EOPSY LAB6

Operation on files

Michał Skarzyński 293054

Table of content

TASK

Write a program which copies one file to another. Syntax:

copy [-m] <file_name> <new_file_name> copy [-h]

Without option -m use read() and write() functions to copy file contents. If the option -m is given, do not use neither read() nor write() but map files to memory regions with mmap() and copy the file with memcpy() instead.

If the option -h is given or the program is called without arguments print out some help information. Important remarks:

- use getopt() function to process command line options and arguments,
- the skeleton of the code for both versions (with read/write and with mmap) should be the same, but in some place either copy_read_write(int fd_from, int fd_to) or copy_mmap(int fd_from, int fd_to) should be called,
- check error codes after each system call.

ARGUMENTS

The first thing to do is process the arguments that appear in the console. For this purpose I used the function getopt which executed in the while loop until it returns -1 will sequentially get arguments from the console.

```
int getopt(int argc, char * const argv[],
                  const char *optstring);
```

One of the most important arguments of this function is optstring, which sets the format of such arguments. In our case, we want to be able to specify the -h argument and the -m argument, so the format will look "hm", I will not treat the file name as an additional value for the -m command mainly because it will be easier to process it later and it will give us an additional option whose file names will be given earlier than the -m argument (I think it is more convenient for the programmer if he forgot, for example, and there is no way in this program to understand such a structure differently).

The analysis of such arguments is quite simple. We can create a MODE variable that will have a value of 0 when no parameters are given (where the parameters are obviously -m or -h) but the names of two files are given, which means that program will perform copying using read and write .

The variable mode can be set to 1 when help is displayed - when the -h parameter is given without additional arguments, and when no parameters and no arguments are given, which means that help will be displayed.

The variable can be also 2, when parameter -m and two file names are specified. Then program will use mmap and memcpy.

In every other case MODE will have value -1, and appropriate error message will be displayed.

As I wrote earlier, file names will not be strictly taken as arguments for the -m parameter, but will be parsed after passing through the entire console parameter line, using the getop function mechanism which holds the optind variable by which we can find out which arguments given to the program have not been parsed.

OPENING FILES

I will use the open function to get file descriptors of files.

```
int open(const char *pathname, int flags);
```

For the source file I will use arguments:

```
open(input, O_RDONLY);
```

For the output file I will use arguments:

```
open(output, O_RDWR | O_CREAT | O_TRUNC);
```

READ AND WRITE CASE

To load data using the read function at the very beginning we need to create a buffer holding data chunk.

```
data = malloc(size);
```

Then in the while loop we will load the appropriate amount of data into this buffer using the read command, of course checking errors everywhere, we will also write the amount of data that we were able to load in the variable bytes when it is equal to 0, which means that we arrived at the end of the file, while when it will be less than zero that means there was an error in the writing.

```
	bytes = read(inp_fd, data, size);
		if (bytes < 0) {
		perror("Reading data failed");
		free(data); // dealocating buffer

		return -1; // error during reading data
		}
		else
		{
			if (bytes == 0) // EOF
			break; // break coping loop no more data to read
		}
```

At this point, we can proceed to write data from this buffer. It may happen that not all data will be saved at once, so we will have to use another function while checking to make sure that we have saved everything, or that there was an error in saving.

```
ptr = data; // begin
	 end = data + bytes; // end

		while (ptr < end) { // ensure that whole buffer will be written if
possible
			bytes = write(out_fd, ptr, (size_t)(end - ptr)); // write to
output file (end-ptr) starting at ptr
			if (bytes <= 0) { // number of written bytes is <=0 means error in
writing

			free(data); // dealocate buffer

			perror("Writing data failed");
			return -1;
			}
			else
			{
			ptr += bytes; // in case we couldn't write whole buffer shift
pointer
			}
		}// write while
```

If everything went well, of course at the end we deallocate the buffer

```
	free(data); // dealocate buffer
```

The file should now be successfully copied.

MMAP CASE

If we want to map the destination file to the memory area, we must first make sure that we choose its size properly. To do this, at the very beginning we will get information about the source file and check its size and then trim the size of the resulting file accordingly.

```
fstat(inp_fd,&file_status);
size_t filesize = file_status.st_size;
ftruncate(out_fd, filesize);
```

Our filesize variable at this point will behave more like information, how MUCH we still have to load the data.

In this case our loop will work different, namely until the filesize (amount data to load) is greater than zero.

If it turns out that the that the amount of data we need to load is smaller than our chunk size, we can reduce the chunk size to the amount we need to load and set amount of data we need to load to zero. Otherwise, when filesize is larger than the assumed chunk size, we will reduce the amount of data we need to load by the size of the chunk.

```
while(filesize > 0)
       {
              if(filesize < size)
              {
              size = filesize;
              filesize = 0;
              }
              else
              {
               filesize = filesize - size;
              }
. . .
```

When using the mmap function, we will create an additional offset variable that tells us where we are in the source file.

An interesting aspect is the fact that the mmap function works a bit like malloc so you don't have to worry about manual buffor allocation.

```
if((data=mmap(NULL,size,PROT_READ,MAP_SHARED,inp_fd,offset))==MAP_FAILED)
              {
              perror("mmap error, mapping input");

                     if((munmap(data,size))==-1)printf("munmap error : input");
              return -1;
              }
```

However, we must treat our variable as dynamically allocated and in case of an error take care of freeing memory using the munmap function.

If everything went well at this point, we maped the appropriate chunk of source file, moved by offset to our buffer variable.

We proceed with the source file anbalogically and map it to another variable, in my case it is the variable dst.

```
            if((dst=mmap(NULL,size,PROT_READ |
PROT_WRITE,MAP_SHARED,out_fd,offset))==MAP_FAILED)
            {
            perror("mmap error, mapping output");
                  if((munmap(dst,size))==-1)printf("munmap error : output");
               return -1;
            }
```

In this case, the memory will have slightly different flags.

Specifically, the PROT_WRITE flag that allows us to write to this page.

All we have to do now is perform the memcpy function.

```
memcpy(dst,data,size);
```

And then we can throw this data chunk from memory.

```
            if((munmap(dst,size))==-1)printf("munmap error : output");
            if((munmap(data,size))==-1)printf("munmap error : input");
```

Of course, we also need to move in the file by a given length, ergo should be increment the offset.

```
            offset = offset + size;
```

If all went well after the loop, our file will be copied successfully.

RESULTS

```
osboxes@osboxes:~/Desktop/EOPSY/LAB6$ ./copy
copy [-m] <file_name> <new_file_name>
copy [-h]
Without option -m use read() and write() functions to copy file contents.
If the option -m is given, maps files to memory regions with mmap()
        and copy the file with memcpy() instead.
If the option -h is given or the program is called without arguments
        print out some help information.
osboxes@osboxes:~/Desktop/EOPSY/LAB6$
```

```
osboxes@osboxes:~/Desktop/EOPSY/LAB6$ ./copy -h
copy [-m] <file_name> <new_file_name>
copy [-h]
Without option -m use read() and write() functions to copy file contents.
If the option -m is given, maps files to memory regions with mmap()
        and copy the file with memcpy() instead.
If the option -h is given or the program is called without arguments
        print out some help information.
osboxes@osboxes:~/Desktop/EOPSY/LAB6$
```

```
osboxes@osboxes:~/Desktop/EOPSY/LAB6$ ./copy -h -h
To many arguments
osboxes@osboxes:~/Desktop/EOPSY/LAB6$ ./copy -h -m
To many arguments
osboxes@osboxes:~/Desktop/EOPSY/LAB6$ ./copy -h -a
./copy: invalid option -- 'a'
Unknow argument
osboxes@osboxes:~/Desktop/EOPSY/LAB6$ ./copy -h a
To many arguments
osboxes@osboxes:~/Desktop/EOPSY/LAB6$ ./copy -h a.txt
To many arguments
```

```
osboxes@osboxes:~/Desktop/EOPSY/LAB6$ ./copy file1.txr
Unknow combination of arguments
osboxes@osboxes:~/Desktop/EOPSY/LAB6$ ./copy -m file.txt
Missing argument: file name
osboxes@osboxes:~/Desktop/EOPSY/LAB6$ ./copy -m file.txt file2.txt file3.txt
Missing argument: file name
osboxes@osboxes:~/Desktop/EOPSY/LAB6$ gcc copy.c -o copy
osboxes@osboxes:~/Desktop/EOPSY/LAB6$ ./copy file.txt
Wrong combination of arguments
osboxes@osboxes:~/Desktop/EOPSY/LAB6$ ./copy -m file.txt
Missing argument: file name
osboxes@osboxes:~/Desktop/EOPSY/LAB6$ ./copy -m file.txt file2 file3
To many arguments: file name
osboxes@osboxes:~/Desktop/EOPSY/LAB6$
```

```
osboxes@osboxes:~/Desktop/EOPSY/LAB6$ ./copy file.txt a.txt
File input: file.txt
File output: a.txt
file.txt: No such file or directory
osboxes@osboxes:~/Desktop/EOPSY/LAB6$
```