SdP 2022 – OS161 Laboratory – 5

To face this laboratory it is necessary:

- ***having carried out (and <u>understood</u>) laboratories 2 and 4,***
  - ***laboratory 2 is needed to understand read and write system calls (partially implemented),***
  - ***laboratory 4 for process management;***
- ***having seen and understood the os161-userprocess lesson, in which, in addition to an overall view of user processes and an implementation (partial, limited to the console) of the read/write system calls, the low-level operations needed to read/write a file are described in the context of load_elf()***

Implement the file system support: open/close system calls and (complete) read/write

OS161 lacks support for the file system, intended as the set of system calls that provide file operations, such as: `open()`, `read()`, `write()`, `lseek()`, `close()`, `dup2()`, `chdir()`, `__getcwd()`. This support, in addition to requiring the implementation of individual functions, also needs appropriate data structures (tables for open files, directories, etc.) that allow the search of files and their identification starting from an integer `id`.

We ask you to implement a partial and simplified form of support (`open()`, `close()`, `read()` and `write()`), after getting familiar with the data structures and kernel functions involved:

- The Virtual File System (VFS): .c in `kern/vfs` and .h `kern/include/vfs.h`. In this context, the `struct vnode` represents a file. The `vfs_open()` and `vfs_close()` functions are used to open and close the file (see for example `runprogram()`).

- Support for data transfer between files, kernel memory and user memory: `uio` module. The functions of main interest in this context are: `uio_kinit()` (followed by VOP_READ or VOP_WRITE) for kernel IO (for IO with buffers in userspace a manual variant of `uio_kinit()` is required) and `uiomove()` (called by `copyout()`, `copyin()` and `copystr()`), which transfers data between user and kernel memory. Note that `copyout()` and `copyin()` are used to copy data between user memory and kernel memory safely, that is, without the kernel crashing in case of an incorrect user pointer.
  For example, given a string of length `len` in user space (`userptr_t stru`) to be copied in `strk` (in kernel space: `char *strk`), the copy should be made with
  ```
  copyin(stru, strk, len+1);
  ```
  instead of
  ```
  for (i=0; i<=len; i++) {
    strk[i] = ((char *)stru)[i];
  }
  ```
  or
  ```
  memcpy (strk, (void *)stru, len+1);
  ```
  Strings properly allocated have been considered. It was also assumed that the string terminator (`'\0'`) would also be copied.

We recommend that you observe how the IO is managed in `runprogram/load_elf/load_segment`, from which you can take inspiration for further research on the functions involved. In particular, it is suggested to see how a file is opened and closed in `runprogram()`, and to look at `load_segment()` to perform (in `sys_read` and `sys_write`) reading/writing between files (given a `vnode`) and user buffer (a `vaddr_t`).
In particular, reading and writing are carried out with VOP_READ and VOP_WRITE, preceded by an appropriate initialization of `struct iovec` and `struct uio` (see `load_segment()`).

To test `open/close`, it is recommended to use `testbin/filetest` (without arguments, if you want to use arguments to the main you should first create its support, as indicated in the optional part that follows).

We suggest you to implement `open()` and `close()` in the same files already used for `read()` and `write()`. For a given user process, it is necessary to generate a table of `vnode` pointers (for simplicity, create an array of these pointers where you save a `vnode` pointer for each new file created), or a table of struct where one of the fields is a `vnode` pointer. It is not yet required (although possible) to implement the double table (user and kernel), which would allow file sharing between processes.
A file is opened and closed with the `vfs_open()/vfs_close()` functions (see `runprogram()`). The `OPEN_MAX` constant (`limits.h`) defines the maximum number of files open for a process. Each open file is assigned a file descriptor, a non-negative integer (the minimum among those not occupied at the time of the `open()`).

Warning: unless they are redirected to file, `stdin`, `stdout` and `stderr` are associated to the console (`kern/dev/generic/console.c`) managed by `kprintf()` (which indirectly calls `putch()`) and `kgets()` (which calls `getch()`). The file table must be managed in such a way that the presence or absence of a file on which `stdin`, `stdout` or `stderr` has been redirected, determines the type of IO to be performed (console or file).

Passing arguments to the main **(OPTIONAL TOPIC)**

In order to pass arguments (`argv`) to a user program, it is necessary to load the arguments into the program address space (on the stack of the user address space): both the strings and the `argv` array of arguments (an array of pointers).

Warning! A char can be located in any virtual address, while a pointer must be located at an address multiple of 4 (padding). A similar constraint exists for the stack pointer, which must be at an address multiple of 8 (since the largest representable data (double) is 8 bytes). To understand how passing `argv` and `argc` works, consider, for example, `testbin/tail`, which requires two arguments. The `main()` of `tail` is:

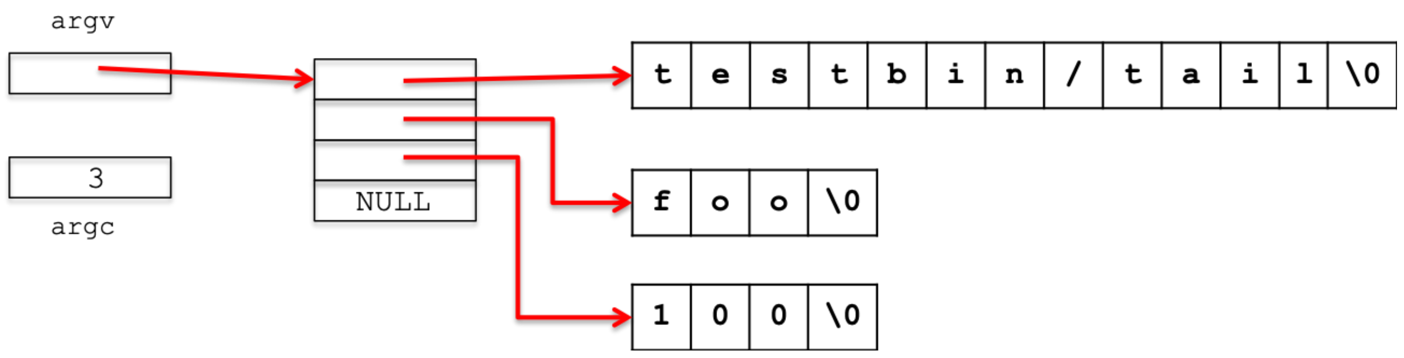
```
int main(int argc, char **argv) {

        int file;
        if (argc < 3) {
               errx(1, "Usage: tail  ");
        }
        file = open(argv[1], O_RDONLY);
        if (file < 0) {
               err(1, "%s", argv[1]);
        }
        tail(file, atoi(argv[2]), argv[1]);
        close(file);
        return 0;
}
```

`argv` is a pointer to a char pointer – it points to an array of character pointers, each pointing to one of the arguments. For example, if `tail` was called with a command line:

```
OS/161 kernel [? for menu]: p testbin/tail foo 100
```

The `argv` and `argc` variables would be as in the figure:


argv
3
argc
NULL
t e s t b i n / t a i l \0
f o o \0
1 0 0 \0