`dsh`: A Diagnostic Shell

Noah Brubaker

March 14, 2016

Course: CSC456 – Operating Systems

Instructor: Dr. Jeff McGough

Program Description

This program is a simple diagnostic shell that will emulate some of the functionality of the standard Bash shell. The main purpose of this shell is process identification, and to provide a platform for further development.

The program will provide the following features:

- **Prompt:** The prompt `dsh>` will be displayed. This is where the user will enter commands.
- **Intrinsic Commands:** Seven shell intrinsic commands will be implemented: `cmdnm, signal, systat, exit, cd, pwd, hb`. `cmdnm` prints the command that initiated a process. `signal` sends a signal to another process. `systat` displays some information about the system, including version, uptime, memory usage, and CPU info. `exit` will exit the shell nicely. `cd` implements the `chdir` command to change the directory via the relative or absolute path provided. `pwd` prints the working directory. `hb` prints the current time every specified interval for a specified length of time.
- **Single Program Command:** Any single command (plus arguments), will be executed by the shell and return any stdout.
- **Pipes and Redirects:** This shell supports nearly unlimited piping, with the proviso that all pipes must come before redirects or remote pipes (which really have the flavor of redirects). Redirects include appending writes.
- **Pthreads:** This program uses threading for education purpose on `cmdnm`, `systat`, and `hb` intrinsic shell commands.

Submission Details

The submission includes a tar-ball, `prog2.tgz`, which contains all files relevant to the program. This includes the source code, Makefile, and documentation.

`prog2.tgz` contains:

- `dsh.c`: This file implements the command prompt, command line input, input parsing, and the main event loop for the program.
- `run.c` This file implements the intrinsic commands, as well as `fork/exec` for single commands with arguments.

- `special.c` This file implements the new file piping and redirection commands. This file could easily be adapted to handle all fork and exec needs for the shell.
- `Makefile` This file builds the program `dsh` using source files `dsh.c`, `run.c`, and `special.c`.
- `prog2.pdf` This file provides documentation for the program `dsh`, its source files and its Makefile.

Compilation and Usage

The Makefile builds the program in the following way:

```
gcc -o -lm -pthread special.o run.o dsh.o
```

Others files are compiled individually using `-c` instead of `-o` to create object files.

The program can be run by typing `dsh` is a bash shell.

Libraries

The source code includes the following libraries.

- `assert.h`
- `pthread.h`
- `netdb.h`
- `netinet/in.h`
- `signal.h`
- `stdio.h`
- `string.h`
- `stdlib.h`
- `sys/resource.h`
- `sys/socket.h`
- `sys/stat.h`
- `sys/types.h`
- `sys/wait.h`
- `time.h`
- `unistd.h`

Structure and Functions

The general flow of the program has the following format.

Program structure

```
do
 getInput - gets user input at command line
 parseInput - parse said input
 parseSpecial - looks for special operators
 status = handler(input,mode)
while status ≡ 0
```

Function Descriptions

Defined in dsh.c

Name: `dsh_prompt[dsh.c(34)]`

Description:
This function receives command line input for the shell. The storage is dynamically allocated for the input stream in blocks of 256 bytes.

Output:

`char** input` A pointer to a character array which will store the input taken at the prompt.

Returns:

`int -1` Failed to allocate memory for input

`int 0` Function successful took input

`int 1` No input received on commandline

`int 2` Exit command received

Name: `parse_input[dsh.c(106)]`

Description:
This function parses input gathered from the command line.

Input:

`char * input` The input string returned by prompt.

Output:

`char *** argv` A pointer to the new parsed argument list, passed by reference.

Returns:

`int argc` The number of arguments in the input string.

Name: `run_command[dsh.c(222)]`

Description:
This function takes the argument list from Main and directs it to either the fork/exec code for single functions or to the intrinsic commands.

The first argument is expected to be the command name.

Input:

`int args` Number of arguments

`char ** arg_list` List of arguments

Returns:

`int ret` Returns the valued returned by `Run` or `New_Process`.

Name: `main[dsh.c(251)]`

Description:
This function implements the main event loop for the shell. It waits for the exit command to terminate.

Returns:

`int 0` Always returns 0.

Defined in run.c

Name: `cmdnm[run.c(46)]`

Description:
This function gets the command that started a process by accessing `/proc/<pid>/comm`. NEW: This command now uses thread function `cmdnm_getCmdnm` to get the name. The thread passes this info along to the main code.

Input:

`char * pid` A character array holding the process identification number.

Returns:

`int 0` Successful.

`int -1` Couldn't find process.

Name: `send_signal[run.c(76)]`

Description:
This function sends a signal to a process using the kill command. It checks if the arguments are in the proper ranges, switching them if not.

Input:

`char * sig_no` A character array holding the desired signal number.

`char * process_id` A character array holding the process identification number.

Returns:

`0` Successful.

`-1` Failed to send signal to process.

Name: `systat[run.c(195)]`

Description:

This function gets some information about the system and displays it for the user in stdout. The specific information it provides is as follows:

-Linux version and system uptime

-Memory Usage: memtotal and memfree

-CPU Information: vendor id through cache size

NEW: Threads now go and fetch each piece of output concurrently.

Returns:

`int 0` Successful.

`int neg` Couldn't access directory.

Name: `cd[run.c(242)]`

Description:

This function implements the change directory intrinsic command.

Input:

`char * path` The absolute or relative path to the desired directory.

Returns:

`0` Successful.

`-1` No such file or directory.

Name: `pwd[run.c(269)]`

Description:

This function implements the print working directory intrinsic command.

Returns:

`0` Always returns 0.

Name: `hb[run.c(348)]`

Description:

Prints the current time every `<tinc>` s/ms until `<tend>`.

Input:

`int tinc` The desired time increment.

`int tend` The desired amount of time to wait.

`char * tval` Should be either s for second of ms for milliseconds.

Returns:

`0` Successfully completed.

Name: `Run[run.c(425)]`

Description:
This function directs the program to run the intrinsic commands, checking for correct number of arguments where applicable. Uses shared memory buffer in for communication between thread and process.

Input:

`int cmd_num` Number specifying desired command.

`int args` The number of arguments.

`char ** arg_list` The null-terminated list of arguments.

Returns:

`int ret` The return value of function it calls

`int neg` Wrong number of inputs or similar error.

`int 2` Exit code.

Name: `New_Process[run.c(480)]`

Description:
Creates a new process to run the given single command received at the command line in the diagnostic shell.

Input:

`char ** arg_list` The list of arguments for the given command.

Returns:

`int 0` If fork and exec operations were successful.

`int -1` An error occured. Either couldn't find command or failed to execute it.

Defined in special.c

Name: `parse_set_free[special.c(77)]`

Description:
This function frees a struct `parse_set` pointer.

Input:

`struct parse_set * arg_set` Thing to be freed.

Name: `pipeBuilder[special.c(113)]`

Description:
This function finds and packages all necessary pipe commands. Result is an array of NULL-terminated arglists which can be used by `execvp`.

Input:

`int command_count` Number of commands involved in pipes.

`int arg_list` The original `arg_list` from `parse_input` to be divided and packaged.

Output:

`pipe_builder * pip` Returns a pointer to a `pipe_builder` struct storing the number of commands and the `arg_lists` package.

Name: `parse_special[special.c(163)]`

Description:
Parses the argument list returned by `parse_input` by looking for special redirection operators and handling everything accordingly. Arguments are passed to other functions using `spec_res` which wraps the main `parse_set` struct pointer.

Input:

`char ** arg_list` The original argument list to be parsed.

Output:

`char ** spec_res` holds address of the `parse_set` struct pointer which is generated by the function.

`int mode` The modes of operation due to operators.

Name: `dclient[special.c(406)]`

Description:
This function returns the file descriptor to the socket located at port on server.

Input:

`char * port` The port number to connect to.

`char * ipv6` The ip address to connect to.

Return:

`int socket_fd` The file descriptor for the socket.

Name: `dserver[special.c(454)]`

Description:
This creates a new port on the local server in preparation for the sending pipe.

Input:

`char * port` The port number to use.

Return:

`int newsocket_fd` The file descriptor for the socket.

Name: `run_special[special.c(498)]`

Description:
This function does all the awesome stuff that the program document asks us to do. It implements pipes, remote pipes, and redirects. It takes the special struct returned by `parse_special` and does magic with it.

Input:

`char * spec_res` Wrapper for magical struct storing extra-parsed information.

`int mode` The thing that tells use what needs to get done.

Return:

`int status` Returns 0 for success!

Testing and Verification

This program was lightly tested and verified by trying each required command at separate times. The code was developed by continually adding functionality to a functional program. Since each required feature was largely independent of the others, debugging was straight-forward.

A known bug: The overly complicated parsing function doesn't always provide the proper error message for some parse errors and will just return saying nothing.

There are no known major errors, however, more testing could expose some.