Advanced Operating Systems
Practical 3: System calls and programming of a μshell

UAH, Departamento de Automática, ATC-SOL
`http://atc1.aut.uah.es`

Themes 1-5

Abstract

The goal of this practical is to study the system calls related to files and to practise their use programming a small *shell* (command interpreter).

1 Introduction

The goal of this practical is to study some system calls and to practise their use. Section 2 describes the basic system calls for accessing files, and proposes making a simple program that uses them. Section 3 proposes making a small *shell* (command interpreter), which we will call μshell. The μshell requires the use of other system calls.

2 Basic system calls for managing files

2.1 Open a file: `open()`

The `open()` system call is the fist step that every process must take in order to access the data contained in a file. It enables the process to make later read/write operations accessing the open file. Its syntax is:

```
int open (const char *pathname, int flags, mode_t modes)
```

Parameters:

- `pathname`: name of the file —optionally preceded by the path—.
- `flags`: open mode —read, write, etc.—.
- `modes`: access permissions of the file in case it has to be created.

Returned value: integer representing the file descriptor; greater or equal to 0 if everything was OK. The file descriptor will be used every time the process wants to access the file.

2.1.1 Recommended tests for `open()`:

Refer to the manual[1,2] and program as many experiments as you need to clearly understand the behaviour and the value returned by `open()` in the next cases:

1. An existing file and a non-existing file are opened.
2. A file is opened by a process with the required access privileges and another file is opened by a process without the required access privileges.
3. Several consecutive calls to `open()` are carried out for different files, and some more for one single file specifying the same open mode.
4. Two different programs call `open()` for different files, and for the same file. Use several terminals or background execution (&) to check if the programs can access the same file at the same time.

2.2 Read the contents of a file: `read()`

The `read()` system call allows processes to access to the contents of a file by storing a copy of these data in memory cells —variables—. Its syntax is:

```
ssize_t read (int fd, void *buffer, size_t count)
```

Parameters:

- `fd`: file descriptor —returned by a previous call to `open()`— for the file on which the read operation will be carried out.
- `buffer`: memory address where the read data will be copied.
- `count`: number of bytes to read from the file.

Returned value: integer indicating the number of bytes actually read; greater or equal to 0 if everything was OK.

The number of bytes read allows the program to detect the event of reaching the end of the file. There is no specific End Of File mark. If the pointer —the current position in the file— is at the end of the file, the call to `read()` will simply return 0 —"zero bytes read"—.

2.2.1 Recommended tests for `read()`:

Refer to the manual[3] and program as many experiments as you need to clearly understand the behaviour and the value returned by `read()` in the next cases:

5. Read a few bytes from a file, and try to read more bytes than there are in the file.
6. Try reading something from a file when all the data have already been read —equivalent to reading from an empty file—.
7. Read from a file using two different descriptors —two `open()` calls on the same file—.

[1] System calls are in section 2 of the manual. In the command line, type: `man 2 open`
[2] We recommend to have the next packages installed: `manpages-dev glibc-doc build-essential`
[3] In the command line, type: `man 2 read`

2.3 Write the contents of a file: `write()`

The `write()` system call allows processes to modify the contents of files, either overwriting existing data or appending additional data at the end. The data to be written are taken from memory locations —variables—. The syntax of `write()` is:

```
ssize_t write (int fd, const void *buffer, size_t count)
```

Parameters:

- `fd`: file descriptor —returned by a previous call to `open()`— for the file on which the write operation will be carried out.
- `buffer`: memory address of the source data to be written.
- `count`: number of bytes to write in the file.

Returned value: integer indicating the number of bytes written; greater or equal to 0 if everything was OK.

2.3.1 Recommended tests for `write()`:

Refer to the manual[4] and program as many experiments as you need to clearly understand the behaviour and the value returned by `write()` in the next cases:

8. Overwrite some bytes in a file, and try to overwrite more bytes than there are in the file.
9. Write in a file when positioned at its end —equivalent to writing in an empty file—.
10. Write in a file using two different descriptors —two `open()` calls on the same file—.

2.4 Reposition the read/write pointer of a file: `lseek()`

The `lseek()` system call allows to modify the position, inside an open file, where the next read/write operation will take place. That is, `lseek()` modifies the value of the pointer stored in the corresponding entry of the open files table. Its syntax is:

```
off_t lseek (int fd, off_t offset, int whence)
```

Parameters:

- `fd`: file descriptor —returned by a previous call to `open()`— for the file on which the read/write pointer must be modified.
- `offset`: number of bytes to move the pointer. It can be positive or negative. The exact meaning of this parameter depends on the value of the parameter `whence`.

[4] In the command line, type: `man 2 write`

- `whence`: position from which to add the offset indicated in the previous parameter. It can be SEEK_SET —beginning of the file—, SEEK_CUR —current position of the pointer— or SEEK_END —end of the file—.

Returned value: resulting position, in bytes, from the beginning of the file —value of the pointer—, or -1 if an error occurred.

The pointer can be positioned beyond the end of the file. If a process writes at such position, the intermediate unwritten space will be filled with zeroes.

2.4.1 Recommended tests for `lseek()`:

Refer to the manual[5] and program as many experiments as you need to clearly understand the behaviour and the value returned by `lseek()` in the next cases:

11. Jump to the end of the file and write some data.
12. Jump to the beginning of the file and read some data.
13. Jump beyond the end of the file and write some data.
14. Find out the current position with a jump to {`offset`=0, `whence`=SEEK_CUR}.
15. Find out the size of the file.

2.5 Map to memory the contents of a file: `mmap()`

The `mmap()` system call allows processes to map the contents of files to memory positions. After the mapping, they can access the contents of the mapped files through variables that point to the mapped zone. In other words, after calling `mmap()`, a process can read or write the contents of the mapped file by using the corresponding positions of the array of bytes that starts at the memory position where the file has been mapped. The syntax of `mmap()` is:

```
void *mmap (void *start, size_t length, int prot,
            int flags, int fd, off_t offset)
```

Parameters:

- `start`: suggested memory address for the mapping. Adress 0 (NULL) is usually specified, indicating the operating system to pick any valid address.
- `length`: number of bytes to map.
- `prot`: desired memory protection. It can be PROT_EXEC —the pages will be executable—, PROT_READ —the pages will be readable—, PROT_WRITE —the pages will be writeable— or PROT_NONE —some virtual space will be reserved, but the pages will not be accessible until the protection is changed with `mprotect()`—. These values can be combined with the bitwise *OR* operation (operator `|` in `C` language)[6].

[5] In the command line, type: `man 2 lseek`
[6] For example: `PROT_READ|PROT_WRITE`

- `flags`: mapping options. It can be MAP_FIXED —do not use an address different from `start`—, MAP_SHARED —the mapped zone can be shared with other processes— or MAP_PRIVATE —the memory zone is private—. The call must specify exactly one of the last two options. In addition, MAP_FIXED can be specified in combination with MAP_SHARED or MAP_PRIVATE through a bitwise *OR* operation[7].
- `fd`: file descriptor —returned by a previous call to `open()`— for the file that will be memory-mapped.
- `offset`: position of the file where the mapping will start. It must be a multiple of the page size[8].

Returned value: memory position where the mapping has been created, if the operation was successful, or MAP_FAILED[9] if there was an error or if the solicited address could not be used.

Accessing files with memory mappings is faster than accessing through read/write calls for several reasons. First, with access via mapping it is not necessary to access intermediate tables on every operation. Second, accesses outside the memory space of a process are also slower. Finally, accesses through `read()` and `write()` require a system call on every operation, while with mapping, thanks to space and time locality, most operations will not require any immediate action by the operating system.

Mapping part of a file requires the same time as mapping the whole file, because the operating system only establishes the relationship between virtual memory positions and disc positions, leaving the read/write operations for the moments when they are unavoidable. For this reason, files are usually mapped in full, which requires knowing their size in advance.

2.5.1 Recommended tests for `mmap()`:

Refer to the manual[10] and program as many experiments as you need to clearly understand the behaviour and the value returned by `mmap()` in the next cases:

16. Read the data contained in a file.

2.6 Free the memory where a file is mapped: `munmap()`

The `munmap()` system call allows processes to release the memory addresses associated to a previously mapped file. Its syntax is:

```
int munmap(void *start, size_t length)
```

Parameters:

- `start`: starting memory address of the map to be freed. It must be the value returned by `mmap()` when the memory map was created.

[7] For example: `MAP_SHARED|MAP_FIXED`
[8] The page size can be obtained with `sysconf(_SC_PAGE_SIZE)`
[9] MAP_FAILED is defined as `(void*)-1`
[10] In the command line, type: `man 2 mmap`

- `length`: number of bytes to be freed.

Returned value: integer indicating whether the operation was successful (0) or not (-1).

After freeing the memory used for a mapping, all subsequent accesses to that region will be invalid memory references.

2.6.1 Recommended tests for `munmap()`:

Refer to the manual[11] and program as many experiments as you need to clearly understand the behaviour and the value returned by `munmap()` in the next cases:

17. Try to read data of a memory-mapped file after calling `munmap()`.

2.7 Close a file: `close()`

The `close()` system call releases a previously used file descriptor, releasing also all the information that the operating system stores related to the open file. When a process is not going to use any more an open file, it must close it in order to ensure that all the operations carried out on the file are completed —the buffering system used by UNIX systems delays some operations in order to enhance the overall system performance—. The syntax of this call is:

```
int close (int fd)
```

Parameters:

- `fd`: file descriptor —returned by a previous call to `open()`— for the file that will be closed.

Returned value: integer indicating whether the operation was successful (0) or not (-1).

When a file is closed, the corresponding entry of the file descriptor table is released. If no other file descriptor —i.e., from another process— points to the same entry of the open files table, this latter entry is released too.

2.7.1 Recommended tests for `close()`:

Refer to the manual[12] and program as many experiments as you need to clearly understand the behaviour and the value returned by `close()` in the next cases:

18. Close a non-existing descriptor.
19. Read or write data in a file after closing it.

[11] In the command line, type: `man 2 munmap`
[12] In the command line, type: `man 2 close`

2.8 Count, with system calls, the occurrences of a character in a file

Following the instructions detailed below, make a program that counts the occurrences of a given character in a file.

The program must be composed of three modules written in `C` language, one containing the main program, the argument checking and the visualization of results, a second module containing a function that will return the existing character in a given position of an already open file and a third module that returns the number of occurrences of a character within an already open file.

These last two modules do not invoke each other, but provide alternative methods of access to the file. The function that reads one character must be implemented without memory mapping, and the function that counts the number of occurrences must be implemented using memory mapping.

Create also a `Makefile` to ease the compilation of the different modules.

The steps for making the program are:

1. Create a `Makefile` to generate an executable program called `count` from the object files that will result from compiling the source modules `main.c`, `readchar_R.c` and `count_M.c`. There will also be two separated files called `readchar_R.h` and `count_M.h` that will contain the declaration of the corresponding functions. These functions will be invoked from `main.c`. Remember that all compilations must be carried out with the modifier `-Wall`. The object files containing the implemented functions will be stored together in a library called `libcount.a` with the command `ar`[13]. Refer to the *man* page of the `ar` command. Figure 1 shows a scheme of the compilation and linkage process that the `Makefile` must specify.

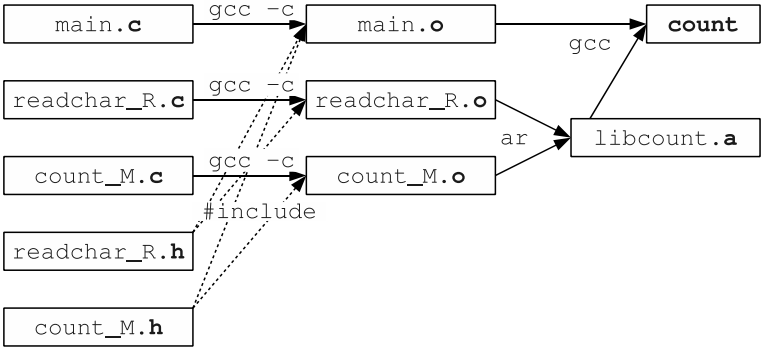

Figure 1: Scheme of Compilation and linkage for the program `count`

[13]Usage example of the `ar` command and the resulting `.a` file:

```
user@host:$ ar -r libcolours.a red.o blue.o yellow.o
user@host:$ gcc -o program main.o -L . -l colours
```

2. Create the header files `readchar_R.h` and `count_M.h` containing, respectively, the next declarations:

```
char ReadCharacter (int fd, int position);
int CountCharacters (int fd, char character);
```

3. Create the modules `readchar_R.c` and `count_M.c`, which will contain the definition of the functions declared in the corresponding header files. The function `ReadCharacter()` must return, without using memory mapping, the character stored in the indicated position of the file whose descriptor is passed. The function `CountCharacters()` must calculate, using memory mapping, the number of occurrences of the specified character in the file.

4. Create the module `main.c`, which will check that the number of arguments is correct and open the file passed as argument. Then it will call the corresponding function and print in the standard output the result returned by the said function. The syntax for calling the executable is:

   For counting while reading with `read()`:
   `user@host:$ ./count R` *file* *character*

   For counting with memory-mapping.
   `user@host:$ ./count M` *file* *character*

   Por ejemplo:

```
user@host:$ ./count R file a
The character a has 43 ocurrences
user@host:$ ./count M file a
The character a has 43 ocurrences
```

After generating the executable —remember that the generation should not provoke any warning, even with the `-Wall` option—, verify that the program executes correctly. Debug the program if you consider it necessary.

3 Programming of a μshell

3.1 Theoretical basis: what is a *shell* and how does it work

As it is already known, a command interpreter is a program that requests keywords —commands— from the standard input and, depending on the word entered, performs one task or another. The execution cycle of a command interpreter —usually known as "shell"— can be summarized in four stages:

1. **Wait for a command from the user.** The program usually prints a message —called "prompt"— to invite the user to type a command.

2. **Command parsing.** In this stage, a set of transformations are applied to the line typed by the user. We can distinguish five types, although their inclusion depends on the capabilities of the shell: history substitutions, alias, variable substitution, command substitution and file expansion.

3. **Execution of the command.** It is in this stage when the command is located and executed. The commands are classified in two types:

   - Internal commands: those executed by the shell itself. The code that carries out the operation is part of the code of the shell.
   - External commands: carried out by executable programs, which are not part of the shell. In order to execute these commands, the shell duplicated itself —creates a new process by calling `fork()`— and then the image of the new process is replaced with that of the external program —through a call to `exec()` or similar—.

4. **Wait for command completion.** Unless the command is executed in the background —which would be indicated by a trailing '&'—, the initial process —the shell— must wait for the completion of the new process before returning to stage 1.

3.2 Program a μshell that recognises some internal commands for managing files and directories

Make a μshell —a small command interpreter— following the instructions detailed below.

The source files `parser.h` and `parser.c`, provided along with this text, facilitate the command parsing stage. Study the program `example.c` —also provided with this text—, which illustrates the use of `parser.h` and `parser.c`. The following listing shows an example of compiling and running the program `example.c`:

```
user@host:$ gcc -Wall example.c parser.c -o example
user@host:$ ./example
Type commands (press Ctrl-D to finish)
$ my_program par\ 1 <in.txt 2>err.txt "par 2" &
        Raw command: "my_program par\ 1 <in.txt 2>err.txt "par 2" &"
        Number of arguments: 3
                argv[0]: "my_program"
                argv[1]: "par 1"
                argv[2]: "par 2"
                argv[3]: NULL
        Input: "in.txt"
        Err. output: "err.txt"
        Execute in background: Yes
$
```

Use the `main()` function of `example.c` as a base for the μshell. As the previous listing shows, the provided parser is capable of interpreting quotation marks, redirections of input/output streams, the escape character '\', and the background execution character '&'. In order to simplify the exercise, ignore redirections and background execution. The μshell will just execute internal commands, without obeying redirections nor calling `fork()` and `exec()`.

The internal commands to implement are the next ones:

1. `mypwd`, which will show the full path and name of the current directory.
2. `myls`, which must list the entries of the directory specified in the argument —the current directory if no directory name is provided—. In addition, if the optional argument `-l` is added, the listing will detail the next fields for every entry:
   - UID (user identifier) of the owner
   - GID (group identifier) of the owner
   - access permissions
   - file type
   - size
   - last modification date
   - name of the file or directory
3. `mymkdir`, which will create a new directory with the name passed as argument.
4. `myrmdir`, which will delete the directory passed as argument.
5. `mycd`, which will change the current directory, going to the directory passed as argument or, if no argument is passed, to the directory where the command interpreter was initially launched.
6. `mycat`, which will print the contents of the file passed as argument. Implement this command with memory-mapping.
7. `mycp`, which will copy, using `read()` and `write()`, the contents of the file passed as first argument in the file passed as second argument.
8. `myrm`, which will delete —well, just *unlink*— the file passed as parameter.
9. `exit`, which will end the execution of the μshell.

Structure the program in several `.c` and `.h` files, and write a `makefile` to ease the compilation process. Write, at least, one independent function for every internal command. We recommend using additional functions to perform tasks that will be required in different parts of the program[14].

[14] For example, the function `main()` will need to find out the current directory before starting to parse commands, while the command `mypwd` should find out the current directory each time it is executed.

In addition to the system calls of section 2, you will need the next system calls:

- Create a file:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int creat (const char *pathname, mode_t mode);
```

- Delete[15] a file:

```
#include <unistd.h>

int unlink (const char *pathname);
```

- Obtain the state of a file in a structure of the type `struct stat`:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int stat (const char *path, struct stat *buf);
int fstat (int fd, struct stat *buf);
int lstat (const char *path, struct stat *buf);
```

- Open a directory (returns some sort of directory descriptor):

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir (const char *name);
```

- Obtain the information of an entry of the directory whose descriptor is passed as parameter:

```
#include <dirent.h>

struct dirent *readdir (DIR *dirp);
```

- Close the directory whose descriptor is passed as parameter:

```
#include <sys/types.h>
#include <dirent.h>

int closedir (DIR *dirp);
```

[15] More precisely, `unlink()` decreases the count of entries that lead to the file, deleting the file only if the count reaches zero

- Create a directory:

```
#include <sys/stat.h>
#include <sys/types.h>

int mkdir (const char *pathname, mode_t mode);
```

- Delete a directory:

```
#include <unistd.h>

int rmdir (const char *pathname);
```

- Obtain the absolute path and name of the current directory:

```
#include <unistd.h>

char *getcwd (char *buf, size_t size);
```

- Change the current directory:

```
#include <unistd.h>

int chdir (const char *path);
```

Refer to the on-line manual —command `man`— for more detailed information. Remember that you will probably need to specify the section number —2 or 3— to prevent `man` from showing the pages of previous sections. For example, `man mkdir` shows the page for the `mkdir` command of the shell (section 1); while `man 2 mkdir` shows the page for the `mkdir()` system call (section 2).