operating systems lab - week 11: exercise

Nicolo Colombo

December 7, 2020

In this lab, you will implement a program that create two concurrent threads for merging the content of two input files. Each thread will read from one of the two input files and write in the shared output file. You will use the POSIX thread-handling functions defined in `pthread.h` to create and synchronize the procedures. After both threads are done, the main program will read the content of the output file and print it on the terminal.

Two sample input files are available on `files.zip` and consists of a set of words[1] that may or may not contain *numerical characters*, e.g. `'1'` , `'2'`, ..., `'0'`. The content of the files is shown in Section 3 but we suggest you use the original ones instead of copying them from this document.

The two procedures will:

- extract the numerical characters and ignore all non-numerical characters, e.g. `'a'`, `';'`, or `'B'`, in the words of the input file,
- convert each word into the corresponding integer, e.g. `"one1two23Three-four4"` will be converted into the integer 1234 (*thousand two-hundred thirty-four*), and
- copy the integer [2] into the output file

The main program will read all integers in the output file and print them on `stdout` by calling `printf` with the integer-format specifier `%d`.

This week there are no summative quizzes. This is to let you work on Assignment 3: C mini project, but we suggest you try to complete this lab-sheet before starting your project, as the last question of the assignment will ask you to implement a multi-thread program.

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` [3], 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.

[1] Groups of characters separated by white spaces, `' '`.

[2] Using the formatted-I/O functions defined in `stdio.h`

[3] `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.

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[4] and run the command

```
$mkdir week11
```

to create a new sub-directory called `week11`. 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. [5] You can create a new C file or open an existing C file, `file_name.c`, by running the command

```
$editorName file_name.c
```

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. [6]

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 Merging files with 2 threads

To focus on this week's subject, i.e. *thread-programming*, an example of a program that merges two files with no threads is given in Section 2. The provided single-thread program reads the two files one after the other, converts their content into integers and prints them on an output file, `outputSingle.txt` and on the terminal. After reading Section 2, copy the code and two input files, e.g. `input1.txt` and `input2.txt` in `files.zip` , into the same directory and compile and run the program. The multi-thread version of the provided code you will implement in this lab should perform the same task and have a similar output on both the file and the terminal. Have a look a the examples in Section 3 to get a general idea.

The code in Section 2 is written in a way that it is almost ready for *parallelization*. The main steps for transforming `singleThread.c` into a multi-thread program are outlined in Video 11 . As a guideline you can also copy the code of the examples shown in Slides 11 and try to adapt them to solve required task. Alternatively, you may follow the step-by-step approach suggested below:

- copy the code given in Sections 2.1, 2.2, and 2.3 into a new file called `multiThreads.c`
- copy the headers, definitions, and functions calls that you need to create and handle a pthread from one of the examples in Slides 11
- check the syntax and argument list of `pthread_create` and `pthread_join` on Slides 11 or have a look at the references suggested there
- in `main`, replace each of the `writeIntegers` calls with a call of `ptrhead_create`
- move the second call of `fopen` (and the corresponding call of `fclose`) into the reading subroutine `printIntegers`
- add two calls of `pthread_join`, as in the third example in Slides 11 , to make the updated value of `nIntegers` available to the main program

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

[5] 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.

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

- try to compile your program with
  ```
  gcc -Wall -Werror -pthread
  ```
  to see what happens and fix all errors. Your program is not thread-safe for the moment but you should be able to compile and run it

- outside `main`, define two *mutex* variables by writing
  ```
  pthread_mutex_t file_mutex=PTHREAD_MUTEX_INITIALIZER;
  pthread_mutex_t nIntegers_mutex=PTHREAD_MUTEX_INITIALIZER;
  ```
  and use `file_mutex` the shared output file and `nIntegers_mutex` to protect the counting variable, `nIntegers`.

- as in the fourth example of Slides 11 , define the opportune critical regions in `printIntegers` so that the two procedure do not attempt to write on the output or change the value of `nIntegers` at the same time

2 A file merging program with no threads

The single-thread program, `singleThread.c`, presented in this section performs the following tasks:

- reads two input files, e.g. `input1.txt` and `input2.txt` in `files.zip` , one after the other
- converts their content into a series of integers
- prints the integers on an output file, e.g. `outputSingle.txt` also in `files.zip` , and on the terminal

The structure of the code is unnaturally complicated to make it ready to be parallelized. In particular, the integers extraction and file-writing operations are performed by a dedicated function called `printIntegers`. The integers are not printed directly on the terminal, as it would be more convenient. To write on `stdout`, we have defined a `while` loop where the output file is read integer-by-integer and printed on the terminal by calling `printf`. Have a look at the code shown in the next sections and try to understand how it works. Note that `main` (Section 2.1) does not require to include the `pthread.h` and is very similar to other programs that you have written so far. The file-reading subroutine (Section in 2.2) is what you will need to modify to make your program parallelizable and thread-safe. Note that all arguments are passed to the function through an object of type `struct args` defined in `main`. The reason is that the thread-creation function `pthread_create` requires the subroutine associated with the thread task to accept a *single* parameter. Finally, the auxiliary functions called for reading and parsing the input (Section 2.3) are functions that you have already implemented and used elsewhere.

2.1 `singleThread.c`: main

```
#include <stdio.h>
#include <unistd.h>

#define MAXCHARS 200
#define NINTEGERS 15

struct args{
        int *nIntegers;
        char *inputName;
        char *outputName;
};

int getInteger(int *end, FILE * fileIn);
int getWord(char *buf, int *end, int maxLength, FILE *fileIn);
int printIntegers(struct args *argT);
```

```
int main(){
        int nIntegers = 0;
        struct args argT1 = {&nIntegers, "input1.txt", "outputSingle.txt"};
        struct args argT2 = {&nIntegers, "input2.txt", "outputSingle.txt"};
        FILE *f = fopen("output.txt", "w");
        fclose(f);
        printIntegers(&argT1);
        printIntegers(&argT2);
        int n = 0;
        FILE *outputFile = fopen("outputSingle.txt", "r");
        for (int i = 0; i < nIntegers; i++){
                fscanf(outputFile, "%d", &n);
                if (i == (nIntegers - 1)) printf("%d \n", n);
                else printf("%d ", n);
        }
        fclose(outputFile);
        return 0;
}
```

2.2 singleThread.c: reading subroutine

```
int printIntegers(struct args* argT){
        FILE *fileIn = fopen(argT->inputName, "r");
        FILE *outputFile = fopen(argT->outputName, "a");
        int end = 0, n;
        while (!end){
                n = getInteger(&end, fileIn);
                if (n && (*(argT->nIntegers)<NINTEGERS)){
                        fprintf(outputFile, "%d ", n);
                        *(argT->nIntegers) = *(argT->nIntegers) + 1;
                }
        }
        fclose(outputFile);
        fclose(fileIn);
        return 0;
}
```

2.3 singleThread.c: auxiliary functions

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

int getInteger(int *end, FILE *fileIn){
        char buf[MAXCHARS];
        int j = getWord(buf, end, MAXCHARS, fileIn);
        if (j < 1) return 0;
        char c;
```

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

3 Examples

3.1 input files

3.1.1 input1.txt

```
here are the numbers of the first file:

one,

"2" = two,

three

and "4" = four.

And there is also these one:
0902,
0904,
0906,
0908
01902,
01904
01906,
01908
011902,
011904
011906,
011908,
0111902!
end
```

3.1.2 input2.txt

```
here are the numbers of the second file:

one="1"

two,

three = "3"

and four, and "5",i.e. five

And there is also this one:
~0oooo901
```

```
~0oooo903
~0oooo905
~0oooo907
~0oooo909
~0oooo9011
~0oooo9013
~0oooo9015
~0oooo9017
~0oooo9019
~0oooo90111
~0oooo90113
~0oooo90115
~0oooo90117
~ 0oooo90119


-------------
```

3.2 output of `singleThread.c`

3.2.1 `outputSingle.txt`

```
2 4 902 904 906 908 1902 1904 1906 1908 11902 11904 11906 11908 111902
```

3.2.2 `stdout`

```
cim-ts-node-01$ ./a.out
2 4 902 904 906 908 1902 1904 1906 1908 11902 11904 11906 11908 111902
cim-ts-node-01$
```

3.3 output of `multiThreads.c`

The output depends on the specific run of the program. Here are two examples of what you may see on the terminal.

```
cim-ts-node-01$ ./a.out
2 4 902 904 906 908 1902 1904 1906 1908 11902 11904 11906 11908 111902
cim-ts-node-01$ ./a.out
1 3 5 901 903 905 907 909 9011 9013 9015 9017 9019 90111 90113
```

Try to change the number of integers printed on `output.txt` or suspend the thread for a random amount of time for each read to see what happens.