Introduction to Linux File System Calls

Objective

To introduce Linux I/O system calls and work with the stdin (standard input), stdout (standard output) and stderr (standard error).

Description

System calls are a set of functions, an API, provided by the operating system by which an application or a user can request services from the OS. Things to keep in mind when dealing with system calls:

- The use of system calls is expensive as they cause switching to kernel mode when called. when the system call is don, the OS switches back to user mode and the process continues execution where it left off.
- System calls are less portable than standard library functions because, and unless they comply to a common standard, they are different for different operating systems.
- It is warranted to check if the system call succeeded before moving on with the rest of the program.

A subset of system calls deal with file I/O through which, a user can create, open, close, read and write files. In Linux, a file is identified by a nonnegative integer, file descriptor, that is used in subsequent system calls (read(2), write(2), lseek(2), fcntl(2), etc.) to refer to the open file.

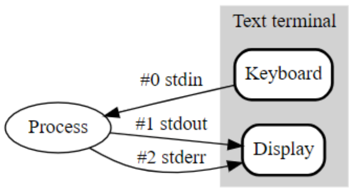
In general, operating systems put limits on how many files a process can open and on how many files can be open at any time. Since tracking the file descriptors of all open files consumes computer resources and takes time, it is always recommended to close all files that are no longer needed.

The main system calls Linux provides for I/O are:

- `int open(char *path, int flags, int mode);`
  http://man7.org/linux/man-pages/man2/open.2.html
- `int close(int fd);`
  http://man7.org/linux/man-pages/man2/close.2.html
- `int read(int fd, char *buf, int size);`
  http://man7.org/linux/man-pages/man2/read.2.html
- `int write(int fd, char *buf, int size);`
  http://man7.org/linux/man-pages/man2/write.2.html
- `off_t lseek(int fd, off_t offset, int whence);`
  http://man7.org/linux/man-pages/man2/lseek.2.html
- `int access(const char *pathname, int mode);`
  http://man7.org/linux/man-pages/man2/access.2.html
- `void *mmap(void addr[.length], size_t length, int prot, int flags, int fd, off_t offset);`
- `int munmap(void addr[.length], size_t length);`
  https://man7.org/linux/man-pages/man2/mmap.2.html

Under normal circumstances, newly created Linux processes have a set of I/O file descriptors that are opened and can be accessed by the process:

- `stdin` (standard input): defaults to the keyboard with a file descriptor of 0.
- `stdout` (standard output): defaults to the screen with a file descriptor of 1.
- `stderr` (standard error): defaults to the screen with file descriptor of 2.



Source: https://en.wikipedia.org/wiki/Standard_streams

C standard libraries provide consistent interface to the programmer while their implementation relies on the system call provided by the underlying operating system. In some cases, using standard libraries improve performance by minimizing the number of times system calls are invoked which reduces the time lost in switching to and from kernel mode. Known as buffered I/O, library functions read and write to a buffer in memory and only when the buffer is full or empty, a system call is invoked.

Submission:

- *One compressed file that contains all C files*
- *Please make sure to build your code in a modular fashion (functions) that you can use in different programs.*
- To get used to system calls, please pay attention to the following:
  - Avoid using library functions (e.g. printf, scanf,..) and use only system calls
  - Check the success of every system call you use
  - Close files when done using them

1. **Echo user input in same case but reverse order (revEcho.c): (40 Points)**

   Write a C program that reads input from keyboard and writes it to the terminal in reverse. When the user presses enter, this input is first stored into a buffer of fixed size (*e.g. 128 byte*). These characters should then be reversed into a dynamically allocated array of the appropriate size and then printed to the terminal. The dynamically sized array should be allocated using `mmap()`. See Figure 1.

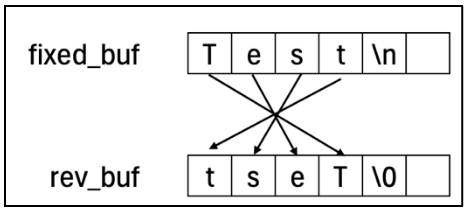
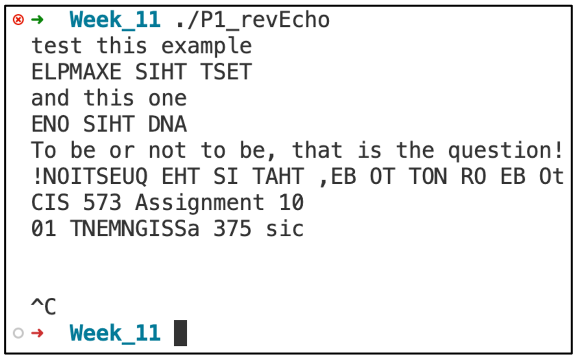


Figure 1 – Input reads into a fixed buffer and is then reversed into a dynamically allocated array.

Your program should run in an infinite loop, terminating only when the user enters CTRL-C. See Figure 2. Test and report the behavior of your tool in the following 3 cases:

```
$lastnameREVECHO > temp0
$lastnameREVECHO < temp0
$lastnameREVECHO < temp0 > temp1
```

Please include a short text description of the behavior observed in each case in a separate text file revEcho.txt.



```
⊗ ➜  Week_11 ./P1_revEcho
  test this example
  ELPMAXE SIHT TSET
  and this one
  ENO SIHT DNA
  To be or not to be, that is the question!
  !NOITSEUQ EHT SI TAHT ,EB OT TON RO EB Ot
  CIS 573 Assignment 10
  01 TNEMNGISSa 375 sic


  ^C
○ ➜  Week_11 
```

Figure 2 – Expected behavior of revEcho

2. Concatenate two files (myCAT.c) (30 Points)

Write a C program (call it `myCat`) that, given two input files (`file1` and `file2`), it adds their contents and place the results into a new file (`file3`). The command takes a flag that determines how the files will be added. The command accepts only the following flags:

- '-s': start with file1
- '-e': end with file1

For examples and given the following files:

`file1` contains: "This is the content of file 1."
`file2` contains: "This is the content of file 2."

The following command will create `file3`:
`$myCAT -s file1 file2 file3`
`file3` contains: This is the content of file 1.

This is the content of file 2.

While the following command will create `file3`:

`$myCAT -e file1 file2 file3`
`file3` contains: This is the content of file 2.

This is the content of file 1.

Your command should print proper error messages in case the user does not give the command the right number of arguments or the wrong flag (anything other than '-s' or '-e'):

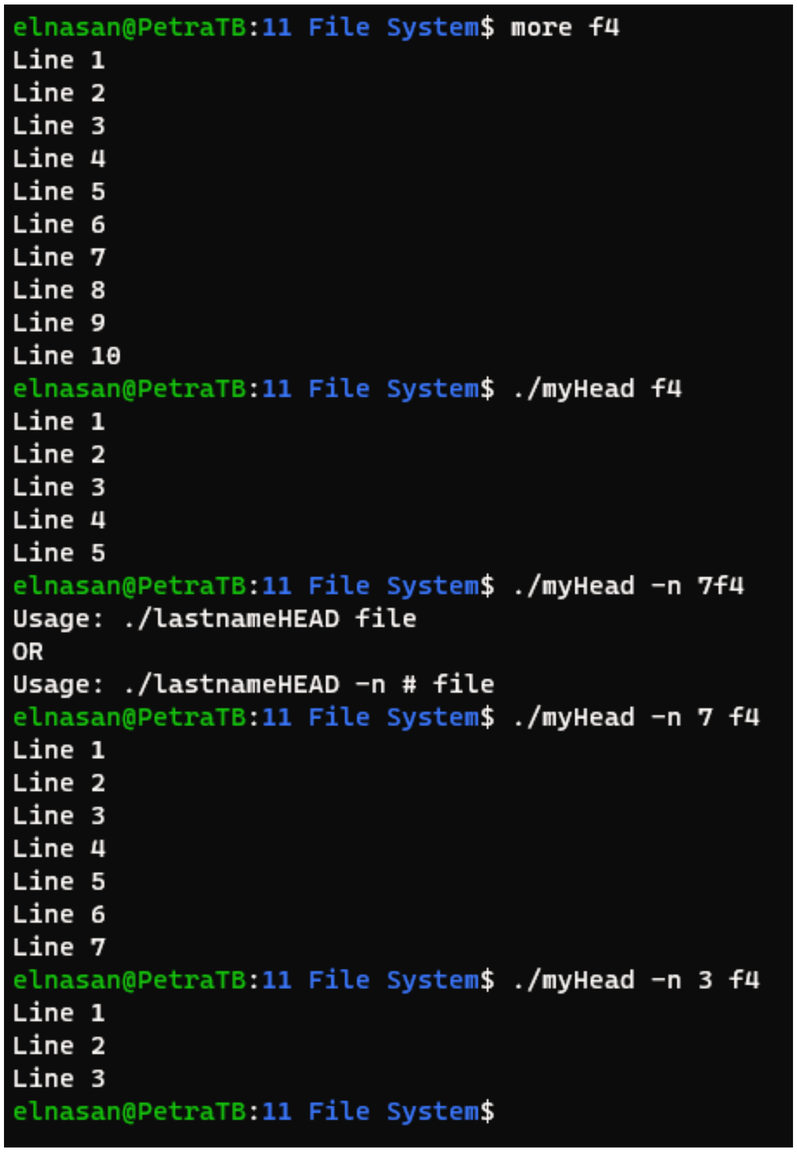
`$myCAT -e file2 file1`

Error! Usage: `myCAT` <s,e> file1 file2 file3

`$myCAT -q file2 file1 file3`
Error! Usage: you should use either '-s' or '-e' then the file names

3. **Print the first 5 lines of a file (myHEAD.c): (30 Points)**
Write a C program that takes a file as an argument and, by default, prints its first 5 lines (equivalent to using `head -n 5`). If the user wants to print the first n lines (not the default value of 5) then the user can use -n flag to specify how many lines to print (e.g. to print first 10 lines from file someData: myHEAD -n 10 someData). If this program does not receive the correct number of arguments, or if it fails to open the given file, it should return errors.

Hint: Write a function `readLine()` that reads characters until it hits a newline character – run this function in a loop to get and print top lines. **Assume that the file has enough data lines.**



```
elnasan@PetraTB:11 File System$ more f4
Line 1
Line 2
Line 3
Line 4
Line 5
Line 6
Line 7
Line 8
Line 9
Line 10
elnasan@PetraTB:11 File System$ ./myHead f4
Line 1
Line 2
Line 3
Line 4
Line 5
elnasan@PetraTB:11 File System$ ./myHead -n 7f4
Usage: ./lastnameHEAD file
OR
Usage: ./lastnameHEAD -n # file
elnasan@PetraTB:11 File System$ ./myHead -n 7 f4
Line 1
Line 2
Line 3
Line 4
Line 5
Line 6
Line 7
elnasan@PetraTB:11 File System$ ./myHead -n 3 f4
Line 1
Line 2
Line 3
elnasan@PetraTB:11 File System$
```

Figure 1 – Expected behavior of myHEAD (data file f4)