Operating Systems Laboratory (CS39002)

Spring Semester 2022-2023

Assignment 2: Use of syscall

Assignment given on: January 25, 2022

Assignment deadline: February 07, 2022, 11:59 PM [for both parts]

- You learned about syscalls in your lectures. Syscalls are function calls that can be used by your program (i.e., user space program) to let OS perform privileged tasks for you (e.g., reading directly from the keyboard or sending data to your printer). Specifically, the following syscalls in Linux might be useful for you in this assignment and in general.
    - dup
    - excelp
    - fork
    - signal
    - pipe
    - select
    - poll
    - open, read, write
    - chmod
- In this assignment, we will build a Linux-type shell with some basic and some advanced functionalities. The assignment has two parts.

Implement a shell that will run as an application program on top of the Linux kernel. The shell will accept user commands (one line at a time), and execute the same. The following features must be implemented:

Part 1

a) **Run an external command**

The external commands refer to executables that are stored as files. They have to be executed by spawning a child process and invoking **execlp()** or some similar system calls. Example user commands:

./a.out myprog.c

cc –o myprog myprog.c

ls -l

b) **Run an external command by redirecting standard input from a file**

The symbol "**<**" is used for input redirection, where the input will be read from the specified file and not from the keyboard. You need to use a system call like **dup()** or **dup2()** to carry out the redirection. Example user command:

./a.out < infile.txt

sort < somefile.txt

c) **Run an external command by redirecting standard output to a file**

The symbol "**>**" is used for output redirection, where the output will be written to the specified file and not to the screen. You need to use a system call like **dup()** or **dup2()** to carry out the redirection. Example user commands:

./a.out > outfile.txt

ls > abc

d) **Combination of input and output redirection**

Here we use both "<" and ">" to specify both types of redirection. Example user command:

./a.out < infile.txt > outfile.txt

e) **Run an external command in the background with possible input and output redirections**

We use the symbol "**&**" to specify running a command in the background. The shell prompt will appear and the next command can be typed while the said command is being executed in the background. Example user commands:

./a.out &

./myprog < in.txt > out.txt &

f) **Run several external commands in the pipe mode**

The symbol "**|**" is used to indicate pipe mode of execution. Here, the standard output of one command will be redirected to the standard input of the next command, in sequence. You need to use the **pipe()** system call to implement this feature. Example user commands:

ls | more

cat abc.c | sort | more

g) **Interrupting commands running in your shell (using signal call)**

- Implement a feature to halt a command running in your shell during runtime. For instance, if the user presses "Ctrl - c" while a program is executing, the program should stop executing, and the shell prompt should reappear. Note that the shell should not stop if the user presses "Ctrl - c".

- Implement a feature to move a command in execution to the background. If the user presses "Ctrl - z" while a program is executing, the program execution should move to the background and the shell prompt should reappear.

Part 2

h) Implementing cd and pwd

Implement the “cd” command which changes the current working directory of the shell, and the “pwd” command which prints the name of the working directory.

i) Handling wildcards in commands (‘*’ and ‘?’)

Wildcards are a feature that allows for a single command to be performed on multiple files matching the wildcard. The ‘*’ character matches 0 or more characters. The ‘?’ character matches a single character.

Examples: Suppose there are files named file1.txt, file2.txt, file3.txt in directories named dir1, dir2, dir3 respectively.

The following commands open all 3 files.

- gedit dir?/file?.txt
- gedit dir1/file1.txt dir2/file2.txt dir3/file3.txt
- gedit d*/f*

The following commands concatenate the 3 files and sort them.

- sort dir?/file?.txt
- sort dir1/file1.txt dir2/file2.txt dir3/file3.txt
- sort d*/f*

Implement a feature such that your shell program handles the wildcards ‘*’ and ‘?’

j) Implementing searching through history using up/down arrow keys

Maintain a history of the last 1000 commands run in your shell (hint: check how bash saves a history in a file). Pressing the ‘up’ arrow key should show the previous command run in your shell. If no previous command exists, then do nothing.
When some previous command is being shown (since the up arrow key was pressed), pressing the down arrow key should show the command run after the command being shown (similar to a typical shell). When showing the current command, pressing the down arrow key should do nothing.

An example: Suppose the following commands were run in your shell
>> touch myfile.txt
>> cd foldername
>> <Current command>

1. Pressing the up key when you are at current command should change the current command to “cd foldername”. Pressing the down arrow key at the current command should do nothing.
2. Pressing the up key when “cd foldername” is being shown should change the command to “touch myfile.txt”. Pressing the down key when “cd foldername” is being shown should show the current command (empty if nothing has been typed).
3. Pressing the up key when “touch myfile.txt” is being shown should do nothing. Pressing the down key when “touch myfile.txt” is being shown should change the command to “cd foldername”.
4. If the 1000th previous command is being shown, then pressing the up key should do nothing. (“touch myfile.txt” is the second previous command in the example given here).

k) Command to detect a simple malware

A process can spawn multiple processes in the system, a technique used by malwares (e.g., trojans). A malware spawns multiple child processes and then goes to sleep and the child processes wreck havoc in your system. So, of course you can run 'top' command, check which processes are using most cpu and kill them using signals, but it will do you no good. The sleeping malware parent process will wake up and spawn even more processes.

To deal with this situation, implement a command "sb" (short for "squashbug") in your shell that will create a child which does the job of detecting malwares. This command starts from a given process id (that the user identifies from htop as a suspected process and supplies as argument) and displays the *parent, grandparent, …* of the given process. Note that the actual malware can be any of the parent, grandparent, of the given process, or the given process itself. Keep a flag "-suggest" which will additionally, based on a heuristic, detect which process id is the root of all trouble (one way: check the time spent for each process and / or the number of child processes that a process has spawned, find anything suspicious). Explain and justify the heuristic in a report (to be submitted), name the report assgn2.squashbug.heuristic.txt.

Also write a test-case for this command – a process P that sleeps for 2 minutes, then wakes up and spawns 5 processes and again goes to sleep. Each of these 5 spawned processes will again spawn 10 processes each, and then run an infinite loop. If the sb command is run, this process P should be identified as the malware. This program should NOT be a part of the shell program, rather a separate program.

l) Command to check for file locks

Some processes start reading/writing a file after locking (using flock) and never release it, making it impossible to delete a file. Implement a command "delep" (short for delete with extreme prejudice) which takes a filepath as argument and spawns a child process. The child process will help to list all the process pids which have this file open as well as all the process pids which are holding a lock over the file. Then the parent process will show it to the user and ask permission to kill each of those processes using a yes/no prompt. On putting yes, all of those processes will be killed using signal and then the file will be deleted. Also create a test code which will spawn a process which locks a file and try to write to a file using a while 1 loop.

m) Features to help editing commands

In a shell often after writing a long command you need to go back to the beginning of the command. In your shell add a feature "ctrl + 1" will bring the cursor to the beginning of your typed command, and a feature "ctrl + 9" that will bring the cursor at the end of the typed command.

Implementation Help:

For redirecting the standard input or output, you can refer to the book: "*Design of the Unix Operating System*" by Maurice Bach. Actually, the kernel maintains a file descriptor table or FDT (one per process), where the first three entries (index 0, 1 and 2) correspond to standard input (**stdin**), standard output (**stdout**), and standard error (**stderr**). When files are opened, new entries are created in the table. When a file is closed, the corresponding entry is logically deleted. There is a system call **dup(xyz)**, which takes a file descriptor **xyz** as its input and copies it to the first empty location in FDT. So if we write the two statements: **close(stdin); dup(xyz);** the file descriptor **xyz** will be copied into the FDT entry corresponding to **stdin**. If the program wants to read something from the keyboard, it will actually get read from the file corresponding to **xyz**. Similarly, to redirect the standard output, we have to use the two statements: **close(stdout); dup(xyz);**

Normally, when the parent forks a child that executes a command using **execlp()** or **execvp()**, the parent calls the function **wait()**, thereby waiting for the child to terminate. Only after that, it will ask the user for the next command. However, if we want to run a program in the background, we do not give the **wait()**, and so the parent asks for the next command even while the child is in execution.

A pipe between two processes can be created using the **pipe()** system call, followed by input and output redirection. Consider the command: **ls | more**. The parent process finds out there is a pipe between two programs, creates a pipe, and forks two child processes (say, X and Y). X will redirect its standard output to the output end of the pipe (using **dup()**), and then call **execlp()** or **execvp()** to execute **ls**. Similarly, Y will redirect its standard input to the input end of the pipe (again using **dup()**), and then call **execlp()** or **execvp()** to execute **more**. If there is a pipe command involving N commands in general, then you need to create N-1 pipes, create N child processes, and connect each pair of consecutive child processes by a pipe.

Check the GNU Readline library. It may help you in some of the parts in Part 2.

Submission Guideline:

You need to submit a single compressed file (.zip) containing the following;

- A single C program for the shell, named Assignment2_<groupno>_<roll no. 1>_< roll no. 2>_< roll no. 3>_< roll no. 4>.c (replace <groupno> and <roll no.> by your group number and roll numbers).
- A C program to serve as a proxy for a malware, for part (k)
- The report assgn2.squashbug.heuristic.txt, for part (k)

You must show the running version of the program(s) to your assigned TA during the evaluation.

Evaluation Guidelines:

Items	Marks
Part 1	
Process creation and running an external command	8
Redirection of stdin	8
Redirection of stdout	3
Redirection of stdin and stdout together	3
Running an external command in background with I/O redirection	8
Running several external commands in pipe mode	10
Interrupting commands running in your shell	10
Total	**50**
Part 2	
Implementing cd and pwd	4
Handling wildcards in commands (‘*’ and ‘?’)	8
Implementing searching through history using up/down arrow keys	8
Command to detect malware	15
Command to check for file locks	15
Features to help editing commands	15
Total	**65**