DASH
Documentation

Zachary Pierson

January 22, 2015

Contents

1 Introduction

The c program dash is a result of the Computer Science Course 456, Operating Systems. The purpose of assignment 1 is to familiarize ourselves with the proc filesystem. Each process has a unique process ID, pid, associated with the process. the kernel exports a significant amount of process information in the /proc directory. There you can see numerical directories, indicating running processes. You can see some examples by typing ps aux at the command prompt of the shell.

1.1 Description of Program

dash is a process identification program. It allows the user to find process id's based on a command string, and find command strings based on process id's. dash is also capable of providing the user with process information.

Once invoked, dash will enter an event loop awaiting user input. Acceptable command options can be found in Table 1 on page 3.

1.2 Compiling Instructions

The file can be found on github at: https://github.com/zjpierson/operatingSystems/prog1 The Makefile supplied in the git repository is set up to compile the program into an executable called dash. This can be achived simply by typing `make` on the command line. To compile using gcc on the command line:

```
gcc -o dash dash.c cmdArgs.c
```

1.3 Using Dash

First you must run the executable. This will automatically start the dash command prompt. You will then be able to use the available commands as seen in Table 1 on page 3 for information on processes.

```
user@host$ ./dash
dash>
```

Command	Description
cmdnm	finds command string based on given pid
pid	finds pid based on given command string
systat	process information dump
help	prints information on available commands
exit/quit	exits the program

Table 1: dash commands

The `cmdnm` command has an optional `[pid]` argument for finding the command string. If no arguments are passed, dash will return a list of all command strings. An example of it use is below.

```
dash> cmdnm 1
systemd
dash> cmdnm
Please specify pid.  Here is a list of all cmd names:
systemd  kthreadd   ksoftirqd/0
rcuos/0  bioset     ksmd
crond    watchdog/3 bash
dash
```

The `pid` command is practically a mirror image of the `cmdnm` command. It takes an optional command string argument `[cmdnm]` and returns a list of all pid's that has a matching substring with the `[cmdnm]` argument. If no arguments are passed, dash will retun a list of all process id's. An example of its use is below.

```
dash> pid systemd
1
dash> pid
Please specify cmd name.  Here is a list of all pid's:
1  2  3  5
7  8  9  10
11 12  13
14
```

The `systat` command prints diagnostic information about the process. There is no argument call to this function. The following information is what `systat` will return:

1. Linux Version
2. System Uptime
3. Total Memory
4. Free Memory
5. CPU information
6. Cache size

The `help` command displays usage information about the program and possible arguments for the command prompt. Finally the next two commands `quit` and `exit` terminates the program and can be used interchangeably. In fact they do **exactly** the same thing as they call the same function.

Libraries	Functions Used
<stdio.h>	`fopen(), fclose(), fscanf()` `printf(),fgets()`
<string.h>	`strcpy(), strcat(), strcmp(), strch()` `strspn(), strstr(), strtok(), strlen()`
<stdlib.h>	`exit()`
<dirent.h>	`opendir(), readdir(), closedir()`

Table 2: Library Functions

1.4 Libraries Used

As Illustrated in Table 2, the included libraries provide the program with useful functions. The `<stdio.h>` library provides standard input output functionality to communicate with the user. These functions are scattered everywhere within the program code.

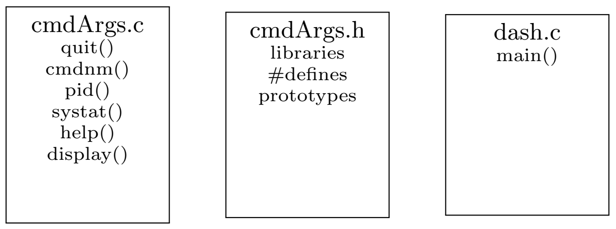
The `<string.h>` library is used for string manipulation. This library has the most diverse use of its functions mainly because there is a lot of string parsing that happens within the proc filesystem in order to retrieve process information.

The `<stdlib.h>` library is included solely for one purpose; to allow the quit function to exit the dash process when called on the command prompt. This library is not essential because there are other ways to quit the program.

The `<dirent.h>` library is used to find all directory names within /proc. This is especially useful when the pid of a command string is unknown and we have to look in every process directory to find the matching command string.

1.5 Program Structure

The program is broken up into 3 separate files: dash.c, cmdArgs.c, and cmdArgs.h. A structure that is important to note is the array of function pointers declared externally in cmdArgs.h and defined in cmdArgs.c. That array is closely used with the commands array also defined externally in cmdArgs.h and defined in cmdArgs.c. These two arrays are used together in the event loop to call the approproate function. Section 2 will cover this more in depth. Below is the file structure.



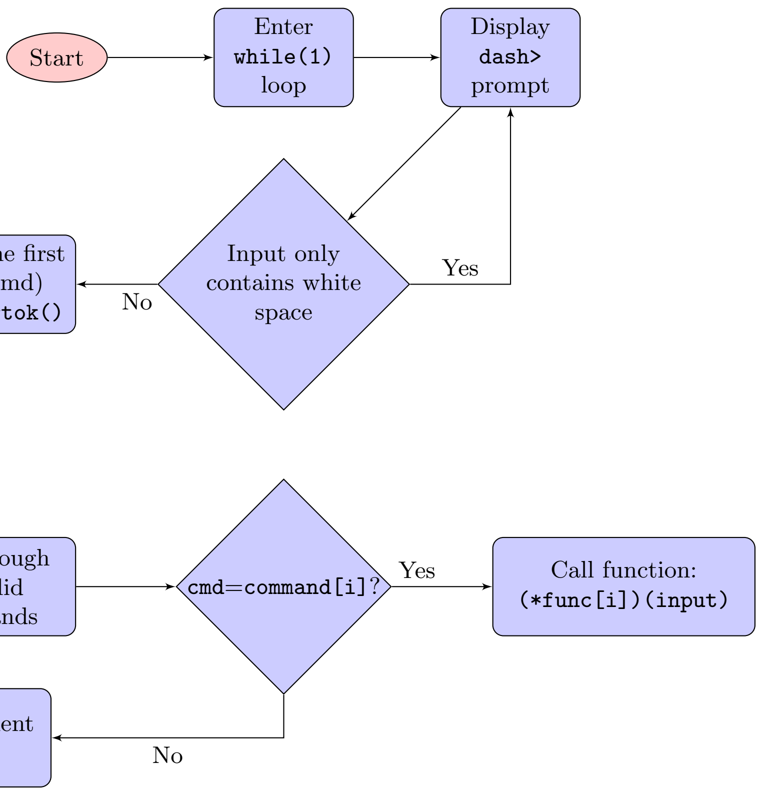

Start
Enter while(1) loop
Display dash> prompt
Input only contains white space
Yes
No
extract the first word (cmd) using strtok()
Loop though all valid commands
cmd=command[i]?
Yes
Call function: (*func[i])(input)
No
increment i

Figure 1: Event Loop

2 Algorithms

As briefly mentioned in Section 1.5, there is an array of function pointers used in the event loop to call the appropriate function. For modularity sake, the `#define NUM_CMDS` is used to keep track of the number of commands the user can enter. To add another command, one would simply increase `NUM_CMDS` and add the function to both the `commands` array and the `func` array. Obviously the function would need to be declared in cmdArgs.h and defined in cmdArgs.c as well.

2.1 Event Loop

As stated before in both Sections 1.5 & 2, the event loop makes use of function pointers to call the appropriate method. The `main()` function in dash.c contains and handles the event loop. Figure 1 demonstrates the structure.

2.2 Functions

The purpose of this section is to describe the algorithms for each of the commands that dash supports. A list of possible commands can be found in Table 1 on page 3.

2.2.1 cmdnm

The function `cmdnm()` first checks if an argument was passed. White space does not count as an argument and therefore calls another function `display_cmdNames()` which displays all command strings. If an argument is passed, then `cmdnm()` will attempt to open the file `/proc/<pid>/comm`. On success, the contents of the file will be displayed to the user (The `/comm` file contains the command string). Figure 2 on page 8 is a flowchart of the algorithm used.

2.2.2 pid

The function `pid()` first checks if an argument was passed, similar to `cmdnm()`. White space does not count as an argument and therefore calls another function `display_pid()` which displays all the process ids. If an argument is passed, then `pid()` enters a loop that goes though every process directory in `/proc`. In this manner, every process's `comm` file will be opened and checked for substring matches with the argument. All processes that match a substring will be displayed for the user. Figure 3 on page 9 is a flowchart of the algorithm used.

2.2.3 systat

The `systat()` function is much simpler than the other `cmdnm()` and `pid()` functions. The Table 3 on page 7 shows the contents of certain files that `systat()` uses. The function opens each file and extracts the information using many of the string functions that can be found in Table 2 on page 5.

2.2.4 exit

The `exit()` function is literally the sole reason why `<stdlib.h>` was included, for the use of it's exit() command.

Files	Information
`/proc/version`	Linux Version information
`/proc/uptime`	System uptime
`/proc/meminfo`	Total Memory, Free Memory
`/proc/cpuinfo`	Vendor id though cache size

Table 3: Useful process files

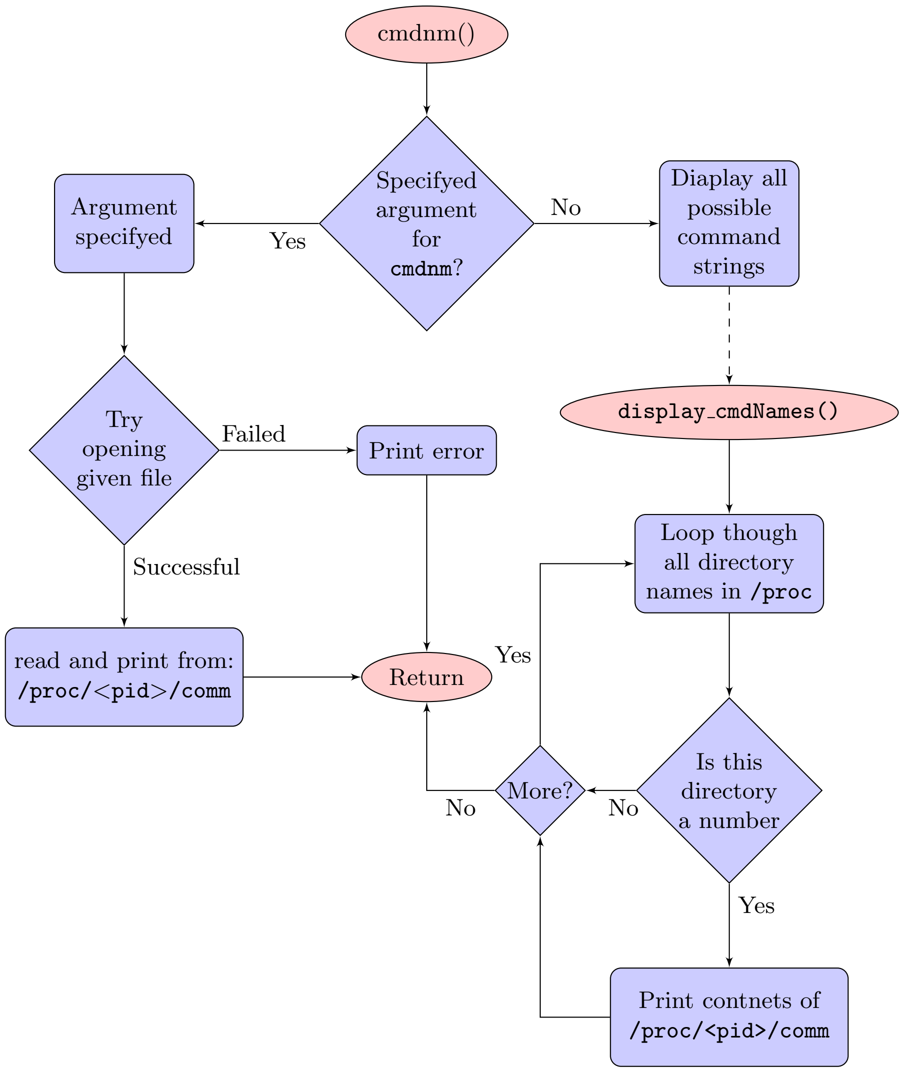

cmdnm()
Specifyed argument for cmdnm?
Yes
No
Argument specifyed
Diaplay all possible command strings
Try opening given file
Failed
Print error
display_cmdNames()
Successful
Loop though all directory names in /proc
read and print from: /proc/<pid>/comm
Return
Yes
More?
No
No
Is this directory a number
Yes
Print contnets of /proc/<pid>/comm

Figure 2: The `cmdnm` function

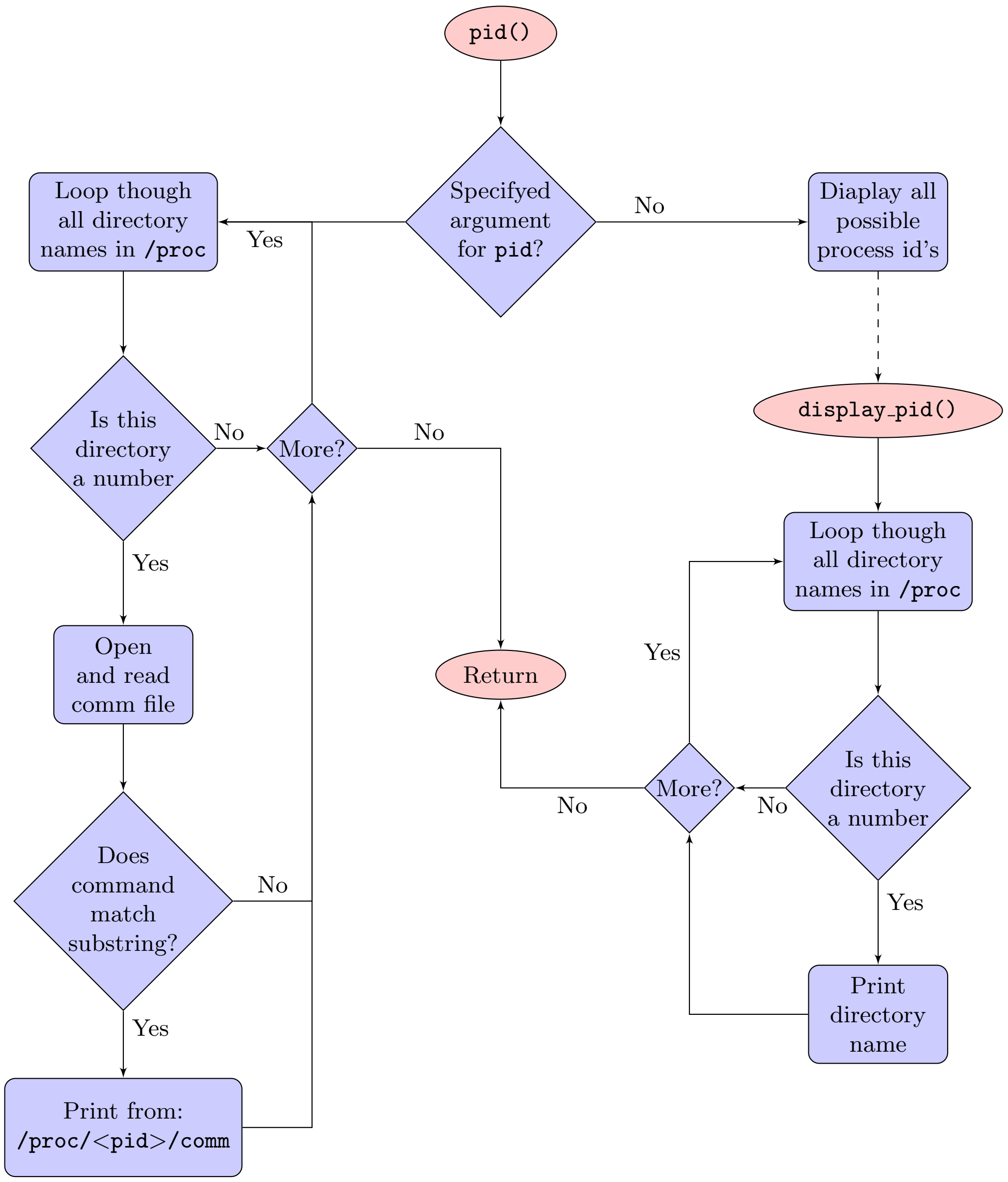

pid()
Specifyed argument for pid?
No
Diaplay all possible process id's
Yes
Loop though all directory names in /proc
Is this directory a number
No
More?
No
Yes
Open and read comm file
Does command match substring?
No
Yes
Print from: /proc/<pid>/comm
Return
display_pid()
Loop though all directory names in /proc
Is this directory a number
No
More?
Yes
No
Yes
Print directory name

Figure 3: The `pid` function

3 Testing and Verification

All functions were tested and have passed inspections. The purpose of this section is demonstrate the testing and verification process. Multiple attempts at trying to break the program will be covered and any unwanted behavior will be reported in this section.

3.1 Event Loop

The Event loop was thoroughly tested. No amount of white spaces or random characters seems to crash the program. Error messages are robust and include detailed information on what failed and how to use dash correctly. The event loop was tested and passed all inspections.

3.2 cmdnm

As with the event loop, no amount of white spaces or random characters seems to crash the program. The main concern with the `cmdnm()` function was weather a non-process id directory would crash the program. However, other directories such as `bus` and `driver` when passed to `cmdnm()` does not crash the program. Any additional arguments to this function will be ignored.

3.3 pid

As with the other functions, no amount of white spaces or random characters seems to crash the program. There were not many concerns about this function and is proving to be accurate and robust. Any additional arguments to this function will be ignored.

3.4 systat

There was not many test cases to run on the `systat()` function. The output of this function matches the actual system process information gathered from `ps`. Like the other functions, any additional arguments supplied to `systat()` will be ignored.

3.5 exit

The `exit()` function does exit the program and therefore works as expected. This function passes inspection.

4 Description of Submission

The submission of this program will consist of the following items within `prog1.tgz` file.

1. prog1.pdf
2. dash.c
3. cmdArgs.c
4. cmdArgs.h
5. Makefile

5 APPENDICES

A Program Assignment

CSC 456 Homework #1 – Due 2/6/2015

Introduction to the POSIX programming environment: Diagnostic Shell

It is always good to have program for diagnostics. Your first programming assignment is to write a process identification program. This program will introduce UNIX environment, system calls, signals and the proc filesystem. Your program will be a command line program with executable name: `dash`. It should present a prompt with a prompt string: `dash>` and accept commands to be executed at the prompt.

Background:

Each process has a unique process ID, pid, associated with the process. The kernel exports a significant amount of process information in the `/proc` directory. There you can see all the running processes. Connecting into the process directory (via the process number) you can gain information about the process. You can see some examples by typing `ps aux` at the command prompt of the shell. The `ps` command is the process status command and options to this command will provide plenty of data.

Required elements:

- Typing `dash` will invoke your program.
- (35 pts) Implement the commands:
  - `cmdnm`
    - (10/35 pts) Return the command string (name) that started the process for a given process id
    - Usage: `cmdnm <pid>`
  - `pid`
    - (10/35 pts) Return the process ids for a given command string (match all substrings)
    - Usage: `pid <command>`
  - `systat`
    - (10/35 pts) Print out some process information
    - print (to stdout) Linux version information, and system uptime.
    - print memory usage information: memtotal and memfree at least.
    - print cpu information: vendor id through cache size.
    - using /proc/* files
  - `exit`
    - (5/35 pts) Exit the program
    - program termination
- (15 pts) Handle errors gracefully.
- Incorrect file or program names will not be accepted.

Solution method:

- Start the documentation and figure out how to generate the PDF: 1/23/15
- You will do the work in a subdirectory named prog1
  - `tar -czf prog1.tgz  prog1`
  - Note: **no prog1.tar.gz, no Prog1.tgz, no program1.tgz, no prog1.zip**,
- Write the parsing code: 1/26/15
- You will need to produce a Makefile to build the executable (sample below): 1/26/15
- Modify this parsing code to read from stdin (command line) and display a prompt. 1/28/15
- Create an event loop. 1/28/15
- Handle an `<enter>` without a string. 1/30/15
- Modify the event loop code to recognize the exit command. 1/30/15
- Add the `cmdnm` command. 1/31/15
- Compare to output from `ps aux`. 1/31/15
- Add the `systat` command 2/02/15
- Add the `pid` command. 2/04/15
- Add the error handling command. 2/05/15

Testing:

- Try to break the code.
- Type at least the following and hit enter:
  - `<space>`
  - `<tab>`
  - random chars
  - no chars [blank line]
  - `cmdnm abc`   (should not die even if it is expecting an integer)

Code structure:

- Clean, modular, well documented code is required.
- Functions must be used to handle the user commands (a single large block is not acceptable).

Submission contents:

- Documentation - all of the following should be in: `prog1.pdf`
  - Description of the program.
  - Description of the algorithms and libraries used.
  - Description of functions and program structure.
  - How to compile and use the program.
  - Description of the testing and verification process.
  - Description of what you have submitted: `Makefile`, external functions, main, etc.
  - Format: PDF (write in any word processor and export as PDF if the option is available, or convert to PDF)
- Main program `dash.c`.
- Any required external functions `*.c`.
- Any include files you use (other than the standard ones).

- The `Makefile` to build the program `dash`.
- Tar the directory and then gzip using the correct filename.
  [And I do know that there are better compression routines than gzip.]

Notes and hints:

- You are not required to write this in C. C++ has some great string handling abilities and so would be useful in this aspect. However, this code is only a few pages of C and is not difficult in plain C.
- The point of this is to learn about system calls and some of the exported data. You will not use `fork/exec, popen(), system().`
- The kernel provides information about the processes through the `/proc` tree of the filesystem. The kernel exports the data as text files. You may read these files like any other text file. To learn about `/proc, cd /proc` and do a file list (`ls`). You can cd to a directory with a numerical name. This is all the kernel exported stats on the process with that ID number.
- If you include any of the sample code, you must completely document it. The sample code is under the Gnu Public License. Thus your code will be as well. You can find out more information about Gnu codes and the GPL at http://www.gnu.org/.
- You will need to work with command lines and strings, the following examples can give you some hints. Example of reading the arguments. If you would like more, you can find it here: Example of parsing them. You don't have to use these, they are just here to give you ideas about parsing.
- Check your code with incorrect strings or just with an empty line (plain enter). It should not seg fault.
- The cmdline in proc: `/proc/<pid>/cmdline` is a file of null terminated strings which is the parsed command line. You will need to replace the nulls with spaces to get your string compare to work correctly.
- The kernel provides information about the kernel state and kernel variables through the `/proc` tree of the filesystem. The kernel exports the data as text files. You may read these files like any other text file. Files of primary interest are `cpuinfo`, `meminfo`, `uptime`, and `version`.
- The kernel maintains data (entry into the kernel) in the /proc tree. Your program can read `/proc/cpuinfo` to get cpu speed, etc. Example of reading `/proc/cpuinfo` .

Grading approach :

- copy from the submit directory to my grading directory
- `tar -xzf prog1.tgz`
- `cd prog1`
- `make`
- `dash`
- enter commands and compare output
- assign a grade
- `cd ..`
- `rm -rf prog1`

Submission method:

You will use the submit page to submit your program. The files need to be tarred and gzipped prior to submission. Please empty the directory prior to tar/zip. [You will tar up the directory containing the files: `tar -czf prog1.tgz prog1`]

Sample Makefile to build "crash" (your version will be slightly different):

```
#---------------------------------------------------------------------
---

# Use the GNU C/C++ compiler:
CC = gcc
CPP = g++

# COMPILER OPTIONS:

CFLAGS = -c

#OBJECT FILES
OBJS = crash.o foo.o


crash: crash.o foo.o
	${CC} -lm ${OBJS} -o crash
crash.o: crash.c
foo.o: foo.c


#---------------------------------------------------------------------
---
```

Note: indents are tabs - NOT SPACES. Tab is a command (yes, horrible design to have whitespace commands).