Candidate: Davide Arcolini
Candidate: Giulia Bianchi
Academic Year: 2022/2023

System and Device Programming

Project c2: Shell (SHELL)

Teacher:	Gianpiero Cabodi
Course:	System and Device Programming
Credits:	10 CFU
Academic Year:	2022/2023

Candidate:	Davide Arcolini
Contact:	davide.arcolini@studenti.polito.it
Candidate:	Giulia Bianchi
Contact:	s294547@studenti.polito.it

Date of last revision: December 20, 2022

Contents

Introduction

OS/161 is a teaching operating system, that is, a simplified system used for teaching undergraduate operating systems classes [1].

The purpose of this project is to support running multiple processes at once from actual compiled programs stored on disk. The programs will be loaded from the disk to the userland, where they will run under the kernel control with the `/bin/sh` process underneath. In order to make the kernel work within the shell environment, the most important and useful function we had to implement was `execv()`. Clearly, this function itself is not enough to make everything completely functional. Therefore, in order to implement a system in the intended goal, we went through the implementation of several features:

- **Virtual Memory management**: we redefined in a simple and non-optimal (however, better than default) way the management of pages in the RAM space. We used a very simple version of a **bitmap** in which the allocation and de-allocation of new pages is managed based on requests of free pages (and, eventually, releasing of those occupied). All the details are presented in the section: A simple bitmap.
- **File management**: here we had to implement the system file table and process file table to keep track of opened and closed files. Moreover, we needed a mechanism to perform reading and writing operations on, among various files, `stdin` and `stdout`; also, when needed, duplicate a file descriptor for another process. All the details are presented in the chapter: Files Management.
- **Directory management**: in order to move through the file system inside the shell environment, we needed to implement a very basic set of operations to be performed, i.e. moving to another directory and getting the current working directory. These two system calls are better analyzed in chapter: Directories Management.
- **Process management**: the core feature of this project is, indeed, the management of the processes running on the operating system. In particular, we needed to:
  - keep track of all process relations (parent-children);
  - manage gracefully the termination of a process (and relative communication with children);
  - forking a process, creating a new address space for the new process (`fork()`);
  - replacing the image of a program with a new one, modifying the already existing address space for the process (`execv()`).

  All the details are presented in the chapter: Processes Management.

Finally, the whole project is built in such a way to gracefully handle all the different error codes, in according to the `testbin/badcall` assertions test, and to return the correct values in case of success. All the details are presented in the final chapter: Results Analysis.

We will present in the following chapters the implementation of the various system calls and data structures used to complete the project. As a reminder, the **kernel** of the operating system can be found in the `kern` directory of this repository. Indeed, all the low level details can be found there, well commented

and presented in readable and well-written fashion. This report has only the purposes of presenting a panoramic view of the kernel with higher levels of details.

Before proceeding, a couple of additional information:

- all the implemented system calls are declared with a `_SHELL` postfix, such as: `sys_open_SHELL()`, `sys_fork_SHELL()` and so on.

1.1 Configuration of the kernel

As required by the assignment, all the code changes are enabled through the conditional `#if OPT_SHELL` statement. `OPT_SHELL` is a global flag defined in the automatically generated file `opt-shell.h` (path: `kern/compile/SHELL/opt-shell.h`), whose configuration is managed in the configuration file `conf.kern` (path: `kern/conf/conf.kern`):

```
defoption shell

optfile shell syscall/syscall_FILE.c
optfile shell syscall/syscall_PROC.c
optfile shell syscall/exec.c
```

The generated file contains indeed the declaration of the flag:

```
/* Automatically generated; do not edit */
#ifndef _OPT_SHELL_H_
#define _OPT_SHELL_H_
#define OPT_SHELL 1
#endif /* _OPT_SHELL_H_ */
```

The flag is therefore used to wrap all the code changes, allowing compilation based on the specific configuration required:

```
#if OPT_SHELL
    /* come code here */
    ...
#endif
```

1.2 System Calls dispatcher

The system call dispatcher (path: `kern/arch/mips/syscall.c`) is a MIPS kernel utility that *dispatch* the received **trapframe** function calls to the desired system call routine. During an exception, a pointer to a specifically crafted trapframe is created and this function is invoked. The parameter mechanism follows the most conventional calling parameters for the system calls management:

- the system call number is passed in the `v0` register. All the registers are retrieved from the trapframe. This number is used by the `sycall(struct trapframe *tf)` function to understand which system call it is necessary to run. The system call numbers can be retrieved from the `syscall.h` header file (path: `kern/include/kern/syscall.h`);
- the first four 32-bit arguments are passed in the four argument registers `a0-a3`;
- 64-bit arguments are passed in aligned pairs of registers (like: `a0-a1` and `a2-a3`);
- eventually, more arguments are passed in the user-side stack, starting at `sp` with byte-offset of `16`;
- the return value is passed back in the `v0` register if it is a 32-bit value, otherwise in the couple `v0-v1`.

As already stated, we wrapped all the system call dispatched in the `OPT_SHELL` flag.

As an example, the `open()` system call is dispatched in the following way:

```
        /* open() SYSTEM CALL */
        case SYS_open:
            err = sys_open_SHELL(
                (userptr_t) tf->tf_a0,
                (int) tf->tf_a1,
                (mode_t) tf->tf_a2,
                &retval_low32
            );
        break;
```

where:

- `SYS_open` is the system call number retrieved from the trapframe, used to initially dispatch the system call.
- `err` is the error value eventually returned from the system call routine.
- `tf->tf_a0`, `tf->tf_a1` and `tf->tf_a2` are the parameters passed to the system call routine.
- `retval_low32` is the expected return value (in case of success) of the open system call (note that this is a 32-bit value, so we just need a single 32-bit variable to handle it).

1.2.1 Entering in a forked process

When a newly created process needs to be started, the operating system triggers the system call dispatcher as if an exception has occurred. We had implemented the routines that manage this kind of operations in the `proc.c` file (path: `kern/proc/proc.c`):

```
void call_enter_forked_process(void *tfv, unsigned long dummy);
```

This routine receives the trapframe crafted by the fork operation and then calls, on its own, the function used to enter in user mode for a newly forked process, whose implementation is the following:

```
void enter_forked_process(struct trapframe *tf) {
    struct trapframe forkedTf = *tf;        // copy trap frame onto kernel stack

    forkedTf.tf_v0 = 0;         // return value is 0
    forkedTf.tf_a3 = 0;         // return with success
    forkedTf.tf_epc += 4;       // return to next instruction

    as_activate();

    mips_usermode(&forkedTf);
}
```

1.3 Error handling

In order to make the whole system the most compliant with the Linux error codes management system, we designed all the system calls to follow the given conventions. In particular, all the routines, in case of failures, return the corresponding error code that can be found in `errno.h` (path: `kern/include/kern/errno.h`). As an example, in the fork routine, if we have run out of virtual memory to host another process, the `ENOMEM` error code will be returned:

```
    /* creating new runnable process */
    struct proc *newproc = proc_create_runprogram(curproc->p_name);
    if (newproc == NULL) {
        return ENOMEM;          // Sufficient virtual memory for the new process was not available.
    }
```

Virtual Memory Management

os161, as it is, has a very basic virtual memory manager that only performs contiguous allocation of real memory, without ever releasing it. As a matter of fact, inside the `kmain()` function, there is a call to the `boot()` function which performs the complete bootstrap of the whole system, without taking into account any form of virtual memory management. The virtual memory related functions can be found at `kern/arch/mips/vm/dumbvm.c`.

In this chapter, we present a very simple and naive mechanism to handle the virtual memory system. While this was not the purpose of the assignment, we believed that this could have helped the management of memory during the rest of the work.

2.1 A simple bitmap

We exploited a simple bitmap system in order to keep track of allocated RAM pages, resolving pages requests and managing releasing of space. The **bitmap** is defined as a couple of arrays:

```
1 static unsigned int *bitmap_freePages = NULL;       /* position of the free pages */
2 static unsigned long *bitmap_sizePages = NULL;      /* size of the free pages */
```

which are allocated in the `vm_bootstrap()` call at start-up, with a computed size of

```
1 number_ram_pages = ((int) ram_getsize()) / PAGE_SIZE;
```

i.e., the total size of the RAM divided by the size of a single page. Pretty straightforward.

The first array, `bitmap_freePages`, stores the position of every allocated pages with a boolean value: the first element of the array corresponds to the first page in the RAM, the second element in the array corresponds to the second page in the RAM, and so on. We used `1` to indicate that the page is free, `0` to indicate that the page has already been occupied.

The second array, `bitmap_sizePages`, stores the occupational size of each pages. Indeed, the first element indicates how many pages have been occupied starting from the first page of the RAM, the second element indicates how many pages have been occupied starting from the second page of the RAM, and so on.

2.2 Allocation and release of pages

The main operations to be performed are allocation and release of pages. When RAM space is requested for a process, the kernel calls the `alloc_kpages(unsigned npages)` routine, which, on its own, will understand if there is the possibility to allocate pages using the bitmap or it is necessary to call `ram_stealmem()` and steal already occupied RAM pages. If the kernel is able to allocate free pages using the bitmap, it will call the `getfreeppages()` routine.

```
static paddr_t getfreeppages(unsigned long npages);
```

This function searches with a linear time execution all the positions in the bitmap. When it finds an available position, it runs another linear time loop to check if the request fits in. While this algorithm is highly inefficient ($\mathcal{O}(n^2)$), it is a first attempt to manage the virtual memory system than can, eventually, be improved. This search is protected by a **spinlock**, in order to prevent multiple processes that concurrently start pages requests, to access the same memory space, ending up in **race conditions**.

On the other hand, the release of the pages is done by `free_kpages(vaddr_t addr)`, which internally computes the physical requested address

```
/* moving from virtual to physical address */
paddr_t paddr = addr - MIPS_KSEG0;
long int start_addr = paddr/PAGE_SIZE;

/* freeing the pages */
freeppages(paddr, bitmap_sizePages[start_addr]);
```

and then calls our defined function:

```
static void freeppages(paddr_t paddr, unsigned long size);
```

Please note that `MIPS_KSEG0` is a global defined variable (path: `kern/arch/mips/include/vm.h`) which refers to the address of the `kseg0` kernel memory area (unmapped and cached). The `kseg0` memory area starts at address `0x80000000` and runs up to address `0x9fffffff`.

Again, the release is protected by the spinlock `spinlock_acquire(&freemem_lock)`.

Files Management

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()` and `dup2()`. This support, in addition to requiring the implementation of individual functions, also needs appropriate data structures (tables for open files, etc.) that allow the search of files and their identification starting from an integer `id`.

A **virtual file system** or virtual filesystem switch is an abstract layer on top of a more concrete file system. The purpose of a VFS is to allow client applications to access different types of concrete file systems in a uniform way. [2]

In *os161*, the VFS is defined in `vfs.h` (path: `/kern/include/vfs.h`). In this context, the struct `vnode` (path: `/kern/include/vnode.h`) represents a file.

3.1 Utilities

For a given user process, it is necessary to generate a table of `vnode` pointers or a table of struct where one of the fields is a `vnode` pointer. In our implementation, we chose the second option because it was more intuitive and allows more versability if ever needed.

First of all, we defined the table structure called `openfile` (path: `kern/include/syscall_SHELL.h`):

```
/**
 * @brief struct defining a pointer to a specific vnode.
 *        Used to build an array of pointers to vnodes (i.e. files) opened for
 *        a given user process.
*/
struct openfile {
    struct vnode *vn;   /* Pointer to the vnode storing the file */
    off_t offset;   /* Define the current offset for the file */
    int mode_open;  /* Define the mode for the current file (i.e., read-only, write-only, etc...) */
    unsigned int count_refs;    /* Count the number of processes which have currently opened this file */
    struct lock *lock;  /* Define the lock for this open file */
};
```

This structure contains everything needed:

- a pointer to the `vnode` that stores the file. This is the main objective field of this structure;
- an offset value that indicates how far in the file we currently are in;
- the mode in which the file has been opened, think about write-only or read-only or their combination;
- a counter that counts how many processes have this `vnode` opened for something;
- a lock which manage the concurrent access to this structure.

We proceeded in our data structures implementation with the definition of the table (i.e, an array) of this `openfile` structure. The table is defined in a global level scope:

```
struct openfile systemFileTable[SYSTEM_OPEN_MAX];
```

where `SYSTEM_OPEN_MAX` is arbitrarily defined as ten times the `OPEN_MAX` variable, i.e. the maximum number of files that a single process can open. Having a global table means that every single opened file has an unique identifier, allowing a more structured way of representing the `vnode` to process relation. However, every process has its own file table, defined in the `struct proc` (path: `kern/include/proc.h`):

```
/**
* @brief file table of this specific process. Each process can have at maximum
*         OPEN_MAX files opened in the table.
*
*         This struct will be initialized in proc_create() and freed in proc_destroy().
*/
struct openfile *fileTable[OPEN_MAX];
```

At this point, we were able to implement the required system calls.

3.2 System calls

In this section we will present the system calls related to the management of files.

3.2.1 `open()` and `close()` system calls

First things first, we needed a way to open and close a given file. Therefore, we defined the `open()` and `close()` system calls in `syscall_FILE.c` (path: `/kern/syscall/syscall_FILE.c`) as:

```
int sys_open_SHELL(userptr_t pathname, int openflags, mode_t mode, int32_t *retval);
```

and

```
int sys_close_SHELL(int fd);
```

The `open()` system call is used to open a `vnode` through the `vfs_open` utility. At the very beginning, it checks whether the parameters passed are valid and eventually copy the `pathname` to the kernel side (using `copyinstr`). This is done also in other system calls which requires a `pathname` or similar as an argument. Copying from userland to kernel side is done from two reasons (which will be written explicitly here and left implicit in the rest of this chapter):

- for **security reasons**: having a string on the user side is not secure, since it can be easily modified by any other thread, even after the acquisition of the system call routine;
- for **integrity reasons**: the `vfs_` utilities may happen to destroy the user buffer or modified it. Therefore, it can be useful to work on a copy of it instead of the original buffer.

Once everything is ready, the `vnode` is opened and a new position for the file is searched in the `curproc` (the current process) file table. Note that the search starts from index 3. This is due to the fact the indices 0, 1 and 2 have been already assigned to the `stdin`, `stdout` and `stderr` respectively. This last operation is performed when the process is created (refer to chapter: Processes Management).

```
/* ‘of‘ is the openfile reference */
int fd = 3; // skipping stdin, stdout and stderr
if (of == NULL) {
    return ENFILE;  // system file table is full
} else {
    for (; fd < OPEN_MAX; fd++) {
        if (curproc->fileTable[fd] == NULL) {
```

```
            curproc->fileTable[fd] = of;
            break;
        }
    }

    if (fd == OPEN_MAX - 1) {
        return EMFILE;  // process file table is full
    }
}
```

Once the position has been found, three management operations are performed:

- offset management, based on the `openflag` parameter passed to the routine;
- references management, to indicate that another process has opened this file;
- mode management, based on the `openflag` parameter passed to the routine.

Finally, a lock for the file is crated.

The `open()` system call returns a 32-bit value corresponding to the opened file descriptor assigned.

On the other hand, the `close()` system call simply uses the `vfs_close()` function, releasing memory and gracefully manage global variables (such as offset and references counter).

3.2.2 `write()` and `read()` system calls

Next on the list, the `write()` and `read()` system calls, also used to perform operations on the `stdin`, `stdout` and `stderr` files. The declaration are the following:

```
ssize_t sys_write_SHELL(int fd, const void *buf, size_t buflen, int32_t *retval);
```

and

```
ssize_t sys_read_SHELL(int fd, const void *buf, size_t buflen, int32_t *retval);
```

The two functions work in a similar way. Both of them perform some checks on the file descriptor passed as argument:

```
/* checking file descriptor */
if (fd < 0 || fd >= OPEN_MAX) {     /* fd should be a valid number */
    return EBADF;
} else if (curproc->fileTable[fd] == NULL) {    /* fd should refer to a valid entry in the fileTable */
    return EBADF;
} else if (curproc->fileTable[fd]->mode_open == O_WRONLY) {     /* fd should refer to a file allowed to be
      read/write */
    return EBADF;
} else if (buf == NULL) {
    return EFAULT;
}
```

Moreover:

- they both manage a kernel buffer by `copyin()` the buffer provided in case of the `write()` and by `copyout()` the result buffer in case of the `read()`. Again, having a kernel buffer respect the conditions already stated about security and integrity;
- they both exploit their corresponding `VOP_WRITE()` and `VOP_READ()` utilities;
- they both manage the offset value of the file, by re-positioning it to the correct location;
- finally, they both release the used memory for the kernel buffer and then return the number of bytes read or written.

As an example, we reported here the `VOP_WRITE()` operation:

```
struct iovec iov;
struct uio kuio;
struct openfile *of = curproc->fileTable[fd];
struct vnode *vn = of->vn;

lock_acquire(of->lock);
uio_kinit(&iov, &kuio, kbuffer, buflen, of->offset, UIO_WRITE); // initialize the iovec and uio structures
int error = VOP_WRITE(vn, &kuio);   // actually perform write operation
if (error) {
    kfree(kbuffer);
    return error;
```

3.2.3 `lseek()` and `dup2()` system calls

Last but not least, we needed to implement two more system calls in order to perform explicit management of the offset of a given file and the file descriptor duplication.

The `lseek()` system call is declared as:

```
int sys_lseek_SHELL(int fd, off_t pos, int whence, int32_t *retval_low32, int32_t *retval_upp32);
```

It is important to notice that this function returns the new offset of the file, which can end up being a pretty large value, depending on the file size. Therefore, we needed to split the return value into two 32-bits variable, considering the lower half and the upper half of the offset. For more details on the return values, refer to the section System Calls dispatcher.

This routine alters the current seek position of the file handle `fd`, seeking to a new position based on `pos` and `whence`. If whence is:

- `SEEK_SET`: the new position is `pos`
- `SEEK_CUR`: the new position is the current position plus `pos`
- `SEEK_END`: the new position is the position of end-of-file plus `pos`

Inside, nothing strange happens. After some preliminary checks, the `whence` parameter is dispatched to understand what type of seek operation is required. If `SEEK_END` is used, the size of the file is discovered through the `VOP_STAT()` routine.

On the other hand, the declaration of the `dup2()` system call is the following:

```
int sys_dup2_SHELL(int oldfd, int newfd, int32_t *retval);
```

This routine clones the file handle `oldfd` onto the file handle `newfd`. If `newfd` names an open file, that file is closed. Again, inside, nothing unexpected happens. The function perform some preliminary checks and then it copies the file descriptor, updating the corresponding entries in the file table. The new file descriptor is returned.

Directories Management

Each process has its own information about the dynamically assigned working directory. When the shell is spawned, the process that runs the shell needs to be able to retrieve the **current working directory** and, eventually, change it based on its intentions.

In this chapter we present the two system calls we needed to implement in order to perform these operations. Note that in this report, we decided to separate the concept of directories from the concept of files, even if the differences are subtle.

4.1 System calls

In this section we will present the system calls related to the management of directories.

4.1.1 `getcwd()` system call

The `getcwd()` system call is used by a process to retrieve the current working directory. Its implementation is quite simple, since it exploits the already implemented `vfs_getcwd()` utility.

The prototype of the function is the following:

```
int sys_getcwd_SHELL(const char *buf, size_t buflen, int32_t *retval);
```

After some initial assertions (in order to check whether the current process exists and it is well-defined), we were required to craft a `struct uio` object able to store the result of the operation:

```
struct uio u;
struct iovec iov;
iov.iov_ubase = (userptr_t) buf;    // the destination buffer
iov.iov_len = buflen;       // the length of the iovec buffer
u.uio_iov = &iov;   // the iovec structure
u.uio_iovcnt = 1;   // how many iovec we have
u.uio_resid = buflen;   // the total length of the buffer
u.uio_offset = 0;
u.uio_segflg = UIO_USERSPACE;   // where the buffer will be stored
u.uio_rw = UIO_READ;    // which type of operation is required
u.uio_space = curthread->t_proc->p_addrspace;   // the current addresspace
```

Please note that this function, besides storing in `buf` the path to the current working directory, it returns the actual length of the buffer in a 32-bit variable:

```
*retval = buflen - u.uio_resid;
```

4.1.2 `chdir()` system call

On the other hand, having a way to retrieve the current working directory is not sufficient to survive in the shell environment. We needed a routine to change the current working directory to an arbitrary one. The declaration of such routine is:

```
1 int sys_chdir_SHELL(const char *pathname);
```

This function performs the following operations:

- some initial assertions (in order to check whether the current process exists and it is well-defined);
- copy of the pathname from the userland to a new kernel buffer (again, for security and integrity reasons);
- creation of a `vnode` storing the given directory (it exploits, again, the `vfs_open()` routine);
- change of the current directory to the given one through the `vfs_setcurdir(vnode *vn)` utility.

We can see, indeed, that the `vfs_open()` utility map a specific name (stored kernel-side in `kbuffer`) to a new created `vnode`. Note also that, again, the function is compliant with the error codes number.

```
1 /* open directory at pathname */
2 struct vnode *vn = NULL;
3 err = vfs_open(kbuffer, O_RDONLY, 0644, &vn);
4 if (err) {
5     kfree(kbuffer);
6     return err; // may return EINVAL, ENOENT
7 }
8 kfree(kbuffer);
```

Processes Management

Process management is the core chapter of the whole assignment. Here, we present how we managed to make processes work concurrently, how they are managed at the system level, how to duplicate and replace a process, how to terminate a process and how to wait for a process termination.

5.1 Utilities

We present in this section the utility functions and data structure used to manage the processes.

5.1.1 Process Table and `struct proc`

Since we were going to run multiple processes at once, we needed a way to keep track of all the active processes. In particular, questions like "*how many processes can the operating system run at once?*" or also "*how can the OS identify a specific process?*" came to our minds. For these reasons, we defined the system `processTable` as:

```
#define PROC_MAX 100                /* maximum number of allowed running process    */
static struct _processTable {
    bool is_active;                 /* table is active and ready to use */
    struct proc *proc[PROC_MAX+1];  /* [0] not used, PID >= 1   */
    pid_t last_pid;                 /* last PID used in the table   */
    struct spinlock lk;             /* lock for this structure  */
} processTable;
```

This table stores the complete list of processes that are currently active in the operating system. It can have at maximum `PROC_MAX` (i.e. 100) processes (arbitrary number. However, memory constraints should have been taken into account, but this was not the purpose of the assignment). The process table is protected by a spinlock. The process table is initialized at start-up in the following routine:

```
/*
 * Create the process structure for the kernel.
 */
void proc_bootstrap(void) {

    /* kernel process init */
    kproc = proc_create("[kernel]");
    if (kproc == NULL) {
        panic("proc_create for kproc failed\n");
    }

    /* user process init */
#if OPT_SHELL
    spinlock_init(&processTable.lk);        /* lock init*/
    processTable.proc[0] = kproc;   /* registering kernel process in the process table  */
    KASSERT(processTable.proc[0] != NULL);
    for (int i = 1; i <= PROC_MAX; i++) {
        processTable.proc[i] = NULL;
```

```
19      }
20      processTable.is_active = true;  /* activating the process table     */
21      processTable.last_pid = 0;      /* last used PID */
22  #endif
23  }
```

The process table is managed through a bunch of routines defined in `proc.c` (path: `/kern/proc/proc.c`), such as:

- `int find_valid_pid(void);`
- `int proc_add(pid_t pid, struct proc *proc);`
- `void proc_remove(pid_t pid);`
- `struct proc *proc_search(pid_t pid);`

Moreover, the process table stores the process identifier (`pid`) of the last added process, such that it is trivial to add a newly created process. But, how is a process defined?

The process data structure can be found in `proc.h` (path: `/kern/include/proc.h`) and it has the following definition:

```
1   struct proc {
2       char *p_name;           /* Name of this process */
3       struct spinlock p_lock;     /* Lock for this structure */
4       unsigned p_numthreads;      /* Number of threads in this process */
5
6       /* VM */
7       struct addrspace *p_addrspace;  /* virtual address space */
8
9       /* VFS */
10      struct vnode *p_cwd;        /* current working directory */
11
12  #if OPT_SHELL
13      int p_status;               /* current process status   */
14      pid_t p_pid;                /* current process PID      */
15      pid_t parent_pid;           /* parent process PID       */
16      struct child_list* children_list;  /*list of children          */
17      struct cv *p_cv;            /* used for waitpid() syscall */
18      struct lock *p_locklock;    /* used for waitpid() syscall */
19
20      /**
21       * @brief file table of this specific process. Each process can have at maximum
22       *        OPEN_MAX files opened in the table.
23       *
24       *        This struct will be initialized in proc_create() and freed in proc_destroy().
25       */
26      struct openfile *fileTable[OPEN_MAX];
27  #endif
28  };
```

Here we can see that:

- each process stores the **current process status**. This will be useful in the `exit()` and `waitpid()` system calls. Refer to the section `exit()` system call for more details;
- each process stores the information about the children and a children process list.
- each process has a set of locks and conditional variables, used to protect its own fields.
- each process has its own array (table) of `openfile`, as we saw in Files Management. This array is initialized in the `proc_create()` routine through a call to the `bzero()` function as: `bzero(proc->fileTable, OPEN_MAX * sizeof(struct openfile*));`.

Process initialization and de-initialization is performed through two routines:

```
static int proc_deinit(struct proc *proc);
static int proc_init(struct proc *proc, const char *name);
```

5.1.2 `exec.c` utilities

In this section we describe the implementation details of the set of utilities we have coded in order to make the `execv()` system call works. The `exec.c` is a new file added to the system that can be found at `/kern/syscall/exec.c`. Overall, we were required to implement a set of utilities that could accomplish two kinds of operations:

- **load the arguments** into the stack;
- **load the actual executable** image (the implementation of this function is somewhat similar to the `runprogram()` function defined in `/kern/syscall/runprogram.c`).

5.1.2.1 Loading the arguments

The arguments of the new program execution are passed in a buffer defined at the user level. Therefore, we needed to perform two operations:

- copying the buffer from the userland to the kernel side;
- copying the buffer from the kernel side to the user stack.

It is important to notice that the destination of this operation is the **user stack**. As we know. the stack has a specific alignment of 4 bytes for every element that get pushed inside. Therefore, we needed to take this into consideration either when moving from user to kernel or when moving from kernel to user (stack). We chose the second implementation. The two main utilities that manage this procedure are `argbuf_copyin()` and `argbuf_copyout()`, which copy a buffer from userland to kernel side and vice-versa.

We start our description from the management of the kernel buffer, which will contain (for security and integrity reasons) the list of arguments passed from the user level. As already stated, we did not take into consideration the byte-alignment at this step.

We have defined the `struct argbuf_s`, a structure that wraps an `argv[]` in the kernel side during the `execv()` system call execution. The implementation of this structure is the following:

```
typedef struct argbuf_s {

    char *data;     // the actual data
    size_t len;
    size_t max;
    int nargs;  // the number of arguments in the argv[] buffer
    bool tooksem;   // indicates whether the buffer

} argbuf_t;
```

The initialization and cleanup of this structure is managed by two functions:

```
void argbuf_init(argbuf_t *buf);
void argbuf_cleanup(argbuf_t *buf);
```

Which basically reset all the fields to the default values.

Note that the structure contains a boolean value `tooksem` that indicates whether the buffer is exploiting the functionality of the `execthrottle` semaphore. The `execthrottle` semaphore is declared in `exec.h` (path: `/kern/include/exec.h`) in order to limit the number of processes that can run the `exec` at once,

or, rather, the number trying to use large `exec` buffers at once. Its initialization is performed at start-up, during the bootstrap of all the system's components, using the function:

```
#define EXEC_BIGBUF_THROTTLE    1
void exec_bootstrap(void) {

    /* sempahore bootstrap */
    execthrottle = sem_create("exec", EXEC_BIGBUF_THROTTLE);
    if (execthrottle == NULL) {
        panic("Cannot create exec throttle semaphore\n");
    }
}
```

To perform the copy of the buffer to the kernel side, we have defined a function:

```
/**
 * @brief Wrapper function to copy an argv from user side to kernel side
 *
 * @param buf buffer in the kernel side
 * @param uargv argv in the user side
 * @return zero on success, an error value in case of failure
 */
int argbuf_fromuser(argbuf_t *buf, userptr_t uargv);
```

that basically:

- tries to allocate a small buffer of 4096 bytes that, hopefully, will be able to contain all the parameters;
- if the size is not sufficient, it will over-allocate the buffer to the maximum size available (1024×1024 bytes).

```
int argbuf_fromuser(argbuf_t *buf, userptr_t uargv) {

    /* attempt a small allocation */
    argbuf_allocate(buf, PAGE_SIZE);

    result = argbuf_copyin(buf, uargv);
    if (result == E2BIG) {

        /* destroy the buffer and over-allocate it */
        argbuf_cleanup(buf);
        argbuf_init(buf);

        argbuf_allocate(buf, ARG_MAX);
        argbuf_copyin(buf, uargv);
    }
}
```

However, this procedure is inefficient for a variety of (obvious) reasons: trial-and-error allocations, over-allocation, destruction and initialization of data structure already potentially valid, etc.... We chose not to take into consideration efficiency aspects in the implementation of these functions.

The `argbuf_allocate()` simply allocate space:

```
int argbuf_allocate(argbuf_t *buf, size_t size) {

    /* allocating space for data */
    buf->data = (char *) kmalloc(size * sizeof(char));
    if (buf->data == NULL) {
        return ENOMEM;
    }
    buf->max = size;
    return 0;
}
```

Now we can start look at the `argbuf_copyin()` routine, which performs the copy from userland to kernel buffer. Its declaration is the following:

```
int argbuf_copyin(argbuf_t *buf, userptr_t uargv);
```

And it will basically goes through all the list of arguments passed in the user buffer and copy them in the kernel buffer (previously allocated) exploiting the `copyin()` routine:

```
int argbuf_copyin(argbuf_t *buf, userptr_t uargv) {

    /* loop every argv[] */
    while (true) {

        /* first, grab the pointer at argv (argv is incremented at the end of the loop). */
        copyin(uargv, &thisarg, sizeof(userptr_t));

        /* check end of array */
        ...

        /* fetch argument string */
        ...

        /* move to the next argument. Note: thisarglen includes the \0. */
        buf->len += thisarglen;
        uargv += sizeof(userptr_t);
        buf->nargs++;
    }

    return 0;
}
```

On the other hand, the `argbuf_copyout()` routine has to manage the stack alignment. Indeed, this routine is used to copy the kernel buffer to the user stack, so byte alignment is mandatory. The declaration is the following:

```
int argbuf_copyout(argbuf_t *buf, vaddr_t *ustackp, int *argc_ret, userptr_t *uargv_ret);
```

And it will basically goes through all the list of arguments passed in the kernel buffer, byte-align them in order to be compliant with the stack rule and finally copy them in the user stack exploiting the `copyout()` routine. We will not report the details of the function here, since it will be too much. The details can be found in the `kern` directory of this repository.

5.1.2.2 Loading the executable

Once the buffer containing the list of arguments has been successfully copied from userland to the kernel side, we are ready to **load the executable image in the address space**. It is important to notice that this operation is critical: the old address space **will be destroyed** in order to let space for the new image. If this operation fails, there will be no more a safe place to return, and the system will `panic()`.

We defined the `loadexec()` function, which will perform almost the same set of operations as the `runprogram()` utility, without actually starting the process (this will be taken into account later on, refers to `execv()` system call).

The declaration of the function is the following:

```
/**
 * @brief Load an executable (common code with runprogram)
 *
 * @param path pathname of the executable to run
 * @param entrypoint virtual address of the starting point
 * @param stackptr virtual address of the stack
 * @return zero on success, an error value in case of failure
 */
int loadexec(char *path, vaddr_t *entrypoint, vaddr_t *stackptr);
```

This routine will perform the following operations:

- open the **executable file** exploiting the `vfs_open()` utility;

- create a new **address space** that will be substitute to the old one through the `proc_setas()` and `as_activate()` utilities;
- **load the executable**, exploiting the `load_elf()` utility. Note that `entrypoint` will contain the actual address of the entrypoint of the ELF file. It will be used later on by the `execv()` system call in order to actually start the execution (`enter_new_process()`);
- define the **user stack** in the address space: `as_define_stack(newas, stackptr)`;
- close the file and **destroy** the old address space.

5.2 System calls

In this section we will present the system calls related to the management of processes.

5.2.1 `getpid()` and `waitpid()` system calls

While it is trivial to get the current process identifier:

```
int sys_getpid_SHELL(pid_t *retval) {

    /* retrieving pid from curproc */
    KASSERT(curproc != NULL);
    *retval = curproc->p_pid;

    return 0;   /* getpid() does not fail.  */
}
```

it is definitely more complex to wait for a process termination.

The declaration of the `waitpid()` system call is the following:

```
/**
 * @brief Wait for the process specified by pid to exit, and return an encoded exit status
 *        in the integer pointed to by status. If that process has exited already, waitpid
 *        returns immediately. If that process does not exist, waitpid fails.
 *
 * @param pid pid of the process to wait
 * @param status exit status of the process to wait
 * @param options not implemented
 * @return zero on success, an error value in case of failure.
 */
int sys_waitpid_SHELL(pid_t pid, int *status, int options, int32_t *retval);
```

This function allows the caller process to get to sleep through the `cv_wait(proc->p_cv, proc->p_locklock);` call, and waits for the specified process to terminate and execute the `exit()` system call. The implementation of this function is the following:

- it checks whether we fall in some trivial case, i.e. the waited process is the same of the calling process or it is not one of its children; it checks whether the status value is `NULL` or not and if its address is a valid kernel address.
- it checks whether the `WNOHANG`option has been used. In this case, it simply returns the waited process identifier, in order to avoid hanging on the condition variable.
- it retrieves the waited process structure given the process identifier passed as argument.
- it waits!

```
/* waiting termination of the process */
    lock_acquire(proc->p_locklock);
    cv_wait(proc->p_cv, proc->p_locklock);
    lock_release(proc->p_locklock);
```

Eventually, it will return some values and destroy the process:

```
*status = proc->p_status;
*retval = proc->p_pid;

proc_destroy(proc);
```

5.2.2 `exit()` system call

When a process terminates, it will trigger the `exit()` system call in order to allow management of processes' statuses, memory de-allocation and other logistic stuff. The definition of the function is the following:

```
/**
 * @brief Cause the current process to exit.
 *
 * @param exitcode reported back to other process(es) via the waitpid() call.
 * @return does not return.
 */
#if OPT_SHELL
void sys_exit_SHELL(int exitcode) {

    /* retrieve status of the current process */
    struct proc *proc = curproc;
    proc->p_status = _MKWAIT_EXIT(exitcode);    /* exitcode & 0xff */

    /* removing thread before signalling due to race conditions */
    proc_remthread(curthread);        /* remove thread from current process */

    /* signalling the termination of the process */
    lock_acquire(proc->p_locklock);
    cv_signal(proc->p_cv, proc->p_locklock);
    lock_release(proc->p_locklock);

    /* main thread terminates here */
    thread_exit();

    /* should not reach here */
    panic("[!] Wait! You should not be here. Some errors happened during thread_exit()...\n");
}
#endif
```

It is important to notice that there may be race conditions when a process exits and signal its termination. This is due to the fact that the `proc_destroy()` function requires that the process being destroyed (data structure) no longer has active threads (see the `process_destroy` code, which contains the assertion `KASSERT(proc->p_numthreads == 0)`). Since `sys__exit()` signals the end of the process before calling `thread_exit()`, it is possible that the kernel on hold (in `common_prog()`) is woken up and calls `proc_destroy()` before `thread_exit()` "detaches" the thread from the process. (see the `thread_exit()` code, especially the call to `proc_remthread()`).

We solved this problem by adding the following lines in the `thread_exit()` routine (path: `/kern/thread/thread.c`):

```
#if OPT_SHELL
    if (cur->t_proc != NULL) {
        proc_remthread(cur);
    }
#else
    proc_remthread(cur);
#endif
```

A complete description of this behavior can be found on my (Davide Arcolini) GitHub page: here

5.2.3 `fork()` system call

The `fork()` system call is one of the two core utilities we were required to implement for this project. When a process is running the shell, it may be required to execute other commands (e.g., run an executable file) without destroying the shell itself. To do such a task, the process duplicates its address space, creating *de facto* another process, called **child process**, which, on its own, will call the `execv()`, running the requested binary file. Indeed, the `fork()` system call is a routine that creates new processes, giving them a new personal address space. The child process has an exact copy of all the memory segments of the parent process. Moreover, the child process starts off with a copy of its parent's file descriptors (typically used to exploit pipes or named pipes in inter process communication; however, this is not the case for this project).

The declaration of the function, which can be found in `syscall_PROC.c` (path: `/kern/syscall/syscall_PROC.c`), is the following:

```
int sys_fork_SHELL(struct trapframe *ctf, pid_t *retval);
```

First thing to notice is that this function returns the process identifier (`pid`) of the newly created process, the child process. This value is returned in a `pid_t` variable, which is nothing more than a 32-bit type definition of an integer value. Moreover, this function receives in input a `struct trapframe`. It contains the parent's trapframe, with all the useful information that will be copied into the child's trapframe before calling the already explained `call_enter_forked_process`.

But let's see the implementation of this routine.

After some preliminary checks (current process exists and it is well-defined), the routine looks in the process table in order to find availability for a new process identifier. It exploits our defined `find_valid_pid()` (path: `/kern/proc/proc.c`). This function searches in the process table with a circular buffer fashion, starting from the last assigned process identifier:

```
int index = (processTable.last_pid + 1 > PROC_MAX) ? 1 : processTable.last_pid + 1;
while (index != processTable.last_pid) {
    if (processTable.proc[index] == NULL) {
        break;
    }

    index ++;
    index = (index > PROC_MAX) ? 1 : index;
}
```

eventually, it returns a valid process identifier.

After that, the routine calls the already defined `proc_create_runprogram()` (path: `/kern/proc/proc.c`). This function create a fresh new runnable process. However, for any given process, as already stated in Files Management, the first file descriptors (0, 1, and 2) are considered to be standard input (`stdin`), standard output (`stdout`), and standard error (`stderr`). These file descriptors should start out attached to the console device (`con:`). Therefore, we modified the function adding the console initialization:

```
/* console initialization */
if (console_init("STDIN", newproc, 0, O_RDONLY) == -1) {
    return NULL;
} else if (console_init("STDOUT", newproc, 1, O_WRONLY) == -1) {
    return NULL;
} else if (console_init("STDERR", newproc, 2, O_WRONLY) == -1) {
    return NULL;
}
```

The console initialization simply create space in the process file table and assign to the first three entries of the table the three default file descriptors. Its implementation can be found in `/kern/proc/proc.c`, whose prototype is the following:

```
static int console_init(const char *lock_name, struct proc *proc, int fd, int flag);
```

Once the new process space has been defined, the `fork()` routine exploits the `as_copy()` utility to copy its own **address space** to the child address space.

```
int err = as_copy(curproc->p_addrspace, &(newproc->p_addrspace));
if (err) {
    proc_destroy(newproc);
    return err;
}
```

Once the destination process is ready, the parent also copy the input trapframe to the newly created one, exploiting the `memmove()` function:

```
struct trapframe *tf_child = (struct trapframe *) kmalloc(sizeof(struct trapframe));
if(tf_child == NULL){
    proc_destroy(newproc);
    return ENOMEM;
}
memmove(tf_child, ctf, sizeof(struct trapframe));
```

Next in the schedule: the parent process needs to add the child process to its own list of children. We defined a function (path: `/kern/proc/proc.c`):

```
int add_new_child(struct proc* proc, pid_t child_pid);
```

that does exactly this job. If the space in the children list is full, the function returns -1 raising an error in the `fork()` system call. On the other hand, if everything works out smoothly, we link the parent with the child:

```
newproc->parent_pid = father->p_pid;
```

If the linking procedure is completed successfully and we reach this point in the execution, we can add the child process to the process table of the system. The created process has now its own identity in the system table! In order to do so, we defined another function:

```
int proc_add(pid_t pid, struct proc *proc) {

    /* parameters checks */
    if (pid <= 0 || pid > PROC_MAX+1 || proc == NULL) {
        return -1;
    }

    /* adding process */
    spinlock_acquire(&processTable.lk);
    processTable.proc[pid] = proc;

    /* update last pid position */
    processTable.last_pid = pid;
    spinlock_release(&processTable.lk);

    /* task completed successfully */
    return 0;
}
```

Now, the most important operation that actually starts the new process. The parent process calls the `thread_fork()` routine (path: `/kern/thread/thread.c`), which will create a new thread based on an existing one, the parent. This routine is already present in the basic version of *os161* and we did not need to edit it in any way. The function call is the following:

```
err = thread_fork(
    curthread->t_name,              /* same name as the parent  */
    newproc,                        /* newly created process    */
    call_enter_forked_process,      /* routine to start         */
    (void *) tf_child,              /* child trapframe          */
    (unsigned long) 0               /* unused                   */
);
```

Once the new thread has started in the child process, the `fork()` can return the child process identifier:

```
*retval = newproc->p_pid;
```

5.2.4 `execv()` system call

As we have already seen, the `fork()` system call is able to duplicate the address space of a process and create a child process which will run on its own. However, having a duplicated process is not sufficient to make the shell works as intended. Indeed, when a command is given on the shell, the running process will duplicate itself. However it needs a way to replace the image of its own program with the image of the specified program. When we run the shell with the command `p /bin/sh`, a process will be created and it will be loaded with the image of the `/bin/sh` binary file. Now, from that shell, if we want to run any commands, e.g. `/testbin/palin/`, the shell process will duplicate itself and then replace the image of `/bin/sh` with the image of `/testbin/palin`.

The `execv()` system call does exactly this job.

The `execv()` system call is defined in the `syscall_PROC.c` file (path: `/kern/syscall/syscall_PROC.c`); the declaration is the following:

```
1 /**
2  * @brief Replaces the currently executing program with a newly loaded program image. This occurs
3  *        within one process; the process id is unchanged.
4  *
5  * @param progname pathname of the program where execution will start
6  * @param argv an array of 0-terminated strings. The array itself should be terminated by a NULL pointer.
7  * @return zero on success, and error value in case of failure
8 */
9 int sys_execv_SHELL(const char *progname, char *argv[])
```

Before discussing the implementation of this routine, refer to the `exec.c` utilities section, in which the details of the utilities used for the `execv()` are described.

After some preliminary checks (current process exists and it is well-defined), the function performs the following operations:

- allocate a kernel buffer through the `argbuf_init()` routine;
- copy the user buffer to the kernel buffer through the `argbuf_fromuser()` routine;
- load the new executable image through the `loadexec()` routine;
- copy the kernel buffer to the user stack through the `argbuf_copyout()` routine;
- switch to user mode through the `enter_new_process()` routine;

Note that `execv()` must not return, since the execution is passed to the usermode with the `enter_new_process()` call. In case of return of this last function, the system will `panic()`!

Results Analysis

In this chapter, we provide pictures of the result execution performed through some test program along with some considerations and insights about the whole development.

First thing to notice is that the available amount of RAM has been increased in order to allow concurrent execution of multiple process. In particular, from having a `ramsize` of `512K` we moved to a `ramsize` of `2048K`. This value is stored in the `ramsize` variable defined in the `sys161.conf` file (path: `root/sys161.conf`). As an example, the `testbin/forktest` works as expected:

```
OS/161 kernel [? for menu]: p testbin/forktest
[!] 0 is waiting for 1...
(program name unknown): Starting. Expect this many:
|---------------------------|
AABBBBCCCCCCCCDDDDDDDDDDDDDDDD
(program name unknown): Complete.
[!] process 1 terminated with exit status 0
Operation took 1.072911152 seconds
```

Figure 6.1: Results of the forktest operation.

6.1 `testbin/badcall` test results

In order to check the correctness of the error management and the limit case resilience of the implemented functions, the `testbin/badcall` has been used. The `testbin/badcall` provides a menu in which it is possible to choose a particular stress tests among different routines and retrieve a report of the results. Typically, limit case of parameters value and return values are checked against the expected case.

Here's the result of the execution of the badcalls related to the processes routine.

```
badcall: exec with NULL program... Bad memory reference                passed
badcall: exec with invalid pointer program... Bad memory reference      passed
badcall: exec with kernel pointer program... Bad memory reference       passed
badcall: exec the empty string... Invalid argument                      passed
badcall: exec with NULL arglist... Bad memory reference                 passed
badcall: exec with invalid pointer arglist... Bad memory reference      passed
badcall: exec with kernel pointer arglist... Bad memory reference       passed
badcall: exec with invalid pointer arg... Bad memory reference          passed
badcall: exec with kernel pointer arg... Bad memory reference           passed
```

(a) execv() system call

```
badcall: wait for pid -8... No child processes                          passed
badcall: wait for pid -1... No child processes                          passed
badcall: pid zero... No child processes                                 passed
badcall: nonexistent pid... No child processes                          passed
badcall: wait with NULL status... Success                               passed
badcall: wait with invalid pointer status... Bad memory reference       passed
badcall: wait with kernel pointer status... Bad memory reference        passed
badcall: wait with unaligned status... Bad memory reference             passed
badcall: wait with bad flags... Invalid argument                        passed
badcall: wait for self... No child processes                            passed
badcall: wait for parent...
   from child:            No child processes                            passed
   from parent:           Success                                       passed
badcall: siblings wait for each other...
   sibling (pid 8)                    No child processes                passed
   sibling (pid 9)                    No child processes                passed
   overall                                                              passed
```

(b) waitpid() system call

Figure 6.2: Results of the badcall related to the processes management.

Here's the result of the execution of the badcalls related to the files routines and directories routines.

Finally, here's an example of the execution of the shell (`p bin/sh`) and subsequent access to the file

```
badcall: open with NULL path... Bad memory reference                    passed
badcall: open with invalid-pointer path... Bad memory reference          passed
badcall: open with kernel-pointer path... Bad memory reference           passed
badcall: open null: with bad flags... Invalid argument                   passed
badcall: open empty string... Invalid argument                           passed
```

(a) open() system call

```
badcall: close using fd -1... Bad file number                            passed
badcall: close using fd -5... Bad file number                            passed
badcall: close using closed fd... Bad file number                        passed
badcall: close using impossible fd... Bad file number                    passed
badcall: close using fd OPEN_MAX... Bad file number                      passed
```

(b) close() system call

```
badcall: write using fd -1... Bad file number                            passed
badcall: write using fd -5... Bad file number                            passed
badcall: write using closed fd... Bad file number                        passed
badcall: write using impossible fd... Bad file number                    passed
badcall: write using fd OPEN_MAX... Bad file number                      passed
badcall: write using fd opened read-only... Bad file number              passed
badcall: write with NULL buffer... Bad memory reference                  passed
badcall: write with invalid buffer... Bad memory reference               passed
badcall: write with kernel-space buffer... Bad memory reference          passed
```

(c) write() system call

```
badcall: read using fd -1... Bad file number                             passed
badcall: read using fd -5... Bad file number                             passed
badcall: read using closed fd... Bad file number                         passed
badcall: read using impossible fd... Bad file number                     passed
badcall: read using fd OPEN_MAX... Bad file number                       passed
badcall: read using fd opened write-only... Bad file number              passed
badcall: read with NULL buffer... Bad memory reference                   passed
badcall: read with invalid buffer... Bad memory reference                passed
badcall: read with kernel-space buffer... Bad memory reference           passed
```

(d) read() system call

```
badcall: lseek using fd -1... Bad file number                            passed
badcall: lseek using fd -5... Bad file number                            passed
badcall: lseek using closed fd... Bad file number                        passed
badcall: lseek using impossible fd... Bad file number                    passed
badcall: lseek using fd OPEN_MAX... Bad file number                      passed
badcall: lseek on device... Illegal seek                                 passed
badcall: lseek stdin when open on file...                                ------
badcall: try 1: SEEK_SET... Success                                      passed
badcall: try 2: SEEK_END... Success                                      passed
badcall: lseek to negative offset... Invalid argument passed
badcall: seek past/to EOF...                                             passed
badcall: lseek with invalid whence code... Invalid argument              passed
```

(e) lseek() system call

```
badcall: dup2 using fd -1... Bad file number                             passed
badcall: dup2 using fd -5... Bad file number                             passed
badcall: dup2 using closed fd... Bad file number                         passed
badcall: dup2 using impossible fd... Bad file number                     passed
badcall: dup2 using fd OPEN_MAX... Bad file number                       passed
badcall: dup2 to -1... Bad file number                                   passed
badcall: dup2 to -5... Bad file number                                   passed
badcall: dup2 to impossible fd... Bad file number                        passed
badcall: dup2 to OPEN_MAX... Bad file number                             passed
badcall: copying stdin to test with... badcall: dup2 to same fd...       passed
badcall: fstat fd after dup2 to itself... Unknown syscall 82
                                          Function not implemented       ------
badcall: lseek fd after dup2 to itself... Illegal seek                   passed
```

(f) dup2() system call

Figure 6.3: Results of the badcall related to the files management.

```
badcall: getcwd with NULL buffer... Bad memory reference                 passed
badcall: getcwd with invalid buffer... Bad memory reference              passed
badcall: getcwd with kernel-space buffer... Bad memory reference         passed
```

(a) getcwd() system call

```
badcall: chdir with NULL path... Bad memory reference                    passed
badcall: chdir with invalid-pointer path... Bad memory reference         passed
badcall: chdir with kernel-pointer path... Bad memory reference          passed
badcall: chdir to empty string... Invalid argument                       passed
```

(b) chdir() system call

Figure 6.4: Results of the badcall related to the directories management.

system in order to read the content of a file (`cat testscripts/test.py`):

```
cpu0: MIPS/161 (System/161 2.x) features 0x0


[!]     Project c2: SHELL
        ------------------------
        Academic Year: 2021/2022
        Authors:
                Davide Arcolini
                Giulia Bianchi



OS/161 kernel [? for menu]: p bin/sh
[!] 0 is waiting for 1...
(program name unknown): Timing enabled.
OS/161$ cat testscripts/test.py
#!/usr/bin/env
# test.py - run some test material
# usage: auto/test.py [options] test-commands
# options:
#     --menuprompt=STR   Change menu prompt string
#     --shellprompt=STR  Change shell prompt string
#     --conf=sys161.conf Use alternate sys161 config
#     --ram=N            Force RAM size (default from sys161 config)
#     --cpus=N           Force number of cpus (default from sys161 config)
#     --doom=N           Set doom counter to N (default none)
#     --progress=N       Progress monitoring with N-second timeout (default 30)
#     --no-progress      Disable progress monitoring
#     --timeout=N        Global timeout, in seconds (default 300)
#     --kernel=KERNEL    Choose kernel to run (default "kernel")
#
#
# This is a directly executable wrapper around runtest.py. You can use
# it from the host OS shell to run more or less arbitrary scripts, or
# you can write your own Python scripts using runtest.py directly.
#
# See the top of runtest.py for an explanation of the arguments.
#

import sys
from optparse import OptionParser

import runtest
```

Figure 6.5: Example of the shell environment.

Bibliography

[1] David A. Holland. *The OS/161 Instructional Operating System*. Harvard College. 2016. URL: http://www.os161.org/.

[2] Wikipedia contributors. *Virtual file system — Wikipedia, The Free Encyclopedia*. [Online; accessed 22-November-2022]. 2022. URL: https://en.wikipedia.org/w/index.php?title=Virtual_file_system&oldid=1083307651.