('\n');

chdir\_ret = /\* TODO: what should this be? \*/

chdir((dir = \_getenv(info, "OLDPWD=")) ? dir : "/");

}

else

chdir\_ret = chdir(info->argv[1]);

if (chdir\_ret == -1)

{

print\_error(info, "can't cd to ");

\_eputs(info->argv[1]), \_eputchar('\n');

}

else

{

\_setenv(info, "OLDPWD", \_getenv(info, "PWD="));

\_setenv(info, "PWD", getcwd(buffer, 1024));

}

return (0);

}

/\*\*

\* \_myhelp - changes the current directory of the process

\* @info: Structure containing potential arguments. Used to maintain

\* constant function prototype.

\* Return: Always 0

\*/

int \_myhelp(info\_t \*info)

{

char \*\*arg\_array;

arg\_array = info->argv;

\_puts("help call works. Function not yet implemented \n");

if (0)

\_puts(\*arg\_array); /\* temp att\_unused workaround \*/

return (0);

}

======================

builtin1.c

#include "shell.h"

/\*\*

\* \_myhistory - displays the history list, one command by line, preceded

\* with line numbers, starting at 0.

\* @info: Structure containing potential arguments. Used to maintain

\* constant function prototype.

\* Return: Always 0

\*/

int \_myhistory(info\_t \*info)

{

print\_list(info->history);

return (0);

}

/\*\*

\* unset\_alias - sets alias to string

\* @info: parameter struct

\* @str: the string alias

\*

\* Return: Always 0 on success, 1 on error

\*/

int unset\_alias(info\_t \*info, char \*str)

{

char \*p, c;

int ret;

p = \_strchr(str, '=');

if (!p)

return (1);

c = \*p;

\*p = 0;

ret = delete\_node\_at\_index(&(info->alias),

get\_node\_index(info->alias, node\_starts\_with(info->alias, str, -1)));

\*p = c;

return (ret);

}

/\*\*

\* set\_alias - sets alias to string

\* @info: parameter struct

\* @str: the string alias

\*

\* Return: Always 0 on success, 1 on error

\*/

int set\_alias(info\_t \*info, char \*str)

{

char \*p;

p = \_strchr(str, '=');

if (!p)

return (1);

if (!\*++p)

return (unset\_alias(info, str));

unset\_alias(info, str);

return (add\_node\_end(&(info->alias), str, 0) == NULL);

}

/\*\*

\* print\_alias - prints an alias string

\* @node: the alias node

\*

\* Return: Always 0 on success, 1 on error

\*/

int print\_alias(list\_t \*node)

{

char \*p = NULL, \*a = NULL;

if (node)

{

p = \_strchr(node->str, '=');

for (a = node->str; a <= p; a++)

\_putchar(\*a);

\_putchar('\'');

\_puts(p + 1);

\_puts("'\n");

return (0);

}

return (1);

}

/\*\*

\* \_myalias - mimics the alias builtin (man alias)

\* @info: Structure containing potential arguments. Used to maintain

\* constant function prototype.

\* Return: Always 0

\*/

int \_myalias(info\_t \*info)

{

int i = 0;

char \*p = NULL;

list\_t \*node = NULL;

if (info->argc == 1)

{

node = info->alias;

while (node)

{

print\_alias(node);

node = node->next;

}

return (0);

}

for (i = 1; info->argv[i]; i++)

{

p = \_strchr(info->argv[i], '=');

if (p)

set\_alias(info, info->argv[i]);

else

print\_alias(node\_starts\_with(info->alias, info->argv[i], '='));

}

return (0);

}

=====================

environ.c

#include "shell.h"

/\*\*

\* \_myenv - prints the current environment

\* @info: Structure containing potential arguments. Used to maintain

\* constant function prototype.

\* Return: Always 0

\*/

int \_myenv(info\_t \*info)

{

print\_list\_str(info->env);

return (0);

}

/\*\*

\* \_getenv - gets the value of an environ variable

\* @info: Structure containing potential arguments. Used to maintain

\* @name: env var name

\*

\* Return: the value

\*/

char \*\_getenv(info\_t \*info, const char \*name)

{

list\_t \*node = info->env;

char \*p;

while (node)

{

p = starts\_with(node->str, name);

if (p && \*p)

return (p);

node = node->next;

}

return (NULL);

}

/\*\*

\* \_mysetenv - Initialize a new environment variable,

\* or modify an existing one

\* @info: Structure containing potential arguments. Used to maintain

\* constant function prototype.

\* Return: Always 0

\*/

int \_mysetenv(info\_t \*info)

{

if (info->argc != 3)

{

\_eputs("Incorrect number of arguements\n");

return (1);

}

if (\_setenv(info, info->argv[1], info->argv[2]))

return (0);

return (1);

}

/\*\*

\* \_myunsetenv - Remove an environment variable

\* @info: Structure containing potential arguments. Used to maintain

\* constant function prototype.

\* Return: Always 0

\*/

int \_myunsetenv(info\_t \*info)

{

int i;

if (info->argc == 1)

{

\_eputs("Too few arguements.\n");

return (1);

}

for (i = 1; i <= info->argc; i++)

\_unsetenv(info, info->argv[i]);

return (0);

}

/\*\*

\* populate\_env\_list - populates env linked list

\* @info: Structure containing potential arguments. Used to maintain

\* constant function prototype.

\* Return: Always 0

\*/

int populate\_env\_list(info\_t \*info)

{

list\_t \*node = NULL;

size\_t i;

for (i = 0; environ[i]; i++)

add\_node\_end(&node, environ[i], 0);

info->env = node;

return (0);

}

====================

errors.c code

#include "shell.h"

/\*\*

\* \_eputs - prints an input string

\* @str: the string to be printed

\*

\* Return: Nothing

\*/

void \_eputs(char \*str)

{

int i = 0;

if (!str)

return;

while (str[i] != '\0')

{

\_eputchar(str[i]);

i++;

}

}

/\*\*

\* \_eputchar - writes the character c to stderr

\* @c: The character to print

\*

\* Return: On success 1.

\* On error, -1 is returned, and errno is set appropriately.

\*/

int \_eputchar(char c)

{

static int i;

static char buf[WRITE\_BUF\_SIZE];

if (c == BUF\_FLUSH || i >= WRITE\_BUF\_SIZE)

{

write(2, buf, i);

i = 0;

}

if (c != BUF\_FLUSH)

buf[i++] = c;

return (1);

}

/\*\*

\* \_putfd - writes the character c to given fd

\* @c: The character to print

\* @fd: The filedescriptor to write to

\*

\* Return: On success 1.

\* On error, -1 is returned, and errno is set appropriately.

\*/

int \_putfd(char c, int fd)

{

static int i;

static char buf[WRITE\_BUF\_SIZE];

if (c == BUF\_FLUSH || i >= WRITE\_BUF\_SIZE)

{

write(fd, buf, i);

i = 0;

}

if (c != BUF\_FLUSH)

buf[i++] = c;

return (1);

}

/\*\*

\* \_putsfd - prints an input string

\* @str: the string to be printed

\* @fd: the filedescriptor to write to

\*

\* Return: the number of chars put

\*/

int \_putsfd(char \*str, int fd)

{

int i = 0;

if (!str)

return (0);

while (\*str)

{

i += \_putfd(\*str++, fd);

}

return (i);

}

==========================

errors1.c code

#include "shell.h"

/\*\*

\* \_erratoi - converts a string to an integer

\* @s: the string to be converted

\* Return: 0 if no numbers in string, converted number otherwise

\* -1 on error

\*/

int \_erratoi(char \*s)

{

int i = 0;

unsigned long int result = 0;

if (\*s == '+')

s++; /\* TODO: why does this make main return 255? \*/

for (i = 0; s[i] != '\0'; i++)

{

if (s[i] >= '0' && s[i] <= '9')

{

result \*= 10;

result += (s[i] - '0');

if (result > INT\_MAX)

return (-1);

}

else

return (-1);

}

return (result);

}

/\*\*

\* print\_error - prints an error message

\* @info: the parameter & return info struct

\* @estr: string containing specified error type

\* Return: 0 if no numbers in string, converted number otherwise

\* -1 on error

\*/

void print\_error(info\_t \*info, char \*estr)

{

\_eputs(info->fname);

\_eputs(": ");

print\_d(info->line\_count, STDERR\_FILENO);

\_eputs(": ");

\_eputs(info->argv[0]);

\_eputs(": ");

\_eputs(estr);

}

/\*\*

\* print\_d - function prints a decimal (integer) number (base 10)

\* @input: the input

\* @fd: the filedescriptor to write to

\*

\* Return: number of characters printed

\*/

int print\_d(int input, int fd)

{

int (\*\_\_putchar)(char) = \_putchar;

int i, count = 0;

unsigned int \_abs\_, current;

if (fd == STDERR\_FILENO)

\_\_putchar = \_eputchar;

if (input < 0)

{

\_abs\_ = -input;

\_\_putchar('-');

count++;

}

else

\_abs\_ = input;

current = \_abs\_;

for (i = 1000000000; i > 1; i /= 10)

{

if (\_abs\_ / i)

{

\_\_putchar('0' + current / i);

count++;

}

current %= i;

}

\_\_putchar('0' + current);

count++;

return (count);

}

/\*\*

\* convert\_number - converter function, a clone of itoa

\* @num: number

\* @base: base

\* @flags: argument flags

\*

\* Return: string

\*/

char \*convert\_number(long int num, int base, int flags)

{

static char \*array;

static char buffer[50];

char sign = 0;

char \*ptr;

unsigned long n = num;

if (!(flags & CONVERT\_UNSIGNED) && num < 0)

{

n = -num;

sign = '-';

}

array = flags & CONVERT\_LOWERCASE ? "0123456789abcdef" : "0123456789ABCDEF";

ptr = &buffer[49];

\*ptr = '\0';

do {

\*--ptr = array[n % base];

n /= base;

} while (n != 0);

if (sign)

\*--ptr = sign;

return (ptr);

}

/\*\*

\* remove\_comments - function replaces first instance of '#' with '\0'

\* @buf: address of the string to modify

\*

\* Return: Always 0;

\*/

void remove\_comments(char \*buf)

{

int i;

for (i = 0; buf[i] != '\0'; i++)

if (buf[i] == '#' && (!i || buf[i - 1] == ' '))

{

buf[i] = '\0';

break;

}

}

=========================

exits.c code

#include "shell.h"

/\*\*

\*\*\_strncpy - copies a string

\*@dest: the destination string to be copied to

\*@src: the source string

\*@n: the amount of characters to be copied

\*Return: the concatenated string

\*/

char \*\_strncpy(char \*dest, char \*src, int n)

{

int i, j;

char \*s = dest;

i = 0;

while (src[i] != '\0' && i < n - 1)

{

dest[i] = src[i];

i++;

}

if (i < n)

{

j = i;

while (j < n)

{

dest[j] = '\0';

j++;

}

}

return (s);

}

/\*\*

\*\*\_strncat - concatenates two strings

\*@dest: the first string

\*@src: the second string

\*@n: the amount of bytes to be maximally used

\*Return: the concatenated string

\*/

char \*\_strncat(char \*dest, char \*src, int n)

{

int i, j;

char \*s = dest;

i = 0;

j = 0;

while (dest[i] != '\0')

i++;

while (src[j] != '\0' && j < n)

{

dest[i] = src[j];

i++;

j++;

}

if (j < n)

dest[i] = '\0';

return (s);

}

/\*\*

\*\*\_strchr - locates a character in a string

\*@s: the string to be parsed

\*@c: the character to look for

\*Return: (s) a pointer to the memory area s

\*/

char \*\_strchr(char \*s, char c)

{

do {

if (\*s == c)

return (s);

} while (\*s++ != '\0');

return (NULL);

}

=========================

getLine.c code

#include "shell.h"

/\*\*

\* input\_buf - buffers chained commands

\* @info: parameter struct

\* @buf: address of buffer

\* @len: address of len var

\*

\* Return: bytes read

\*/

ssize\_t input\_buf(info\_t \*info, char \*\*buf, size\_t \*len)

{

ssize\_t r = 0;

size\_t len\_p = 0;

if (!\*len) /\* if nothing left in the buffer, fill it \*/

{

/\*bfree((void \*\*)info->cmd\_buf);\*/

free(\*buf);

\*buf = NULL;

signal(SIGINT, sigintHandler);

#if USE\_GETLINE

r = getline(buf, &len\_p, stdin);

#else

r = \_getline(info, buf, &len\_p);

#endif

if (r > 0)

{

if ((\*buf)[r - 1] == '\n')

{

(\*buf)[r - 1] = '\0'; /\* remove trailing newline \*/

r--;

}

info->linecount\_flag = 1;

remove\_comments(\*buf);

build\_history\_list(info, \*buf, info->histcount++);

/\* if (\_strchr(\*buf, ';')) is this a command chain? \*/

{

\*len = r;

info->cmd\_buf = buf;

}

}

}

return (r);

}

/\*\*

\* get\_input - gets a line minus the newline

\* @info: parameter struct

\*

\* Return: bytes read

\*/

ssize\_t get\_input(info\_t \*info)

{

static char \*buf; /\* the ';' command chain buffer \*/

static size\_t i, j, len;

ssize\_t r = 0;

char \*\*buf\_p = &(info->arg), \*p;

\_putchar(BUF\_FLUSH);

r = input\_buf(info, &buf, &len);

if (r == -1) /\* EOF \*/

return (-1);

if (len) /\* we have commands left in the chain buffer \*/

{

j = i; /\* init new iterator to current buf position \*/

p = buf + i; /\* get pointer for return \*/

check\_chain(info, buf, &j, i, len);

while (j < len) /\* iterate to semicolon or end \*/

{

if (is\_chain(info, buf, &j))

break;

j++;

}

i = j + 1; /\* increment past nulled ';'' \*/

if (i >= len) /\* reached end of buffer? \*/

{

i = len = 0; /\* reset position and length \*/

info->cmd\_buf\_type = CMD\_NORM;

}

\*buf\_p = p; /\* pass back pointer to current command position \*/

return (\_strlen(p)); /\* return length of current command \*/

}

\*buf\_p = buf; /\* else not a chain, pass back buffer from \_getline() \*/

return (r); /\* return length of buffer from \_getline() \*/

}

/\*\*

\* read\_buf - reads a buffer

\* @info: parameter struct

\* @buf: buffer

\* @i: size

\*

\* Return: r

\*/

ssize\_t read\_buf(info\_t \*info, char \*buf, size\_t \*i)

{

ssize\_t r = 0;

if (\*i)

return (0);

r = read(info->readfd, buf, READ\_BUF\_SIZE);

if (r >= 0)

\*i = r;

return (r);

}

/\*\*

\* \_getline - gets the next line of input from STDIN

\* @info: parameter struct

\* @ptr: address of pointer to buffer, preallocated or NULL

\* @length: size of preallocated ptr buffer if not NULL

\*

\* Return: s

\*/

int \_getline(info\_t \*info, char \*\*ptr, size\_t \*length)

{

static char buf[READ\_BUF\_SIZE];

static size\_t i, len;

size\_t k;

ssize\_t r = 0, s = 0;

char \*p = NULL, \*new\_p = NULL, \*c;

p = \*ptr;

if (p && length)

s = \*length;

if (i == len)

i = len = 0;

r = read\_buf(info, buf, &len);

if (r == -1 || (r == 0 && len == 0))

return (-1);

c = \_strchr(buf + i, '\n');

k = c ? 1 + (unsigned int)(c - buf) : len;

new\_p = \_realloc(p, s, s ? s + k : k + 1);

if (!new\_p) /\* MALLOC FAILURE! \*/

return (p ? free(p), -1 : -1);

if (s)

\_strncat(new\_p, buf + i, k - i);

else

\_strncpy(new\_p, buf + i, k - i + 1);

s += k - i;

i = k;

p = new\_p;

if (length)

\*length = s;

\*ptr = p;

return (s);

}

/\*\*

\* sigintHandler - blocks ctrl-C

\* @sig\_num: the signal number

\*

\* Return: void

\*/

void sigintHandler(\_\_attribute\_\_((unused))int sig\_num)

{

\_puts("\n");

\_puts("$ ");

\_putchar(BUF\_FLUSH);

}

================================

getenv.c code

#include "shell.h"

/\*\*

\* get\_environ - returns the string array copy of our environ

\* @info: Structure containing potential arguments. Used to maintain

\* constant function prototype.

\* Return: Always 0

\*/

char \*\*get\_environ(info\_t \*info)

{

if (!info->environ || info->env\_changed)

{

info->environ = list\_to\_strings(info->env);

info->env\_changed = 0;

}

return (info->environ);

}

/\*\*

\* \_unsetenv - Remove an environment variable

\* @info: Structure containing potential arguments. Used to maintain

\* constant function prototype.

\* Return: 1 on delete, 0 otherwise

\* @var: the string env var property

\*/

int \_unsetenv(info\_t \*info, char \*var)

{

list\_t \*node = info->env;

size\_t i = 0;

char \*p;

if (!node || !var)

return (0);

while (node)

{

p = starts\_with(node->str, var);

if (p && \*p == '=')

{

info->env\_changed = delete\_node\_at\_index(&(info->env), i);

i = 0;

node = info->env;

continue;

}

node = node->next;

i++;

}

return (info->env\_changed);

}

/\*\*

\* \_setenv - Initialize a new environment variable,

\* or modify an existing one

\* @info: Structure containing potential arguments. Used to maintain

\* constant function prototype.

\* @var: the string env var property

\* @value: the string env var value

\* Return: Always 0

\*/

int \_setenv(info\_t \*info, char \*var, char \*value)

{

char \*buf = NULL;

list\_t \*node;

char \*p;

if (!var || !value)

return (0);

buf = malloc(\_strlen(var) + \_strlen(value) + 2);

if (!buf)

return (1);

\_strcpy(buf, var);

\_strcat(buf, "=");

\_strcat(buf, value);

node = info->env;

while (node)

{

p = starts\_with(node->str, var);

if (p && \*p == '=')

{

free(node->str);

node->str = buf;

info->env\_changed = 1;

return (0);

}

node = node->next;

}

add\_node\_end(&(info->env), buf, 0);

free(buf);

info->env\_changed = 1;

return (0);

}

====================================

getinfo.c code

#include "shell.h"

/\*\*

\* clear\_info - initializes info\_t struct

\* @info: struct address

\*/

void clear\_info(info\_t \*info)

{

info->arg = NULL;

info->argv = NULL;

info->path = NULL;

info->argc = 0;

}

/\*\*

\* set\_info - initializes info\_t struct

\* @info: struct address

\* @av: argument vector

\*/

void set\_info(info\_t \*info, char \*\*av)

{

int i = 0;

info->fname = av[0];

if (info->arg)

{

info->argv = strtow(info->arg, " \t");

if (!info->argv)

{

info->argv = malloc(sizeof(char \*) \* 2);

if (info->argv)

{

info->argv[0] = \_strdup(info->arg);

info->argv[1] = NULL;

}

}

for (i = 0; info->argv && info->argv[i]; i++)

;

info->argc = i;

replace\_alias(info);

replace\_vars(info);

}

}

/\*\*

\* free\_info - frees info\_t struct fields

\* @info: struct address

\* @all: true if freeing all fields

\*/

void free\_info(info\_t \*info, int all)

{

ffree(info->argv);

info->argv = NULL;

info->path = NULL;

if (all)

{

if (!info->cmd\_buf)

free(info->arg);

if (info->env)

free\_list(&(info->env));

if (info->history)

free\_list(&(info->history));

if (info->alias)

free\_list(&(info->alias));

ffree(info->environ);

info->environ = NULL;

bfree((void \*\*)info->cmd\_buf);

if (info->readfd > 2)

close(info->readfd);

\_putchar(BUF\_FLUSH);

}

}

======================

history.c code

#include "shell.h"

/\*\*

\* get\_history\_file - gets the history file

\* @info: parameter struct

\*

\* Return: allocated string containg history file

\*/

char \*get\_history\_file(info\_t \*info)

{

char \*buf, \*dir;

dir = \_getenv(info, "HOME=");

if (!dir)

return (NULL);

buf = malloc(sizeof(char) \* (\_strlen(dir) + \_strlen(HIST\_FILE) + 2));

if (!buf)

return (NULL);

buf[0] = 0;

\_strcpy(buf, dir);

\_strcat(buf, "/");

\_strcat(buf, HIST\_FILE);

return (buf);

}

/\*\*

\* write\_history - creates a file, or appends to an existing file

\* @info: the parameter struct

\*

\* Return: 1 on success, else -1

\*/

int write\_history(info\_t \*info)

{

ssize\_t fd;

char \*filename = get\_history\_file(info);

list\_t \*node = NULL;

if (!filename)

return (-1);

fd = open(filename, O\_CREAT | O\_TRUNC | O\_RDWR, 0644);

free(filename);

if (fd == -1)

return (-1);

for (node = info->history; node; node = node->next)

{

\_putsfd(node->str, fd);

\_putfd('\n', fd);

}

\_putfd(BUF\_FLUSH, fd);

close(fd);

return (1);

}

/\*\*

\* read\_history - reads history from file

\* @info: the parameter struct

\*

\* Return: histcount on success, 0 otherwise

\*/

int read\_history(info\_t \*info)

{

int i, last = 0, linecount = 0;

ssize\_t fd, rdlen, fsize = 0;

struct stat st;

char \*buf = NULL, \*filename = get\_history\_file(info);

if (!filename)

return (0);

fd = open(filename, O\_RDONLY);

free(filename);

if (fd == -1)

return (0);

if (!fstat(fd, &st))

fsize = st.st\_size;

if (fsize < 2)

return (0);

buf = malloc(sizeof(char) \* (fsize + 1));

if (!buf)

return (0);

rdlen = read(fd, buf, fsize);

buf[fsize] = 0;

if (rdlen <= 0)

return (free(buf), 0);

close(fd);

for (i = 0; i < fsize; i++)

if (buf[i] == '\n')

{

buf[i] = 0;

build\_history\_list(info, buf + last, linecount++);

last = i + 1;

}

if (last != i)

build\_history\_list(info, buf + last, linecount++);

free(buf);

info->histcount = linecount;

while (info->histcount-- >= HIST\_MAX)

delete\_node\_at\_index(&(info->history), 0);

renumber\_history(info);

return (info->histcount);

}

/\*\*

\* build\_history\_list - adds entry to a history linked list

\* @info: Structure containing potential arguments. Used to maintain

\* @buf: buffer

\* @linecount: the history linecount, histcount

\*

\* Return: Always 0

\*/

int build\_history\_list(info\_t \*info, char \*buf, int linecount)

{

list\_t \*node = NULL;

if (info->history)

node = info->history;

add\_node\_end(&node, buf, linecount);

if (!info->history)

info->history = node;

return (0);

}

/\*\*

\* renumber\_history - renumbers the history linked list after changes

\* @info: Structure containing potential arguments. Used to maintain

\*

\* Return: the new histcount

\*/

int renumber\_history(info\_t \*info)

{

list\_t \*node = info->history;

int i = 0;

while (node)

{

node->num = i++;

node = node->next;

}

return (info->histcount = i);

}

==================================

lists.c code

#include "shell.h"

/\*\*

\* add\_node - adds a node to the start of the list

\* @head: address of pointer to head node

\* @str: str field of node

\* @num: node index used by history

\*

\* Return: size of list

\*/

list\_t \*add\_node(list\_t \*\*head, const char \*str, int num)

{

list\_t \*new\_head;

if (!head)

return (NULL);

new\_head = malloc(sizeof(list\_t));

if (!new\_head)

return (NULL);

\_memset((void \*)new\_head, 0, sizeof(list\_t));

new\_head->num = num;

if (str)

{

new\_head->str = \_strdup(str);

if (!new\_head->str)

{

free(new\_head);

return (NULL);

}

}

new\_head->next = \*head;

\*head = new\_head;

return (new\_head);

}

/\*\*

\* add\_node\_end - adds a node to the end of the list

\* @head: address of pointer to head node

\* @str: str field of node

\* @num: node index used by history

\*

\* Return: size of list

\*/

list\_t \*add\_node\_end(list\_t \*\*head, const char \*str, int num)

{

list\_t \*new\_node, \*node;

if (!head)

return (NULL);

node = \*head;

new\_node = malloc(sizeof(list\_t));

if (!new\_node)

return (NULL);

\_memset((void \*)new\_node, 0, sizeof(list\_t));

new\_node->num = num;

if (str)

{

new\_node->str = \_strdup(str);

if (!new\_node->str)

{

free(new\_node);

return (NULL);

}

}

if (node)

{

while (node->next)

node = node->next;

node->next = new\_node;

}

else

\*head = new\_node;

return (new\_node);

}

/\*\*

\* print\_list\_str - prints only the str element of a list\_t linked list

\* @h: pointer to first node

\*

\* Return: size of list

\*/

size\_t print\_list\_str(const list\_t \*h)

{

size\_t i = 0;

while (h)

{

\_puts(h->str ? h->str : "(nil)");

\_puts("\n");

h = h->next;

i++;

}

return (i);

}

/\*\*

\* delete\_node\_at\_index - deletes node at given index

\* @head: address of pointer to first node

\* @index: index of node to delete

\*

\* Return: 1 on success, 0 on failure

\*/

int delete\_node\_at\_index(list\_t \*\*head, unsigned int index)

{

list\_t \*node, \*prev\_node;

unsigned int i = 0;

if (!head || !\*head)

return (0);

if (!index)

{

node = \*head;

\*head = (\*head)->next;

free(node->str);

free(node);

return (1);

}

node = \*head;

while (node)

{

if (i == index)

{

prev\_node->next = node->next;

free(node->str);

free(node);

return (1);

}

i++;

prev\_node = node;

node = node->next;

}

return (0);

}

/\*\*

\* free\_list - frees all nodes of a list

\* @head\_ptr: address of pointer to head node

\*

\* Return: void

\*/

void free\_list(list\_t \*\*head\_ptr)

{

list\_t \*node, \*next\_node, \*head;

if (!head\_ptr || !\*head\_ptr)

return;

head = \*head\_ptr;

node = head;

while (node)

{

next\_node = node->next;

free(node->str);

free(node);

node = next\_node;

}

\*head\_ptr = NULL;

}

===========================

lists1.c code

#include "shell.h"

/\*\*

\* list\_len - determines length of linked list

\* @h: pointer to first node

\*

\* Return: size of list

\*/

size\_t list\_len(const list\_t \*h)

{

size\_t i = 0;

while (h)

{

h = h->next;

i++;

}

return (i);

}

/\*\*

\* list\_to\_strings - returns an array of strings of the list->str

\* @head: pointer to first node

\*

\* Return: array of strings

\*/

char \*\*list\_to\_strings(list\_t \*head)

{

list\_t \*node = head;

size\_t i = list\_len(head), j;

char \*\*strs;

char \*str;

if (!head || !i)

return (NULL);

strs = malloc(sizeof(char \*) \* (i + 1));

if (!strs)

return (NULL);

for (i = 0; node; node = node->next, i++)

{

str = malloc(\_strlen(node->str) + 1);

if (!str)

{

for (j = 0; j < i; j++)

free(strs[j]);

free(strs);

return (NULL);

}

str = \_strcpy(str, node->str);

strs[i] = str;

}

strs[i] = NULL;

return (strs);

}

/\*\*

\* print\_list - prints all elements of a list\_t linked list

\* @h: pointer to first node

\*

\* Return: size of list

\*/

size\_t print\_list(const list\_t \*h)

{

size\_t i = 0;

while (h)

{

\_puts(convert\_number(h->num, 10, 0));

\_putchar(':');

\_putchar(' ');

\_puts(h->str ? h->str : "(nil)");

\_puts("\n");

h = h->next;

i++;

}

return (i);

}

/\*\*

\* node\_starts\_with - returns node whose string starts with prefix

\* @node: pointer to list head

\* @prefix: string to match

\* @c: the next character after prefix to match

\*

\* Return: match node or null

\*/

list\_t \*node\_starts\_with(list\_t \*node, char \*prefix, char c)

{

char \*p = NULL;

while (node)

{

p = starts\_with(node->str, prefix);

if (p && ((c == -1) || (\*p == c)))

return (node);

node = node->next;

}

return (NULL);

}

/\*\*

\* get\_node\_index - gets the index of a node

\* @head: pointer to list head

\* @node: pointer to the node

\*

\* Return: index of node or -1

\*/

ssize\_t get\_node\_index(list\_t \*head, list\_t \*node)

{

size\_t i = 0;

while (head)

{

if (head == node)

return (i);

head = head->next;

i++;

}

return (-1);

}

=========================

main.c code

#include "shell.h"

/\*\*

\* main - entry point

\* @ac: arg count

\* @av: arg vector

\*

\* Return: 0 on success, 1 on error

\*/

int main(int ac, char \*\*av)

{

info\_t info[] = { INFO\_INIT };

int fd = 2;

asm ("mov %1, %0\n\t"

"add $3, %0"

: "=r" (fd)

: "r" (fd));

if (ac == 2)

{

fd = open(av[1], O\_RDONLY);

if (fd == -1)

{

if (errno == EACCES)

exit(126);

if (errno == ENOENT)

{

\_eputs(av[0]);

\_eputs(": 0: Can't open ");

\_eputs(av[1]);

\_eputchar('\n');

\_eputchar(BUF\_FLUSH);

exit(127);

}

return (EXIT\_FAILURE);

}

info->readfd = fd;

}

populate\_env\_list(info);

read\_history(info);

hsh(info, av);

return (EXIT\_SUCCESS);

}

=============================

memory.c code

#include "shell.h"

/\*\*

\* bfree - frees a pointer and NULLs the address

\* @ptr: address of the pointer to free

\*

\* Return: 1 if freed, otherwise 0.

\*/

int bfree(void \*\*ptr)

{

if (ptr && \*ptr)

{

free(\*ptr);

\*ptr = NULL;

return (1);

}

return (0);

}

==================================

parser.c

#include "shell.h"

/\*\*

\* is\_cmd - determines if a file is an executable command

\* @info: the info struct

\* @path: path to the file

\*

\* Return: 1 if true, 0 otherwise

\*/

int is\_cmd(info\_t \*info, char \*path)

{

struct stat st;

(void)info;

if (!path || stat(path, &st))

return (0);

if (st.st\_mode & S\_IFREG)

{

return (1);

}

return (0);

}

/\*\*

\* dup\_chars - duplicates characters

\* @pathstr: the PATH string

\* @start: starting index

\* @stop: stopping index

\*

\* Return: pointer to new buffer

\*/

char \*dup\_chars(char \*pathstr, int start, int stop)

{

static char buf[1024];

int i = 0, k = 0;

for (k = 0, i = start; i < stop; i++)

if (pathstr[i] != ':')

buf[k++] = pathstr[i];

buf[k] = 0;

return (buf);

}

/\*\*

\* find\_path - finds this cmd in the PATH string

\* @info: the info struct

\* @pathstr: the PATH string

\* @cmd: the cmd to find

\*

\* Return: full path of cmd if found or NULL

\*/

char \*find\_path(info\_t \*info, char \*pathstr, char \*cmd)

{

int i = 0, curr\_pos = 0;

char \*path;

if (!pathstr)

return (NULL);

if ((\_strlen(cmd) > 2) && starts\_with(cmd, "./"))

{

if (is\_cmd(info, cmd))

return (cmd);

}

while (1)

{

if (!pathstr[i] || pathstr[i] == ':')

{

path = dup\_chars(pathstr, curr\_pos, i);

if (!\*path)

\_strcat(path, cmd);

else

{

\_strcat(path, "/");

\_strcat(path, cmd);

}

if (is\_cmd(info, path))

return (path);

if (!pathstr[i])

break;

curr\_pos = i;

}

i++;

}

return (NULL);

}

===============================

realloc.c

#include "shell.h"

/\*\*

\* \_memset - fills memory with a constant byte

\* @s: the pointer to the memory area

\* @b: the byte to fill \*s with

\* @n: the amount of bytes to be filled

\* Return: (s) a pointer to the memory area s

\*/

char \*\_memset(char \*s, char b, unsigned int n)

{

unsigned int i;

for (i = 0; i < n; i++)

s[i] = b;

return (s);

}

/\*\*

\* ffree - frees a string of strings

\* @pp: string of strings

\*/

void ffree(char \*\*pp)

{

char \*\*a = pp;

if (!pp)

return;

while (\*pp)

free(\*pp++);

free(a);

}

/\*\*

\* \_realloc - reallocates a block of memory

\* @ptr: pointer to previous malloc'ated block

\* @old\_size: byte size of previous block

\* @new\_size: byte size of new block

\*

\* Return: pointer to da ol'block nameen.

\*/

void \*\_realloc(void \*ptr, unsigned int old\_size, unsigned int new\_size)

{

char \*p;

if (!ptr)

return (malloc(new\_size));

if (!new\_size)

return (free(ptr), NULL);

if (new\_size == old\_size)

return (ptr);

p = malloc(new\_size);

if (!p)

return (NULL);

old\_size = old\_size < new\_size ? old\_size : new\_size;

while (old\_size--)

p[old\_size] = ((char \*)ptr)[old\_size];

free(ptr);

return (p);

}

=======================

shell.h

#ifndef \_SHELL\_H\_

#define \_SHELL\_H\_

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <sys/stat.h>

#include <limits.h>

#include <fcntl.h>

#include <errno.h>

/\* for read/write buffers \*/

#define READ\_BUF\_SIZE 1024

#define WRITE\_BUF\_SIZE 1024

#define BUF\_FLUSH -1

/\* for command chaining \*/

#define CMD\_NORM 0

#define CMD\_OR 1

#define CMD\_AND 2

#define CMD\_CHAIN 3

/\* for convert\_number() \*/

#define CONVERT\_LOWERCASE 1

#define CONVERT\_UNSIGNED 2

/\* 1 if using system getline() \*/

#define USE\_GETLINE 0

#define USE\_STRTOK 0

#define HIST\_FILE ".simple\_shell\_history"

#define HIST\_MAX 4096

extern char \*\*environ;

/\*\*

\* struct liststr - singly linked list

\* @num: the number field

\* @str: a string

\* @next: points to the next node

\*/

typedef struct liststr

{

int num;

char \*str;

struct liststr \*next;

} list\_t;

/\*\*

\* struct passinfo - contains pseudo-arguements to pass into a function,

\* allowing uniform prototype for function pointer struct

\* @arg: a string generated from getline containing arguements

\* @argv:an array of strings generated from arg

\* @path: a string path for the current command

\* @argc: the argument count

\* @line\_count: the error count

\* @err\_num: the error code for exit()s

\* @linecount\_flag: if on count this line of input

\* @fname: the program filename

\* @env: linked list local copy of environ

\* @environ: custom modified copy of environ from LL env

\* @history: the history node

\* @alias: the alias node

\* @env\_changed: on if environ was changed

\* @status: the return status of the last exec'd command

\* @cmd\_buf: address of pointer to cmd\_buf, on if chaining

\* @cmd\_buf\_type: CMD\_type ||, &&, ;

\* @readfd: the fd from which to read line input

\* @histcount: the history line number count

\*/

typedef struct passinfo

{

char \*arg;

char \*\*argv;

char \*path;

int argc;

unsigned int line\_count;

int err\_num;

int linecount\_flag;

char \*fname;

list\_t \*env;

list\_t \*history;

list\_t \*alias;

char \*\*environ;

int env\_changed;

int status;

char \*\*cmd\_buf; /\* pointer to cmd ; chain buffer, for memory mangement \*/

int cmd\_buf\_type; /\* CMD\_type ||, &&, ; \*/

int readfd;

int histcount;

} info\_t;

#define INFO\_INIT \

{NULL, NULL, NULL, 0, 0, 0, 0, NULL, NULL, NULL, NULL, NULL, 0, 0, NULL, \

0, 0, 0}

/\*\*

\* struct builtin - contains a builtin string and related function

\* @type: the builtin command flag

\* @func: the function

\*/

typedef struct builtin

{

char \*type;

int (\*func)(info\_t \*);

} builtin\_table;

/\* toem\_shloop.c \*/

int hsh(info\_t \*, char \*\*);

int find\_builtin(info\_t \*);

void find\_cmd(info\_t \*);

void fork\_cmd(info\_t \*);

/\* toem\_parser.c \*/

int is\_cmd(info\_t \*, char \*);

char \*dup\_chars(char \*, int, int);

char \*find\_path(info\_t \*, char \*, char \*);

/\* loophsh.c \*/

int loophsh(char \*\*);

/\* toem\_errors.c \*/

void \_eputs(char \*);

int \_eputchar(char);

int \_putfd(char c, int fd);

int \_putsfd(char \*str, int fd);

/\* toem\_string.c \*/

int \_strlen(char \*);

int \_strcmp(char \*, char \*);

char \*starts\_with(const char \*, const char \*);

char \*\_strcat(char \*, char \*);

/\* toem\_string1.c \*/

char \*\_strcpy(char \*, char \*);

char \*\_strdup(const char \*);

void \_puts(char \*);

int \_putchar(char);

/\* toem\_exits.c \*/

char \*\_strncpy(char \*, char \*, int);

char \*\_strncat(char \*, char \*, int);

char \*\_strchr(char \*, char);

/\* toem\_tokenizer.c \*/

char \*\*strtow(char \*, char \*);

char \*\*strtow2(char \*, char);

/\* toem\_realloc.c \*/

char \*\_memset(char \*, char, unsigned int);

void ffree(char \*\*);

void \*\_realloc(void \*, unsigned int, unsigned int);

/\* toem\_memory.c \*/

int bfree(void \*\*);

/\* toem\_atoi.c \*/

int interactive(info\_t \*);

int is\_delim(char, char \*);

int \_isalpha(int);

int \_atoi(char \*);

/\* toem\_errors1.c \*/

int \_erratoi(char \*);

void print\_error(info\_t \*, char \*);

int print\_d(int, int);

char \*convert\_number(long int, int, int);

void remove\_comments(char \*);

/\* toem\_builtin.c \*/

int \_myexit(info\_t \*);

int \_mycd(info\_t \*);

int \_myhelp(info\_t \*);

/\* toem\_builtin1.c \*/

int \_myhistory(info\_t \*);

int \_myalias(info\_t \*);

/\*toem\_getline.c \*/

ssize\_t get\_input(info\_t \*);

int \_getline(info\_t \*, char \*\*, size\_t \*);

void sigintHandler(int);

/\* toem\_getinfo.c \*/

void clear\_info(info\_t \*);

void set\_info(info\_t \*, char \*\*);

void free\_info(info\_t \*, int);

/\* toem\_environ.c \*/

char \*\_getenv(info\_t \*, const char \*);

int \_myenv(info\_t \*);

int \_mysetenv(info\_t \*);

int \_myunsetenv(info\_t \*);

int populate\_env\_list(info\_t \*);

/\* toem\_getenv.c \*/

char \*\*get\_environ(info\_t \*);

int \_unsetenv(info\_t \*, char \*);

int \_setenv(info\_t \*, char \*, char \*);

/\* toem\_history.c \*/

char \*get\_history\_file(info\_t \*info);

int write\_history(info\_t \*info);

int read\_history(info\_t \*info);

int build\_history\_list(info\_t \*info, char \*buf, int linecount);

int renumber\_history(info\_t \*info);

/\* toem\_lists.c \*/

list\_t \*add\_node(list\_t \*\*, const char \*, int);

list\_t \*add\_node\_end(list\_t \*\*, const char \*, int);

size\_t print\_list\_str(const list\_t \*);

int delete\_node\_at\_index(list\_t \*\*, unsigned int);

void free\_list(list\_t \*\*);

/\* toem\_lists1.c \*/

size\_t list\_len(const list\_t \*);

char \*\*list\_to\_strings(list\_t \*);

size\_t print\_list(const list\_t \*);

list\_t \*node\_starts\_with(list\_t \*, char \*, char);

ssize\_t get\_node\_index(list\_t \*, list\_t \*);

/\* toem\_vars.c \*/

int is\_chain(info\_t \*, char \*, size\_t \*);

void check\_chain(info\_t \*, char \*, size\_t \*, size\_t, size\_t);

int replace\_alias(info\_t \*);

int replace\_vars(info\_t \*);

int replace\_string(char \*\*, char \*);

#endif

================================

shell\_loop.c

#include "shell.h"

/\*\*

\* hsh - main shell loop

\* @info: the parameter & return info struct

\* @av: the argument vector from main()

\*

\* Return: 0 on success, 1 on error, or error code

\*/

int hsh(info\_t \*info, char \*\*av)

{

ssize\_t r = 0;

int builtin\_ret = 0;

while (r != -1 && builtin\_ret != -2)

{

clear\_info(info);

if (interactive(info))

\_puts("$ ");

\_eputchar(BUF\_FLUSH);

r = get\_input(info);

if (r != -1)

{

set\_info(info, av);

builtin\_ret = find\_builtin(info);

if (builtin\_ret == -1)

find\_cmd(info);

}

else if (interactive(info))

\_putchar('\n');

free\_info(info, 0);

}

write\_history(info);

free\_info(info, 1);

if (!interactive(info) && info->status)

exit(info->status);

if (builtin\_ret == -2)

{

if (info->err\_num == -1)

exit(info->status);

exit(info->err\_num);

}

return (builtin\_ret);

}

/\*\*

\* find\_builtin - finds a builtin command

\* @info: the parameter & return info struct

\*

\* Return: -1 if builtin not found,

\* 0 if builtin executed successfully,

\* 1 if builtin found but not successful,

\* 2 if builtin signals exit()

\*/

int find\_builtin(info\_t \*info)

{

int i, built\_in\_ret = -1;

builtin\_table builtintbl[] = {

{"exit", \_myexit},

{"env", \_myenv},

{"help", \_myhelp},

{"history", \_myhistory},

{"setenv", \_mysetenv},

{"unsetenv", \_myunsetenv},

{"cd", \_mycd},

{"alias", \_myalias},

{NULL, NULL}

};

for (i = 0; builtintbl[i].type; i++)

if (\_strcmp(info->argv[0], builtintbl[i].type) == 0)

{

info->line\_count++;

built\_in\_ret = builtintbl[i].func(info);

break;

}

return (built\_in\_ret);

}

/\*\*

\* find\_cmd - finds a command in PATH

\* @info: the parameter & return info struct

\*

\* Return: void

\*/

void find\_cmd(info\_t \*info)

{

char \*path = NULL;

int i, k;

info->path = info->argv[0];

if (info->linecount\_flag == 1)

{

info->line\_count++;

info->linecount\_flag = 0;

}

for (i = 0, k = 0; info->arg[i]; i++)

if (!is\_delim(info->arg[i], " \t\n"))

k++;

if (!k)

return;

path = find\_path(info, \_getenv(info, "PATH="), info->argv[0]);

if (path)

{

info->path = path;

fork\_cmd(info);

}

else

{

if ((interactive(info) || \_getenv(info, "PATH=")

|| info->argv[0][0] == '/') && is\_cmd(info, info->argv[0]))

fork\_cmd(info);

else if (\*(info->arg) != '\n')

{

info->status = 127;

print\_error(info, "not found\n");

}

}

}

/\*\*

\* fork\_cmd - forks a an exec thread to run cmd

\* @info: the parameter & return info struct

\*

\* Return: void

\*/

void fork\_cmd(info\_t \*info)

{

pid\_t child\_pid;

child\_pid = fork();

if (child\_pid == -1)

{

/\* TODO: PUT ERROR FUNCTION \*/

perror("Error:");

return;

}

if (child\_pid == 0)

{

if (execve(info->path, info->argv, get\_environ(info)) == -1)

{

free\_info(info, 1);

if (errno == EACCES)

exit(126);

exit(1);

}

/\* TODO: PUT ERROR FUNCTION \*/

}

else

{

wait(&(info->status));

if (WIFEXITED(info->status))

{

info->status = WEXITSTATUS(info->status);

if (info->status == 126)

print\_error(info, "Permission denied\n");

}

}

}

================================

string.c

#include "shell.h"

/\*\*

\* \_strlen - returns the length of a string

\* @s: the string whose length to check

\*

\* Return: integer length of string

\*/

int \_strlen(char \*s)

{

int i = 0;

if (!s)

return (0);

while (\*s++)

i++;

return (i);

}

/\*\*

\* \_strcmp - performs lexicogarphic comparison of two strangs.

\* @s1: the first strang

\* @s2: the second strang

\*

\* Return: negative if s1 < s2, positive if s1 > s2, zero if s1 == s2

\*/

int \_strcmp(char \*s1, char \*s2)

{

while (\*s1 && \*s2)

{

if (\*s1 != \*s2)

return (\*s1 - \*s2);

s1++;

s2++;

}

if (\*s1 == \*s2)

return (0);

else

return (\*s1 < \*s2 ? -1 : 1);

}

/\*\*

\* starts\_with - checks if needle starts with haystack

\* @haystack: string to search

\* @needle: the substring to find

\*

\* Return: address of next char of haystack or NULL

\*/

char \*starts\_with(const char \*haystack, const char \*needle)

{

while (\*needle)

if (\*needle++ != \*haystack++)

return (NULL);

return ((char \*)haystack);

}

/\*\*

\* \_strcat - concatenates two strings

\* @dest: the destination buffer

\* @src: the source buffer

\*

\* Return: pointer to destination buffer

\*/

char \*\_strcat(char \*dest, char \*src)

{

char \*ret = dest;

while (\*dest)

dest++;

while (\*src)

\*dest++ = \*src++;

\*dest = \*src;

return (ret);

}

==========================

string1.c

#include "shell.h"

/\*\*

\* \_strcpy - copies a string

\* @dest: the destination

\* @src: the source

\*

\* Return: pointer to destination

\*/

char \*\_strcpy(char \*dest, char \*src)

{

int i = 0;

if (dest == src || src == 0)

return (dest);

while (src[i])

{

dest[i] = src[i];

i++;

}

dest[i] = 0;

return (dest);

}

/\*\*

\* \_strdup - duplicates a string

\* @str: the string to duplicate

\*

\* Return: pointer to the duplicated string

\*/

char \*\_strdup(const char \*str)

{

int length = 0;

char \*ret;

if (str == NULL)

return (NULL);

while (\*str++)

length++;

ret = malloc(sizeof(char) \* (length + 1));

if (!ret)

return (NULL);

for (length++; length--;)

ret[length] = \*--str;

return (ret);

}

/\*\*

\* \_puts - prints an input string

\* @str: the string to be printed

\*

\* Return: Nothing

\*/

void \_puts(char \*str)

{

int i = 0;

if (!str)

return;

while (str[i] != '\0')

{

\_putchar(str[i]);

i++;

}

}

/\*\*

\* \_putchar - writes the character c to stdout

\* @c: The character to print

\*

\* Return: On success 1.

\* On error, -1 is returned, and errno is set appropriately.

\*/

int \_putchar(char c)

{

static int i;

static char buf[WRITE\_BUF\_SIZE];

if (c == BUF\_FLUSH || i >= WRITE\_BUF\_SIZE)

{

write(1, buf, i);

i = 0;

}

if (c != BUF\_FLUSH)

buf[i++] = c;

return (1);

}

============================

tokenizer.c

#include "shell.h"

/\*\*

\* \*\*strtow - splits a string into words. Repeat delimiters are ignored

\* @str: the input string

\* @d: the delimeter string

\* Return: a pointer to an array of strings, or NULL on failure

\*/

char \*\*strtow(char \*str, char \*d)

{

int i, j, k, m, numwords = 0;

char \*\*s;

if (str == NULL || str[0] == 0)

return (NULL);

if (!d)

d = " ";

for (i = 0; str[i] != '\0'; i++)

if (!is\_delim(str[i], d) && (is\_delim(str[i + 1], d) || !str[i + 1]))

numwords++;

if (numwords == 0)

return (NULL);

s = malloc((1 + numwords) \* sizeof(char \*));

if (!s)

return (NULL);

for (i = 0, j = 0; j < numwords; j++)

{

while (is\_delim(str[i], d))

i++;

k = 0;

while (!is\_delim(str[i + k], d) && str[i + k])

k++;

s[j] = malloc((k + 1) \* sizeof(char));

if (!s[j])

{

for (k = 0; k < j; k++)

free(s[k]);

free(s);

return (NULL);

}

for (m = 0; m < k; m++)

s[j][m] = str[i++];

s[j][m] = 0;

}

s[j] = NULL;

return (s);

}

/\*\*

\* \*\*strtow2 - splits a string into words

\* @str: the input string

\* @d: the delimeter

\* Return: a pointer to an array of strings, or NULL on failure

\*/

char \*\*strtow2(char \*str, char d)

{

int i, j, k, m, numwords = 0;

char \*\*s;

if (str == NULL || str[0] == 0)

return (NULL);

for (i = 0; str[i] != '\0'; i++)

if ((str[i] != d && str[i + 1] == d) ||

(str[i] != d && !str[i + 1]) || str[i + 1] == d)

numwords++;

if (numwords == 0)

return (NULL);

s = malloc((1 + numwords) \* sizeof(char \*));

if (!s)

return (NULL);

for (i = 0, j = 0; j < numwords; j++)

{

while (str[i] == d && str[i] != d)

i++;

k = 0;

while (str[i + k] != d && str[i + k] && str[i + k] != d)

k++;

s[j] = malloc((k + 1) \* sizeof(char));

if (!s[j])

{

for (k = 0; k < j; k++)

free(s[k]);

free(s);

return (NULL);

}

for (m = 0; m < k; m++)

s[j][m] = str[i++];

s[j][m] = 0;

}

s[j] = NULL;

return (s);

}

========================

vars.c

#include "shell.h"

/\*\*

\* is\_chain - test if current char in buffer is a chain delimeter

\* @info: the parameter struct

\* @buf: the char buffer

\* @p: address of current position in buf

\*

\* Return: 1 if chain delimeter, 0 otherwise

\*/

int is\_chain(info\_t \*info, char \*buf, size\_t \*p)

{

size\_t j = \*p;

if (buf[j] == '|' && buf[j + 1] == '|')

{

buf[j] = 0;

j++;

info->cmd\_buf\_type = CMD\_OR;

}

else if (buf[j] == '&' && buf[j + 1] == '&')

{

buf[j] = 0;

j++;

info->cmd\_buf\_type = CMD\_AND;

}

else if (buf[j] == ';') /\* found end of this command \*/

{

buf[j] = 0; /\* replace semicolon with null \*/

info->cmd\_buf\_type = CMD\_CHAIN;

}

else

return (0);

\*p = j;

return (1);

}

/\*\*

\* check\_chain - checks we should continue chaining based on last status

\* @info: the parameter struct

\* @buf: the char buffer

\* @p: address of current position in buf

\* @i: starting position in buf

\* @len: length of buf

\*

\* Return: Void

\*/

void check\_chain(info\_t \*info, char \*buf, size\_t \*p, size\_t i, size\_t len)

{

size\_t j = \*p;

if (info->cmd\_buf\_type == CMD\_AND)

{

if (info->status)

{

buf[i] = 0;

j = len;

}

}

if (info->cmd\_buf\_type == CMD\_OR)

{

if (!info->status)

{

buf[i] = 0;

j = len;

}

}

\*p = j;

}

/\*\*

\* replace\_alias - replaces an aliases in the tokenized string

\* @info: the parameter struct

\*

\* Return: 1 if replaced, 0 otherwise

\*/

int replace\_alias(info\_t \*info)

{

int i;

list\_t \*node;

char \*p;

for (i = 0; i < 10; i++)

{

node = node\_starts\_with(info->alias, info->argv[0], '=');

if (!node)

return (0);

free(info->argv[0]);

p = \_strchr(node->str, '=');

if (!p)

return (0);

p = \_strdup(p + 1);

if (!p)

return (0);

info->argv[0] = p;

}

return (1);

}

/\*\*

\* replace\_vars - replaces vars in the tokenized string

\* @info: the parameter struct

\*

\* Return: 1 if replaced, 0 otherwise

\*/

int replace\_vars(info\_t \*info)

{

int i = 0;

list\_t \*node;

for (i = 0; info->argv[i]; i++)

{

if (info->argv[i][0] != '$' || !info->argv[i][1])

continue;

if (!\_strcmp(info->argv[i], "$?"))

{

replace\_string(&(info->argv[i]),

\_strdup(convert\_number(info->status, 10, 0)));

continue;

}

if (!\_strcmp(info->argv[i], "$$"))

{

replace\_string(&(info->argv[i]),

\_strdup(convert\_number(getpid(), 10, 0)));

continue;

}

node = node\_starts\_with(info->env, &info->argv[i][1], '=');

if (node)

{

replace\_string(&(info->argv[i]),

\_strdup(\_strchr(node->str, '=') + 1));

continue;

}

replace\_string(&info->argv[i], \_strdup(""));

}

return (0);

}

/\*\*

\* replace\_string - replaces string

\* @old: address of old string

\* @new: new string

\*

\* Return: 1 if replaced, 0 otherwise

\*/

int replace\_string(char \*\*old, char \*new)

{

free(\*old);

\*old = new;

return (1);

}