Program 1: Diagnostic Shell

CSC 456 - Operating Systems

South Dakota School of Mines and Technology
Dr. Jeff S. McGough

Christine N. Sorensen
2/12/16

Description

Diagnostic shell. A process identification program and a simple shell.
Purpose: To introduce Unix environment, system calls, signal, the proc system, Harvard commas, and for/exec system calls.

When ran, it should look like this:

```
dsh>
dsh> pwd
/home/student/1959287/Documents/csc456/dsh
dsh>cd ..
dsh> pwd
/home/student/1959287/Documents/csc456
dsh> cd /home/student/1959287/Downloads
dsh> pwd
/home/student/1959287/Downloads
dsh> ls –a
.     ..    spankjeff.gif    twilight.txt twilight_german.mp4
dsh> cmdnm 1
/user/lib/system/system
dsh>systat
Linux version: Linux version 4.2.8-200.fc22.x86_64
(mockbuild@bkernel02.phx2.fedoraproject.org) (gcc version 5.3.1
20151207 (Red Hat 5.3.1-2) (GCC) ) #1 SMP Tue Dec 15 16:50:23 UTC 2015
System uptime: 333390.32 2369338.04
MemTotal: 16307348 kB
MemFree: 12657404 kB
vendor_id    : GenuineIntel
cpu family    : 6
model        : 60
model name    : Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz
stepping    : 3
microcode    : 0x1c
cache size    : 8192 KB
vendor_id    : GenuineIntel
dsh> signal 9 4
dsh> exit
1959287@linux07 dsh >>
```

Compile

```
>gcc dsh.c –o dsh
```

A Makefile is also included with the program by running:

>make

```
#---------------------------------------------------------------------------
# GNU C compiler and linker:
LINK = gcc
```

```
# Preprocessor and compiler flags (turn on warning, optimization, and debugging):
# CPPFLAGS = <preprocessor flags go here>

CFLAGS = -Wall -O -g
CXXFLAGS = $(CFLAGS)
#---------------------------------------
# Targets
all: dsh
#-----------------------------------------------------------------------
dsh: dsh.o
	$(LINK) -o $@ $^ $(LIBS)
#-----------------------------------------------------------------------
clean:
	rm -f *.o *~ core
cleanall:
	rm -f *.o *~ core dsh
```

Usage

```
>./dsh
```

Libraries

unistd.h – used for standard and miscellaneous symbolic constants, types, and functions.
Used for functions chdir(), get_current_dir_name(), fork(), and execvp()

stdlib.h – general purpose library for C that handles process control, memory allocation, and conversions.
Used for exit() and atoi()

stdio.h – standard library for file input and output.
Used for fopen(), fclose(), fgets(), printf(), and type FILE*.

string.h – library to manipulate cstrings
Used for strtok(), strlen(), strcmp(), strcat().

ctype.h – c standard library used for mapping characters.
Used for tolower()

signal.h – library for informing processes of events.
Used for kill()

sys/types.h – Copyright © 2002 by Sunmicrosystems, Inc. Contains various types.
Used for size_t.

fcntl.h – Copyright © 2007 Apple Inc. Library for file control options

sys/wait.h – Copyright © 2007 Free Software Foundation, Inc. For the delectations of waiting.
Used for wait()

Functions:

Main

Start of the dsh program. Creates a flag that is used while loop. The loop continuously calls the program. Each time, >dshThe flag value will change by the user when `exit` is submitted and therefore leave the loop, thus ending the program.

```
int main( int argc, char** argv )
{
    // flag to control the loop
    int flag = 1;

    // each line read in from the user
    char line[80];

    while( flag == 1 )
    {
        // print the shell name
        printf( "dsh> " );

        // get the line from the user
        fgets( line, sizeof(line), stdin );

        // remove that annoying newline character at the end
        // that fgets seems to have necessary to include
        size_t len = strlen( line );
        if( len > 0 && line[len-1] == '\n' )
        {
            line[--len] = '\0';
        }

        // call the dsh function
        flag = dsh( line );
    }

    return 1;
}
```

dsh

The heart and soul of this program. This is called by main continuously from main(). It takes a line that the user submits, tokenizes the words from it into an array. The first word of the array is considered the

command. It is compared with the commands written for this program along with the number of parameters (it only checks for the minimum needed parameters. If there is more than necessary, the program will ignore them). It calls the corresponding function and passes its parameters. If a user input does not match one of the commands, it is passed into dsh_fork().

Each function returns a value, stored in return_val. It the value is -1, it means there was an error. The commands are then sent to dsh_fork(). The return_val is returned at the end of dsh() function. This tells the while loop in main() to keep going or not.

```
int dsh( char* line )
{
    // delimiters, split at spaces and tabs
    const char delim[3] = " \t";

    // string to hold each token
    char* token = NULL;

    // array to hold the tokens
    char* tokens[64];
    int i = 0;

    // extract the first word
    token = strtok( line, delim );

    // collect the words
    while( token != NULL )
    {
        // add to list of tokens
        tokens[i] = token;
        i++;

        // get the next word
        token = strtok( NULL,delim );
    }

    // call the appropriate function
    // check for the correct number of parameters
    // if there are more parameters than needed, it will ignore the
excess
    // lowercase the command
    char* command = toLowerCase( tokens[0] );

    // default to return successful
    int return_val = 1;

    // the number of parameters to check for
    int num_params = i;
    if( strcmp( command, "cmdnm" ) == 0 )
```

```
    {
        if( num_params >= 2 )
        {
            // check to see if pid is an int
            int flag = isInt( tokens[1] );
            if( flag != -1 )
            {
                // pass in the pid
                return_val = cmdnm( tokens[1] );
            }
            else
            {
                return_val = -1;
            }
        }
        else
        {
            return_val = -1;
        }
    }
    else if( strcmp( command, "signal" ) == 0 )
    {
        // pass in the signal number and pic
        if( num_params >= 3 )
        {
            int i1 = isInt( tokens[1] );
            int i2 = isInt( tokens[2] );
            if( i1 != -1 && i2 != -1 )
            {
                return_val = dsh_signal( atoi(tokens[1]),
atoi(tokens[2]) );
            }
        }
        else
        {
            return_val = -1;
        }
    }
    else if( strcmp( command, "systat" ) == 0 )
    {
        return_val = systat();
    }
    else if( strcmp( command, "exit" ) == 0 )
    {
        return_val = dsh_exit();
    }
    else if( strcmp( command, "cd" ) == 0 )
    {
```

```
        // pass in path
        if( num_params >= 2 )
        {
            return_val = cd( tokens[1] );
        }
        else
        {
            return_val = -1;
        }
    }
    else if( strcmp( command, "pwd" ) == 0 )
    {
        return_val = pwd();
    }
    else if( strcmp( command, "kill" ) == 0 )
    {
        if( num_params >= 3 )
        {
            int i1 = isInt( tokens[1] );
            int i2 = isInt( tokens[2] );
            if( i1 != -1 && i2 != -1 )
            {
                return_val = dsh_kill( atoi(tokens[1]),
atoi(tokens[2]) );
            }
        }
        else
        {
            return -1;
        }
    }
    else
    {
        // it it doesn't match one of the custom commands above,
        // fork it
        return_val = dsh_fork( tokens, num_params );
    }

    // if one of the custom commands failed, fork it
    if( return_val == -1 )
    {
        return_val = dsh_fork( tokens, num_params );
    }

    // return val determines whether to continue the dsh loop
    return return_val;
}
```

Command functions

These functions are specifically towards the dsh program.

cmdnm

This returns the command string that started the process from the process id it was given. The pid is passed in. The filename containing the command name is located in /proc/<pid>/cmdline. When the pid is passed in, it is concatenated with /proc/ and /cmdline in order to find that file. The file opened (or an error is returned if it fails to open). The function simply grabs the string containing the command name and prints it. 1 is returned for success.

```
int cmdnm( char* pid )
{
    FILE* fin;
    char filename[32] = {NULL};

    // create the filename with the pid
    strcat( filename, "/proc/" );
    strcat( filename, pid );
    strcat( filename, "/cmdline" );

    // holds the output of the cmdnm file
    char cmdnm[128] = {NULL};

    // open the file with the name of the process
    if( (fin = fopen( filename, "r" ) )  == NULL )
    {
        // if it fails
        return -1;
    }

    // grab the pid's name
    fgets( cmdnm, sizeof( cmdnm ), (FILE*)fin );

    // print to output
    printf( "%s\n", cmdnm );

    // return to continue loop
    return 1;
}
```

dsh_signal

This function signals a process. It is given a pid to signal and the value to signal to it. It only calls dsh_kill() with the parameters which will be explained later in the document.

```
int dsh_signal( int signal_num, int pid )
{
```

```
    // call the signal using the kill function
    dsh_kill( pid, signal_num );

    return 1;
}
```

sysat

This function prints out various stats about the system using the /proc/* files. It prints:

- Linux version
- System uptime
- Memory usage:
  - Memory total
  - Memory free
- CPU information:
  - Vendor ID through Cache size found in /cpuinfo

This function opens each corresponding file, retrieves the needed data, and prints it to the screen.

```
int systat()
{
    // used for all file opening
    FILE* fin;

    //** linux version **//
    // found in /proc/version
    char version[256];

    // open the file containing the version of linux
    if( ( fin = fopen( "/proc/version", "r" ) ) == NULL )
    {
        return -1;
    }

    // get the version from the file
    fgets( version, sizeof( version ), (FILE*)fin );

    // Remove that newline
    size_t len = strlen( version );
    if( len > 0 && version[len-1] == '\n' )
    {
        version[--len] = '\0';
    }

    // close the version file
    fclose( fin );

    // print that sob out!
    printf( "Linux version: %s\n", version );
```

```
    //** system uptime **//
    // found in /proc/uptime
    char uptime[64];

    // open uptime file
    if( ( fin = fopen( "/proc/uptime", "r" ) ) == NULL )
    {
        return -1;
    }

    // get the uptime from the file
    fgets( uptime, sizeof( uptime ), (FILE*)fin );

    // Remove that newline
    len = strlen( uptime );
    if( len > 0 && uptime[len-1] == '\n' )
    {
        uptime[--len] = '\0';
    }

    // close the uptime file
    fclose( fin );

    // print uptime
    printf( "System uptime: %s\n", uptime );

    //** memory usage: memtotal and memfree **//
    // found in /proc/meminfo
    char* memtotal = NULL;
    char* memfree = NULL;
    char line[256];
    int memFlag = 0;
    char* label = NULL;
    char* metric = NULL;

    // open up folder with the memory information
    if( ( fin = fopen( "/proc/meminfo", "r" ) ) == NULL )
    {
        return -1;
    }

    // delimiter to tokenize the memory info
    const char delim[3] = " \t";

    // get the entirty of meminfo one line at a time
    while( fgets( line, sizeof( line ), (FILE*)fin ) )
    {
        // extract the label
```

```
        label = strtok( line, delim );

        // if it's the memory free information
        if( strcmp( label, "MemFree:" ) == 0 )
        {
            memFlag++;

            // extract the value and the metric
            memfree = strtok( NULL, " \t" );
            metric = strtok( NULL, " \t" );

            // gaaahhhhh, take out that newline!!!!
            len = strlen( metric );
            if( len > 0 && metric[len-1] == '\n' )
            {
                metric[--len] = '\0';
            }

            // Print the memory free to the console
            printf( "MemFree: %s %s\n", memfree, metric );
        }

        // if it's the memory total information
        else if( strcmp( label, "MemTotal:" ) == 0 )
        {
            memFlag++;

            // grab the value and the metric
            memtotal = strtok( NULL, " \t" );
            metric = strtok( NULL, " \t" );

            // Remove that gd newline at the end of the metric
            len = strlen( metric );
            if( len > 0 && metric[len-1] == '\n' )
            {
                metric[--len] = '\0';
            }

            // print out the memory total
            printf( "MemTotal: %s %s\n", memtotal, metric );
        }

        // stop reading the file once you get the two needed values
        if( memFlag == 2 )
        {
            break;
        }
    }
```

```
    // close the meminfo file
    fclose( fin );

    //** print vendor id through cache size **//
    // found in /proc/cpu info

    // count for number of lines needed to print
    int cpuFlag = 0;
    char linePrint[256];

    // delimiter to tokenize the memory info
    const char delim2[4] = " \t";

    // open the file with the cpu info
    if( ( fin = fopen( "/proc/cpuinfo", "r" ) ) == NULL )
    {
        return -1;
    }

    // get entirity of cpuinfo one line at a time
    while( fgets( line, sizeof( line ), (FILE*) fin ) )
    {
        // copy the read in line to a variable that will print it
        strcpy( linePrint, line );

        // extract the label
        label = strtok( line, delim2 );

        // compare the labels
        if( strcmp( label, "vendor_id" ) == 0 )
        {
            printf( "%s", linePrint );
            cpuFlag++;
        }
        else if( strcmp( label, "cpu" ) == 0 )
        {
            if( strcmp( strtok( NULL, delim2 ), "family" ) == 0 )
            {
                printf( "%s", linePrint );
                cpuFlag++;
            }
        }
        // Note here: model name will short circuit here before seeing
the next
        // else-if to avoid printing a duplicate line with model
        else if( strcmp( label, "model" ) == 0 &&  strcmp( strtok(
NULL, delim2 ), "name" ) == 0 )
```

```
        {
            printf( "%s", linePrint );
            cpuFlag++;
        }
        else if( strcmp( label, "model" ) == 0 )
        {
            printf( "%s", linePrint );
            cpuFlag++;
        }
        else if( strcmp( label, "stepping" ) == 0 )
        {
            printf( "%s", linePrint );
            cpuFlag++;
        }
        else if( strcmp( label, "microcode" ) == 0 )
        {
            printf( "%s", linePrint );
            cpuFlag++;
        }
        else if( strcmp( label, "cpu" ) == 0 && strcmp( strtok( NULL,
delim2 ), "MGz" ) == 0 )
        {
                printf( "%s", linePrint );
                cpuFlag++;
        }
        else if( strcmp( label, "cache" ) == 0 && strcmp( strtok(
NULL, delim2 ), "size" ) == 0 )
        {
            printf( "%s", linePrint );
            cpuFlag++;
        }

        // If you have printed all of the needed cpu infos, stop
reading from file
        if( cpuFlag >= 8 )
        {
            break;
        }
    }

    // close the cpuinfo file
    fclose( fin );

    return 1;
}
```

pwd

This prints the current working directory to the screen. It utilizes the function get_current_dir_name() to retrieve it and print it.

```
int pwd()
{
    // get the current working directory
    char* cwd;
    cwd = (char*)get_current_dir_name();

    // print to console
    printf( "%s\n", cwd );

    return 1;
}
```

dsh_kill

This function uses kill() to signal a process with a number.

The signal values:

- 1 – Hang up
- 2 – Interrupt from keyboard
- 9 – Kill signal
- 15 – Termination signal
- 17, 19, 23 – Stop the process

```
int dsh_kill( int pid, int signal_num )
{
    // signal pid. set return to flag to return
    int flag;
    flag = kill( pid, signal_num );

    // return the result from the kill process
    return flag;
}
```

dsh_fork

This is the fork/exec function handler.

The layout of this function is similar to Dr. McGough's code which was taken from "Advanced Linux Programming," by CodeSourcery LLC

This function takes the arguments from the user input as well as the number of params. The process is duplicated. It is check to see if it's the child process. If it's the parent process, the function waits. If it's the child process, the function calls execvp() with the arguments.

```
int dsh_fork( char* args[], int num_params )
{
    // duplicate the process
    int c_pid;
    c_pid = fork();
```

```
    // child process
    if( c_pid == 0 )
    {
        // execute the program
        execvp( args[0], args );
        // only returns when an erro occurs
        exit(0);
    }
    // parent process
    else
    {
        int waiting;
        wait( &waiting );
    }

    return 1;
}
```

Miscellaneous functions

These functions were used for supporting tasks to the program.

toLowerCase

This function takes in a string and turns it to all lowercase. This is used for to avoid errors if the user mistypes a command in uppercase.

```
char* toLowerCase( char* token )
{
    int i = 0;
    char c;

    // go through each letter in the string
    while( token[i] )
    {
        c = token[i];
        token[i] = tolower( c );
        i++;
    }

    // return the newly lower-cased token
    return token;
}
```

isInt

This function is passed in a character string and determines whether it is an integer or not. It first checks to see if the first character is a minus sign for negative. The it checks the other characters to see if the ASCII value is in the boundaries of zero(48) and nine (57). It returns with a -1 if it hits a non-integer character. 1 is returned if it is an integer.

This function was used in error checking from the user. If a command required integer parameters, it was checked for valid parameters.

```
int isInt( char* num )
{
    int i = 0;

    // if it's a negative number, ignore the - sign
    if( num[0] == 45 )
    {
        i = 1;
    }

    // go through each decimal in the number passed in
    while( num[i] )
    {
        // if it's in the boundaries of an int
        if( num[i] < 48 || num[i] > 57 )
        {
            return -1;
        }

        i++;
    }

    // else, it's an int
    return 1;
}
```