DASH - a DiAgnostics SHell

Program Description

DASH is a DiAgnostics SHell program. It runs a loop prompting the user for a command with a 'dash>' prompt. A list of valid commands and their usage can be pulled up using the 'help' command. The program handles running a set of intrinsic commands like 'pid', 'cmdnm', 'systat', 'signal', and 'cd' while giving access to Linux commands from the /bin/ directory. The shell can also handle redirecting stdout and/or stdin as well as piping one commands output as another commands input. The program achieves this by running each command as a forked process and then handling each command, if the command is an intrinsic function of the shell then no other process is forked, if the command is not an intrinsic function, another process is forked, and the command is executed on that process instead, returning when finished. Any signal received by the program will be caught by the diagnostic shell as well as the signal number will be displayed when received.

Program Compilation

GCC is required to compile, a Makefile is included or the program can be compiled using the following input:

```
    $ gcc -c dash.c commands.c signalHandle.c
    $ gcc -o dash dash.o commands.o signalHandle.o
```

Program Usage

The program can be ran by calling the dash executable with the following input:

```
    $ ./dash
```

From there the program will give a dash prompt and expect a command. A help menu can be accessed for a list of valid commands and their usages with the following input:

```
    dash> help
```

Files Included

The included files are “dash.c”, “commands.c”, “commands.h”, “signalHandle.c”, “signalHandle.h”, and “Makefile”. Any other necessary files/libraries/headers should come included with GCC.

Function Documentation

dash.c

int main(int argc, char* argv[])

Description: This is the entry point and main loop of the program. The main function handles circulating through the dash commands by looping through gathering input, determining if piping is necessary, forking off child processes to send out to handle a given command, and output process information after each command.

Inputs: int argc – the number of command line arguments
char* argv[] – a list of strings containing command line arguments

Outputs: 0 if no errors

Testing: This function was tested by entering the loop, gathering and outputting the input string to validate fgets worked correctly, passing through various pipe commands and non-piped commands as well as forkable commands and non-forkable commands (cd and exit can't be forked) and then changing the running boolean to see if it would exit the loop.

struct op handleInput(char* instr)

Description: handleInput declares a delimiter variable to identify what to tokenize on, a pointer to a string to hold the token, and a command struct to organize the tokenized data into. We begin by trimming any new lines and tokenizing the input string into its first token, split on space, and add that token as the command's name. We then loop a maximum of 10 times, or until there are no more tokens, and tokenize the rest of the input string adding each new token as a command argument. The number of command arguments are then stored in the command structure and the structure is output

Inputs: char* instr – an input string to be split into tokens

Outputs: struct op – a command structure containing a new command

Testing: Testing this function included giving various input strings and outputting the full command structure, making sure that the command name was always tokenized correctly, and each subsequent argument was tokenized correctly. Different types of strings were given, including null strings, overpopulated strings, etc. The arbitrary limit of 10 command arguments kept the speed fast and allowed for flexibility on future instructions.

void handleCommand(struct op command, int* running)

Description: handleCommand begins with error checking that a rogue null command or newline character doesn't exist, and either exits out of that command or replaces with a null terminator if a newline exists. First the commands are checked for any redirects and handles swapping our input and output file descriptors with the appropriate redirect, then we handle any intrinsic functions. If the given command is not an intrinsic function then we assume it is a Linux script we are running and we re-organize the command structure into a single array, fork the process, and run execvp on using the new command arguments. If the command still can't run then the command is considered invalid and we give an error statement. When all is done we redirect our output back to stdin and stdout and leave the function

Inputs: struct op command – a command structure containing a valid command
int* running – a pointer to an integer used for halting the program

Outputs: None

Testing: The intrinsic functions weren't tested because they were just re-routing commands to helper functions, but the redirects and non-intrinsic commands required outputting the given arguments before and after redirecting and before running a non-intrinsic command to validate the correct arguments existed and that when passed to execvp the Linux scripts ran without issue.

commands.h

This file is used solely for function prototypes and include statements.

commands.c

int systat()

Description: The systat function is a helper function meant to handle the systat command in its entirety. The function ignores any command arguments as it requires none. It begins by creating a file pointer, buffer, a int for storing the size of the bytes read in, and two strings for managing a large substring between two points of a read file. The function repeats itself 3 times and has a fourth section that is like the first 3 but includes a small subsection to get a specific substring of the larger file. The pattern uses the file pointer to open either "version", "uptime", "meminfo", or "cpuinfo" in the /proc/ directory and the file is read in. Once read in, the contents are dumped to stdout with a header letting the user know the type of file that was read. "/proc/cpuinfo" is the only file that is handled differently in that the file is read in, then two different copies are made from the "vendor_id" line down to eof and "physical id" down to eof, the two substring lengths are subtracted from each other and a null terminator is put in the character index of that difference in the "vendor_id" substring. This string is then output to the terminal instead of the file buffer.
Inputs: None
Outputs: 0 if successful
1 if unable to open/read /proc/version
2 if unable to open/read /proc/uptime
3 if unable to open/read /proc/meminfo
4 if unable to open/read /proc/cpuinfo
Testing: This function only required two different tests. One test was set up for reading in the file and displaying its contents to stdout, this was done by opening the file and checking for NULL values when given various known good and known bad inputs and then reading to the buffer. Once the files were being opened properly the file was read into a buffer and output to stdout where it was checked for accuracy. The cpuinfo file required a bit more testing by trying out different methods of creating a substring and validating that the difference in substring would truncate the first substring to the appropriate size, regardless of the output or order of the output (as long as vendor_id always appeared above physical_id).

int cmdnm(char* pid)

Description: cmdnm takes in a process id as an input string. This string is first checked for a NULL value, if true, the function outputs the default 'not found' message and exits the function. If the input is not NULL, the string is checked for any '\n' endings and replaces any with a '\0'. Once the string is prepared the string is appended to a "/proc/" string and a "/comm" string is attached to that string, this newly made string is a complete file path to a valid process name for a valid process id. At this point a file pointer, character buffer, and integer for bytes read in are created and the file pointer opens the file given by the file path string. If the file did not open successfully, we can assume that the file and folder are nonexistent and that the given process id is invalid, we then output an error message letting the user know the input is invalid. If the file opened successfully, we read the contents of the file and output the contents to stdout, this gives the user the process name for the entered process id.
Inputs: char* pid – an input string representing a valid process id
Outputs: 0 if successful
1 if unable to open/read the process folder/file (invalid pid)
2 if returned from a NULL input string

Testing: This function was tested by entering in NULL strings, empty strings, overly large strings, invalid process id strings, valid process id strings, character strings, mixed alphanumeric strings, etc. The function keeps the input as a string so the issues with conversions between string and int don't appear, as long as the given input has a valid folder in the /proc/ directory and a /comm file in that folder, the file will be read and output.

int pid(char* cmdnm)
Description: pid is given a string representing a valid process name as input. If given a NULL input, the function will output a message and immediately exit, otherwise it will try to replace any ending '\n's with a '\0' instead and enter the /proc/ directory. All files in the proc directory are read, but the folders are filtered by converting each directory to a long, if the long value isn't converted the folder is ignored, otherwise we traverse into that folder's /comm file and compare the contents of /comm to the given input string, if the two match, the process ID is output to stdout and the function moves to the next folder. When all folders have been searched, the function exits. If no matches occur, an error is displayed to let the user know that command name does not have any valid process id's linked with it.
Inputs: char* cmdnm – an input string representing a valid process name
Outputs: 0 if successful
1 if the /proc/ directory could not be opened
2 if the given input string is a NULL value
Testing: To test pid() I gave it invalid input, valid input, no input, unexpected input, etc. Originally, upon entering the /proc/ folder all contents were displayed, then a filter was placed to only let folders with a numeric name be displayed, when the filter worked and was validated to be showing the correct folder, the output was replaced with a sub-function similar to the one in the cmdnm command.

int cd(char *path)
Description: cd is a function that runs the chdir function to change the working directory given a valid path. If the given path is invalid, chdir will fail and perror will output a message explaining why.
Inputs: char *path - the relative or absolute path to change to
Outputs: 0 if successful directory change
-1 if the directory could not switch
Testing: The cd command was tested by giving valid and invalid absolute and relative addresses, on success I expected no output and an 'ls' command to show the changed directory, and on failure I expected a "could not change directory" message and an 'ls' to show no directory change.

int sig(char **args)
Description: sig handles sending a signal number to a specific process. The signal number is passed as the first argument and the pid to send it to is given as the second argument. The function validates the two arguments and then runs the kill function to send the signal if it can. If any issues come up a -1 is returned out, else a 0 is returned.
Inputs: char **args - a list of command arguments
Outputs: 0 if successful
-1 if the input is invalid or kill could not run
Testing: sig was tested by sending signals to various processes running, including the dash process itself, and expected a response of some kind from these processes. When testing signals sent to itself, the signal handling subroutines would spit out which signal the process caught.

int help()
Description: help is just a function that displays information about other valid commands. The function is split into sections where each section outputs command information in the following format:

```
    CMD_NAME [REQUIRED_ARG0] {OPTIONAL_ARG}
     - Function_Description and Notes
```

Inputs: None
Outputs: 0 if successful
Testing: No testing went into this function, just spell-checking.

int toInt(char *str, int *val)
Description: toInt converts a given input string from a c string to an integer, given the input string contains a valid integer and only a valid integer. The output is given through a pass by reference argument and the validity of the input string is given by the function's return statement.
Inputs: None
Outputs: *val – a pass by reference integer containing a converted int
 0 if the string is valid
-1 if the string contains an invalid character (non-integer)
Testing: No testing went into this function, just spell-checking.

signalHandle.h

This file is used solely for handling signals and holding function prototypes and include statements.

signalHandle.c

void handle_signal(int signal)
Description: This is an entry point for incoming signals. All signals are passed through this function first, with a generic print statement declaring what signal has been caught. If the signal can be handled in another way, such as exiting from a SIGSEGV or SIGTERM then this function will pass the control over to that function to handle the given signal.
Inputs: int signal – The signal number being handled
Outputs: None
Testing: Several signals were sent using the kill command and signal command from a separate bash terminal and a separate dash terminal, the sent signal was verified by checking the received signal against it.

Makefile

Description: A script file used to build a program using a set of rules defined in the script. This file first creates *.o files in our case and then links the *.o files into a program executable called dash.
Usage: Can be invoked using 'Make' in a BASH environment.