Home exam in INF1060:

mysh - a command prompt for linux

Delivery: Tuesday, October 30, 2018
Submission: Tuesday, November 13, 2018, at. 23.59

Intro

To send commands to the operating system (OS) and for OS to provide feedback, we often use a *shell* - also called a *command prompt* . Unix systems do not have any built-in window-based interface, but assume a single character-based interface where a user types characters (text) and ends with E NTER or R ETURN .

This task is to program **mysh** (my shell) - such a shell in the programming language C for Linux on la sh, csh or bash, but in a greatly simplified edition. The program should be able to run on IFI's standard Linux machines like those in the Linux cluster - ssh linux . It should be able to execute common commands with parameters and that it should have some built-in commands similar to the common command interpreters.

Remember that this assignment is grade counting for the course (approximately 35%). It is therefore not allowed to retrieve code from another team or share your own code with others - all code must be written by yourself. However, it is permitted (and we strongly encourage) to discuss problems as well as to exchange information and knowledge with other students, but do not share code directly.See also the last section on how to submit the assignment.

The assignment must be delivered in Devilry within the deadline.

The task

This is an extensive task and you may want to take it step by step. The rest of the assignment text is therefore written as follows. Furthermore, part of the task is to print debug information using *fprintf*(see fprintf) to 'stderr' (since a printout sent to

'stderr' is unbuckled, you should see print on the screen even if the program should crash. which goes to 'stdout' can, however, "disappear" in a crash because of buffering. See you stderr).

For example, debug prints can be done as follows:

```
#ifdef DEBUG
fprintf (stderr, ...);
#endif
```

This printf command will only be executed if DEBUG is defined, for example, if the program is compiled with the -DDEBUG option or to have a #define DEBUG in the C file. In addition to the "mandatory" debug prints, specified below and only printed in "debug mode", you may want to load many test prints along the way to get an overview of what's happening. Check especially if all text operations work.

Compilation of the code

To compile the code, create a Makefile so you can write a spouse in the directory where the files are stored to compile the program. It may also be wise (and it's a plus) to add some of the other features (such as the built-in features) into their own files. The make file must also be handed in and give points.

Read command from the keyboard

The command interpreter will go forever and get commands from the user. That is, for each round it should be able to read a command from the keyboard. mysh shall:

•Print a prompt (prompt). This must consist of username @ mysh where *username* is the username of the logged in (H INT: Use the environment *variable* USER where environment *variables* can be read in general, for example *getenv* .) And a number (a counter) indicating the number of commands since the shell was started. :

paalh @ mysh 1>

•Read a line from the default input (keyboard). We can assume that the line is never longer than 120 characters. H INT: Use *fgets* to read the line.

•If there are no more lines to read (the user will quit), mysh must quit with status value 0. NB! This is not the same as reading an empty line! When reading from the keyboard, the user can press Ctrl + D to indicate that there are no more lines to read. H INT: Use the function value that *fries* returns to check if there were more lines to read.

•DEBUG: As debug information, the program will print the read line.

Divide the input line into word command with parameters

In order for the command interpreter to execute a command, the line we read over must be divided into words separated by blank characters. That is, after reading a line in the task above we should be able to:

•Find out what words it contains; We assume there are never more than 20 words on one line and a word is defined as a sequence of non-blank characters. H INT: Use the ' *isspace* ' function to determine if a character is blank.

•One point to a copy of each word is to be stored in the array / vector *char * param [21];* . Parameter element [0] should point to a copy of the first word, param [1] on the second word, etc. The parameter param [n] (where n is the number of words on the line) must contain the value 0 (NULL) to mark the end of the vector. (The attentive may see that *param* is similar to the *argv*parameter to the *main*)

•DEBUG: As debug information, the program should be able to print the contents of the array / vector *param* .

Execute commands

From the above steps we should now have a program that reads input from the user, divides this string into words and puts these into the vector *param* . The next step is to have the command interpreter interpret these words as commands and associated parameters, then perform these commands by creating a child process. We assume that the first word on the line (*param [0]*) contains the command name and the following words are parameters:

•If the command is *quit* , the command prompt will end.

•If the line does not contain any words (that is, it is blank), do not do more with the line.

•Start a childhood process with *fork* . This childcare process will try to execute the specified command with a call on *execve* . It will look among the file areas specified in the environment variable `PATH 'after where the command can be found as executable file. H INT: Ambient *variables* can be read with, for example, *getenv* .

•If no program 'xyz' exists in the areas specified in PATH, an error message will be written before the childcare process ends:

mysh: xyz: command not found

•Check if the line ends with a '&' (optionally followed by blank). If it does, the parent process will only print the childcare PID and continue the loop by asking for a new command. If no '&' finishes the line, the parenting process will wait until the childcare process is complete. Note that this is similar to the behavior, such as in bash, and you can test it in bash to understand what their own program should do.

Built-in Commands

As another shell, mysh has some built-in commands (in addition to *quit*). For example, if you try to find which version of *CD* you are using by performing the CD type command, you are told that this is a built-in command in the shell:

```
[vizzini] 1> CD type
cd is a shell builtin
```

Therefore, you must enter the embedded commands specified in the shell - ie h (history) (cd will not be implemented). **These should not be expanded or executed with execve ()** .

h - execution history

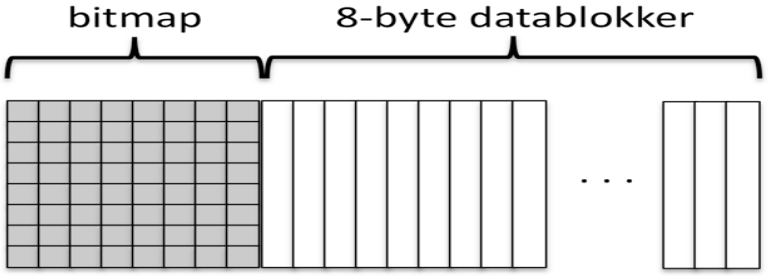
A common built-in command is *h* , which is an alias for *history* . This shows an overview of the last commands (including parameters) shellet has performed. You must enter *h* as a built-in command in **mysh** where the shell will be able to keep the **n** last commands. The number of **n** should vary depending on how much memory is used per command (see below). If the user then performs *h* , a list of the last commands will be printed where the oldest command is located at the top and the newest bottom (which will be h) - a la:

```
paalh @ mysh 10> h

History list of the last 10 commands:
  10: cd
   9: 1s
   8: cd oblig
   7: husband
   6: man gcc
   5: You get it
   4: h
   3: more mysh.c
   2: mysh
   1: h
```

Memory management for history

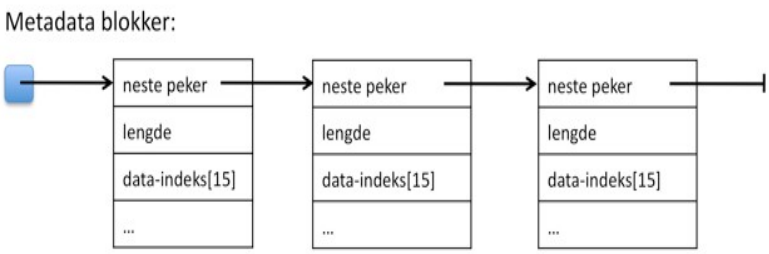
To save the commands for *h* , you must implement a system held in the primary memory. This should be a continuous memory that can accommodate *64* **8 byte** blocks as well as a bitmap to indicate which blocks are in use and not - that is, the bit values should be 0 if the blocks are available and 1 whose blocks is busy. An example of how to allocate and split the memory is shown in the figure below.



Since a command is assumed to be of arbitrary length, but as mentioned above does not exceed 120 characters, a single command may need more such **8-byte** blocks (on pages in memory or disk blocks on the disk). These blocks do not necessarily have to be left behind. When a new command is written (performed by the shell), the system will look for available blocks in the bitmap, allocate the ones needed (set the bits) and use the corresponding blocks to save the command in these memory blocks. If there are not enough available blocks to save the command in history, the system will expel the oldest command from the list, release its allocated data blocks, and try again to enter the new one (possibly repeat to get sufficient space). When a command is deleted from the list, its data blocks must also be deleted, ie remove the contents of them (H INT: Use, for example, *bzero*).

metadata Structure

In addition, you'll need a type of metadata block for each command stored in *history* . The metadata should be inserted into a structure that must at least contain the IDs of the data blocks used to save the command. Here you may want to use "direct pointers" (here are the indexes for the bitmap and data block memory). You need 15 such as to save a command of up to 120 characters. In addition, you need a "length" field to save information about how long the command is. Furthermore, the metadata blocks must be organized using a linked list - the newest command should be in the list first, and the oldest (and any one that should be removed to free space) must last. That is, the data structure also needs a nest pointer to the next metadata block. (If you need other elements in the structure, this should be described and argued.) An example of such a list is shown below.



DEBUG 1: As debug information, the program will print the contents of the bitmap where you print 0's and 1's. Each line must be 32 characters.

```
DEBUG - BITMAP:

10101010101010111100011001100111
11000110011001111010101010101011
```

DEBUG 2: As debug information, the program will print the contents of the data blocks. Each data block must be separated by two "#" (ie, "##"). Each line should print *4* 8-byte blocks. (Note that the blocks marked in the bitmap in the example above NOT match the blocks used in the example below).

```
DEBUG - DATABLOCKS:

## more mys ## hc #### cd oblig ##
## gcc -DDE ## BUG -om ## y_perfec ## t_mysh ##
##########
## mysh.c ########
##########
## h -d 5 #### ls -al * ####
## .pdf ###### you get ##
## nv ########

.  .  .  .

## h ## h 5 ## make ####
```

In summary, this means that when a command is entered, the shell will create a metadata block and add it to that list. The length of the command must be calculated and available data blocks must be allocated, put the bits in the bitmap and insert the indexes of the data blocks into the "direct pointers" in the structure. Finally, do not forget and save the command in the data blocks. Similarly, when a command has to be deleted from *history* , both data blocks and metadata blocks must be released.

hi - do it in the youngest history command

You should expand the functionality of h so that if a number *is* sent as an argument, no list will be printed, but the most recent command will be executed. For example, h 3 will in the example above cause the command more mysh.c to be executed.

h -di - delete entry in command history

You must expand the functionality of h with the option *-d* so if the option -d is entered with a number, it will be deleted from the story *in the* youngest command. For example, h -d 3 will in the example above cause the more mysh .c command to be deleted. Make sure the rest of the list is kept intact even if a command is deleted.

jobs - list all jobs that the shells are running

To get an overview of all the processes that Shellet has started (and still running), here you can list all processes run in the Shellet with PID Program Name / Command Line, ie to know about all processes running in the background . The number of processes running in the background should be arbitrary. There are many ways to do this (for example), and you must find an appropriate way to handle the set of running processes (provide an explanation in the code).

The printout should look like this:

```
mysh 1> emacs &
mysh 2> xterm &
mysh 3> jobs

Pid = 7880
Command line = emacs

Pid = 7891
Command line = xterm
```

kill id - kill the process that has a similar *ID*

Shellet should be able to kill an identified process by running kill . Compared to other versions in other shell, our simplification is the only one that can be sent, the signal is SIGKILL to complete the process. H INT: see man 2 kill

```
Good luck!
```