COMP 304 - Operating Systems: Project 1

Due: December 22nd, 2024 - 23:59

Hakan Ayral Fall 2024

Notes: The project can be done individually or as a team of 2. You may discuss the problems with other teams and post questions to the discussion forum, but the submitted work must be your own.

Any sources, services or material you use from external sources such as any form of AI and the internet resources should be properly cited in your report.

Contact TAs: Doğan Sağbili, Semih Erken

Description

This project is a variation on the programming project **Project 1 - UNIX Shell** at the end of Chapter 3 of our textbook (Operating System Concepts).

The main part of the project requires you to develop an interactive Unix-style operating system shell, called `da`**`sh`** in C. After executing `da`**`sh`** , it will read commands from the user and execute them. Some of these commands will be *builtin* commands, i.e., specific to `da`**`sh`** and implemented in the same executable binary, while it should be able to launch other commands which are available as part of your own linux system such as `ls` and `echo`. The project has four main parts (95 points) in addition to a report (5 points). We suggest starting with the first part and building the rest on top of it.

Part I (15 pts.)

- Use the skeleton program provided as a starting point for your implementation. The skeleton program reads a line of commands from `stdin`, parses it, and separates it into arguments using whitespace as the delimiter. You will implement the action that needs to be taken based on the command and its arguments entered in `da`**`sh`** . Feel free to modify the command line prompt and parser as you wish.
- Use the provided `Makefile` to compile your code. Type `make help` to get a list of build targets.
- Command line inputs, except those matching builtin commands, should be interpreted as program invocation. The shell must **fork** and **execute** the requested programs. Refer to *Part I - Creating a child process* from the book.

- The shell must support **background execution** of programs. An ampersand (&) at the end of the command line indicates that the shell should return the command line prompt immediately after launching the program.
- Do not use the `exec()` family of calls that are prefixed with `p` such as `execp()` that automatically search for executable files. Instead, use the `execv()` library call and implement **path resolution** yourself.
- Your shell must provide a shell prompt like "dash>" and implement `cd` and `exit` as builtin commands.

Part II (10 + 15 pts.)

1. I/O redirection (10 pts.)

In this part of the project, you will implement I/O redirection for da**sh** . For I/O redirection if the redirection character is >, the output file is created if it does not already exist and overwritten if it does exist. For the redirection symbol >> the output file is created if it does not exist and appended to if it does exist. The < character means that input is read from a file. See "**IV. Redirecting Input and Output**" on page P-14 in the textbook.

A sample terminal line is given for I/O redirection below:

```
dash> program arg1 arg2 > outputfile >> appendfile < inputfile
```

2. Piping (15 pts.)

In this part, you will handle program piping for da**sh** . Piping enables passing the output of one command as the input of second command. To handle piping, you would need to execute multiple children, and create a pipe that connects the output of the first process to the input of the second process, etc. It is better to start by supporting piping between two processes before handling longer chain pipes. See "**V. Communication via a Pipe**" on page P-15 in the textbook.

Below is a simple example for piping:

```
dash> ls -la | grep search-this-text | wc
```

Part III (15 + 20 pts.)

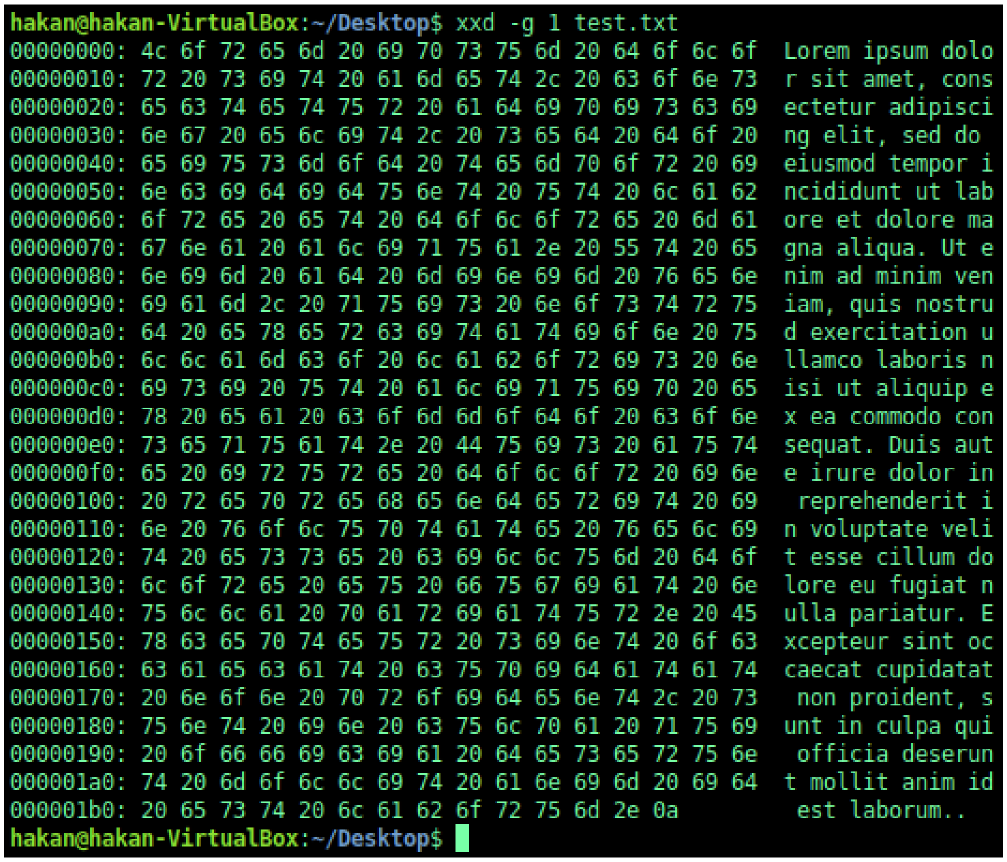
In this part of the project, you will implement new da**sh** commands (builtin commands).

1. Auto-complete (15 pts.)

You are required to handle auto-completion of existing and newly introduced commands in your shell. While typing a command if the Tab key is pressed, da**sh** should automatically complete the command. If there are more than one match found, it should list all the possible matches. If the command is fully typed, then the Tab key is pressed, it should list the files in the current directory. Your implementation should support newly introduced commands in this project.

2. kuhex (20 pts.)

Write a built-in command named **kuhex** to show a hex dump of a given file - that is showing the file contents as hexadecimal numbers. This program is expected to behave similarly to the xxd utility.



```
hakan@hakan-VirtualBox:~/Desktop$ xxd -g 1 test.txt
00000000: 4c 6f 72 65 6d 20 69 70 73 75 6d 20 64 6f 6c 6f  Lorem ipsum dolo
00000010: 72 20 73 69 74 20 61 6d 65 74 2c 20 63 6f 6e 73  r sit amet, cons
00000020: 65 63 74 65 74 75 72 20 61 64 69 70 69 73 63 69  ectetur adipisci
00000030: 6e 67 20 65 6c 69 74 2c 20 73 65 64 20 64 6f 20  ng elit, sed do 
00000040: 65 69 75 73 6d 6f 64 20 74 65 6d 70 6f 72 20 69  eiusmod tempor i
00000050: 6e 63 69 64 69 64 75 6e 74 20 75 74 20 6c 61 62  ncididunt ut lab
00000060: 6f 72 65 20 65 74 20 64 6f 6c 6f 72 65 20 6d 61  ore et dolore ma
00000070: 67 6e 61 20 61 6c 69 71 75 61 2e 20 55 74 20 65  gna aliqua. Ut e
00000080: 6e 69 6d 20 61 64 20 6d 69 6e 69 6d 20 76 65 6e  nim ad minim ven
00000090: 69 61 6d 2c 20 71 75 69 73 20 6e 6f 73 74 72 75  iam, quis nostru
000000a0: 64 20 65 78 65 72 63 69 74 61 74 69 6f 6e 20 75  d exercitation u
000000b0: 6c 6c 61 6d 63 6f 20 6c 61 62 6f 72 69 73 20 6e  llamco laboris n
000000c0: 69 73 69 20 75 74 20 61 6c 69 71 75 69 70 20 65  isi ut aliquip e
000000d0: 78 20 65 61 20 63 6f 6d 6d 6f 64 6f 20 63 6f 6e  x ea commodo con
000000e0: 73 65 71 75 61 74 2e 20 44 75 69 73 20 61 75 74  sequat. Duis aut
000000f0: 65 20 69 72 75 72 65 20 64 6f 6c 6f 72 20 69 6e  e irure dolor in
00000100: 20 72 65 70 72 65 68 65 6e 64 65 72 69 74 20 69   reprehenderit i
00000110: 6e 20 76 6f 6c 75 70 74 61 74 65 20 76 65 6c 69  n voluptate veli
00000120: 74 20 65 73 73 65 20 63 69 6c 6c 75 6d 20 64 6f  t esse cillum do
00000130: 6c 6f 72 65 20 65 75 20 66 75 67 69 61 74 20 6e  lore eu fugiat n
00000140: 75 6c 6c 61 20 70 61 72 69 61 74 75 72 2e 20 45  ulla pariatur. E
00000150: 78 63 65 70 74 65 75 72 20 73 69 6e 74 20 6f 63  xcepteur sint oc
00000160: 63 61 65 63 61 74 20 63 75 70 69 64 61 74 61 74  caecat cupidatat
00000170: 20 6e 6f 6e 20 70 72 6f 69 64 65 6e 74 2c 20 73   non proident, s
00000180: 75 6e 74 20 69 6e 20 63 75 6c 70 61 20 71 75 69  unt in culpa qui
00000190: 20 6f 66 66 69 63 69 61 20 64 65 73 65 72 75 6e   officia deserun
000001a0: 74 20 6d 6f 6c 6c 69 74 20 61 6e 69 6d 20 69 64  t mollit anim id
000001b0: 20 65 73 74 20 6c 61 62 6f 72 75 6d 2e 0a         est laborum..
hakan@hakan-VirtualBox:~/Desktop$ 
```

Figure 1: Sample xxd output

The xxd utility has several program options, but you are only required to implement -g, which determines the number of bytes grouped together in each column. (The example above has the grouping set to -g 1 = columns of one byte.) The number of bytes represented in each line should be 16, therefore ASCII representation on the right hand column will always have 16 characters per line. On the middle hexadecimal part, for -g 1 you will have

16 columns, and for -g 2 you will have 8. For brevity, you only need to support group sizes 1,2,4,8,16.

Part IV (20 pts.)

psvis <PID> <output file> (Must be written in C):

You are required to implement a **psvis** command which retrieves the process tree starting from a process with a given PID as the root and visualize this in a human-readable graph form. Each node in the graph corresponds to a process, and each edge represents a parent-child relationship between the two processes it connects. For visualization, you have to construct an image file showing this tree of processes; in each node, show the PID and the name of the executable binary file (without path) of the respective process.

Obtaining the process tree information requires kernel-level support, you are required to write a loadable kernel module. Please read the sections ”Programming Project - Introduction to Linux Kernel Modules” at the end of Chapter 2, ”Project 2 - Linux Kernel Module for Task Information” and ”Project 3 - Linux Kernel Module for Listing Tasks” at the end of Chapter 3 of the textbook.

When `da`**`sh`** is executed it should check whether the kernel module is already loaded (this can happen if another instance of `da`**`sh`** is already running) and load the kernel module with the **insmod** command using **sudo** only if it is not; if module is already loaded it should just notify the user that the module is already loaded. Read ”II. Loading and Removing Kernel Modules” on page P-3 in textbook.

`da`**`sh`** should remove the module from kernel if it is the last instance (i.e. no other `da`**`sh`** processes running) and is currently exiting.

Any code which is not part of the kernel module must be implemented as part of the `da`**`sh`** executable; but you are free to use **gnuplot**, **graphviz** or a similar library to generate the diagram and/or image file.

Some Hints:

- To make your user level `da`**`sh`** process and kernel module commmunicate with each other, use the **/proc** file system. Read from the textbook sections titled ”III. The /proc File System” on page P-5, ”II. Reading from the /proc File System” on page P-17 and ”I. Writing to the /proc File System” on page P-16.
- In the kernel you can not load any libraries (.so shared objects), therefore do not handle the visualization on the kernel module; only pass the relevant information to the `da`**`sh`** process from the kernel and deal with the visualization at the user level.

- The struct named **task** defined in `linux/sched.h` is the key data structure in the kernel which represents the processes and threads for scheduling purposes. You need to use it in order to obtain necessary information such as process name and process start time.
- Test your kernel module outside of da**sh** first to check if it works.
- You can compare your output to the output of the **pstree** to check for correctness. Use the **-p** flag to list the processes with their PIDs.
- If you get trouble with **insmod** loading your module and you are developing on a real linux system (i.e. as opposed to a virtual machine) make sure that the **secure boot** option is disabled in the BIOS. (secure boot prevents loading of some kernel modules if they are not digitally signed)

Note: You need to be in a working directory with no spaces in the path to build the kernel module with the provided Makefile.

References

We strongly recommend you to start your implementation as early as possible as it may require a decent amount of research. The following links might be useful:

- Writing a simple kernel module:
  https://devarea.com/linux-kernel-development-and-writing-a-simple-kernel-module/
- Task Linked List (scroll down to Process Family Tree):
  https://www.informit.com/articles/article.aspx?p=368650
- Linux Cross-Reference:
  https://elixir.bootlin.com/linux/latest/source

Deliverables and Requirements

You are required to submit the following in a zip file (name it username1-username2.zip) to the new Moodle based LearnHub.

- You must push your work to GitHub classroom in addition to LearnHub submission. We will be checking the commits as part of project evaluation, your commits must reflect the progression of your work.
- Although not *required*, we highly encourage you to use the provided `.clang-format` file to autoformat your code. Run the following command to apply formatting.

```
find . -name '*.[ch]' -exec clang-format -i {} \;
```

- `.c` source file that implements the `da`**`sh`** shell. Your code must include enough comments to let another computer engineer understand how it works.
- `.c` source file of the Kernel module used by `psvis`.
- Any supplementary files for your implementation (Makefiles, header files, etc.)
- (5 points) a report describing your implementation. Include screenshots in your report and submit it as a `pdf` file.
- You should keep your GitHub repo updated from the start to the end of the project. Do not commit at the very end when you are finished, instead make consistent commits as you make progress. You will be graded accordingly.
- Implement your code for the Linux OS and also test your code on a different clean Linux installation (i.e. a VM) to make sure that you don't have parts of code which only works on your local setup but not on other systems.