OS161 PROJECT: C2 - SHELL

Professor Gianpiero Cabodi

Students:

- **Grazia D'Onghia** - S284633
- **Alessandro Rea** - S280145
- **Vincenzo Sagristano** - S292447

Abstract: project's summary

The purpose of this project is to support running multiple processes at once from actual compiled programs stored on disk. These programs will be loaded into OS161 and executed in user mode, under the control of your kernel and the command shell in bin/sh (menu command: `p bin/sh`). The project is highly based on the availability of the `execv` and `dup2` system calls. The project can be limited to the EMUFS emulated file system.
To achieve this goal, we have been in charge of the implementation of the following system calls:

- **`open`**

- `read`
- `write`
- `close`
- `lseek`
- `dup2`
- `chdir`
- `getcwd`
- `getpid`
- `fork`
- `execv`
- `waitpid`
- `_exit`

Kernel

Accondingly to the convention, the configuration file in the /config folder is named **SHELL**.

`waitpid`

Prototype: **`int sys_waitpid(pid_t pid, userptr_t statusp, int options, pid_t *err);`**

`sys_waitpid` has been implemented in `kern/syscall/waitpid.c`, essentially following the same logic developed during OS161 lab 4. The basic structure is the following:

`sys_waitpid` takes as parameters :

- the `pid` of the waited process
- a user pointer to the status of the waited process (`statusp`)
- an `options` flag, which is handled just for one value: `if options == WNOHANG` and the waited process is not exited yet (`!p->p_exited`), then `sys_waitpid` returns 0. A flag `p_exited` has been added to `struct proc` and it is initialized to `false` inside `proc_create` function, while it is set to `true` inside `sys__exit`
- a pointer to the error code `*err`. If something goes wrong then the error code is placed in `*err` and the function returns `-1`

At the beginning some validity checks are performed, especially on `statusp` , `pid` and `curproc`, i.e the one which is waiting:

- `statusp` must be properly aligned by 4 since it is a `userptr_t`, hence it can not even point to a kernel memory region (`statusp <= 0x80000000`). Finally `statusp` must be a valid pointer (`statusp != 0x40000000`). If `statusp` is `NULL` then it is not required to collect the exit status.
- `pid` cannot be negative and it must belong to an existing process
- `curproc` checks, performed after function `proc_search_pid` is called, are the most interesting ones:
  - a process cannot wait for itself
  - a process can perform waitpid **just** on its children, i.e waiting for itself or for a parent is not allowed

If all these checks are passed, then `proc_wait` function is called. Inside the latter two wait options are possible:

1. the parent process waits on child process semaphore (`P(proc->p_sem)`)

2. the parent process waits on a condition variable, hence `struct cv *p_cv` and `struct lock *p_cv_lock` fields have been added to `struct proc`. the condition is `while (!proc->p_exited)`

At the end the return status is copied onto user space through `copyout` and the waited process' pid is returned.

`exit`

Prototype: **`void sys__exit(int status);`**

`sys__exit` has been implemented in `kern/syscall/exit.c`

This system call takes as parameter the exit status `status` of the process which is exiting and doesn't return because the last action executed is calling `thread_exit`, which should not return.

At the beginning all open files of `curproc` are closed by calling `cloas_all_files(struct proc *p)`, then `status` is assigned to `curproc->p_status` and the current thread is removed from the process through `proc_remthread(struct thread *t)`. Of course `waitpid` and `exit` system calls are related, that is `p->p_exited` flag is set to true and then either the semaphore or the condition variable is signaled.

A call to `sys__exit` has been added inside `kill_curthread`, so that if a user program tries to access to an illegal memory region it immediately exits with `code` parameter passed to `kill_curthread`.

`getpid`

Prototype: `pid_t sys_getpid(void);`

This system call has been implemented during lab 4 and it simply returns the pid of the current process.

`fork`

Prototype: **`int sys_fork(struct trapframe *ctf, pid_t *retval);`**

`sys_fork` has been implemented in `kern/syscall/fork.c` , requiring the addition of two new fields in `struct proc`:

1. `struct proc *p_parent`, which is a pointer to the parent process. It is set to `curproc` (since `curproc` is the parent) inside `sys_fork` after the child process has been created.
2. `struct array *p_children`, which is the children array of the parent process. `array` is a special data structure defined in OS161 which handles arrays of several types since it is structured like this:

   ```
   struct array {
     void **v;
     unsigned num, max;
   };
   ```

   it also has specific methods and functions, making easier children handling in `sys_fork`.

`sys_fork` takes as parameter the parent's trapframe `ctf` and the pointer `*retval`. At the very beginning default validity checks are performed onto `curproc`. Then the following workflow is done:

1. child process `child` is created through `proc_create_runprogram` : `proc` data structure is allocated and `curproc->p_cwd` is copied inside child process. After child creation some validity checks are performed on `child->p_pid`
2. child's trapframe `tf_child` is allocated and `ctf` is copied to it through `memcpy(void *dest, const void *src, size_t len)` function, which copies `len` bytes from `src` to `dest`
3. parent's address space is copied to child through `as_copy(struct addrspace *old, struct addrspace **ret)`
4. child points to parent, i.e `p_parent` field of `child` is set to `curproc`

5. parent calls `thread_fork` with `call_enter_forked_process` as entrypoint. The latter is the caller function for `enter_forked_process`. It's the child process which executes this function. This is what happens inside `enter_forked_process`:

    - child's trapframe is duplicated so that it's on kernel stack and not on the heap.
    - return values are set inside trapframe fields. `$a3` stores 0 , and `$v0` stores return value (child process returns with zero) or `$v0:$v1` if retval is 64-bit; on failure, `$a3` stores 1, and `$v0` stores error code.
    - child's address space is activated
    - `mips_usermode` is called, and the newly created child process can start running as user process.

    `thread_fork` will set newly created child thread runnable and try to switch to it immediately. So it's highly possible that before `thread_fork` returns, the child thread is already running. This could be a problem if the child thread starts running before parent's `fileTable` is copied, because child thread would run without a `fileTable`. This problem has been handled by copying parent's `fileTable` to child process inside `thread_fork`, in mutual exclusion.

6. parent process returns with child's pid: `*retval = child->p_pid`. If something goes wrong then the error code is placed inside `*retval` and the function returns `-1`. Afterwards error checking is performed inside `syscall` function: if the return value is `<0`, then the error code is placed inside the `err` variable.

`execv`

Prototype: **`int sys_execv(char *program, char **args);`**

`sys_execv` has been implemented in `kern/syscall/execv.c`.

The overall flow of `sys_execv` is:

1. Copy arguments from user space into kernel buffer
2. Open the executable, create a new address space and load the elf into it
3. Copy the arguments from kernel buffer into user stack
4. Return user mode using `enter_new_process`

`sys_execv` takes as parameters `char *program` (the program name) and `char **args` (array with arguments), both passed from user space. After some initial validity checks on arguments and `curproc`, `argc` (number of arguments) is computed and function `copy_args_to_kbuf(char **args, int argc, int *buflen)` is called. The two global static arrays, `char kargbuf[ARG_MAX]` and `karg[ARG_MAX]` represent respectively the kernel buffer (`kargbuf` is treated as a byte array) and the i-th argument that has to be copied onto kernel buffer (there's actually no need to store the whole argument array). These two variables, being global, are protected with the lock `exec_lock`, created inside `proc_bootstrap` in `kern/proc/proc.c` and acquired in `sys_execv` just before copying arguments. The main additional function implemented to make `sys_execv` work is `copy_args_to_kbuf(char **args, int argc, int *buflen)` .

The latter takes as parameters the user arguments, the number of arguments and a pointer to `buflen`, which is how much does the user stack have to move to store the arguments from kernel buffer `kbuf`. `copy_args_to_kbuf` uses pointers arithmetic to build `kargbuf`. This is the approach followed by the function:

- at the beginning `kbuf` is initialized, that is:
    - `unsigned char *p_begin` points to the beginning of the array
    - `int last_offset = argc * sizeof(char *)` is the dimension of the portion of `kbuf` used to store the argument offset
    - `int offset` deals with the "second" portion of `kbuf`, used to store the position of arguments in the stack. This will be used to copy arguments to user stack.
    - `unsigned char *p_end = p_begin + last_offset` is the memory location that stores the actual argument `karg`

    - `int offset` stores the offset of `karg` and it is placed inside memory region pointed by `p_begin`
- a `for` loop contains all the copy process:
    - user arguments are copied inside `karg` through `copyinstr` function
    - `offset` is computed moving from `last_offset` by `padding`
    - `padding` is computed through `padded_length` function, which aligns by 4 (as required by OS161 user stack) the passed string. It basically exploits modular arithmetic to return an `int` value which is the required padding in order to obtain a multiple of 4 as length of `karg`.
    - `karg` is copied in `p_end` through `memcpy` function. `void *memcpy(void *dest, const void *src, size_t len)` copies `n` characters from memory area `src` to memory area `dest`
    - `offset` is stored in memory region pointed by `p_begin`, so it is saved inside `kbuf` .
    - final updates: `p_end` moves by `padding` to store the next argument in the next iteration. `p_begin` moves by 4 bytes (i.e `sizeof(char *)`) in order to store the offset of the next argument in the next iteration. `last_offset` is set to `offset` and `*buflen` is incremented by `padding + sizeof(char *)`

At the end of the `while` loop `*buflen` is finally incremented by 4 bytes and the function returns zero. At this point `kbuf` contains both the arguments and their offset in kernel stack.

Now `sys_execv` copies `program` (program name) in kernel memory through `copyinstr` and performs the same actions performed by `runprogram`: open the executable related to `kprogram` , create and activate the new address space, load the ELF executable and create a stack for the new address space.

Finally the arguments have to be copied again to user space through `copyout` function, so that `enter_new_process` can be called and the user program is executed.

However, before `copyout((void *)kbuf, (userptr_t)stackptr, buflen)` the kernel buffer has to be adjusted for user stack: at this point the function `change_kbuf_for_userstack` is called. This function basically takes the old offset (the `offset` computed in `copy_args_to_kbuf`), adds it to `stackptr` and stores this new value `new_offset` inside `kbuf` at the same position as before. Of course before calling `change_kbuf_for_userstack` the stack pointer has to lower by `*buflen`.

Testing of the execv system call

In order to test `fork, waitpid, getpid` and `exit` , `testbin/forktest` has been chosen among `testbin` programs. In this user program a process forks several times, generating processes "geometrically". Basically `sys_fork` is called 4 times, generating a tree which is printed on `stdout` through letters A, B, C, D.

After each `fork`, `sys_getpid` is called through `check()` function: the latter calls `getpid` in a `for` loop to make sure each fork has its own address space.

In order to avoid out of memory errors in `testbin/forktest` , the ramsize has been increased in `root/sys161.conf` from `512K` to `4096K`. `sys_waitpid` is also tested because once the parent process has forked several times, it waits for its children.

If `sys_execv` is properly implemented, the OS161 shell should work. Namely, a user program should run (even in background) without crashing the kernel. When a user program (e.g `testbin/palin`) is run inside OS161 shell, which performs the following:

1. the shell, i.e the process `/bin/sh`, forks
2. The system search for the program until it finds it, calling`sys_execv` is called 3 times: once for `/bin/palin`, once for `/sbin/palin` and finally for `testbin/palin` and the user program is successfully executed.

`runprogram` uses, let's say, the last part of execv, namely the copy of arguments from kernel stack onto user stack, in order to achieve arguments passing from main. This has been implemented and tested through `testbin/argtest`, which prints on `stdout` all the passed arguments. However this feature has been implemented differently from `sys_execv` : instead of a static array of bytes, a dynamic array of strings `argvptr` is used and the alignment check is done by an AND operation with `0x3`.

These different implementations are due to the fact that `runprogram` has been improved while doing OS161 labs, while `execv` has been written later, basically following the explanation in *Pearls in Life* blog http://jhshi.me/2012/03/11/os161-execv-system-call/index.html#.YcBaFbso-3c .

`chdir`

Prototype: **`int sys_chdir(userptr_t path, int *err);`**

Design of the chdir system call

`sys_chdir` has been implemented in `kern/syscall/curdir_syscalls.c`.

- Parameters:
  - `userptr_t path`: contains the destination path which we want to move at.
  - `int *err`: this integer pointer will collect the information about the eventual error.
- Return value:
  - 0, if the operation has been successfully completed;
  - -1, if there's been an error (see err for the error code).

The path parameter is checked: since it contains an address, it must be

- not NULL
- not equal to 0x40000000
- lower than 0x80000000
  Since the path has been read from the command line, the validity of this raw string is checked by the function `dir_parser`.
  After the allocation of the kpath, which will contain the userptr_t string into the kernel address space, we use copyinstr to indeed copy the content of `path` into the kernel-level `kpath`.

The `vfs_chdir` function (vfs.h) does the most of the work, getting the `kpath` string. In case of error, this will return an error code, which is a positive number greater than zero that can be interpreted by the syscall.c mechanism to be verbose about the error occurred.

Testing of the chdir system call

The `sys_chdir` system call has been tested using the `cd` command provided by OS161 in their menu. There've been notably created different folders in the /root folder in order to test this system call.

`getcwd`

Prototype: **`int sys___getcwd(userptr_t buf, size_t size, int *err);`**

Design of the getcwd system call

`sys___getcwd` has been implemented in `kern/syscall/curdir_syscalls.c`.

- Parameters:
    - `char *buf`: array of char in which it's going to be stored the current working directory path.
    - `size_t size`: size of the `buf`string.
    - `int *err`: this integer pointer will collect the information about the eventual error.
- Return value:
    - the size of the string read, if the operation has been successfully completed;
    - -1, if there's been an error (see err for the error code).

The buf parameter is checked: since it contains an address, it must be

- not NULL
- not equal to 0x40000000
- lower than 0x80000000
  First of all, we check for whether the current working directory is a NULL pointer, setting the error parameter to ENOTDIR (17, "Not a directory").
  It's been introduced an ad-hoc function for setting the user space without involving the kernel level address space: after uio_kinit has been called, there are three major fields to be properly set:
- `iovec->iov_ubase = buf;`
- `uio->uio_segflg = UIO_USERSPACE;`
- `uio->uio_space = proc_getas();`

The `vfs_getcwd` function (vfs.h) does the most of the work, getting as parameter the pointer to `uio`(`struct uio`) and returns a value which has to be checked to detect eventual errors.

The returned value is the difference between the `size` of the buffer in which the string has been stored and the `uio_resid` field, which stores the remaining amount of data to transfer: then, the returned value is the amount of data that has been read.

Testing of the getcwd system call

Testing the `sys___getcwd` system call has been quite tricky. The only directory we've been able to retrieve has been the one in the root, which returns: `emu0:`.
We have then implemented the `sys_fstat` and the `sys_mkdir` in order to mount another filesystem with the procedure described on the OS161 official website, that is:

`OS/161 kernel [? for menu]: p /sbin/mksfs lhd1raw: myDisk`

`OS/161 kernel [? for menu]: mount sfs lhd1`

`OS/161 kernel [? for menu]: cd lhd1:`

but even in this case, running the mkdir command has not been working as wished.
Indeed, the `sys___getcwd`has retrieved the `lhd1:`string, acting the same way of when it's been called in the default filesystem.

`dup2`

`int sys_dup2(int old_fd, int new_fd, int *err);`

Design of the dup2 system call

`sys_dup2` has been implemented in `kern/syscall/file_syscalls.c`.

- Parameters:
    - `int old_fd`: file descriptor related to the file to which the new_fd will be linked at the end of the call
    - `int new_fd`: file descriptor that will be associated to the file related to the old_fd at the end of the call
    - `int *err`: this integer pointer will collect the information about the eventual error.
- Return value:
    - 0, if the operation has been successfully completed;
    - -1, if there's been an error (see err for the error code).

First of all, there's the acquisition of the process' spinlock in order to achieve exclusive access to the fileTable of the process.
The fileTable is a pointer to an array of pointers of struct openfile whose size is **OPEN_MAX** (a constant defined in limits.h, representing the maximum number of files that a process can keep in its fileTable). The struct openfile is made up this way:

```
struct openfile
{
  struct vnode *vn;
  mode_t mode;
  off_t offset;
  int accmode;
  struct lock *file_lock;
  unsigned int ref_count;
};
```

After this operations, the two file descriptors passed as parameters are checked: they must be valid file descriptors (in the `[0, OPEN_MAX]` set) and the `old_fd` must be present inside the `fileTable`.
If the two file descriptors are equals, the system call can terminate with no efforts, returning the `new_fd` value.
Then, if the `new_fd` is previously opened, the dup2 function will close it calling the `sys_close` system call.
Now, `curproc->fileTable[new_fd]` is NULL, so there's the allocation of a new struct openfile entry.
At this point, there's the increment of the reference counter `ref_count` of the entry of the fileTable `fileTable[old_fd]`, then it's time to copy all the field of the data structure from the `old_fd` entry to the `new_fd` one.

```
  curproc->fileTable[new_fd]->vn = curproc->fileTable[old_fd]->vn;
  curproc->fileTable[new_fd]->mode = curproc->fileTable[old_fd]-
>mode;
  curproc->fileTable[new_fd]->offset = curproc->fileTable[old_fd]-
>offset;
  curproc->fileTable[new_fd]->accmode = curproc->fileTable[old_fd]-
>accmode;
  curproc->fileTable[new_fd]->file_lock = curproc->fileTable[old_fd]-
>file_lock;
  curproc->fileTable[new_fd]->ref_count = curproc->fileTable[old_fd]-
>ref_count;
```

and, eventually, release the spinlock of the process.

Testing of the dup2 system call

Testing the dup2 involved the creation of an ad-hoc test called dup2test.
This test is available at the following repository: https://github.com/lifeofvins/dup2test

If the constant in the define TEST_SECTION is set on 1:
a dup2 call (`dup2(fd, STDOUT_FILENO)`) is performed such that it's possible to use printf/scanf to make r/w operations on a file (which filename is hardcoded for the sake of simplicity).
Check the file "dup-2-test.txt" in the root location to understand if the operation worked.
The final content of the file (dup-2-test.txt) will be these lines of text:
"And what marks did you see by the wicket-gate?"
"None in particular."

"Good heaven! Did no one examine?"
"Yes, I examined, myself."
"And foud nothing?"
"It was all very confused. Sir Charles had evidently stood there for five or ten minutes."
"How do you know that?"
"Because the ash had twice dropped from his cigar."

In another test (not showed here), it's been tested also the usage of `dup2(fd1, fd2)` in order to do r/w operations using the fd2 file descriptor on the file represented by the fd1 file descriptor.

If the constant in the define TEST_SECTION is set on 0:
two dup2 calls are performed:

```
dup2(fd2, STDIN_FILENO);
dup2(fd3, STDOUT_FILENO);
```

in order to read from the file descriptor STDOUT_FILENO and write on the STDIN_FILENO, which have been previously warped such that those operations will be done on other files (x.txt and aaa.txt).

`lseek`

Prototype: **`int sys_lseek(int fd, off_t offset, int whence, int *err);`**

Design of the lseek system call

`sys_lseek` has been implemented in `kern/syscall/file_syscalls.c`.

- Parameters:
    - `int fd`: file descriptor related to the file on which make the lseek operation
    - `off_t offset`: this is an integer number representing the displacement from the position specified by the usage of the current position and the `whence` parameter
    - `int whence`: this value represents the point of the file from which it will be done the lseek operation. There are three different valid values:
        - **SEEK_CUR**: the position is the one at which the file is currently located
        - **SEEK_SET**: the position is set to the diplacement passed as parameter in the `offset` parameter
        - **SEEK_END**: the position is set to the end of the file
    - `int *err`: this integer pointer will collect the information about the eventual error.
- Return value:
    - 0, if the operation has been successfully completed;
    - -1, if there's been an error (see err for the error code).

The first operations to be done are the checks over the parameters:

1. The `fd` file descriptor shall be valid: this is checked with the `is_valid_fd` function, which checks `fd` is in the [0; OPEN_MAX] set and that it's actually present into the `fileTable` of the process.
2. As said before, the `whence` parameter must be check in order to accept only one of the three valid values (**SEEK_SET**, **SEEK_CUR** and **SEEK_END**).

Now, based on the `whence` parameter, it's time to compute the actual position for the file:

1. If **SEEK_SET**, then the actual postion is basically the offset passed as parameter;
2. If **SEEK_CUR**, then the actual position is the current position plus the offset passed as parameter;
3. If **SEEK_END**, then VOP_STAT is called to know the size of the file, then the actual offset is computed summing up the size of the file obtained with this latter call and the offset passed as parameter.

During the entire call to `sys_lseek` the access to the file is protected using the file_lock associated to the relative entry in the fileTable: `curproc->fileTable[fd]->file_lock`.

Testing of the lseek system call

The `lseek` system call has been tested in two different ways.
The first way was using the program `/testbin/tail` which is called as following:
`p /testbin/tail <filename> <offset>`
and it was tested with the x.txt file from the root folder.

Then, in order to test the usage of the `whence` parameter in an exaustive way a test has been written that calls lseek three times as following:

```
...
lseek(fd1, 1, SEEK_SET);
read(fd1, buffer, 13);
buffer[14] = '\0';
printf("%s", buffer);
printf("\n---------------\n");

lseek(fd1, 1, SEEK_CUR);
read(fd1, buffer, 13);
buffer[14] = '\0';
printf("%s", buffer);
printf("\n---------------\n");

lseek(fd1, -13, SEEK_END);
read(fd1, buffer, 13);
buffer[14] = '\0';
printf("%s", buffer);
printf("\n---------------\n");
...
```

and the `sys_lseek` function worked perfectly.

`read`

Design

`sys_read` has been implemented in `kern/syscall/file_syscalls.c`and it is similar to the function implemented during lab 2. The basic structure is the following:

`sys_read` takes as parameters :

- `fd`, the file descriptor
- `buf_ptr` is a user pointer to a buffer used to perform a reading operation
- `size` represents the size of the buffer
- a pointer `*err`, in order to store the error type if an error occurred

At the beginning several checks are done to ensure the parameters correctness :

- `buf_ptr` needs to be a not-null pointer.
- `buf_ptr` is asked to point to a segment that is within the user space.

Inside the function different operations are executed based on the value assumed by `fd`:

- if `fd == STDERR_FILENO || fd == STDOUT_FILENO`, to face the possibility of a previous `dup2` call involving `STDOUT_FILENO` or `STDERR_FILENO`, different scenarios must be taken into account:
    - If `curproc->fileTable[fd] == NULL` it means that does not exist an opened file with `fd` as a file descriptor in the current process file table. An error type is stored in `*err` and `-1` is returned.
    - If `curproc->fileTable[fd]->vn == systemFileTable[fd].vn` we are in the case in which a `dup2` function has not been performed previously. It means that a reading operation is not allowed on `STDOUT_FILENO` or `STDERR_FILENO`, so the value of the file descriptor is not correct for our purposes. An error type is saved into `*err` and `-1` is returned.
    - If `curproc->fileTable[fd] != NULL`, one among `STDOUT_FILENO` and `STDERR_FILENO` is a valid file descriptor of a opened file in the current process file table. In this case, a call to `file_read(fd, buf_ptr, size, err)` is returned.

    - If we are not in one of the previous situations it means that `fd` is a valid file descriptor for a reading operation on standard input. Consequently, a "classic" standard input operation is performed.
- if `fd == STDIN_FILENO`, two situations are possible:
    - the `if (curproc->fileTable[fd] != NULL)` clause extends the functionality of the `dup2` to the case in which `STDIN_FILENO` is the file descriptor of a file located in the `fileTable` of the current process. The call to `file_read(fd, buf_ptr, size, err)` is returned.
    - in the other case a "classic" standard input operation is executed. In this case the `size` of the read bytes is returned.
- if we don't have none of the previous situations, `file_read(fd, buf_ptr, size, err)` is returned, and a file read operation is performed.

The function returns the number of bytes read.

`write`

Design

`sys_write` has been implemented in `kern/syscall/file_syscalls.c`and it is similar to the function implemented during lab 2. The basic structure is the following:

`sys_write` takes as parameters :

- `fd`, the file descriptor
- `buf_ptr` is a user pointer to a buffer used to perform a writing operation
- `size` represents the size of the buffer
- a pointer `*err`, in order to store the error type if an error occurred

At the beginning several checks are done to ensure the parameters correctness :

- `buf_ptr` needs to be a not-null pointer.
- `buf_ptr` is asked to point to a segment that is within the user space.

Inside the function different operations are executed based on the value assumed by `fd`:

- if `fd == STDIN_FILENO`, to consider the possibility of a previous `dup2` call involving `STDIN_FILENO`, different scenarios must be taken into account:
  - If `curproc->fileTable[fd] == NULL` it means that does not exist an opened file with `fd` as a file descriptor in the current process file table. An error type is stored in `*err` and `-1` is returned.
  - If `curproc->fileTable[fd]->vn == systemFileTable[fd].vn` we are in the case in which a `dup2` function has not been performed previously. It means that a writing operation is not allowed on `STDIN_FILENO`, so the value of the file descriptor is not correct for our purposes. An error type is saved into `*err` and `-1` is returned.
  - If `curproc->fileTable[fd] != NULL`, `STDIN_FILENO` is a valid file descriptor of a opened file in the current process file table. In this case, a call to `file_write(fd, buf_ptr, size, err)` is returned.
  - If we are not in one of the previous situations it means that `fd` is a valid file descriptor for a writing operation on standard output. Consequently, a "classic" standard output operation is performed.

- if `fd == STDOUT_FILENO || fd == STDERR_FILENO`, two situations are possible:
    - the `if (curproc->fileTable[fd] != NULL)` clause extends the functionality of the `dup2` to the case in which one among `STDOUT_FILENO` and `STDERR_FILENO` is the file descriptor of a file located in the `fileTable` of the current process. The call to `file_write(fd, buf_ptr, size, err)` is returned.
    - in the other case a "classic" standard output operation is executed. In this case the `size` of the written bytes is returned.
- if we don't have none of the previous situations, `file_write(fd, buf_ptr, size, err)` is returned, and a file write operation is performed.

The function returns the number of bytes written.

`open`

Design

`sys_open` has been implemented in `kern/syscall/file_syscalls.c` starting from the solution provided during the lab5.

`sys_open` takes as parameters :

- `path` is a user pointer to the path of the file
- `openflags` represents the access mode
- `mode` defines the permissions of the file
- a pointer `*err`, in order to store the error type if an error occurred

The important data structure used in this function is `struct openfile`, defined as follows:

```
struct openfile {
struct vnode *vn;
mode_t mode;
off_t offset;
int accmode;
struct lock *file_lock;
unsigned int ref_count;
}
```

1. `struct vnode *vn;` is a pointer to a vnode.
2. `mode_t mode;` indicates the type of permissions allowed for the file to be opened.
3. `off_t offset;` represents the position inside the file where to start the read/write operation.
4. `int accmode;` tells us the access mode of the current file.
5. `struct lock *file_lock;` is used to enforce mutual exclusion between cuncurrent operations performed on the same file.
6. `unsigned int ref_count;` stores the number of processes "active" on this file.

At the beginning several validity checks are implemented:

- if `(path == NULL)` the error type is saved inside `*errp` and `-1` is returned.
- if `accmode` is different from O_RDONLY, O_WRONLY or O_RDWR, there isn't a valid access mode for the file, therfore an error type is saved inside `*errp` and `-1` is returned.

If all the checks are completed successfully the core of the function starts.

First of all `copyinstr` is used in order to copy PATH_MAX bytes from a user-space address `path` to a kernel-space address `fname`. After that, `vfs_open` is called and a virtual node structure (`struct vnode *v`) is obtained from it. If `systemFileTable`(data structure used to track the opened files in the system) contains free slots where to place a openfile struct, then a pointer to an entry of `systemFileTable` is assigned to `of`, and its elements are initialized. Once checked the overall number of files opened in the system, we need to ensure that inside the `fileTable` of the current process there is enough space, and if this is the case, we assign `of` to an entry of `fileTable`. Another situation that needs to be taken into account is the value of `of->offset`. Indeed if the bitwise AND between `openflags` and O_APPEND gives `1` as a result, it means that `of->offset` should no longer be equal to zero but equal to the file size. So we need to check it and use So we need to check it and use `VOP_STAT` to get file size if necessary.

If all went smoothly `sys_open` returns the file descriptor, `-1` otherwise.

`close`

Design

`sys_close` has been implemented in `kern/syscall/file_syscalls.c` starting from the solution provided during the lab5.

`sys_close` takes as parameters :

- `fd`, the file descriptor
- a pointer `*err`, in order to store the error type if an error occurred

The aim of this function is to check possible errors occurred during the file close operation and to de-allocate objects related to the file management.

At the beginning, some validity checks are performed to make sure that:

- `fd` is a valid file descriptor.
- `vn` and `of` are not-null variables.
  If these conditions are not satisfied, an error type is saved inside `*err` and `-1` is returned.

Once passed the latters, the `sys_close` controls if the process which is performing the close operation, is the last one working on the file (`if (of->ref_count == 1)`). If so, the file is closed and the lock destroyed, `of->ref_count` is decremented otherwise.

In any error occurs, `sys_close` returns `0`.

Testing of the functions `sys_open`, `sys_close`, `sys_write` and `sys_read`

The functions inside `file_syscalls.c` have been tested with `f_test.c`, located in `userland/testbin/f_test`.
For our purposes `f_test.c` was run respectively with `1` and `3` as arguments on the command line.
So, executing `p f_test 1` checks different operations performed on a big file instead `p f_test 3` makes sure that concurrent operations on the same file behave properly.
Several other tests that indirectly call system calls included in `file_syscalls.c` have been performed:
`userland/testbin/bigseek`, `userland/testbin/badcall` and `userland/testbin/tail`.

Teamwork organization

Our team has been managed following some teamworking principles:

1. The workload has been divided proportionally in order to obtain the best parellalization of the workflow, whether possible. Meanwhile, the biggest issues related to the implementation of the system calls have been faced together in person (i.e. study rooms) or via communication platforms (i.e. Discord).
2. We have had a call on Discord every two weeks in order to take stock of the developing situation and face together the biggest doubts. We fixed several major deadlines, giving each other freedom and calm about the achivement of minor goals.
3. Bug fixing has been the most challeging phase: eventually, when the majority of the workload had been done, we have debugged the entire code in three different ways: single-person-testing on their own system calls, single-person-testing on system calls implemented by somebody else of the group and group-testing on the general OS161 system.
4. For project management and version control we've used Git: the entire folder (os161-base-2.0.3) has been uploaded in order to get synchronized even on the homemade tests which are resident in the userland folder.