HAW Hamburg - IE6

Operating Systems Lab 2

Date	09.05.2023
Group2	Thishan Warnakulasooriya
	Celestine Machuca
Supervisor	Prof. Dr.-Ing. Holger Gräßner

2.1 Multiple file access

Write a program to cut the length of files to zero. Only the file's content shall be erased but not the file itself. Specify the files to manipulate via command line: Multiple files may be specified by usage of wildcards (*, ?).

a) First, write a C program that is able to access a single file only. Test if the erasing works. Don't change the file's access rights.

c code:

```
#include <fcntl.h>
int main(int argc, char *argv[])
{
    for (int i = 1; i < argc; i++)
        open(argv[i], O_TRUNC | O_WRONLY);
}
```

before testing:

```
$ cat test.txt
```

Output:

```
Hello World!
```

Testing:

```
$ gcc -o lab2_1 lab2_1.c
$ ./lab2_1 test.txt
$ cat test.txt
```

Output:

```
$
```

b) Study the manual of the command line interpreter: How to use wildcards? How to specify multiple file names? Check if the file set specification works properly!

We tested the program with the following files:

test.txt, test1.text, afile.txt.

Testing "*" wildcard, which matches any string, including the empty string:

```
$ ./lab2_1 *.txt
```

Checking if the files contents are empty:

```
$ cat test.txt
$ cat test1.txt
$ cat afile.txt
```

Output:

```
$
$
$
```

Testing "?" wildcard, which matches any single character:

Checking if the files contents are empty:

```
$ cat test.txt
```

Output:

```
$ Hello World!
```

```
$ cat test1.txt
```

Output:

```
$ Hello World!
```

Testing "?" wildcard, which matches any single character:

```
$ ./lab2_1 test?.txt
```

Checking if the files contents are empty:

```
$ cat test.txt
$ cat test1.txt
$ cat afile.txt
```

Output:

```
$ Hello World!
```

```
$
$ Hello World!
```

2.2 Concurrent access to a file

Write a C program running two processes A and B continuously: Process A writes a short text to a file. Then process B has to read this file. After this write-/read-cycle, process A has to erase the file, then the processes change roles: B writes to the file, A reads from the file.

a) Use a lock file first to implement the exclusive usage of the file.

c code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <time.h>
#include <sys/types.h>
#include <signal.h>

#define TEXT_SIZE 100
#define IS_CHILD 0
#define IS_PARENT 1
#define LOOP_ITERATIONS 10
#define FILE_PATH "file.txt"

void truncateFile(const char *filePath)
{
    FILE *fp = fopen(filePath, "w");
    if (fp == NULL)
    {
        printf("Error: Unable to open the file.\n");
        return;
    }
    fclose(fp);
    printf("Successfully erased the file.\n");
}

void writeFileProcess(const char *filePath, const char *text, pid_t
processID)
{
    FILE *fp = fopen(filePath, "w");
    if (fp == NULL)
    {
        printf("Error: Unable to open the file.\n");
        return;
    }
    fprintf(fp, "%s", text);
```

```
    fclose(fp);
    kill(processID, SIGUSR1);
}

void readFileProcess(const char *filePath, int processType)
{
    pause();
    printf("Process %s is reading ---", processType == IS_CHILD ? "child" :
"parent");
    FILE *fp = fopen(filePath, "r");
    if (fp == NULL)
    {
        printf("Error: Unable to open the file.\n");
        return;
    }
    char ch;
    printf("File content: ");
    while ((ch = fgetc(fp)) != EOF)
    {
        printf("%c", ch);
    }
    printf("\n");
    fclose(fp);
    truncateFile(filePath);
}

void signalHandler(int signalNumber)
{
    return;
}

int main()
{
    signal(SIGUSR1, signalHandler);
    pid_t processID = fork();
    if (processID == IS_CHILD)
    {
        pid_t parentProcessID = getppid();
        // sleep(1);
        for (int i = 0; i < LOOP_ITERATIONS; i++)
        {
            // writeFileProcess(FILE_PATH, "Hello from child",
parentProcessID);
            readFileProcess(FILE_PATH, IS_CHILD);
            writeFileProcess(FILE_PATH, "Hello from child", parentProcessID);
        }
    }
    else if (processID > IS_CHILD)
    {
        pid_t childProcessID = processID;
        sleep(1);
        for (int i = 0; i < LOOP_ITERATIONS; i++)
        {
            // readFileProcess(FILE_PATH, IS_PARENT);
            writeFileProcess(FILE_PATH, "Hello from parent", childProcessID);
            readFileProcess(FILE_PATH, IS_PARENT);
```

```
        }
    }
    else
    {
        fprintf(stderr, "Error creating child process\n");
        exit(EXIT_FAILURE);
    }
    return EXIT_SUCCESS;
}
```

Output:

```
Process child is reading ---File content: Hello from parent
Successfully erased the file.
Process parent is reading ---File content: Hello from child
Successfully erased the file.
Process child is reading ---File content: Hello from parent
Successfully erased the file.
Process parent is reading ---File content: Hello from child
Successfully erased the file.
```

b) Implement an exclusive access to the file by usage of fcntl(). Study the manual first. **Check if your program runs correctly. Compare both attempts. Name the advantages and disadvantages of each method.**

c code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <time.h>
#include <sys/types.h>
#include <signal.h>

#define TEXT_SIZE 100
#define IS_CHILD 0
#define IS_PARENT 1
#define LOOP_ITERATIONS 10
#define FILE_PATH "file.txt"

void truncateFile(const char *filePath)
{
    int fileDescriptor = open(filePath, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (fileDescriptor < 0)
    {
        perror("Error opening file");
        exit(EXIT_FAILURE);
    }
    close(fileDescriptor);
}
```

```
void writeFileProcess(const char *filePath, const char *text, pid_t
processID)
{
    int fileDescriptor = open(filePath, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (fileDescriptor < 0)
    {
        perror("Error opening file");
        exit(EXIT_FAILURE);
    }
    struct flock fileLock;
    fileLock.l_type = F_WRLCK;
    fileLock.l_whence = SEEK_SET;
    fileLock.l_start = 0;
    fileLock.l_len = TEXT_SIZE;
    fileLock.l_pid = getpid();
    fcntl(fileDescriptor, F_SETLKW, &fileLock);
    write(fileDescriptor, text, strlen(text));
    printf("Wrote %ld bytes to %s\n", strlen(text), filePath);
    fileLock.l_type = F_UNLCK;
    fcntl(fileDescriptor, F_SETLK, &fileLock);
    close(fileDescriptor);
    kill(processID, SIGUSR1);
}

void readFileProcess(const char *filePath, int processType)
{
    pause();
    printf("Process %s is reading ---", processType == IS_CHILD ? "child" :
"parent");
    int fileDescriptor = open(filePath, O_RDONLY);
    if (fileDescriptor < 0)
    {
        perror("Error opening file");
        exit(EXIT_FAILURE);
    }
    struct flock fileLock;
    fileLock.l_type = F_RDLCK;
    fileLock.l_whence = SEEK_SET;
    fileLock.l_start = 0;
    fileLock.l_len = TEXT_SIZE;
    fileLock.l_pid = getpid();
    fcntl(fileDescriptor, F_SETLKW, &fileLock);
    char buffer[1024];
    int bytesRead = read(fileDescriptor, buffer, 1024);
    if (bytesRead < 0)
    {
        perror("Error reading file");
        exit(EXIT_FAILURE);
    }
    buffer[bytesRead] = '\0';
    printf("Read %d bytes from %s ---", bytesRead, filePath);
    printf("%s\n", buffer);
    fileLock.l_type = F_UNLCK;
    fcntl(fileDescriptor, F_SETLK, &fileLock);
    close(fileDescriptor);
```

```
    truncateFile(filePath);
}

void signalHandler(int signalNumber)
{
    return;
}

int main()
{
    signal(SIGUSR1, signalHandler);
    pid_t processID = fork();
    if (processID == IS_CHILD)
    {
        pid_t parentProcessID = getppid();
        sleep(1);
        for (int i = 0; i < LOOP_ITERATIONS; i++)
        {
            writeFileProcess(FILE_PATH, "Hello from child", parentProcessID);
            readFileProcess(FILE_PATH, IS_CHILD);
        }
    }
    else if (processID > IS_CHILD)
    {
        pid_t childProcessID = processID;
        for (int i = 0; i < LOOP_ITERATIONS; i++)
        {
            readFileProcess(FILE_PATH, IS_PARENT);
            writeFileProcess(FILE_PATH, "Hello from parent", childProcessID);
        }
    }
    else
    {
        fprintf(stderr, "Error creating child process\n");
        exit(EXIT_FAILURE);
    }
    return EXIT_SUCCESS;
}
```

Output:

```
Process child is reading ---read 17 bytes from file.txt ---Hello from parent
wrote 16 bytes to file.txt
Process parent is reading ---read 16 bytes from file.txt ---Hello from child
wrote 17 bytes to file.txt
Process child is reading ---read 17 bytes from file.txt ---Hello from parent
wrote 16 bytes to file.txt
Process parent is reading ---read 16 bytes from file.txt ---Hello from child
wrote 17 bytes to file.txt
Process child is reading ---read 17 bytes from file.txt ---Hello from parent
wrote 16 bytes to file.txt
Process parent is reading ---read 16 bytes from file.txt ---Hello from child
wrote 17 bytes to file.txt
```

In our approach, the first task was done using signals to implement the exclusive usage of a file. This could have also been done by using any other interprocess communication methods in combination. The first method, which involves using fopen and fclose, is simpler to implement and comprehend. However, it comes with a trade-off: we sacrifice some of the fine-grained control over the file. On the other hand, the fcntl approach provides us with greater control over the file, allowing us to perform operations like locking specific sections of the file and using different types of locks. Nonetheless, it is more challenging to implement and grasp due to its complexity.