OS Aspects

Fonaments dels Computadors – Grau en Intel·ligència Artificial – 2021-2022 Q1

Facultat d'Informàtica de Barcelona – Departament d'Arquitectura de Computadors

In this session we will exercise around the services that the OS provides to users and applications. Download the program set from:
http://docencia.ac.upc.edu/FIB/GIA/FC/documents/Lab/S12/S12-FC_GIA.tar.gz.

The execution environment

When a process is executed, it inherits a series of properties from its "parent" process. The parent process is the process that creates another one. The new process is named the "child" process. Those properties include…

- The Current Working Directory (CWD), the directory where the two processes will open files by default
- The files opened in the parent process are also available in the child process. This means that the standard files stdin(0), stdout(1), and stderr(2) will also be available to the child process. These numbers are a sort of "private" index of reachable devices for the process, also known as "file descriptors".

Tool "strace"

Throughout this Lab Session, use the command line "strace -e read -o NAME.txt COMMAND" to record, in the file NAME.txt, the trace log of read systemcalls executed by COMMAND. Obivously, update NAME and COMMAND to introduce the filename and command you want to use. Use the output of this tool to complement the analyses of the exercises.

Familiarize with the "mycat" program

In this session we will use a version of the "cat" program. You already know that "cat" takes a number of files, reads them one after another, and writes their contents to its standard output – stdout(1). The implementation that we present here uses the FILE functions from <stdio.h>: *fopen*, *fgetc*, *putchar*, and *fclose*.

mycat.c:

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

extern void mycat (char * filename);

int main(int argc, char * argv[])
{
        int i;
        for (i = 1; i < argc; i++) {      // Loop over all files provided
           mycat(argv[i]);                // as arguments
        }
        return 0;
}

void mycat (char * filename)
{
        FILE * file = fopen(filename, "r");     // open the file
```

```
        if (file == NULL) {                    // indicate the error, if so
                perror("fopen");
                fprintf(stderr, " ... while opening file \"%s\"\n", filename);
                return;
        }
        int c = fgetc(file);  // read character by character from the open file
        while (c != EOF) {    // while no end-of-file is reached
                putchar(c);   // and put the character to the standard output
           c = fgetc(file);    // read the next character
        }
        fclose(file);       // close the file, as it will not be used any more
}
```

Compile the programs (use the command "make"), and write your answers to the following exercises in the "answers.txt" file for this session.

Exercise 1

Let's determine if "mycat" is equivalent to "cat" when given a number of files as arguments, by doing this experiment:

```
$ ./mycat   quijote.txt  saeta.txt  > output1.txt
$   cat     quijote.txt  saeta.tst  > output2.txt
$ diff output1.txt output2.txt
```

- *Explain what the previous command lines do. For that, check the contents of the input files (quijote.txt and saeta.txt), and the output files*

- *How do you know that the two files output1.txt and output2.txt have the same content? (hint: see man diff, and what it should be its result in case the files have the same content)*

Exercise 2

Now let's determine if "mycat" and "cat" behave the same, when we do not provide an argument to them. Try these command lines:

```
$ ./mycat          # no arguments are provided
$  cat             # no arguments are provided
```

- *Do they behave in the same way? Why?*

Now try:

```
$ ./mycat <saeta.txt   # no arguments, stdin is changed to saeta.txt
$   cat   <saeta.txt  # no arguments, stdin is changed to saeta.txt
```

Why do you think they behave differently? Explain your findings

The "mycat-buffered" program

In this section, we will use another version of the "cat" program. It also takes a number of files, and writes them to the stdout(1) output channel, but it uses a different set of libc services for reading and writing: *fread*, and *fwrite*. It also checks for the end-of-file condition using *feof*.

It is called "mycat-buffered" because it implements "buffering" at the application level. Instead of using fgetc/putchar to read/write a single character at a time, it uses fread/fwrite to read/write a number of characters for each iteration. The characters read from the source file are saved in a "buffer", and then written to the target output file.

```
mycat-buffered.c:
#include <stdio.h>
#include <stdlib.h>

extern void mycat (char * filename);

int main(int argc, char * argv[])
{
        int i;
        for (i = 1; i < argc; i++) {          // Loop over all files provided
           mycat(argv[i]);                     // as arguments
        }
        return 0;
}


#define BUFSIZE 32              // buffer used to hold the characters read from
char buffer[BUFSIZE];           // the source file, and to be written to the target

void mycat (char * filename)
{
        size_t res;
        FILE * file = fopen(filename, "r");
        if (file == NULL) {
             perror("fopen");
             fprintf(stderr, " ... while opening file \"%s\"\n", filename);
             return;
        }
        while (!feof(file)) {
             res = fread(buffer, 1, BUFSIZE, file);    // fills the buffer (fread)
             if (res < 0) {                            // error control
                  perror("fread");
                  fprintf(stderr, " ... while reading from file \"%s\"\n", filename);
                  return;
             }
             printf ("\n\nRead %ld\n\n", res);
             fwrite(buffer, 1, res, stdout);          // writes the buffer to stdout
        }
        fclose(file);
}
```

Exercise 3

Now use "mycat-buffered", and compare it to "cat". Try:

```
$ ./mycat-buffered   saeta.txt
$ cat     saeta.txt
```

Do you see a difference on the output of these two commands? Explain why their output is different.

We have provided the "mycat-buffered" program with a sentence we used to "debug" it: a sentence containing a "printf" to display the amount of characters read by each *fread* call.

Exercise 4

Try to understand how the fread *calls get the data from the input file, and answer:*

- *What is the buffer size?*
- *How many characters are read by each* fread *call?*
- *Is there a pattern in the buffer sizes read? What happens with the last one?*

Exercise 5

As the "debug" implemented with printf breaks the "mycat-buffered" program, making it different from "cat", let's try to solve this issue. As we know that we have "stdout" for the output of the program and "stderr" for the "error" information, let's change the program so that the "debug" information gets written onto "stderr". Do:

- *Replace the 'printf ("\n\nRead %ld\n\n", res);' sentence by*

```
fprintf (stderr, "\n\nRead %d\n\n", res);
```

- Now you can execute:

```
$ ./mycat-buffered saeta.txt 2> debug.out
```

Explain what is the behaviour of your version of the program now:

  - *Where the "saeta*.txt" file is now written?
  - *Where the "Read 32" messages are written?*
  - *Is this version useful now?*

- Or you can now execute:

```
$ ./mycat-buffered quijote.txt saeta.txt  2> debug.out > output3.txt
$ diff output1.txt output3.txt
```

Explain what is the behaviour of your version of the program now:

  - *Where the joint "quijote.txt" and "saeta*.txt" file are now written?
  - *Where the "Read 32" messages are written?*
  - What is the result of the "diff" command? Have the output1.txt and output3.txt files the same contents?

Upload the Deliverable

To collect all contributions to this deliverable, you can use the tar command as follows:

```
# tar  czvf   session12.tar.gz   answers.txt   *.c    Makefile
```

Now go to RACO and upload this recently created file to the corresponding session slot.