# DASH - a DiAgnostics SHell

## Program Description

DASH is a DiAgnostics SHell program that can be queried for information from the /proc/ directory of a Linux system. When ran, the program will prompt the user for input with “dash> ” on stdout. The input is given through stdin and tokenized. The first token will always be considered the command name and any subsequent tokens will be considered command arguments, an arbitrary maximum of 10 arguments can be given per input string, anything past that will be ignored. Any arguments given in excess to the command’s required arguments will also be ignored and only the required arguments will be used. Currently the program can find process names when given a process id using the cmdnm command, find all process ids with a name matching a given process name using the pid command, pull up system version, uptime, memory info, and CPU info when given the systat command, or bring up a help menu when given the help command.

## Program Compilation

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

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

## 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”, and “Makefile”. Any other necessary files/libraries/headers should come included with GCC.

## Function Documentation

### *main.c*

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

**Description:** This is the entry point of the program. We begin by declaring a buffer for our input string, a boolean for exiting the main loop, and a command structure to organize our input into commands. We prompt the user

with a "dash> " prompt, get their input, pass it to a tokenizer function, and then handle the input as a command. We loop indefinitely until our boolean has been changed by an external function.
**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, 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 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 first checks that the given command structure doesn't have a stray '\n' at the end of the string, then attempts to match the command structures name to a valid command and calls that commands helper function. If no valid command is found, the user is prompted to type 'help' for a list of valid commands and their usage. The only exception to this pattern is if the command name is 'exit', where instead of passing the command to a helper function, the function will modify a pointer to an integer by adding 1 to it and exit the function, this modified integer will cause the program to exit.
**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:** This function required little testing because it was meant to route a given command to its helper function. The little testing that did occur was on validating that the input command name wasn't malformed for any types of input and that the if blocks were passing to the helper functions correctly.

### *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 similar to 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 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.

## *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.