Concurrent File Access by Parent and Child Processes Lab. #1

Arif A. Balik

Undergraduate Student
Sytstem Programming
Department of Computer Science
Arel University
Büyükçekmece, İstanbul 34537
Email: arifbalik@outlook.com

March 5, 2019

Abstract

This report include analysis of some system calls in UNIX environment based on a program which provided in System Programming course. First section answers some questions asked in the experiment paper, and second section suggests and implements some improvements and optimisations on the code.

This report powered by LaTeX

Theory of Operation

Program first checks whether expected three argument is given or not, if not program terminates. In case of correct input from console, program opens input file and creates a new file to copy the text. Later on program calls *fork* and creates a child process. Then in both child and parent processes program reads the input file and writes into created file along with a console message based on which part of the program did that (C for child, P for parent). There is a delay between read and write operations in both processes, it causes copied file to be corrupted; for instance parent reads the character x and waits for loop to finish, in the same time child also reads the character $x+1$[i] and finishes the loop first and writes the character $x+1$[ii] in the place where x should be.

[i] When program calls *read* function it reads the n bytes of data and increments the cursor the latest character, therefore even when both processes store the information in their own isolated variables, they effect each other via global cursor value

[ii] 1 is an arbitrary value it can be much more than that

Q&A

Q1. *What is the purpose of this laboratory work?*

The purpose is to analyse the behaviour of the given program by observing and tweaking some code lines, thus getting a sense of how some system calls behave in which conditions in UNIX environment.

Q2. *What is the meaning of two parameters of the function* ***main()****?*

The first parameter **argc** stands for *Argument Count* which holds the information about how many parameter is given when program executed. The second one, **argv** stands for *Argument Value* which stores the values of given arguments during execution. The form of **argv** is called *pointer-to-pointer* which is a char pointer to another char pointer, this gives an incredible flexibility to developer, when there is a need for any kind of text processing. Simply, we can think it as two-dimensional char array.

Q3. *Suppose that you started the program using two names of files. What will be the value of the parameter* ***argc*** *in the* ***main()*** *function?*

The **argc** is always greater than 0, because the first parameter of **argv** is always the name of the executable program (unless if we use a specialised compiler). So in case of given two arguments to the program, we expect **argc** to be 3.

Q4. *What are* ***argv[0]****,* ***argv[1]*** *and* ***argv[2]****?*

As we stated earlier, **argv[0]** should be name of the program, **argv[1]** the first parameter, and **argv[2]** is the second parameter.

Q5. *What is the purpose of system calls* ***open()*** *and* ***creat()****? What is the meaning of the parameters* ***O_RDONLY*** *and* ***0666*** *in these system calls?*

The function **open()** takes mainly two parameter, first is a path name and second is flags. It opens a file and returns a number associated with opened file (*File Descriptor*) and flags determines whether it is a read only or read and write or write only etc. file. It returns -1 if anything goes wrong, and sets the global error flag for developer to further debug the problem.

The function **creat()**[iii] as name calls creates a file and return a file descriptor just like **open()** function. If file already exists, function opens the file and rewrites it. It has two parameter, which first one is path name and second one is a parameter called *mode*. It defines permissions for the file to be created.

But as said earlier **open()** function takes *mainly* two parameters, but it has one more optional parameter which is the parameter also called *mode*. Function behaves

[iii] Interesting fact : *creat* is a typo made by Ken Thompson.

just as same as **creat()** function when the parameter flag is set as following code. The permissions can be defined via third parameter.

```
open(path, O_CREAT|O_WRONLY|O_TRUNC, mode);
```

O_RDONLY is a constant integer, specifies that the file is opened for only reading and **0666** specifies the permissions of created file. 6 gives the permissions read, write, execute, and every digit stands for different users of system, such as owner, others.

Q6. *What system call is used for the creation of a new process in UNIX?*

fork(). Also there is **exec()** but it replaces the existing process with another one.

Q7. *What is the meaning of the variable* ***pid*** *in the program? Has this variable the same value for two processes in the program? Explain your answer in more detail.*

When we call **fork()** program duplicates itself and in order to make those two programs to do different things, developer needs to know which one is which. Therefore fork returns 0 when a child process created, but we can only see that value in the child process, and in the parent process the value of pid must be different than one in the child process and indeed parent process's pid value is a positive number.

Q8. What is the purpose of statement ***break*** *in two cycles in the in the program?*

The **read()** function returns the number of bytes read from the file which is given by developer with parameter *n_byte*. It returns *-1* if something happens during the process and sets the error flag. Therefore the program reads one byte and checks whether function read it or not, if not program breaks the loop and exits if it's in the child process or waits for child process to exit if it is in the parent process.

But program can't tell whether there was a problem while reading or program reached the end of the file, because read function returns *0* when it reaches the end of the file and program does not check *0* or *-1* condition, it treats same for both values. This problem will be fixed later in this report.

Q9. *Suppose that there is no system call* ***exit()*** *in the child portion of the program. What could be the result?*

Program would continue to execute following codes after the if statement, and since there is no lines of code to be executed, it would stop.

We can observe this by just listing currently running processes with **ps** command in the command line.

Q10. *Suppose that there is no system call* ***wait()*** *in the parent portion of the program. What could be the result?*

Same as *Q9*, the parent will exit after reaching the end of the code which is instant since there is also no lines of code after **wait()**. If parent finishes its execution and terminate itself before child process, child becomes an *orphan* and later will be *adopted* by **init()**. So everything should be as same for this specific program.

Q12. *What is the role of cycle "**for**" between read and write operations in both processes? What will be the result without this cycle (in both processes) when the copied file is small?*

The *for* loop in the program is just a large loop with an empty body. Since every cycle of the loop takes some execution time, it allows developer to delay the code between executions. This is a very primitive way for a program to delay since we can not exactly say how much it will take to finish it because UNIX is not a real-time operating system. But, then again it is what suppose to happen for this particular case. With small time nuances in the program, it will take different times between *read* and *write* operations both for child and parent processes and that will create a corrupted file which is the subject of this experiment.

So in the absence of those loops the program would have less uncertainty and less corruption, and also the size of the text is important because when size gets large it is more probable for collisions of processes to happen.

Q12. *Do both processes execute the same statement? If not, show which statements are executed by the parent and child processes.*

Practically they execute the same thing, but they don't. Parent starts as normal program from top to bottom, until function *fork* called. After fork, the program would continue with an identical copy of itself from the point the *fork* function called and then the child will enter the *if* statement (line 32-41) and parent will enter the *else* statement.

Q13. *In two cycles "**for**" the same index variable **i** is used. Since the parent and child processes ron concurrently, is it possible that any of these two processes will use the value of **i** modified by the other process? Explain why yes/no.*

When a child process created, all of the parent process's variables will be copied to a new address space, so they will be independent even if they have the same name, their addresses will be different, therefore, they can not interfere. This can be proven by changing the loop in the parent section with the following code and waiting the child process to exit before parent goes into the loop, if they would have the same address for the variable ***i***, there would be no way for the parent to go into the loop because the value of the variable would be changed by the child process, and condition *i [less than] 50000* would be false.

```
for(; i < 50000; i++);
```

Optimization

Firstly, the definition of the main function was out-of-date, so it has been changed;

Listing 1: Original

```
main(argc , argv)
int argc;
char* argv[];

{
```

Listing 2: Optimised

```
int main(int argc, char** argv){
```

Some variables were unnecessary, for instance *open()* and *creat()* functions return something called *file descriptor* which will be always 3 at first (0, 1 and 2 are reserved) and 4, 5 and so on. Therefore for this program they are pretty much constants since there is no dynamic creation or opening a file, it is only done twice and never again. Thus the variables *fdrd* and *fdwt* were changed with constants 3 and 4 respectively.

Listing 3: Original

```
int fdrd, fdwt;
char parent = 'P';
char child = 'C';
```

Listing 4: Optimised

```
#define _FDRD 3
#define _FDWT 4
#define P 'P'
#define C 'C'
```

Unnecessary but (Crucial for embedded software design) the variable *pid* can be defined as a *char* (or *byte*) object since its value will be small.

Personally I prefer *while* loops instead of *for* loops by means of readability, and try to use backward looping when possible;

Listing 5: Original

```
for(;;)
{
   ...
   for(i = 0; i < 50000; i++);
   ...
   ...

}
```

Listing 6: Optimised

```
while(1){
   ...
   while(--i);
   i = WAIT;
   ...
   ...
}
```

where WAIT is a constant.

Developers often use preprocessors to switch between modes. This feature can be implemented for this program's debug feature by using *#ifdef* preprocessor wherever an information printed for user to see. For example;

Listing 7: Original

```
printf("Some Text");
```

Listing 8: Optimised

```
#ifdef DEBUG
   printf("Some Text");
#endif
```

This will increase the overall performance. Developer just needs to comment the line *#define DEBUG* in the code.

There might be endless discussions about how optimised and efficient this simple program can be, for example using *write* instead of *printf* or using *open* instead of *creat* etc.

There is more optimisation made on this program, which can be seen in the Appendix.

Appendix

```
/*
 * System Programming Course
 * Lab #1
 * Concurrent File Acces by Parent and Child Processes
 * Arif Ahmet Balik - 180303019
 * Text Editor : Sublime Text 3
 * Last Update : 7 March 2019
 * All the files can be found at github.com/arifbalik/SYSTEM_PROG_LAB
*/

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/uio.h>
#include <sys/time.h>
#include <unistd.h>

#define _FDRD 3
#define _FDWT 4

/*
 * Uncomment to enable debug mode.
 * When disabled it saves 3000 microseconds!!!
 */
#define DEBUG
#define WAIT 10000

#define HELP "Please use the command as follows;\n" \
                        ".\\sharefile.c [input_file] [output_file]"

int main(int argc, char** argv){

        #ifdef DEBUG
                struct timeval  t1, t2;
                gettimeofday(&t1, NULL);
        #endif

        char pid, c, error;
        unsigned long i = WAIT;
```

```
if(argc > 3){
        printf("Too much parameter!" HELP "\n");
        exit(1);
}else if(argc < 3){
        printf("Not enough parameter!" HELP "\n");
        exit(1);
}
if(open(argv[1], O_RDONLY) == -1 ||
   open(argv[2], O_CREAT|O_WRONLY|O_TRUNC, 0666) == -1){
        perror("Oops!");
        exit(1);
}

#ifdef DEBUG
        printf("Parent: creating child process \n");
#endif

pid = fork();
if(pid == 0){

        #ifdef DEBUG
                printf("Child process starts, id: %d\n", getpid());
        #endif
        while(1){
                error = read(_FDRD, &c, 1);
                if(error == -1){
                        perror("Oops!");
                        break;
                } else if(!error){
                        #ifdef DEBUG
                                printf("\nChild: End of the file!\n")
                        #endif
                        break;
                }
                while(--i);
                i = WAIT;
                #ifdef DEBUG
                        write(1,"\nChild have read:",17);
                        write(1, &c, 1);
                #endif
                if(write(_FDWT, &c, 1) == -1) perror("Oops!");
        }
        #ifdef DEBUG
```

```
                printf("\nExiting child process\n");
        #endif
        exit(0);
    }
    else{

        #ifdef DEBUG
            printf("Parent process starts, id: %d\n", getpid());
        #endif

        while(1){
        error = read(_FDRD, &c, 1);
        if(error == -1){
            perror("Oops!");
            break;
        } else if(!error){
            #ifdef DEBUG
                printf("\nParent: End of the file!\n");
            #endif
            break;
        }
        while(--i);
        i = WAIT;
        #ifdef DEBUG
            write(1,"\nParent read:",13);
            write(1, &c, 1);
        #endif
        if(write(_FDWT, &c, 1) == -1) perror("Oops!");
    }

    wait(0);
    close(_FDRD);
    close(_FDWT);

    #ifdef DEBUG
        gettimeofday(&t2, NULL);
        printf ("Execution time = %f uSeconds\n",
            (double) (t2.tv_usec - t1.tv_usec));
        printf("Exiting parent process\n");
    #endif

    }
}
```