ECE PARIS · LYON
ÉCOLE D'INGÉNIEURS

ING4 SI03

Operating Systems

Lab 4 : Advanced Input/Output

Author(s) :
Romain BRISSE
Hyunjae LEE

Teacher(s) :
Mr. KHOURY

I attest that this work completely comes from the author(s) mind(s). If it is otherwise you will be able to find the source in the concerned section.

Paris, 21/10/2018

LaTeX

Table des matières

0.1 File Descriptors

0.1.1 The cat function

The cat function is used to concatenate files and print the result on the standard output. Within the command given by this subject we find this function written in the form of *inputFile* > *outputFile*, which means that the > is used to change the standard output. When using this command, you can precise any type of input or output that you want, if nothing is provided, the standard input (Keyboard) and standard output (Terminal) will be used.

0.1.2 The dup2 function

The dup function is used to duplicate a file descriptor. In this cas we will use the dup2 function in order to be able to specify a new file descriptor.

0.1.3 Changing the standard output to a file

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char **argv)
{
        FILE *fptr;
        FILE *fptr2;
        char string[20] = "";

        //ptr = fopen("fileopen","mode");
        fptr = fopen("text1","r");
        fptr2 = fopen("text2","w");

        //redirect standard output to text2
        dup2(fileno(fptr2),STDOUT_FILENO);

        if(fptr!=NULL)
        {
                while(fgets(string,20,fptr) != NULL)
                {
                        printf("%s",string);
                }
        }
        else
        {
                printf("error");
        }


        fclose(fptr);
        return 0;
}
```

FIGURE 1 – How to change the standard output

(a) using dup2

In this program, I simply open two files, one in read mode and one in write mode. Then, right before reading the first file and printing it's result using *printf()* that prints to standard output, I redirect the standard output to the second file, opened in write mode.

0.2 Pipes

0.2.1 Piping two commands

In order to correctly understand what the *ps* function, and the *more* function do when they are used together with a pipe, we first need to decompose their actions :

- ps is used to report a snapshot of the current processes.
- a,u, & x are different arguments added to ps.
  - a –> for all users
  - u –> display the process user/owner
  - x –> also show all processes not attached to a terminal
- | is the ASCII character used to represent a pipe
- more is a function that allows you to display results on more than one screen. It is a filter that allows better viewing of the results by letting the user choose when to see the next result.

0.2.2 Programming the *ps aux | more* command

File Edit View Search Terminal Help

```c
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int main(int argc, char **argv)
{
        //system("ps aux | more");


        int pipefd[2];
        char buf;
        char command[13];

        if(argc != 1){fprintf(stderr,"not the right number of arguments");}

        //creation of the pipe + checking for an error
        if(pipe(pipefd)==-1){perror("pipe"); exit(EXIT_FAILURE);}

        if(fork()==0) //Child process
        {
                //We will run the more operation from here

                //First, close the write end of the pipe
                close(pipefd[1]);
                //then make the standard input to be the rend end of the pipe
                dup2(pipefd[0],STDIN_FILENO);
                //finally execute the more function
                system("more");
                //close the read end of the pipe
                close(pipefd[0]);

                exit(EXIT_SUCCESS);
        }
        else //Parent process
        {
                //We will run the ps aux operation from here

                //first, we close the pipe read end
                close(pipefd[0]);
                //change standard output
                dup2(pipefd[1],STDOUT_FILENO);
                //then, execute the ps aux function
                system("ps aux");
                //close the write end of the pipe
                close(pipefd[1]);
                //wait for the child to finish his operation
                wait(NULL);

                exit(EXIT_SUCCESS);
        }
```

1,1 Top

FIGURE 3 – Programming of ps aux | more

(a) using a pipe

0.3 Non-blocking Calls

0.3.1 Uncommented fcntl() call

```
romain@tuxtop:~/Bureau/s7/OS/lab4$ ./aa
nwrites = -1    error = 11
nwrites = -1    error = 11
nwrites = -1    error = 11
nwrites = -1    error = 11
nwrites = -1    error = 11
nwrites = -1    error = 11
nwrites = -1    error = 11
nwrites = -1    error = 11
nwrites = -1    error = 11
nwrites = -1    error = 11
```

FIGURE 5 – A non-blocking call of read()

This result seems weird. We get a -1 number of keystrokes for each of the ten read calls and what seems to be an error, just like if there was no actual input given !

0.3.2 Commented fcntl() call

```c
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <fcntl.h>

int main()
{
        int i;
        char buf[100];

        // ouvrir un le stdin en lecture non bloquante
        //fcntl(STDIN_FILENO, F_SETFL, O_NONBLOCK);

        for (i = 0; i < 10; i++)
        {
                int nb;
                //try to read something from the standard input
                //and store it in the buffer array
                nb = read(STDIN_FILENO, buf, 100);
                //Then, print
                //the number of characters in the string (including the \0)
                //and if there was an error or not (0 if not)
                printf("nwrites = %d\terror = %d\n", nb, errno);
        }
}
```

FIGURE 6 – A non-blocking call of read()

```
Hello
nwrites = 6     error = 0
,
nwrites = 2     error = 0
world
nwrites = 6     error = 0
!
nwrites = 2     error = 0
I
nwrites = 2     error = 0
am
nwrites = 3     error = 0
testing
nwrites = 8     error = 0
your
nwrites = 5     error = 0
code
nwrites = 5     error = 0
!
nwrites = 2     error = 0
```

FIGURE 7 – A non-blocking call of read()

After commenting the fcntl() call and the code these screenshots are what I obtained. I was able to input a string ten times and get back no errors.

0.3.3 Observations

Using the fcntl() function which is used to manipulate the file descriptor of a process, you can give it the argument O_NONBLOCK. This argument makes it so the calls to the standard input will be non blocked, meaning the program will not wait for the user. And, as it executes in a microsecond, we get errors because there was indeed no input string following the read() call.