/\*----------------------------------------------------------------------------

Meryl Mabin

CSE 410, section 001

proj02 - Shell

---------------------------------------------------------------------------\*/

using namespace std;

#include <iostream>

#include <string.h>

#include <stdlib.h>

#include <unistd.h>

#include <queue>

#include <iostream>

#include <sys/types.h>

#include <sstream>

#include <sys/wait.h>

#include <fcntl.h>

#include <stdio.h>

#include <errno.h>

void processCommand( string input, int seq\_num, queue<string> & history,

bool flag );

void showHist( queue<string> hist\_queue );

bool execute( vector<string> tokens\_queue );

void pipeThis( vector <vector <string> > bigVec, unsigned int place,

int numPipes );

void inputOutputRedirec( vector< vector <string> > bigVec, string tokens );

int main( int Argc, char \*Arg[] )

{

string user\_input,

command\_line;

queue<string> history;

int seq\_num = 1,

flag = 1; // flag for -c

// Environment variables

string userid = getenv( "USER" ),

hostname = getenv( "HOST" ),

firstArg;

if (Argc > 1 ) firstArg = Arg[1];

// If the user typed "-c" into Args, execute the command and quit.

if ( firstArg == "-c" )

{

flag = 0;

for ( int I=2; I < Argc; I++ )

{

command\_line += Arg[I];

command\_line += " ";

}

processCommand( command\_line, seq\_num, history, flag );

}

// Else enter a loop and keep prompting for commands.

else

{

// Loop that prompts for user input

while (true)

{

// Display prompt and get input

cout << "<" << seq\_num << " " << hostname << ":" << userid << " >";

getline(cin, user\_input);

if ( user\_input.substr(0, 4) == "quit" ) break;

else processCommand( user\_input, seq\_num, history, flag );

seq\_num ++;

} // end loop

} // end else

}

/\*----------------------------------------------------------------------------

Name: processCommand

Purpose: Process user input

----------------------------------------------------------------------------\*/

void processCommand( string input, int seq\_num, queue<string> &history,

bool flag )

{

char direcBuffer[256];

const unsigned int QUEUE\_MAX = 10;

vector<string> tokens,

temp,

left,

right;

queue<string> specials,

pipes;

string in\_and\_output;

vector< vector<string> > bigVec;

string build\_hist,

buf,

dirname;

pid\_t mypid,

pID;

stringstream out;

int status;

stringstream ss(input);

// Build the vector of tokens and the vector of vectors

ss << input;

while (ss >> buf) tokens.push\_back(buf);

// Execute user commands if there was user input.

if ( input[0] != '\0' )

{

// Loop through the vector of tokens and store each GROUP delimited by

// |, <, or >

// into its own vector, then store the executable part (as a vector)

// into a vector.

for (unsigned int I=0; I< tokens.size(); I++ )

{

// if ( tokens[I] == "|" || tokens[I] == "<" || tokens[I] == ">" )

if ( tokens[I] == "|" )

{

// Store pipe characters in a queue (and in specials)

pipes.push(tokens[I]);

specials.push(tokens[I]);

// Put temp in the big vector and clear it for next time

bigVec.push\_back(temp);

temp.clear();

}

else if ( (tokens[I] == "<") || (tokens[I] == ">") )

{

// Store in/output characters in a stack (and in specials)

in\_and\_output += tokens[I];

specials.push(tokens[I]);

// Put temp in the big vector and clear it for next time

bigVec.push\_back(temp);

temp.clear();

}

// Otherwise, store it into a temp vector

else temp.push\_back(tokens[I]);

}

// Store the last temp into the big vector

bigVec.push\_back(temp);

// So now do this for EACH vector in the big vector

if (flag)

{

// Add to the history queue

if ( history.size() == QUEUE\_MAX ) history.pop();

out.str("");

out.clear();

out << seq\_num;

build\_hist = out.str();

build\_hist += " " + input;

history.push( build\_hist );

if ( tokens[0] == "quit" ) exit(1);

// If user input is "hist", display history

if ( tokens[0] == "hist" ) showHist( history );

// If user input is "cd", change directories

if ( tokens[0] == "cd" )

{

tokens.pop\_back();

dirname = tokens.front();

if ( chdir(dirname.c\_str()) == -1 ) cout << "Invalid directory name "

<< dirname << "\n";

}

// If user input is "curr", display absolute path of current directory

if ( tokens[0] == "curr" )

{

getcwd( direcBuffer, 256 );

cout << direcBuffer << "\n";

}

// If user input is "curPID", display current process ID

if ( tokens[0] == "curPID" )

{

mypid = getpid();

cout << mypid << "\n";

}

}

// If user input is not a built-in command, it must be external

if ( (tokens[0] != "quit") && (tokens[0] != "hist") && (tokens[0] != "cd" )

&& (tokens[0] != "curPID") && (tokens[0] != "curr" ) )

{

if (bigVec.size() < 2)

{

pID = fork();

// Child process execution

if( pID == 0 )

{

execute( tokens );

exit(1);

}

else if( pID < 0 )

{

cout << "Couldn't create process.\n";

}

// Parent process execution

else

{

// Don't wait if there was a "&" at the end

if ( !(tokens.back() == "&") )

{

while (wait(&status) != pID);

}

sleep(1);

}

} // end big if

else if ( (bigVec.size() - specials.size()) == 1 )

{

if ( in\_and\_output == "<" || in\_and\_output == ">" ||

in\_and\_output == "<>" )

{

int pID = fork();

int status;

if (pID > 0)

{

while(wait(&status) != pID);

sleep(1);

}

else

{

inputOutputRedirec( bigVec, in\_and\_output );

exit(1);

}

}

else if ( specials.front() == "|" )

{

int pID = fork();

int status;

if (pID > 0)

{

while(wait(&status) != pID);

sleep(1);

}

else

{

pipeThis( bigVec, (bigVec.size() - 1), specials.size() );

exit(1);

}

}

else

{

cout << "Ambiguous input/output redirection.\n";

}

}

else

{

cout << "Invalid input.\n";

}

}

} // end

}

/\*----------------------------------------------------------------------------

Name: showHist

Purpose: Display the most recent 10 commands

Input: A vector containing the most recent 10 user commands

----------------------------------------------------------------------------\*/

void showHist( queue<string> hist\_queue )

{

unsigned int limit = hist\_queue.size();

for ( unsigned int I=0; I< limit; I++ )

{

cout << hist\_queue.front() << "\n";

hist\_queue.pop();

}

}

/\*----------------------------------------------------------------------------

Name: execute

Purpose: Call execvp based on the proper parameters

Input: A queue of tokens

----------------------------------------------------------------------------\*/

bool execute( vector<string> tokens )

{

unsigned int limit = tokens.size();

char \*tokenArr[limit];

string myToken;

char \*charToken;

unsigned int I;

if (tokens.back() == "&") tokens.pop\_back();

// Build an array of arguments to pass to execvp

for( I=0; I< tokens.size(); I++)

{

myToken = tokens[I];

charToken = new char[myToken.size()+1];

strcpy(charToken, myToken.c\_str());

tokenArr[I] = charToken;

}

tokenArr[I] = NULL;

if (execvp( tokenArr[0], tokenArr ) < 0)

{

cout << "Error. Failed to execute command.\n";

return 0;

}

else return 1;

}

/\*----------------------------------------------------------------------------

Name: pipeThis

Purpose: Pipe two processes

Input: Two vectors of executables and any additional arguments

----------------------------------------------------------------------------\*/

void pipeThis( vector< vector <string> > bigVec, unsigned int place,

int numPipes)

{

int fd[2],

status;

pipe(fd); // piping two file descriptors

int pid = fork();

// We just called pipeThis, so decrement the number of times left to call it

numPipes --;

if (pid > 0)

{

close(fd[1]);

int dupSuccess = dup2(fd[0],0); //duplicate input to fd[0]

if (dupSuccess < 0) perror("dup2");

execute(bigVec[place]);

wait(&status);

sleep(1);

}

else

{

int dupSuccess = dup2(fd[1],1); // duplicate output to fd[1]

if (dupSuccess < 0) perror("dup2");

// Don't recurse if we're on our last pipe

if (numPipes == 0)

{

execute(bigVec[place-1]);

exit(1);

}

// We still need another pipe to do, so make a recursive call to pipeThis

else

{

place --;

pipeThis( bigVec, place, numPipes );

}

}

}

/\*----------------------------------------------------------------------------

Name: inputOutputRedirec

Purpose: Redirect output into an output file

Input: An executable and the file for output

----------------------------------------------------------------------------\*/

void inputOutputRedirec( vector< vector <string> > bigVec, string tokens)

{

pid\_t pid;

int fd,

fd1,

fd2;

int status;

string myFile,

file1,

file2;

// Output redirection

if ( tokens == ">" )

{

pid = fork();

myFile = bigVec[1][0];

if (pid == 0)

{

fd = open(myFile.c\_str(), O\_WRONLY|O\_CREAT|O\_TRUNC, S\_IRWXU);

if ( fd == -1 ) cout << "Failed to open file.\n";

else

{

dup2(fd, STDOUT\_FILENO); //duplicate output to fd

close(fd);

if ( !execute(bigVec[0]) )

{

cout << "Failed to redirect output.\n";

exit(1);

}

}

}

else

{

close(fd);

wait(&status);

}

}

// Input redirection

else if ( tokens == "<" )

{

pid = fork();

myFile = bigVec[1][0];

if (pid == 0)

{

fd = open(myFile.c\_str(), O\_RDWR);

if ( fd==-1 ) cout << "Failed to open file.\n";

else

{

dup2(fd, STDIN\_FILENO); // duplicate input to fd

close(fd); // Note that STDIN\_FILENO=0

if ( !execute(bigVec[0]))

{

cout << "Failed to redirect input.\n";

exit(1);

}

}

}

else

{

close(fd);

wait(&status);

}

}

else if ( tokens == "<>" )

{

pid = fork();

file1 = bigVec[1][0];

file2 = bigVec[2][0];

if (pid == 0)

{

fd1 = open(file1.c\_str(), O\_RDWR);

fd2 = open(file2.c\_str(), O\_WRONLY|O\_CREAT|O\_TRUNC, S\_IRWXU);

if ( (fd1 == -1) || (fd2 == -1) ) cout << "Failed to open file.\n";

else

{

dup2(fd1, STDIN\_FILENO);

dup2(fd2, STDOUT\_FILENO);

close(fd1);

close(fd2);

if ( !execute(bigVec[0]))

{

cout << "Failed to redirect input/output.\n";

exit(1);

}

}

}

else

{

close(fd1);

close(fd2);

wait(&status);

}

}

}