DASH

Documentation

Zachary Pierson

February 28, 2015

Contents

1 Introduction

The c program dash is a result of the Computer Science Course 456, Operating Systems. The purpose of assignment 2 is to introduce fork and exec commands as well as pipes and redirects. The basic approach for this program is to fork of a process and have the child execute the command via the exec. Signals are also implemented. The kill system call allows the program to send signals to other processes. dash is also able to catch signals that are sent to it.

1.1 Description of Program

Not only is dash a process identification program, it is also an imitation of the bash shell. 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. The second version of this project includes forks, pipes, redirects, and signals to call system calls similar to what the bash shell does.

Once invoked, dash will enter an event loop awaiting user input. Acceptable command options can be found in Table 1 on page 16.

1.2 Compiling Instructions

The file can be found on github at: https://github.com/zjpierson/operatingSystems/prog2 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 16 for information on processes.

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

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.

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.

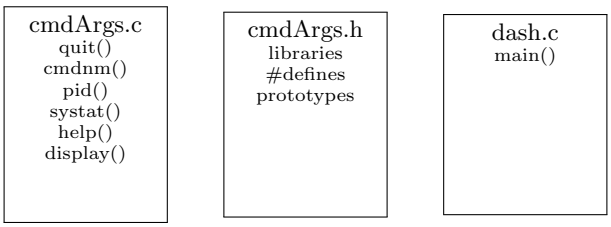
The <`sys/time.h`> and <`sys/resource.h`> libraries are used to mainly to provide process information using the `getrusage` function.

The <`unistd.h`> library allows the program to `fork` off different processes and make system calls using `execvp`. It is also used to change the current working directory by using the `chdir` function.

The <`fcntl.h`> library is used solely for the use of the `O_RDONLY` for opening a read only file for redirected output.

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 declared externally in cmdArgs.h and defined in cmdArgs.c. These two arrays are used together in the `call` function to be able to invoke the correct function. Section 2 will cover this more in depth. Below is the file structure.



2 Algorithms

As briefly mentioned in Section 1.5, there is an array of function pointers used in `call` to invoke the appropriate function. For modularity sake, the

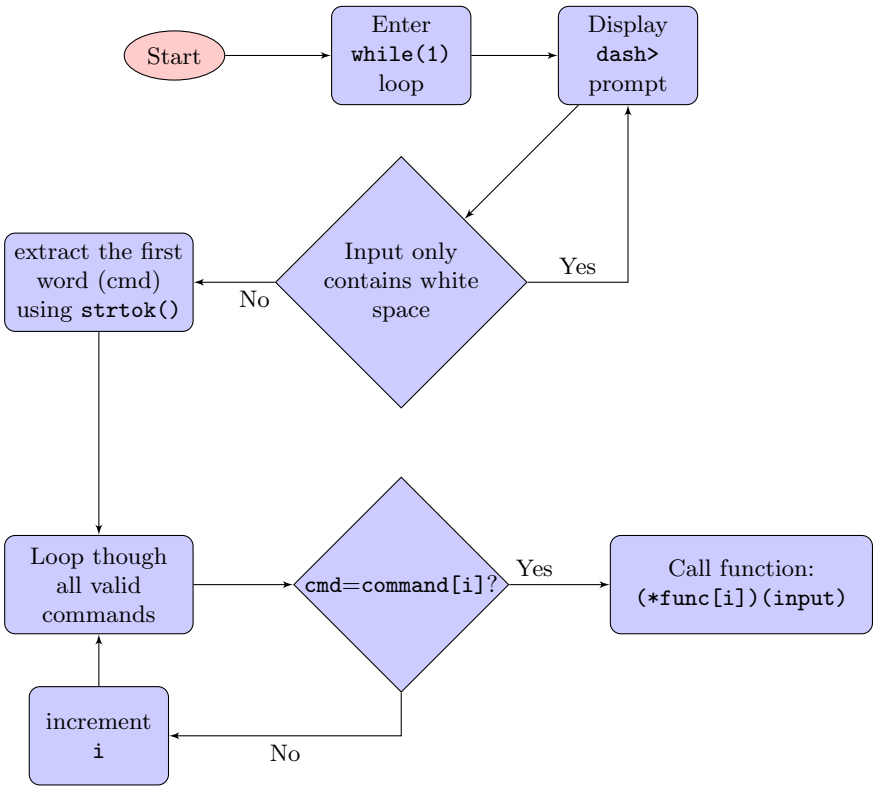


Figure 1: `call` function

`#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 `call` function 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 16.

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 16 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 16.

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.

2.2.5 cd

The `cd()` function is much simpler than many of the other functions. This one simply makes a call to `chdir()` which changes the current working directory.

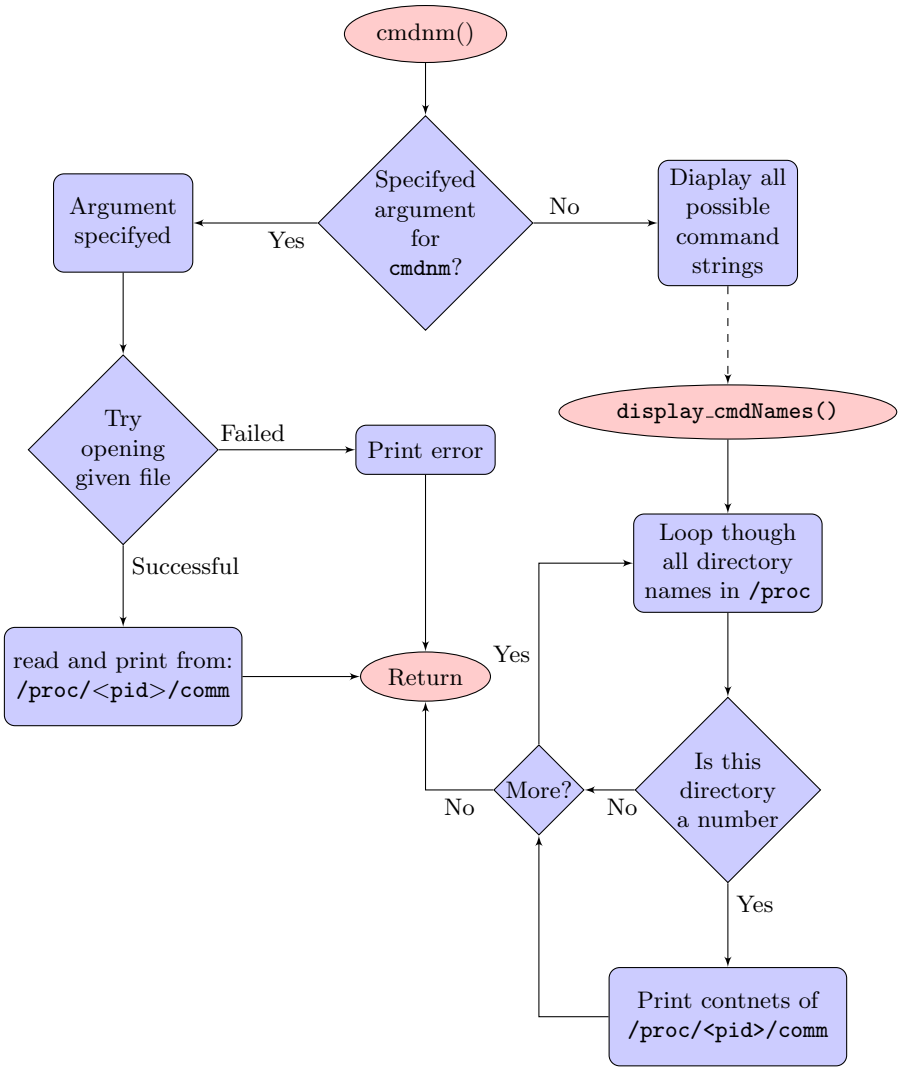


Figure 2: The `cmdnm` function

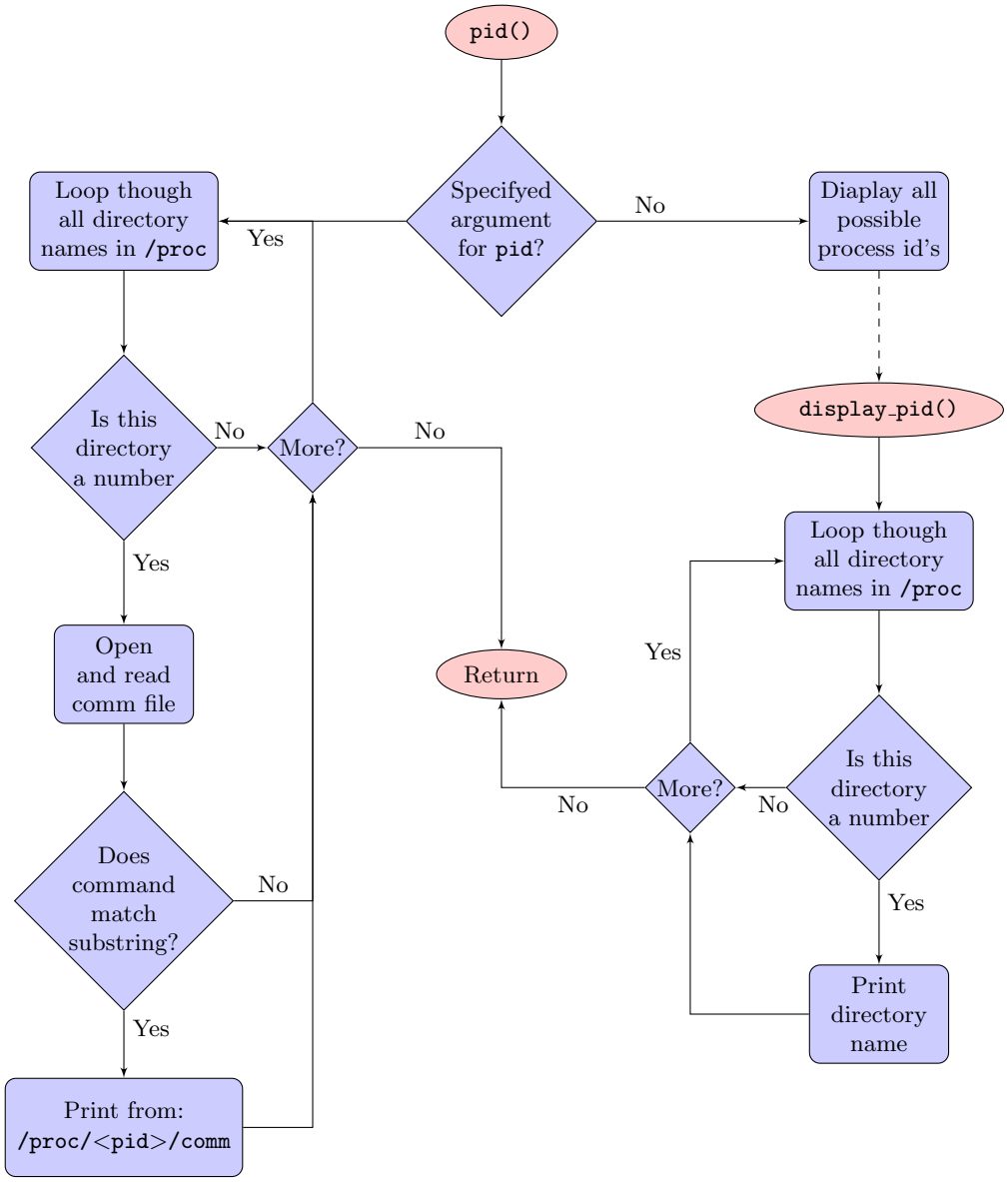


Figure 3: The `pid` function

2.2.6 signal

The `signal` command is used to send signals to other processes. This function is basically a wrapper for the `kill` system call. A new variable called `cmd` is used as the actual argument to `execvp(). cmd` is initialized with "kill", "-s" and appends the rest of the arguments from `args` to `cmd`.

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 cd

Initially, when the `cd` function was implemented, it seemed as if it had not changed the current working directory. However, on latter attempts It was found that once the child process had terminated, the `chdir` no longer contained the correct directory. The solution to this was to call the `cd` command directly in the Event Loop.

3.6 signal

Not only can dash send specific signals to processes, but it can also catch any signal except for `SIGKILL` and `SIGSTOP`. Many tests were done and it is concluded that the signal command is properly functioning.

3.7 pipes and redirected input/output

This is an important section to include here as it is not fully satisfying. First of all, the symbols "`<>|`" need to be surrounded by white space for dash to recognize their meaning. Secondly, the pipes take 2 commands that get piped together. This is a problem when the user wishes to pipe more than two commands together. A note in the program documentation explains that these are to be implemented in the future. Other than that, these features work great and hold up many different test cases.

3.8 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. prog2.pdf
2. dash.c
3. cmdArgs.c
4. cmdArgs.h
5. Makefile

5 APPENDICES

A Program Assignment

CSC 456 Homework #2 – Due 3/1/2015

The second programming assignment is extending your shell, `dash`. This program will introduce the `fork` and `exec` commands as well as `pipes` and `redirects`. You will keep the executable name: `dash`. It should present a prompt with a prompt string: `dash>` and accept commands to be executed at the prompt. The approach is to fork off a child process and run the command in the child process via an `exec`. To send a signal, you will use the kill system call. To get information on this type: `man -s2 kill`. The `-s2` option says look at the second section (programmer section) of the man pages. It gives the includes as well. To see a list of the signals, try man `-s`

Required elements:

- Typing ./`dash` will invoke your shell and display the prompt:
  - dash>
- (30 pts) Your shell should take any command (plus arguments), execute it and return any stdout, for example (these will not be the test commands however):
  - `ls -al`
  - `gcc -o foo foo.c`
  - `mv foo foop`
  - Example code
  - On initiation of the command, the (child) process pid is printed after the prompt.
  - On competition of the command, the shell should print out process stats (using the `getrusage` call):
    - Time for user & system
    - Page faults and swaps
- (10 pts) Implement the shell intrinsic functions (new):
  - `cd`
    - The standard cd command:
    - `cd <absolute path>`
    - `cd <relative path>`
- (15 pts) Implement redirection:
  - `<Unix command> > file`
  - `<Unix command> < file`
  - Example code
- (20 pts) Implement shell pipes:
  - `cmd1 | cmd2`
  - Example code
- Signal (15 pts) This function will send a signal to a process.
  - `signal <signal_num> <pid>`
- Catch and display all signals it receives.
- Implement the shell intrinsic functions (already coded):
  - `cmdnm`
  - `signal`
  - `systat`
  - `exit`
- Handle errors gracefully.
- This means that if a blank line (just a newline) is entered, the prompt is redisplayed.

Submission contents:

- Documentation (10 pts)
  - 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 (as before) `dash.c`.
- Any required external functions `*.c`.
- Any include files you use (other than the standard ones).
- The Makefile to build the program dash.
- You will need to produce a Makefile to build the executable:

Solution method:

- Read the man pages for `fork, exec, wait, getrusage, chdir.` 2/6/15
- You will do the work in a subdirectory named prog2
  - `tar -czf prog2.tgz prog2`
  - Note: no `prog2.tar.gz`
- Start with the example code that uses `fork` and `exec` to run a command. 2/6/14
- Merge with program1 code. 2/9/14
- Add the `cd` and `signals` commands. 2/9/14
- Add the process stats feature. 2/16/14
- Code pipes and redirects. 2/20/14
- Error test your code 2/23/14
- Clean up errors and write pdf. 2/28/14

Testing:

- Try to break the code.
- Type the following and hit enter:
  - `<space>`
  - `<tab>`
  - random chars
  - no chars [don't hand fork a null string]
  - `<space> <space> ..... <space>` some-cmd [don't parse based on position]

Notes and hints:

- Process statistics may be gained using the `getrusage()` system call.
- The `getrusage()` function returns a struct that has current process user time. Example of `getrusage()` call.
- Note that getrusage can return data about the current process or that of all child processes (that have terminated and been waited for): `RUSAGE_SELF` or `RUSAGE_CHILDREN`. If you have multiple child processes, then you may get cumulative data.
- 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/
- No specification was made about environment variables. Use the existing ones.
- Exit should exit out of dsh not the parent process.

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 crunch.o


      crash: ${OBJS}
            ${CC} -lm ${OBJS} -o crash

      crunch.o: crunch.c
            ${CC} -c -lm crunch.c

      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).

Grading approach:

- copy from the submit directory to my grading directory
- `tar -xzf prog2.tgz`
- `cd prog2`
- `make`
- `dash`
- enter commands and compare output
- assign a grade
- `cd ..`
- `rm -rf prog2`

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 prog2.tgz prog2` .]

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
proc_status	prints process status
cd	changes the current working directory
redirect_pipe	finds redirects or pipes in the input
redirected_output	redirects standard output to given file
redirected_input	redirects standard input to a file
do_pipe	pipes output of one command to the input of another
signal	sends given signal to given process

Table 1: dash commands

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

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