operating systems lab - week 9: exercise

Nicolo Colombo

November 24, 2020

In this lab, you will implement a program that creates a message-passing half-duplex pipe between a parent process and a child process. The child will read a string from the terminal and, if a *word*[1] contains numerical characters, i) interpret it as an integer, e.g. `one12two` will be converted into the integer 12, and ii) send the obtained integer to its parent through the pipe. The parent will i) read the integers sent by the child, ii) compute their sum, and iii) print the obtained value on the terminal.

To pass formatted messages through the pipe, you will use *high-level* I/O functions that are defined in `stdio.h` and refer to the pipe *reading* and *writing* ends through pointers to their *file handle*, i.e. pointers to `struct`-objects of type `FILE`.

To write the function that reads words from the terminal and converts them to integers, you can reuse parts of `getWord`, given in `Exercise lab-sheet 6` , and `getInt`, given in `intCalc.c` .

Please try to complete this lab sheet before attempting Lab Quiz 9 , as some of the quiz questions may require you to run, comment, or modify the programs you are asked to write.

Set up

Depending on your OS, use the following instructions to connect to `linux.cim.rhuk.ac.uk`:

Unix Open the terminal and run

```
ssh yyyyxxx@linux.cim.rhul.ac.uk
```

where `yyyyxxx` is your college username, and enter your password to access the teaching server.

Windows Launch the Windows SSH client `puTTY` [2], enter the following

```
linux.cim.rhul.ac.uk
```

in the empty field *Host Name (or IP address)* and click on *Open*. The client opens a new window where you are required to enter your college user name `yyyyxxx` and password.

Once logged in, you should be able to see the content of and navigate in your home directory using the standard UNIX commands, e.g. `ls`, `cd`, `cp`. Go to the `CS2850labs` directory[3] and run the command

```
$mkdir week9
```

to create a new sub-directory called `week9`. We suggest you save and compile all the programs you write this week in this directory.

Use a command-line text editor, e.g. emacs, nano, or vimto open, edit and save your programs. The advantage of command-line editors is that they can be used in a non-graphical SSH session. [4] You can create a new C file or open an existing C file, `file_name.c`, by running the command

```
$editorName file_name.c
```

[1] We let a word be defined as a group of character ending with an empty-space character `' '`

[2] `puTTY` should be installed on all department's machines. If you work on your own Windows machine you can download it at download puTTY and install it as explained.

[3] The parent directory you have created for the first week's lab

[4] If you do not want to open and close the editor every time you modify and save your code you can open a new SSH session and use two shell windows simultaneously.

where, e.g. `editorName` is `vim` We suggest you save separate files for all single parts of this exercise and follow the name suggestions given in each section. [5]

Compile your C code by running

```
$clang -o file_name file_name.c
```

and run the corresponding binary files `file_name` through

```
$./file_name
```

For debugging, we suggest you use the free debugging tools of valgrind, which is already installed on the teaching server `linux.cim.rhul.ac.uk`. To check your code, you just need to run the command

```
$valgrind ./file_name
```

and have a look at the messages printed on the terminal.

1 A *half-duplex* pipe to send formatted messages

In this section, you will write a program, `integerPipe.c`, where:

- a child process and a parent process are created by calling fork
- the child process will convert an input string of words into a series of integers, e.g.

  ```
  ...  one 1 two2 three34four and 5five6six7seven ...
  ```

  will correspond to the integers

  1 2 34 567

  and send them to its parent as separate *formatted* messages through a half-duplex pipe
- the parent will read the integers sent by the child, compute their sum and print the sum on the terminal before exiting

1.1 Create a pipe in the main process

Start by creating a 2-entry integer array declared as

```
int fd[2];
```

whose entries will be loaded with the file descriptors associated with the *reading* and *writing* ends of the pipe (in the order). Create a new channel by writing

```
int good = pipe(fd);
```

that requires the operating system to set up the message-passing pipe. Check whether the pipe has been created successfully by looking at the value of `good` and at the entries of `fd`.

1.2 Convert the file descriptor into a file handle

Use `fdopen`, which is defined in `stdio.h` [6], to convert each file descriptor into a file handle, i.e. let

```
FILE *readEnd = fdopen(fd[0], "r");
```

to obtain the file handle of the *reading* end of the pipe and

```
FILE *writeEnd = fdopen(fd[1], "w");
```

to obtain the file handle of the *writing* end of the pipe. You will need `readEnd` and `writeEnd` to use the formatted I/O functions `fprintf` and `fscanf`.

1.3 Create a child process and close the unused ends of the pipe

Call `fork` to create a child process that will inherit both pointers to the pipe file handles, `readEnd` and `writeEnd`. In the child, who will be sending the messages, close the *reading* end of the pipe by calling

```
fclose(readEnd);
```

In the parent, who will read the child's messages, close the *writing* end of the pipe by calling

[5] This is mainly because some of the Moodle quiz questions may refer to single pieces of code through the suggested file names.

[6] See Section 13.4 of the GNU online manual for more details about `fdopen` and similar functions.

```
fclose(readWrite);
```

Note that you can use the standard library function `fclose`, which is also defined in `stdio.h` [7], because you have now obtained the pointer to the pipe file handle.

1.4 The child process

The child is now able to send messages to the parent through the pipe defined by writing on the file pointed by `writeEnd`. To select the child and the parent processes, introduce an `if (pid == 0)-else` statement, where `int pid` is the return value of `fork`. In the `if` part, i.e. in the child process, perform the following operations:

- Define and set to 0 an integer *stopping* variable that you will use to stop reading from `stdin` when you reach the end of the string.
- Extract the integers to be sent to the parent from the words in the user input string by defining a dedicated function

  ```
  int getInteger(int *end)
  ```

  where:

  - the `end` argument is the address of the stopping variable and

  - the return value is the integer extracted from the current word.
- Include the call of `getInteger` in a `while (end == 0)` loop to keep reading words and converting them to integers until `getInteger` sets the value of the stopping variable to 1
- After extracting a new (valid) integer from the input, send it to the parent by calling another dedicated function

  ```
  int writeMessage(FILE *pipeEnd, int n, int end)
  ```

  where:

  - the `pipeEnd` argument is the file handle of the *writing* end of the pipe,

  - the `n` argument is the integer to be written in the pipe,

  - the `end` argument is used to send a stopping signal to the parents after the last message and

  - the return value is the integer that has been sent
- Outside the `while` loop, close the *writing* end of the pipe and exit by adding a `return` statement.

1.5 The parent process

In the `else` part of the `if (pid == 0)-else` statement, i.e. in the parent process, perform the following operations:

- Declare and set to 0 a stopping variable, e.g. write

  ```
  int end = 0;
  ```
- Declare and set to 0 another integer variable, called `sum`, to keep track of the sum of the integers received from the child, i.e. write

  ```
  int sum = 0;
  ```
- Define a `while(end == 0)` loop where you iteratively call a reading function

  ```
  int readMessage(FILE *pipeEnd, int *sum)
  ```

  where:

  - the `pipeEnd` argument is the file handle of the *reading* end of the pipe,

  - the `sum` argument is the address of the integer variable that keeps track of the sum of the received integers (see above), and

  - the return value is 1 if the received message was a *stopping* message and 0 otherwise.

[7] See Section 12.4 of the GNU online manual for more details about `fclose` and similar functions.

- Outside the `while` loop, close the *reading* end of the pipe, call `wait(NULL)` to make the parent wait for the child to terminate, print on the terminal the sum of the received integers by calling

  ```
  printf("sum = %d \n", sum);
  ```

  and exit by writing a `return` statement.

1.6 Expected output

On `linux.cim.rhul.ac.uk`, your program should produce the following output:

```
cim-ts-node-03$ ./a.out

sum = 0
cim-ts-node-03$ ./a.out
123
sum = 123
cim-ts-node-03$ ./a.out
1 23
sum = 24
cim-ts-node-03$ ./a.out
1 two 3
sum = 4
cim-ts-node-03$ ./a.out
1 two2 and three
sum = 3
cim-ts-node-03$ ./a.out
... one 1 two2 three34four and 5five6six7seven ...
sum = 604
```

Note This week's quiz will ask you to run the program you wrote in this section. Be sure that your program writes on `stdout` as shown in the examples above. In particular, check that

- the value of the sum is correct
- the output is written in the correct format (with all empty spaces and new line character needed)
- no extra messages or values are printed on the terminal

2 Functions

2.1 `getInteger`

Try to write your version of `getInteger`, the function called by the child process to parse the user input as described in Section 2.2. We suggest you start by combining the code from `getWord` you used in `Exercise lab-sheet 6` and `getInt` used in `intCalc.c` . Be sure that the argument and the return values are the same as in the function declaration given in Section 2.2.
If you are stuck, you can use the following implementation

```
int getInteger(int *end){
  char buf[MAXCHARS];
  int j = readOneWord(buf, end, MAXCHARS);
  if (j < 1) return 0;
        char c;
  int n = 0, i = 0;
        while ((c = buf[i++]) != ' ' && c!= '\0'){
                if(c<= '9' && c >= '0'){
                        n  = n * 10 + c - '0';
                }
```

```
        }
        return n;
}
```

where `readOneWord` is the following simplified version of `getWord` from `Exercise lab-sheet 6` :

```
int readOneWord(char *buf, int *end, int maxLength){
        int j = 0;
        char c;
        while(((c = getchar()) != ' ') && (j < maxLength) && (c !='\n')){
    buf[j] = c;
                j++;
        }
  buf[j] = '\0';
        if (c == '\n') *end  = 1;
        return j;
}
```

2.2 `writeMessage` and `readMessage`

The writing and reading subroutines mentioned in Sections and are thin wrappers of these two functions defined in the standard library `stdio.h`[8]:

`int fprintf (FILE *stream, char *format, ...  )`

where:

- the `stream` argument is the pointer to the file handle of the output file

- the `format` argument specifies the format in which the following argument(s), here denoted by . . . , will be printed[9], and

- the return value is the number of characters printed.

In this case, you will have `writeEnd` as the first argument, `%d` as the second argument, and the integer to be put in the channel as the third argument.

`int fscanf (FILE *stream, char *format, ...)`

where:

- the `stream` argument is the pointer to the file handle of the input file

- the `format` argument specifies the format in which the content of the file will be assigned to the following argument(s), here denoted by `...`. [10]

- the return value is the number of successful assignments.

In this case, you will have `readEnd` as the first argument, `%d` as the second argument, and `&n` as the third argument, where `n` is the local variable you use to store the received integers.

Here is a possible implementation of the writing and reading functions you can call from your `main`

```
int writeMessage(FILE *pipeEnd, int n, int end){
  if (n > 0){
    fprintf(pipeEnd, "%d\n", n);
    if (end == 1){
      fprintf(pipeEnd, "%d\n", -1);
    }
  }
  return n;
}
```

[8] See Section 12.12.7 of the GNU online manual and Section 12.14.8 of the GNU online manual for more details.

[9] Similarly to `printf`

[10] Again, the format specification is as for `fscanf` and `printf`, but `...` should be a list of local addresses where the content of the file should be copied. For example, if you want to write the first integer in `stream` to the local variable `n`, you should set `format` to `'%d'` and the following argument to `&n`

```
int readMessage(FILE *pipeEnd, int *sum){
  int n = -1;
  int end = 0;
  fscanf(pipeEnd, "%d", &n);
  if (n >= 0){
    *sum = *sum + n;
  }
  else end = 1;
  return end;
}
```