OS161 SHELL Project

Implement system calls (for file and process management) and exception handling

Bongo Federica 292395
Rizzello Stefano 288013

Project Goals

The purpose of this project is to support running multiple processes at once from actual compiled programs stored on disk.

The system calls implemented for process and file management are:

File System Calls

- Open()
- Read()
- Write()
- Close()

Process System Calls

- Exit()
- Fork()
- Execv()
- Getpid()
- Waitpid()

Preliminary Operations

Preliminary Operations

To implement system calls, changes were made within the file systems.

In **proc.h**, the File Table and Process Table were managed and the structure of the processes was defined by adding their IDs. As far as the synchronisation of threads is concerned, a lock and a flag for their exit were added. In addition, the variable myCV was added for communication between the Parent process and the Child process.

In **synch.c** the Lock has been modified to synchronise threads concerning system calls, and cv for the waitpid system call.

Writing a new system call

Writing a new system call

1. In **kern/arch/mips/syscall/syscall.c** there is switch case in which you can menage the system calls
2. To menage a system call it's necessary an integer code. See: **kern/include/kern/syscall.h**
3. Create your system call in **kern/syscall/my_syscalls.c**
4. Modify **kern/include/syscall.h** adding a prototype of the syscall function
5. Modify **kern/conf/conf.kern** inserting the new file (#System call layer):

   file syscall/my_syscalls.c

6. Finally do the make of the kernel

   In **kern/conf**
   ./config DUMBVM

   In **kern/compile/DUMBVM**
   bmake depend
   bmake
   bmake install

File System Calls

- Open()
- Read()
- Write()
- Close()

Open

int **sys_open**
(const char *filename, int flag, int *retfd)

- **Description**:
open opens the file, device, or other kernel object named by the pathname filename. The flags argument specifies how to open the file.
- **Return Values:**
On success, open returns 0 while the variable “retval” is the file descriptor of the opened object. On error, returns the error of the function that had the error at that moment.

Open

1. Cycle through the File Table until it finds an empty entry.

```
while (curproc->file_table[i] != NULL){
        if (i == OPEN_MAX - 1){

                return EMFILE;          // EMFILE is an error. It means that a process has too many files opened
        }

        i = i + 1;
}
```

2. Convert the filename string from user level to kernel level.

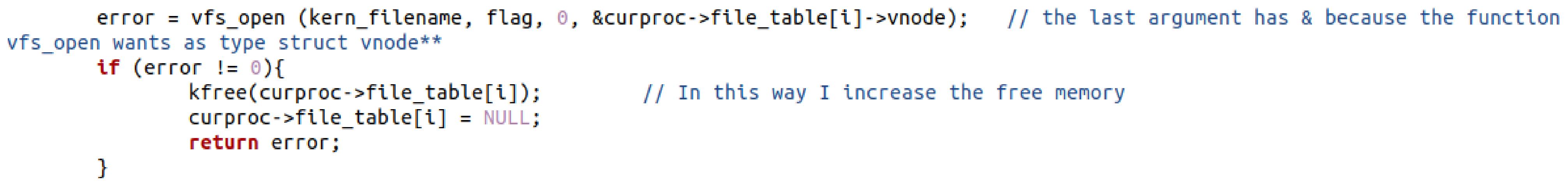
```
        int copyk_filename = copyinstr ((const_userptr_t) filename, kern_filename, PATH_MAX, &actual);      // I am copying a string from
    user level address to kernel level address
        if (copyk_filename != 0){

                return copyk_filename; // It means that copyinstr worked unsuccessfully

        }
```

3. A space in memory is allocated in the System File Table. This location is pointed by the File Table.

```
      curproc->file_table[i] =  (struct open_file*) kmalloc(sizeof(struct open_file)); // In this way I put a connection between the
file_table and the system file table
```

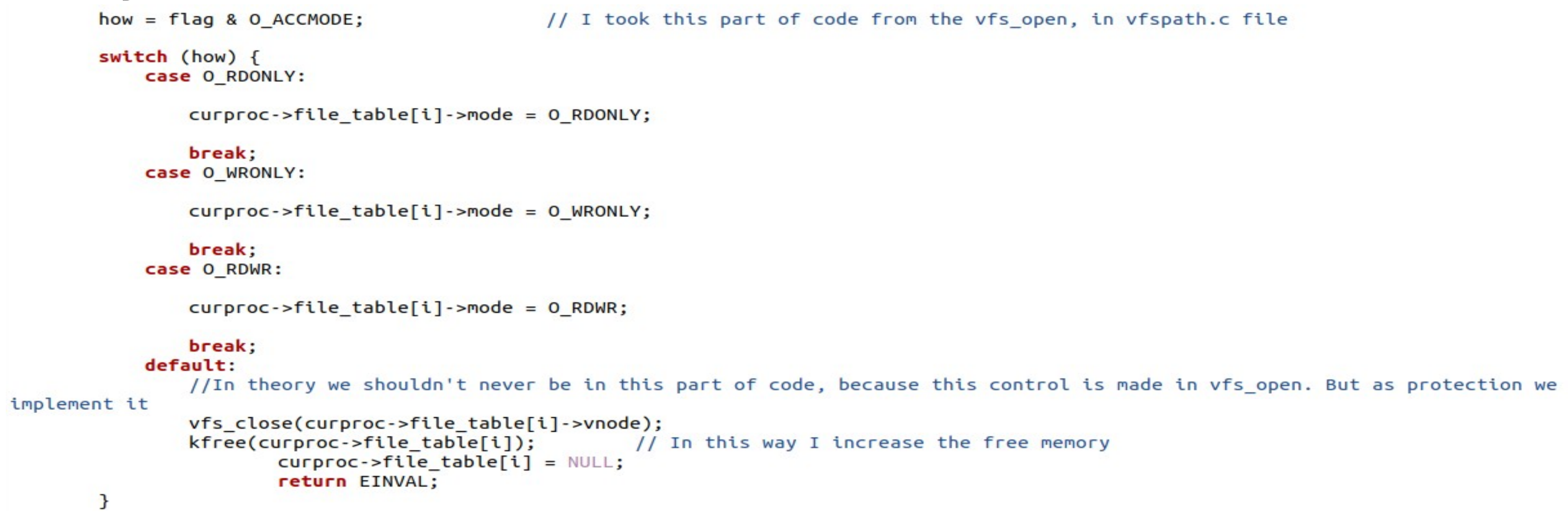
Open

4. The file is opened using the function **vfs_open()**.



```
        error = vfs_open (kern_filename, flag, 0, &curproc->file_table[i]->vnode);    // the last argument has & because the function
vfs_open wants as type struct vnode**
        if (error != 0){
                kfree(curproc->file_table[i]);         // In this way I increase the free memory
                curproc->file_table[i] = NULL;
                return error;
        }
```

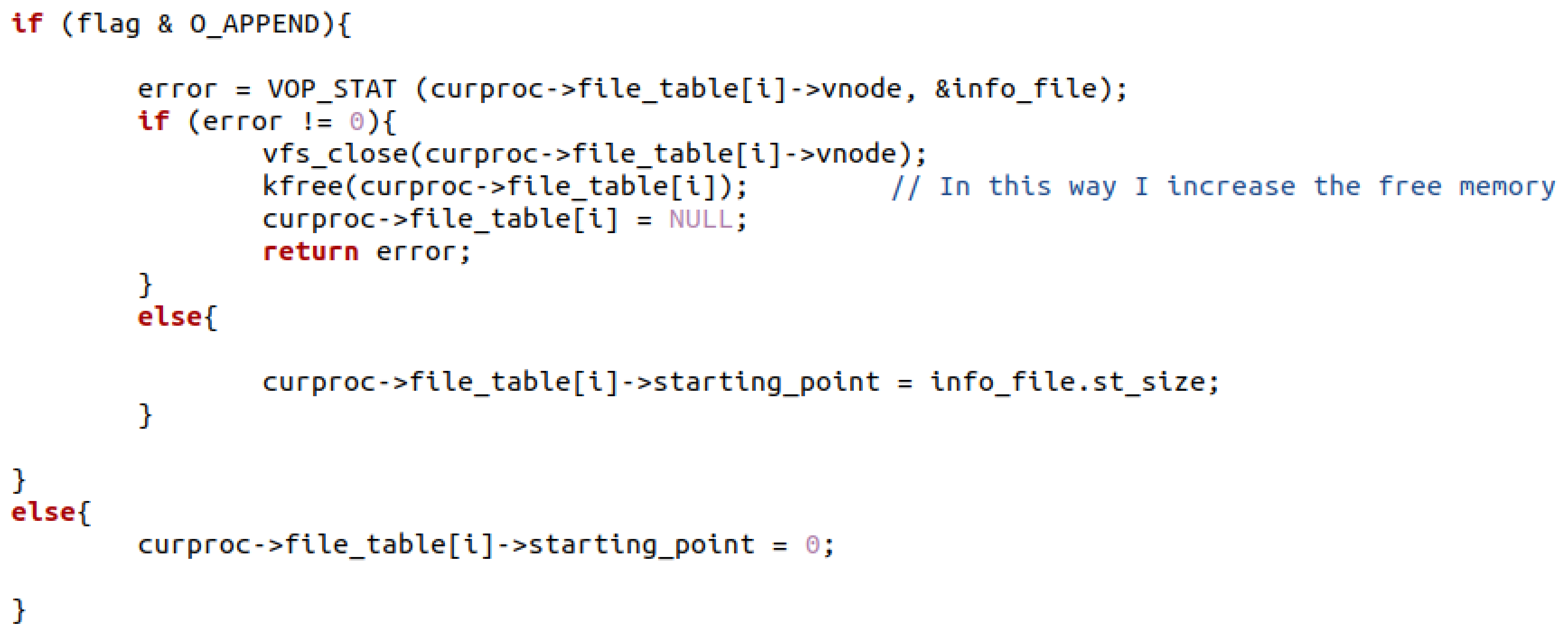
5. In the corresponding row of the System File Table it is saved the mode in which the file is opened (such as Write Only, Read Only, Read and Write).



```
        how = flag & O_ACCMODE;                    // I took this part of code from the vfs_open, in vfspath.c file

        switch (how) {
            case O_RDONLY:

                curproc->file_table[i]->mode = O_RDONLY;

                break;
            case O_WRONLY:

                curproc->file_table[i]->mode = O_WRONLY;

                break;
            case O_RDWR:

                curproc->file_table[i]->mode = O_RDWR;

                break;
            default:
                //In theory we shouldn't never be in this part of code, because this control is made in vfs_open. But as protection we
implement it
                vfs_close(curproc->file_table[i]->vnode);
                kfree(curproc->file_table[i]);         // In this way I increase the free memory
                        curproc->file_table[i] = NULL;
                        return EINVAL;
        }
```

Open

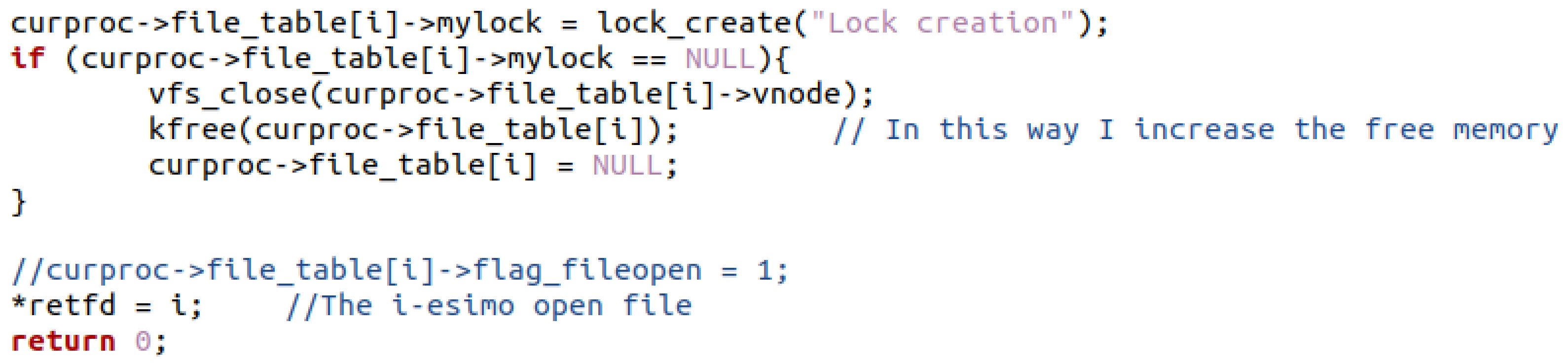
6. If the flag passed as argument is about an operation of append, in the corresponding row of the System File Table it is saved the position of the first free space at the end of the file, otherwise it will be saved a position equal to 0.



```
if (flag & O_APPEND){

        error = VOP_STAT (curproc->file_table[i]->vnode, &info_file);
        if (error != 0){
                vfs_close(curproc->file_table[i]->vnode);
                kfree(curproc->file_table[i]);         // In this way I increase the free memory
                curproc->file_table[i] = NULL;
                return error;
        }
        else{

                curproc->file_table[i]->starting_point = info_file.st_size;
        }

}
else{
        curproc->file_table[i]->starting_point = 0;

}
```

Open

7. A lock is created, and the returned value is set to the index of the File Table position that was found at the beginning, which is the file descriptor.



```
curproc->file_table[i]->mylock = lock_create("Lock creation");
if (curproc->file_table[i]->mylock == NULL){
        vfs_close(curproc->file_table[i]->vnode);
        kfree(curproc->file_table[i]);         // In this way I increase the free memory
        curproc->file_table[i] = NULL;
}

//curproc->file_table[i]->flag_fileopen = 1;
*retfd = i;     //The i-esimo open file
return 0;
```

Read

int **sys_read**
(int fd, userptr_t buf, size_t size, int *retval)

- **Description**:
read reads up to size bytes from the file specified by fd and stores them in the space pointed by buf. The file must be open for reading.

- **Return Values:**
On success, open returns 0 while the variable “retval” is the number of bytes read. On error, returns the error of the function that had the error at that moment.

Read

1. Three preliminary checks are performed: File Descriptor must not be a negative value; the flag associated with the file must not be equal to O_WRONLY; the corresponding row of the File Table must point to an existing region of memory.

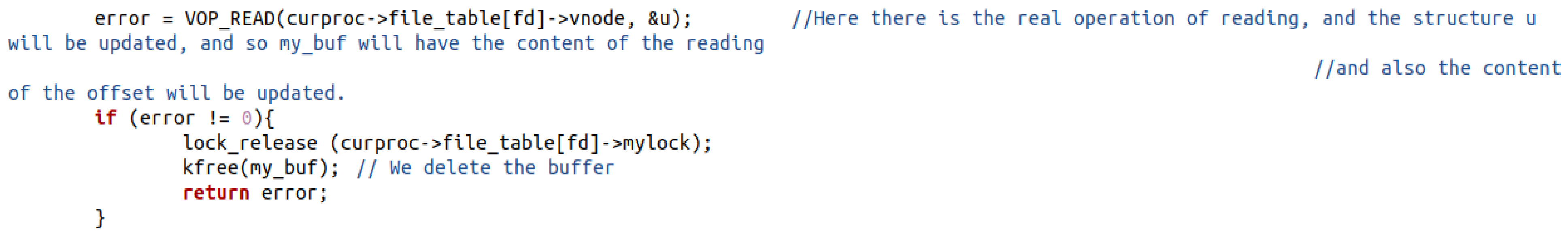
```
if (fd < 0 || curproc->file_table[fd]->mode == O_WRONLY || curproc->file_table[fd] == NULL){

        return EBADF;          //This error means: bad file number
}
```

2. A lock is acquired and this makes sure that the reading operation is performed by only one thread.

```
lock_acquire (curproc->file_table[fd]->mylock);      //In this way the operation of reading will be made only by a thread
```

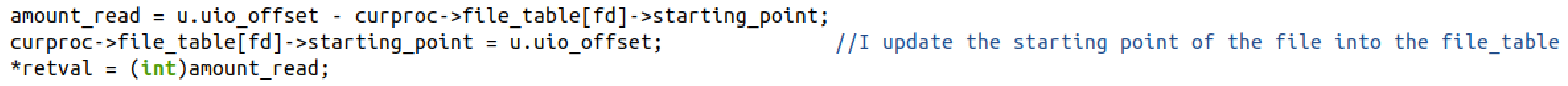
Read

3. The actual reading is performed by the function **VOP_READ()**, and the message read is therefore stored.



```
        error = VOP_READ(curproc->file_table[fd]->vnode, &u);         //Here there is the real operation of reading, and the structure u
will be updated, and so my_buf will have the content of the reading
                                                                                                      //and also the content
of the offset will be updated.
        if (error != 0){
                lock_release (curproc->file_table[fd]->mylock);
                kfree(my_buf); // We delete the buffer
                return error;
        }
```

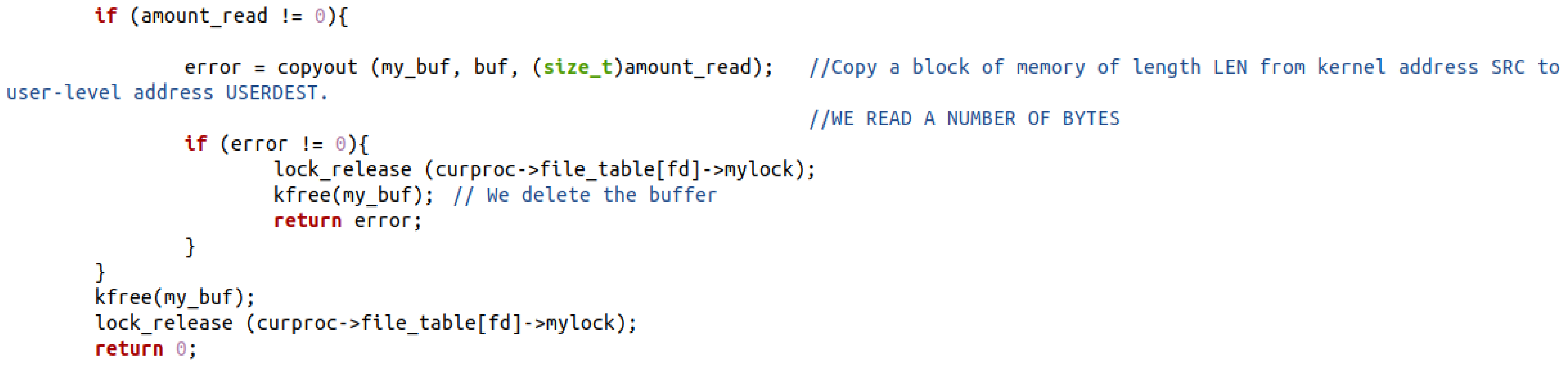
4. The amount of message read is computed and this is the return value. It is saved in the corresponding row of the System File Table the first free space at the end of the file.



```
amount_read = u.uio_offset - curproc->file_table[fd]->starting_point;
curproc->file_table[fd]->starting_point = u.uio_offset;              //I update the starting point of the file into the file_table
*retval = (int)amount_read;
```

Read

5. The message read is copied from kernel address to user-level address and the lock is released.



```
        if (amount_read != 0){

                error = copyout (my_buf, buf, (size_t)amount_read);   //Copy a block of memory of length LEN from kernel address SRC to
user-level address USERDEST.
                                                                      //WE READ A NUMBER OF BYTES
                if (error != 0){
                        lock_release (curproc->file_table[fd]->mylock);
                        kfree(my_buf);  // We delete the buffer
                        return error;
                }
        }
        kfree(my_buf);
        lock_release (curproc->file_table[fd]->mylock);
        return 0;
```

Write

int **sys_write**
(int fd, const userptr_t buf, size_t size, int *retval)

- **Description**:
  write writes up to size bytes to the file specified by fd taking the data from the space pointed by buf.

- **Return Values:**
  On success, open returns 0 while the variable “retval” is the number of bytes written. On error, returns the error of the function that had the error at that moment.

Write

1. Three preliminary checks are performed: File Descriptor must not be a negative value; the flag associated with the file must not be equal to O_RDONLY; the corresponding row of the File Table must point to an existing region of memory.

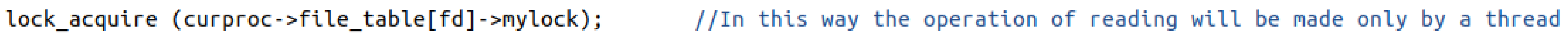
```
if (fd < 0 || curproc->file_table[fd]->mode == O_RDONLY || curproc->file_table[fd] == NULL){

        return EBADF;           //This error means: bad file number
}
```

2. The message passed as argument is copied from user-level address to kernel address.

```
    //It is necessary to translate the message from the user-level address to the kernel-level address, and this is made by copyin

    my_buf = (void*) kmalloc (sizeof(buf)*size);
    return_copyin = copyin(buf, my_buf, size);    //Copy a block of memory of length LEN from user-level address USERSRC to kernel
address DEST
                                                  //WE WRITE A NUMBER OF BYTES
    if (return_copyin != 0){
            kfree(my_buf); // We delete the buffer
            return return_copyin;
    }
```

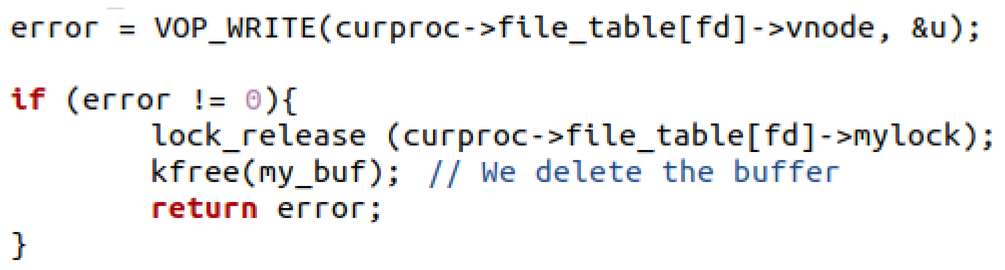
Write

3. A lock is acquired and this makes sure that the writing operation is performed by only one thread.



```
lock_acquire (curproc->file_table[fd]->mylock);      //In this way the operation of reading will be made only by a thread
```

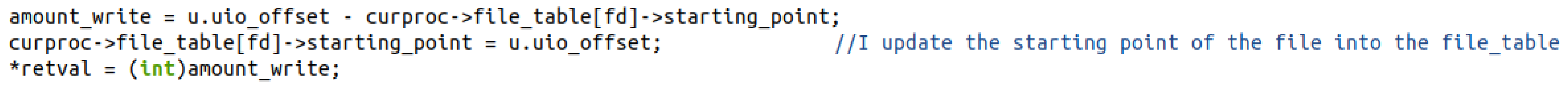
4. The actual writing is performed by the function **VOP_WRITE()**.



```
error = VOP_WRITE(curproc->file_table[fd]->vnode, &u);

if (error != 0){
        lock_release (curproc->file_table[fd]->mylock);
        kfree(my_buf); // We delete the buffer
        return error;
}
```

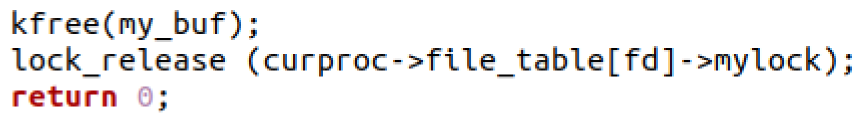
Write

5. The amount of message written is computed and this is the return value. It is saved in the corresponding row of the System File Table the first free space at the end of the file.



```
amount_write = u.uio_offset - curproc->file_table[fd]->starting_point;
curproc->file_table[fd]->starting_point = u.uio_offset;             //I update the starting point of the file into the file_table
*retval = (int)amount_write;
```

6. The lock is released.



```
kfree(my_buf);
lock_release (curproc->file_table[fd]->mylock);
return 0;
```

Close

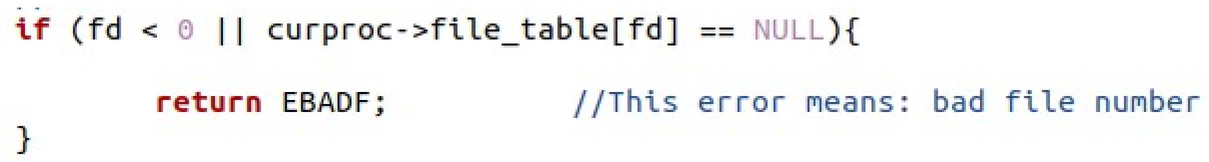
int **sys_close** (int fd)

- **Description**:
  close closes the file handle passed with fd.

- **Return Values:**
  On success, close returns 0.

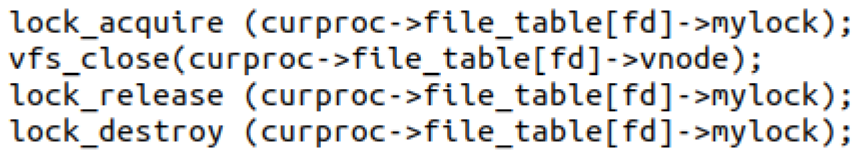
Close

1. Two preliminary checks are performed: File Descriptor must not be a negative value; the corresponding row of the File Table must point to an existing region of memory.



```
if (fd < 0 || curproc->file_table[fd] == NULL){

        return EBADF;           //This error means: bad file number
}
```

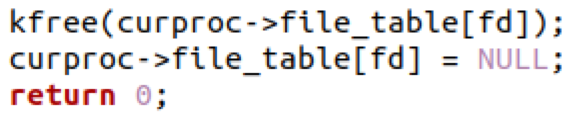
2. A lock is acquired, then the function **VFS_CLOSE()** is performed, then the lock is released and destroyed.



```
lock_acquire (curproc->file_table[fd]->mylock);
vfs_close(curproc->file_table[fd]->vnode);
lock_release (curproc->file_table[fd]->mylock);
lock_destroy (curproc->file_table[fd]->mylock);
```

Close

3. The corresponding row to the file descriptor of the File Table is reset, and the corresponding space into the System File Table is erased.



```
kfree(curproc->file_table[fd]);
curproc->file_table[fd] = NULL;
return 0;
```

Process System Calls

- Exit()
- Fork()
- Execv()
- Getpid()
- Waitpid()

Exit

int **sys_exit**(int exitcode)

- **Description**:

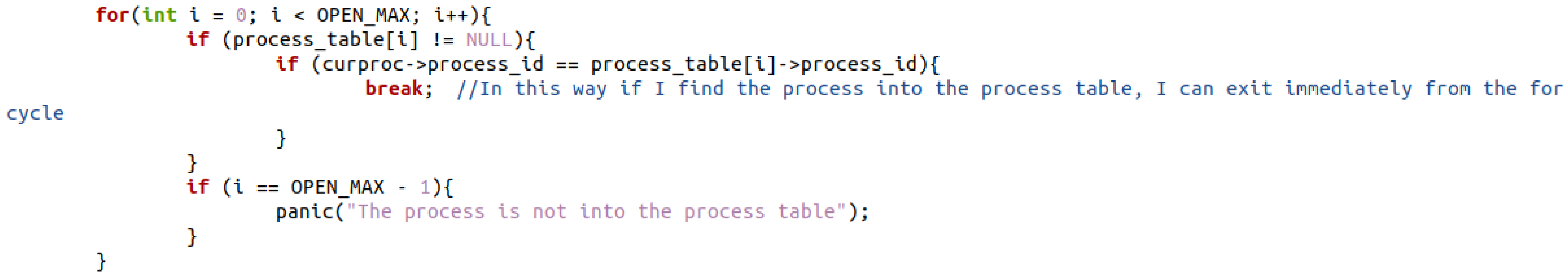
  Cause the current process to exit.

- **Return Values**:

  exit does not return.

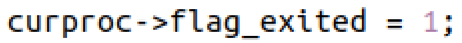
Exit

1. A first check is made to verify that the current process, to which the exit is called, is already present into the process table. Otherwise, an error is generated.



```
	for(int i = 0; i < OPEN_MAX; i++){
		if (process_table[i] != NULL){
			if (curproc->process_id == process_table[i]->process_id){
				break;  //In this way if I find the process into the process table, I can exit immediately from the for
cycle
			}
		}
		if (i == OPEN_MAX - 1){
			panic("The process is not into the process table");
		}
	}
```

2. A flag called “flag_exited” into the process is set equal to 1 to show to other processes that is terminated.



```
curproc->flag_exited = 1;
```

Exit

3. The number, that is put into the exit by the user, is converted in an exit code on 16 bits. This exit code is saved into the process.

```
curproc->exit_code = _MKWAIT_EXIT(exitcode*64);
```

4. Before the process actually ends, its parent process must be waked up from its waiting state.

```
cv_signal(curproc->mycv, curproc->mylock);
```

5. The process ends with the use of the function thread_exit().

```
thread_exit();
```

Fork

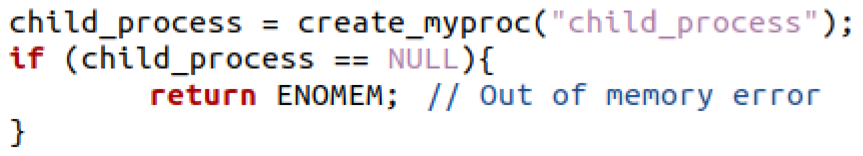
int **sys_fork**
(struct trapframe *tf_parent, pid_t *child_pid)

- **Description**:
  fork duplicates the currently running process. The two copies are identical, except that one (the "new" one, or "child"), has a new, unique process id, and in the other (the "parent") the process id is unchanged.

- **Return Values**:
  On success, fork returns twice, once in the parent process and once in the child process. In the child process, 0 is returned. In the parent process, the process id of the new child process is returned. On error, returns the error of the function that had the error at that moment.

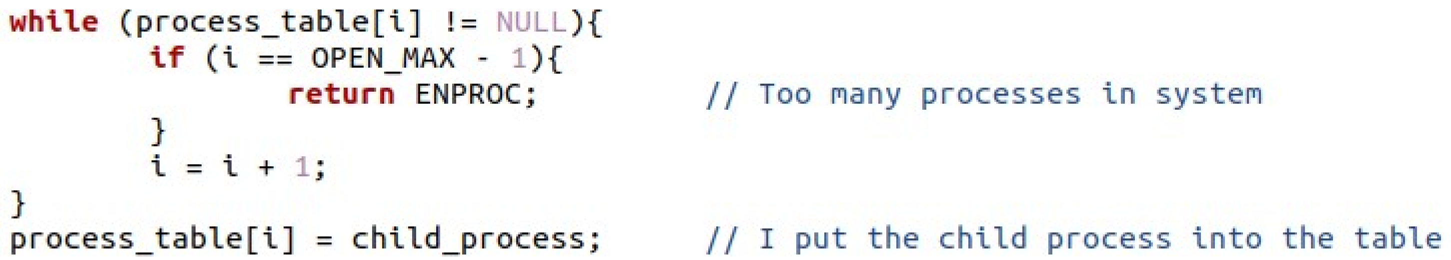
Fork

1. Create a new process that is the child process.



```
child_process = create_myproc("child_process");
if (child_process == NULL){
        return ENOMEM;  // Out of memory error
}
```

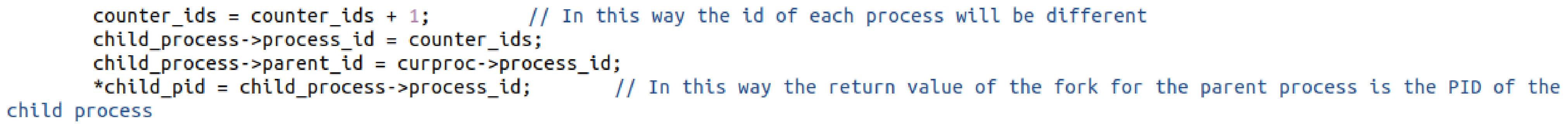
2. Connect the child process to the first free position of the Process Table.



```
while (process_table[i] != NULL){
        if (i == OPEN_MAX - 1){
                return ENPROC;          // Too many processes in system
        }
        i = i + 1;
}
process_table[i] = child_process;       // I put the child process into the table
```

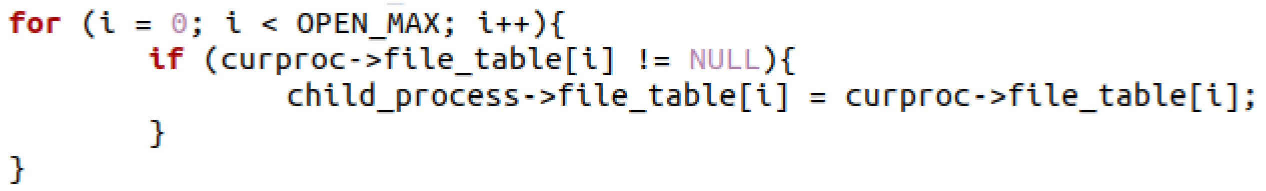
Fork

3. Give the child process a new ID, that will be the return value of the system call for the parent process. Moreover it is updated its parent ID to that of the current process.



```
        counter_ids = counter_ids + 1;          // In this way the id of each process will be different
        child_process->process_id = counter_ids;
        child_process->parent_id = curproc->process_id;
        *child_pid = child_process->process_id;         // In this way the return value of the fork for the parent process is the PID of the
child process
```

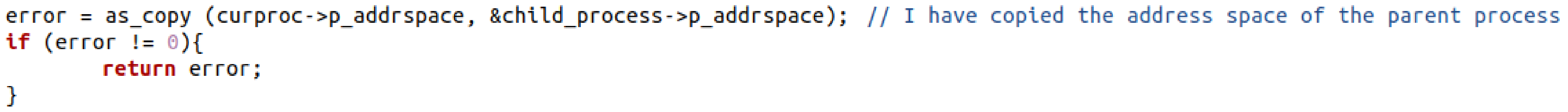
4. Copy into each row of the File Table of the child process the content of the corresponding row of the parent process.



```
for (i = 0; i < OPEN_MAX; i++){
        if (curproc->file_table[i] != NULL){
                child_process->file_table[i] = curproc->file_table[i];
        }
}
```

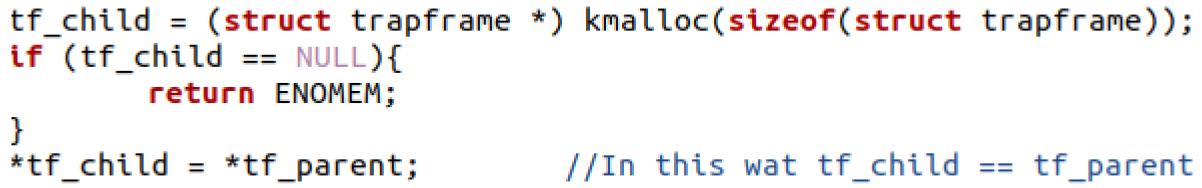
Fork

5. Copy into the address space of the child process the content of the address space of the parent process.



```
error = as_copy (curproc->p_addrspace, &child_process->p_addrspace);  // I have copied the address space of the parent process
if (error != 0){
        return error;
}
```

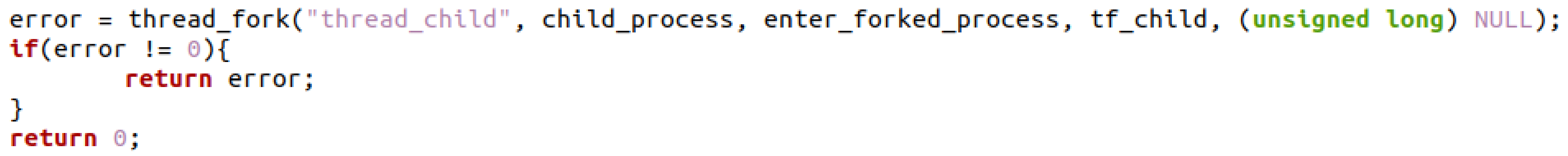
6. Copy into the registers of the trapframe of the child process the content of the corresponding registers of the trapframe of the parent process.



```
tf_child = (struct trapframe *) kmalloc(sizeof(struct trapframe));
if (tf_child == NULL){
        return ENOMEM;
}
*tf_child = *tf_parent;        //In this wat tf_child == tf_parent
```

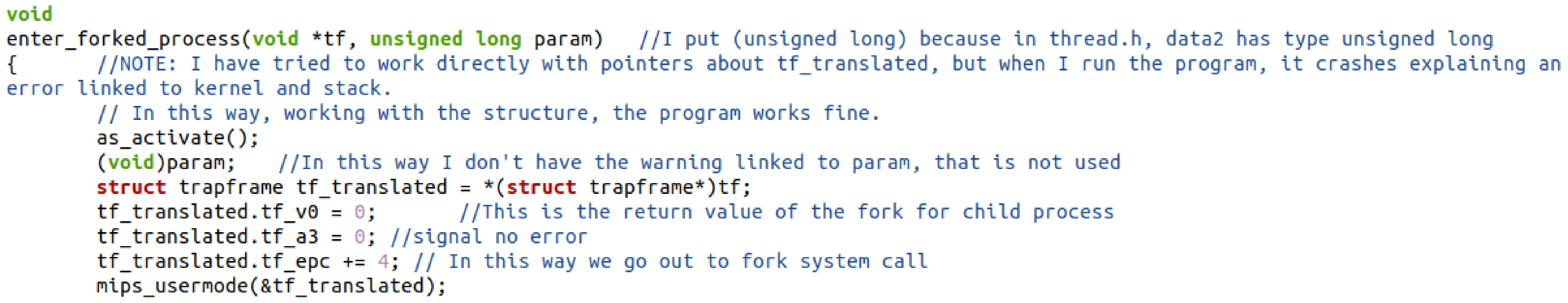
Fork

7. Set appropriately the main registers of the child process trapframe and assign it a new thread, consisting of a copy of the parent thread.



```
error = thread_fork("thread_child", child_process, enter_forked_process, tf_child, (unsigned long) NULL);
if(error != 0){
        return error;
}
return 0;
```

enter_forked_process():



```
void
enter_forked_process(void *tf, unsigned long param)   //I put (unsigned long) because in thread.h, data2 has type unsigned long
{       //NOTE: I have tried to work directly with pointers about tf_translated, but when I run the program, it crashes explaining an
error linked to kernel and stack.
        // In this way, working with the structure, the program works fine.
        as_activate();
        (void)param;    //In this way I don't have the warning linked to param, that is not used
        struct trapframe tf_translated = *(struct trapframe*)tf;
        tf_translated.tf_v0 = 0;        //This is the return value of the fork for child process
        tf_translated.tf_a3 = 0; //signal no error
        tf_translated.tf_epc += 4; // In this way we go out to fork system call
        mips_usermode(&tf_translated);
```

Execv

int **sys_execv**
(const char* program, char**args)

- **Description**:
  execv replaces the currently executing program with a newly loaded program image.
- **Return Values**:
  On success, execv does not return; instead, the new program begins executing. On failure, execv returns the error of the function that had the error at that moment.

Execv

1. Open the executable file.

```
result = vfs_open(progname, O_RDONLY, 0, &v);
if (result != 0) {
        return result;
}
```

2. Create a new address space, and therefore switching to it and activating it.

```
/* Create a new address space. */
as = as_create();
if (as == NULL) {
        vfs_close(v);
        return ENOMEM;
}

/* Switch to it and activate it. */
proc_setas(as);
as_activate();
```

Execv

3. Load the executable file.

```
/* Load the executable. */
result = load_elf(v, &entrypoint);
if (result) {
        /* p_addrspace will go away when curproc is destroyed */
        vfs_close(v);
        return result;
}
```

4. Close the executable file.

```
/* Done with the file now. */
vfs_close(v);
```

5. Define the user stack in the address space.

```
/* Define the user stack in the address space */
result = as_define_stack(as, &stackptr);
if (result) {
        /* p_addrspace will go away when curproc is destroyed */
        return result;
}
```

Execv

6. Copy the arguments into the user stack.

```
//Now I want to copy the arguments from the kernel_buffer to the user stack
stackptr = stackptr - actual_position;
error = copyout(kernel_buffer, (userptr_t) stackptr, actual_position);
```

7. Perform the necessary initialization of the trapframe, paying attention to initialize correctly the stack pointer.

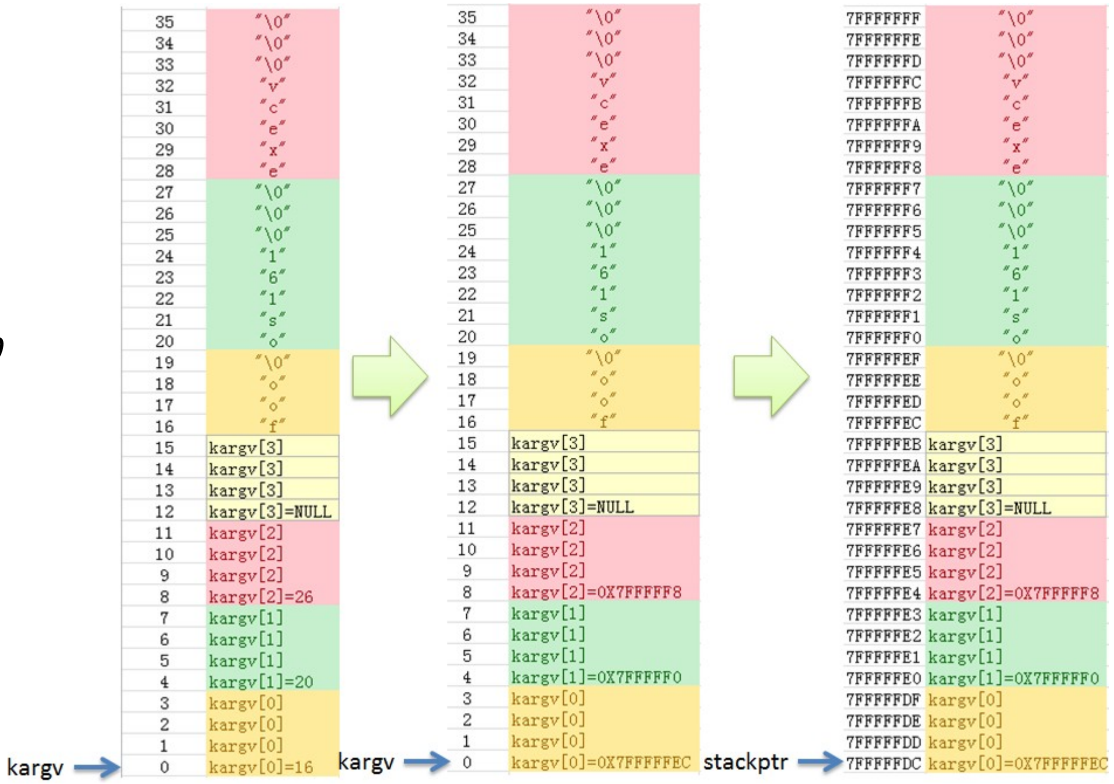
```
stackptr = stackptr - 4;
char* last = NULL;
error = copyout(&last, (userptr_t) stackptr, sizeof(last));
if (error != 0){
        return error;
}

for (j  = i - 1; j >= 0; j--){
        stackptr = stackptr - 4;
        value = USERSTACK - actual_position + starting_values[j];
        last = (char*)value;
        error = copyout(&last, (userptr_t) stackptr, sizeof(last));
        if (error != 0){
                return error;
        }
}
first_argv_offset = (userptr_t) stackptr;
stackptr = stackptr - 4;
```

Execv

8. Enter user mode

```
/* Warp to user mode. */
enter_new_process(i, first_argv_offset,
                  NULL /*userspace addr of environment*/,
                  stackptr, entrypoint);

/* enter_new_process does not return. */
panic("enter_new_process returned\n");
return EINVAL;
```

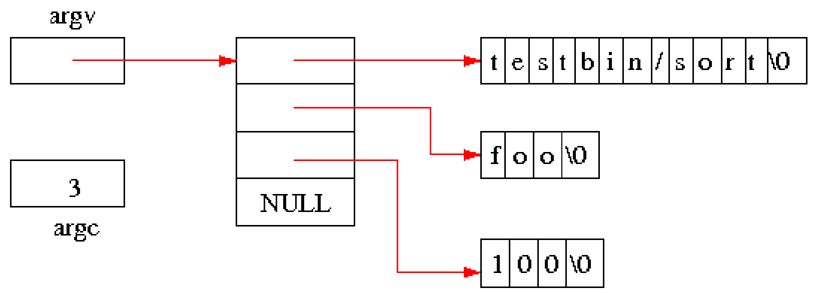
Example: arguments that are copied from kernel buffer to stack user



kargv
kargv
stackptr

35	"\0"	35	"\0"	7FFFFFFF	"\0"
34	"\0"	34	"\0"	7FFFFFFE	"\0"
33	"\0"	33	"\0"	7FFFFFFD	"\0"
32	"v"	32	"v"	7FFFFFFC	"v"
31	"c"	31	"c"	7FFFFFFB	"c"
30	"e"	30	"e"	7FFFFFFA	"e"
29	"x"	29	"x"	7FFFFFF9	"x"
28	"e"	28	"e"	7FFFFFF8	"e"
27	"\0"	27	"\0"	7FFFFFF7	"\0"
26	"\0"	26	"\0"	7FFFFFF6	"\0"
25	"\0"	25	"\0"	7FFFFFF5	"\0"
24	"1"	24	"1"	7FFFFFF4	"1"
23	"6"	23	"6"	7FFFFFF3	"6"
22	"1"	22	"1"	7FFFFFF2	"1"
21	"s"	21	"s"	7FFFFFF1	"s"
20	"o"	20	"o"	7FFFFFF0	"o"
19	"\0"	19	"\0"	7FFFFFEF	"\0"
18	"o"	18	"o"	7FFFFFEE	"o"
17	"o"	17	"o"	7FFFFFED	"o"
16	"f"	16	"f"	7FFFFFEC	"f"
15	kargv[3]	15	kargv[3]	7FFFFFEB	kargv[3]
14	kargv[3]	14	kargv[3]	7FFFFFEA	kargv[3]
13	kargv[3]	13	kargv[3]	7FFFFFE9	kargv[3]
12	kargv[3]=NULL	12	kargv[3]=NULL	7FFFFFE8	kargv[3]=NULL
11	kargv[2]	11	kargv[2]	7FFFFFE7	kargv[2]
10	kargv[2]	10	kargv[2]	7FFFFFE6	kargv[2]
9	kargv[2]	9	kargv[2]	7FFFFFE5	kargv[2]
8	kargv[2]=26	8	kargv[2]=0X7FFFFFF8	7FFFFFE4	kargv[2]=0X7FFFFFF8
7	kargv[1]	7	kargv[1]	7FFFFFE3	kargv[1]
6	kargv[1]	6	kargv[1]	7FFFFFE2	kargv[1]
5	kargv[1]	5	kargv[1]	7FFFFFE1	kargv[1]
4	kargv[1]=20	4	kargv[1]=0X7FFFFFF0	7FFFFFE0	kargv[1]=0X7FFFFFF0
3	kargv[0]	3	kargv[0]	7FFFFFDF	kargv[0]
2	kargv[0]	2	kargv[0]	7FFFFFDE	kargv[0]
1	kargv[0]	1	kargv[0]	7FFFFFDD	kargv[0]
0	kargv[0]=16	0	kargv[0]=0X7FFFFFEC	7FFFFFDC	kargv[0]=0X7FFFFFEC

If the tail program is invoked from the kernel command line like this:
OS/161 kernel [? for menu]: p testbin/sort foo 100
Then the argv and argc variables should be set up as illustrated in the following illustration:



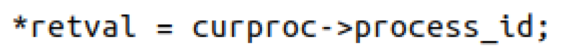
Getpid

int **sys_getpid** (pid_t *retval)

- **Description**:
  getpid returns the process id of the current process.
- **Errors**:
  getpid does not fail.

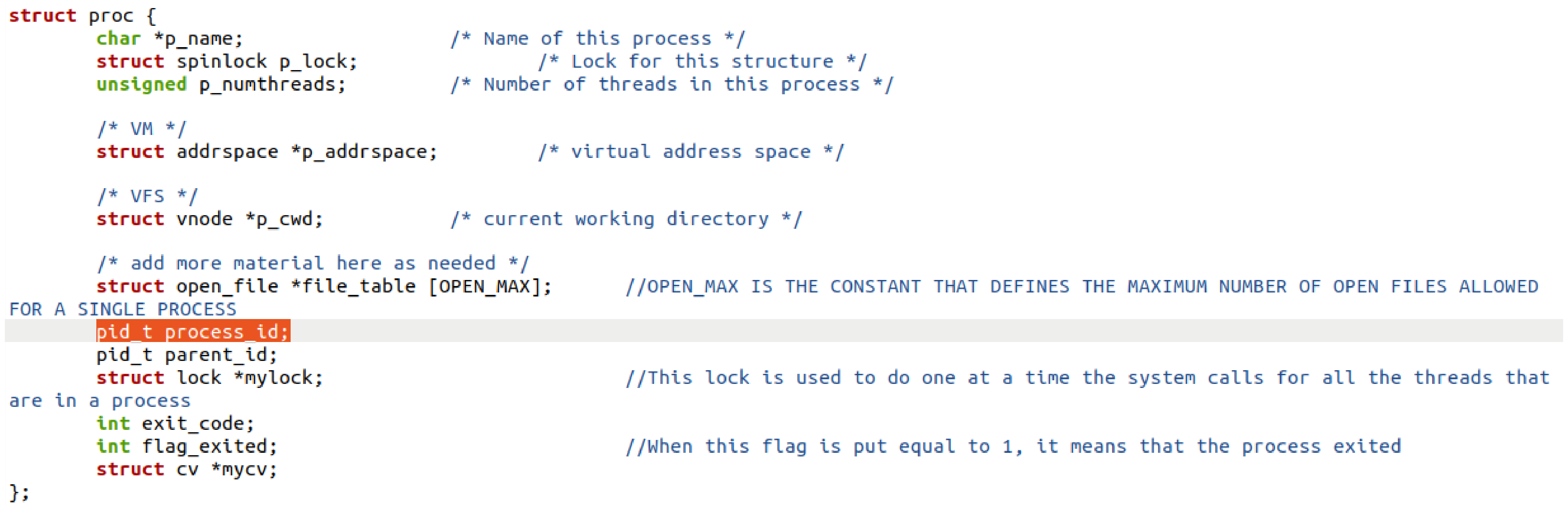
Getpid

Returns the process id of the current process



```
*retval = curproc->process_id;
```

process_id is a variable added in the proc struct:



```
struct proc {
        char *p_name;                   /* Name of this process */
        struct spinlock p_lock;                 /* Lock for this structure */
        unsigned p_numthreads;          /* Number of threads in this process */

        /* VM */
        struct addrspace *p_addrspace;          /* virtual address space */

        /* VFS */
        struct vnode *p_cwd;            /* current working directory */

        /* add more material here as needed */
        struct open_file *file_table [OPEN_MAX];      //OPEN_MAX IS THE CONSTANT THAT DEFINES THE MAXIMUM NUMBER OF OPEN FILES ALLOWED
FOR A SINGLE PROCESS
        pid_t process_id;
        pid_t parent_id;
        struct lock *mylock;                          //This lock is used to do one at a time the system calls for all the threads that
are in a process
        int exit_code;
        int flag_exited;                              //When this flag is put equal to 1, it means that the process exited
        struct cv *mycv;
};
```

Waitpid

int **sys_waitpid**
(pid_t waiting_for_pid, int *status, int options, pid_t *retval)

- **Description**:
  wait for a process, specified by its id, to end.
- **Return Values**:
  waitpid returns the pid value of the child process.

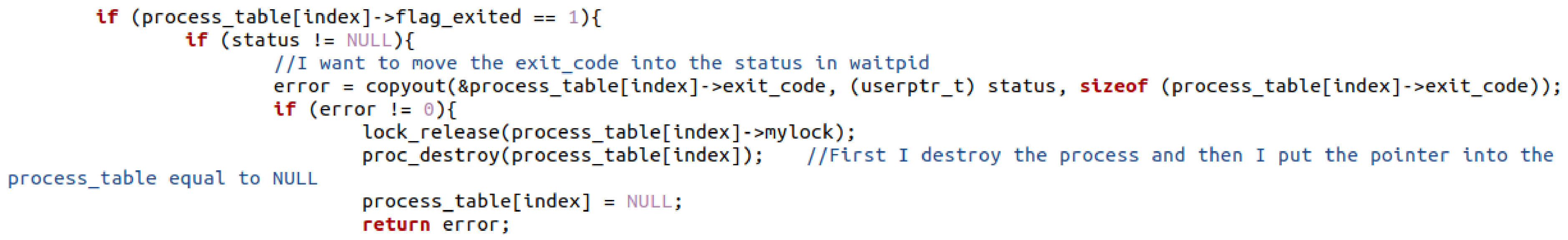
Waitpid

1. It is necessary to understand if the PID passed as argument refers to a really existing process. Therefore that PID has been searched in all the processes belonging to the process table. If successful, the position of the process within the process table is obtained, otherwise an error is generated.

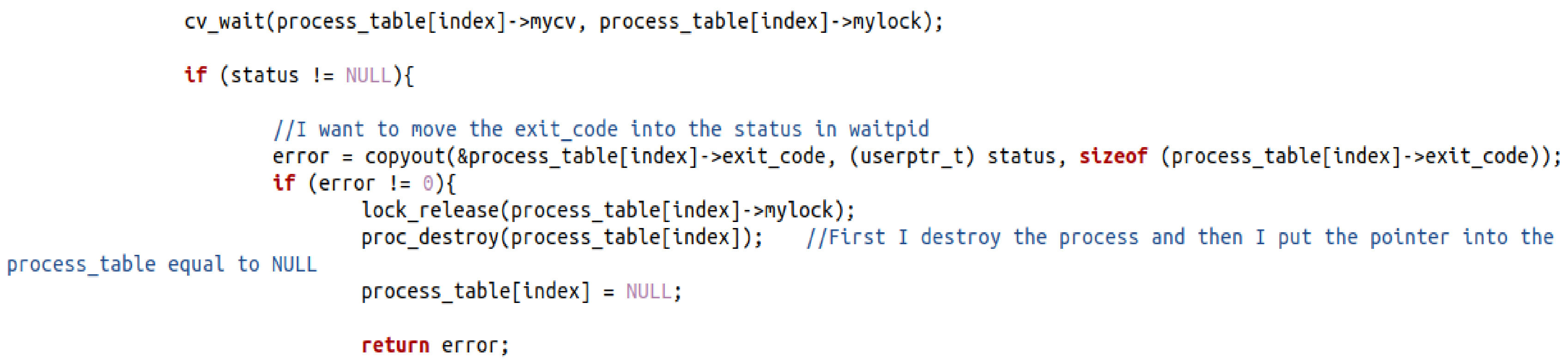
```
	//First control on the waiting_for_pid
	if (waiting_for_pid == curproc->process_id){
		return ECHILD;  //No child processes
	}
	if (options != 0){
		return EINVAL;  //Invalid argument   I have never used a value of options different than 0
	}
	//Now I have to verify that the child process exists, and it is into the process table
	for (index = 0; index < OPEN_MAX; index ++){
		if (process_table[index] != NULL){
			if (waiting_for_pid == process_table[index]->process_id){
				break;  //In this way I go out from the for cycle and I can continue the function with the correct value of
index
			}
			if (index == OPEN_MAX - 1){
				return ESRCH;   //No such process
			}
		}
	}
	//I want to be sure that the waiting_for_pid is the child of the curproc
	if (process_table[index]->process_id != 2){   //This modification is made for the menù
		if (curproc->process_id != process_table[index]->parent_id){
			return ECHILD;  //No child processes
		}
	}
```

Waitpid

2. If **flag_exited** of this process has already a value equal to 1, its exit code is moved into the status; Otherwise, the parent process enters in a sleep state, and it will be awakened by the child process before it ends. At this point its exit code is moved into the status.



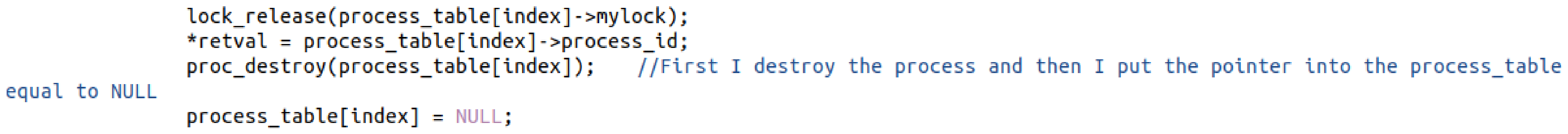
```
        if (process_table[index]->flag_exited == 1){
                if (status != NULL){
                        //I want to move the exit_code into the status in waitpid
                        error = copyout(&process_table[index]->exit_code, (userptr_t) status, sizeof (process_table[index]->exit_code));
                        if (error != 0){
                                lock_release(process_table[index]->mylock);
                                proc_destroy(process_table[index]);    //First I destroy the process and then I put the pointer into the
process_table equal to NULL
                                process_table[index] = NULL;
                                return error;
```



```
            cv_wait(process_table[index]->mycv, process_table[index]->mylock);

            if (status != NULL){

                    //I want to move the exit_code into the status in waitpid
                    error = copyout(&process_table[index]->exit_code, (userptr_t) status, sizeof (process_table[index]->exit_code));
                    if (error != 0){
                            lock_release(process_table[index]->mylock);
                            proc_destroy(process_table[index]);    //First I destroy the process and then I put the pointer into the
process_table equal to NULL

                            process_table[index] = NULL;

                            return error;
```

Waitpid

3. The return value of this system call will be equal to the ID of the child process, and finally the child process will be destroyed and the pointer into the process table will be equal to NULL. In this way the process table can host a new process.



```
                lock_release(process_table[index]->mylock);
                *retval = process_table[index]->process_id;
                proc_destroy(process_table[index]);    //First I destroy the process and then I put the pointer into the process_table
equal to NULL
                process_table[index] = NULL;
```

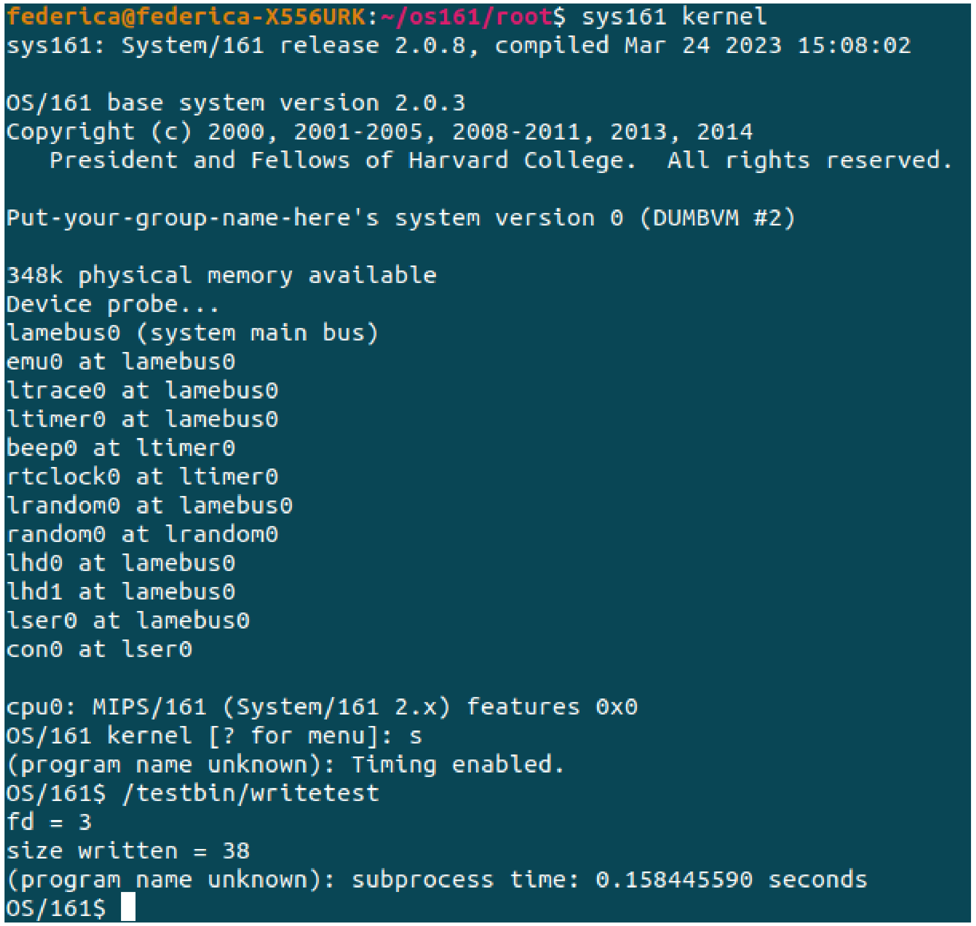
Results

In the following slides will be shown the test of each system call and the corresponding results.

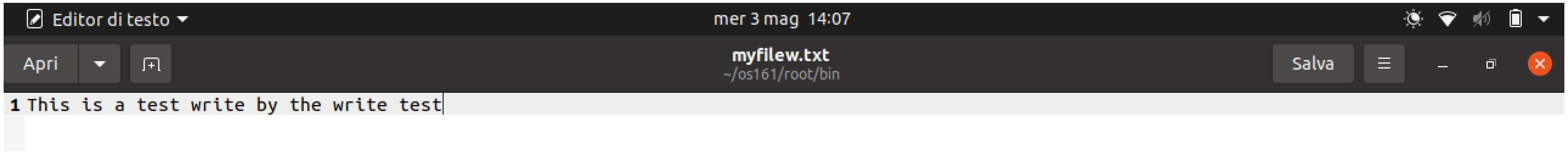
- The **Open** and **Close** system calls are tested in the **Write test** and **Read test**.
- The **Exit**, **Fork** and **Getpid** system calls are tested in the **Waitpid test**.

Write Test

```c
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <err.h>

int main(){

        //WRITING INTO A FILE
        int fd;
        int sz;
        char message [60];
        int len;
        fd = open ("/bin/myfilew.txt", O_WRONLY | O_CREAT);
        printf("fd = %d\n", fd);
        strcpy(message, "This is a test write by the write test");
        len = strlen(message);
        sz = write(fd, message, len);
        close(fd);
        printf("size written = %d\n", sz);

        return 0;

}
```



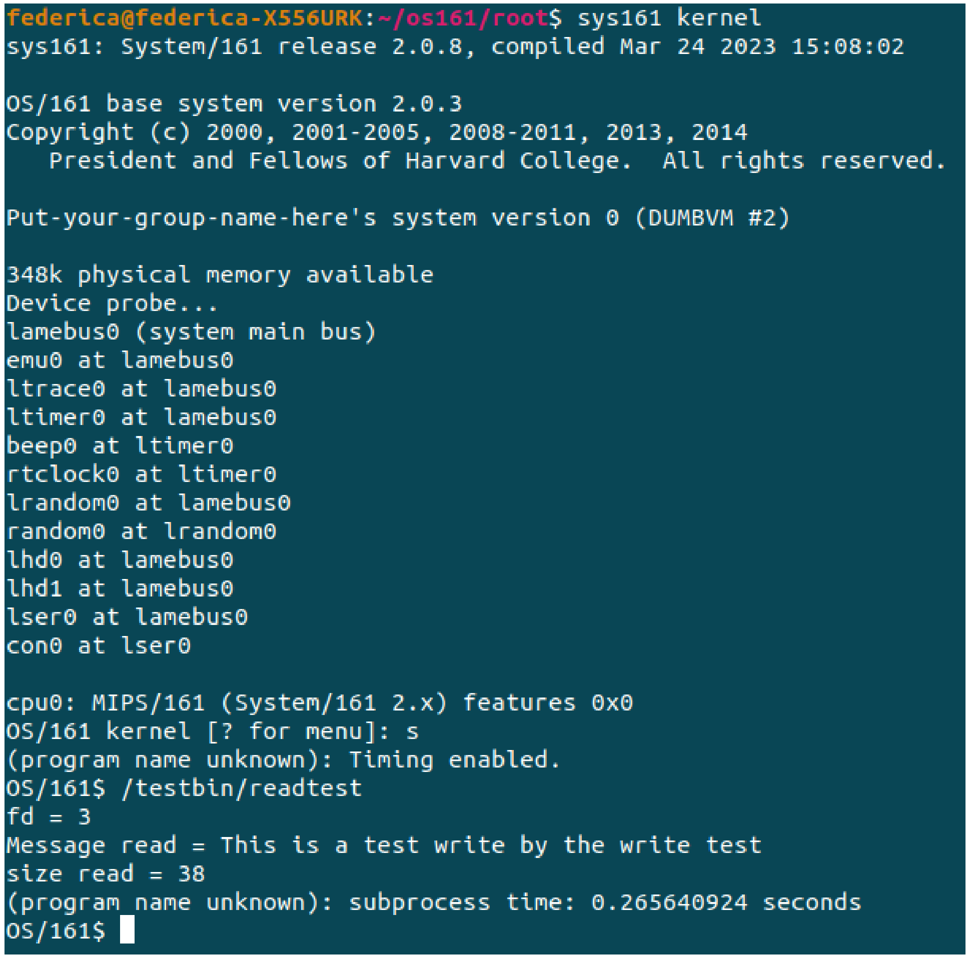
```
federica@federica-X556URK:~/os161/root$ sys161 kernel
sys161: System/161 release 2.0.8, compiled Mar 24 2023 15:08:02

OS/161 base system version 2.0.3
Copyright (c) 2000, 2001-2005, 2008-2011, 2013, 2014
   President and Fellows of Harvard College.  All rights reserved.

Put-your-group-name-here's system version 0 (DUMBVM #2)

348k physical memory available
Device probe...
lamebus0 (system main bus)
emu0 at lamebus0
ltrace0 at lamebus0
ltimer0 at lamebus0
beep0 at ltimer0
rtclock0 at ltimer0
lrandom0 at lamebus0
random0 at lrandom0
lhd0 at lamebus0
lhd1 at lamebus0
lser0 at lamebus0
con0 at lser0

cpu0: MIPS/161 (System/161 2.x) features 0x0
OS/161 kernel [? for menu]: s
(program name unknown): Timing enabled.
OS/161$ /testbin/writetest
fd = 3
size written = 38
(program name unknown): subprocess time: 0.158445590 seconds
OS/161$
```



```
This is a test write by the write test
```

Read Test

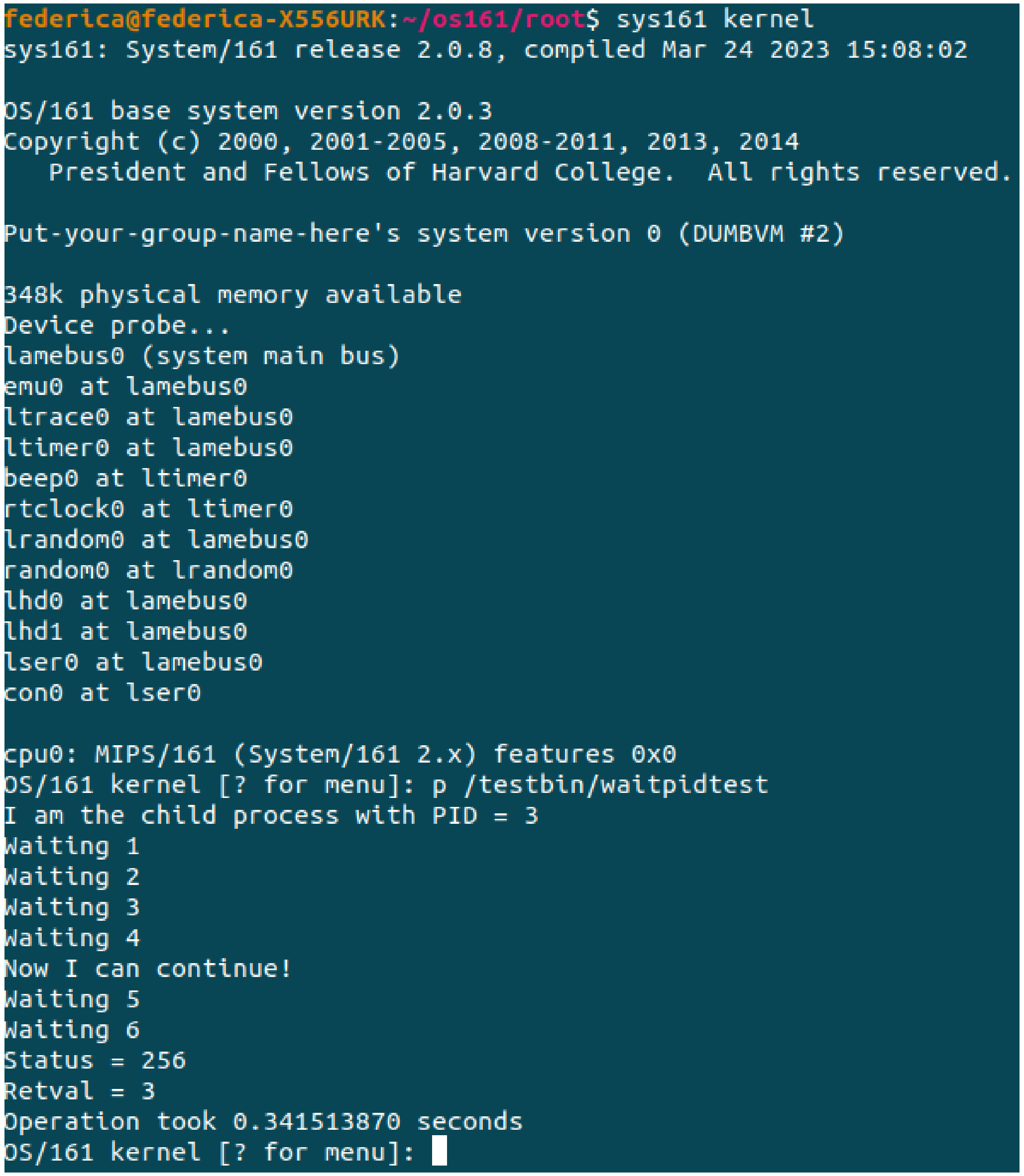
```c
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <err.h>

int main(){


        //READING FROM A FILE
        int fd;
        int sz;
        char message [60];
        fd = open ("/bin/myfilew.txt", O_RDONLY | O_CREAT);
        printf("fd = %d\n", fd);
        sz = read(fd, message, 60);
        printf("Message read = %s\n", message);
        close(fd);
        printf("size read = %d\n", sz);

        return 0;

}
```



```
federica@federica-X556URK:~/os161/root$ sys161 kernel
sys161: System/161 release 2.0.8, compiled Mar 24 2023 15:08:02

OS/161 base system version 2.0.3
Copyright (c) 2000, 2001-2005, 2008-2011, 2013, 2014
   President and Fellows of Harvard College.  All rights reserved.

Put-your-group-name-here's system version 0 (DUMBVM #2)

348k physical memory available
Device probe...
lamebus0 (system main bus)
emu0 at lamebus0
ltrace0 at lamebus0
ltimer0 at lamebus0
beep0 at ltimer0
rtclock0 at ltimer0
lrandom0 at lamebus0
random0 at lrandom0
lhd0 at lamebus0
lhd1 at lamebus0
lser0 at lamebus0
con0 at lser0

cpu0: MIPS/161 (System/161 2.x) features 0x0
OS/161 kernel [? for menu]: s
(program name unknown): Timing enabled.
OS/161$ /testbin/readtest
fd = 3
Message read = This is a test write by the write test
size read = 38
(program name unknown): subprocess time: 0.265640924 seconds
OS/161$
```

Waitpid Test

```c
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <err.h>

int main(){
        pid_t pid;
        int retval;
        int status;
        pid = fork();
        if (pid == 0){
                //Child process
                printf ("I am the child process with PID =
%d\n", getpid());
                printf("Waiting 1\n");
                printf("Waiting 2\n");
                printf("Waiting 3\n");
                printf("Waiting 4\n");
                exit(1);
        }
        else{
                //Parent process
                retval = waitpid(pid, &status, 0);
                printf("Now I can continue!\n");
                printf("Waiting 5\n");
                printf("Waiting 6\n");
                printf("Status = %d\n", status);
                printf("Retval = %d\n", retval);
        }
        return 0;
}
```



```
federica@federica-X556URK:~/os161/root$ sys161 kernel
sys161: System/161 release 2.0.8, compiled Mar 24 2023 15:08:02

OS/161 base system version 2.0.3
Copyright (c) 2000, 2001-2005, 2008-2011, 2013, 2014
   President and Fellows of Harvard College.  All rights reserved.

Put-your-group-name-here's system version 0 (DUMBVM #2)

348k physical memory available
Device probe...
lamebus0 (system main bus)
emu0 at lamebus0
ltrace0 at lamebus0
ltimer0 at lamebus0
beep0 at ltimer0
rtclock0 at ltimer0
lrandom0 at lamebus0
random0 at lrandom0
lhd0 at lamebus0
lhd1 at lamebus0
lser0 at lamebus0
con0 at lser0

cpu0: MIPS/161 (System/161 2.x) features 0x0
OS/161 kernel [? for menu]: p /testbin/waitpidtest
I am the child process with PID = 3
Waiting 1
Waiting 2
Waiting 3
Waiting 4
Now I can continue!
Waiting 5
Waiting 6
Status = 256
Retval = 3
Operation took 0.341513870 seconds
OS/161 kernel [? for menu]: 
```

Other Tests

Using the shell, it is also possible to test individual system calls, with:

- myforktest
- exittest
- getpidtest