System Programming in C for Linux (II)

File management, part II:

Concurrent or exclusive access to files. File locks

Cristian Vidrașcu
cristian.vidrascu@info.uaic.ro

April, 2024

Overview

Introduction

Because `UNIX` operating systems (in particular, `Linux`) are *multi-tasking* systems (*i.e.*, systems that support the “simultaneous” execution of multiple programs), usually the *concurrent access* to files is allowed, meaning multiple processes can access “simultaneously” for reading and/or writing the same file, or even the same record in a file.

However, this concurrent (“simultaneous”) access to a file by different processes can sometimes have side effects (such as destroying the integrity of the data in the file due to *data races*).

For this reason, `UNIX` systems have implemented mechanisms that allow a mode of *exclusive access* to files, *i.e.* a mode of access in which only one process has, at a time, permission to access a file, or even a specific record in a file.

Concurrent (simultaneous) access to files

Outline

Demo (1): An example of concurrent access to a file

Remark: from the programmer's point of view, it must not use any techniques other than those discussed in the previous lesson on file access, in order to "benefit" from concurrent ("simultaneous") access to a file. Everything happens at runtime: if the user is running two or more instances of programs that are usually accessing the same file at the same time, then the file accesses will be "simultaneous" (*i.e.*, about the same time).

Here is an example of a program that can be used to illustrate the effects of concurrent file access: see the program `access_v1.c` ([2]).

First, a run demo illustrating *sequential file access*, *i.e.* a single process wants to access the file in a certain amount of time.
We are creating a `fis.dat` that contains the following line of text: `aaaa#bbbb#cccc#dddd#eeee`
Then we sequentially launch several instances of this program, *e.g.* by using the command:

```
UNIX> ./access_v1 1 ; ./access_v1 2 ; ./access_v1 3
```

What will be the contents of the file after the execution of this command ?

After the execution of the first instance, the file will look like this: `aaaa1bbbb#cccc#dddd#eeee`

After the execution of the second instance, the file will look like this: `aaaa1bbbb2cccc#dddd#eeee`

After the execution of the third instance, the final result will be: `aaaa1bbbb2cccc3dddd#eeee`

Demo (1): An example of concurrent access to a file (cont.)

And now, a run demo illustrating *concurrent file access*: multiple processes (*i.e.*, instances of the program) that want to access the file in the same time frame.
"Reset" the file `fis.dat` with the following line of text: `aaaa#bbbb#cccc#dddd#eeee`
Then we run in parallel ("simultaneous") two instances of this program, by command:

```
UNIX> ./access_v1 1 & ./access_v1 2 &
```

What will be the contents of the file after the execution of this command ?

You probably expect the file to look like this after the execution of this command:
`aaaa1bbbb2cccc#dddd#eeee` or `aaaa2bbbb1cccc#dddd#eeee`
(depending on which of the two processes first managed to overwrite the first character '#' in this file, the other process remaining the second character '#' to overwrite it.)

In reality, repeating the execution of this command as many times as possible, we will always obtain:
`aaaa1bbbb#cccc#dddd#eeee` or `aaaa2bbbb#cccc#dddd#eeee`

Reason: the call `sleep(5)`, which causes a 5-second wait between finding the first record in the file that is '#' and overwriting that record with another character.

Remark: by removing the `sleep(5)` call from the program, repeating this command a sufficiently large number of times, all of the above 4 results can be obtained with different observation frequencies.

Demo: for more detailed explanations, see [FirstDemo] presented in lab support #7 (in romanian).

Exclusive access to files – Locks on files

Outline

flock data structure for file locks

UNIX systems provide programmers with a locking mechanism (*i.e.*, to put "*locks*") on portions of a file for exclusive access.
This mechanism defines an *exclusive access* area in the file. Such a portion will not be able to be accessed concurrently by multiple processes for the duration of the locking.

To specify a lock on a portion of a file (or the entire file), use the data structure `flock`, defined in the header file `fcntl.h` as follows :

```
struct flock
{
  short l_type;   // the type of lock
  short l_whence; // relative position (origin)
  long  l_start;  // starting position, in relation to the origin
  long  l_len;    // the length of the locked portion
  int   l_pid;
}
```

Remark: after filling in the fields of the above structure, the `fcntl` function will be called later to actually put the "lock" on that part of the file.

flock data structure for file locks (cont.)

The meaning of the fields of the `flock` data structure:

- the field `l_type` indicates the type of lock, which may have one of these constants as its value:
  - `F_RDLCK` : read locking
  - `F_WRLCK` : write locking
  - `F_UNLCK` : unlocking
- the field `l_whence` indicates the relative position (*i.e.*, origin) in relation to which the `l_start` field is interpreted, and may have as value one of the following symbolic constants:
  - `SEEK_SET` (=0) : beginning of file
  - `SEEK_CUR` (=1) : current position in file
  - `SEEK_END` (=2) : end of file
- the field `l_start` indicates the position (*i.e.*, the offset relative to the origin `l_whence`) from which the locked portion starts.
  Remark: `l_start` must be negative for `l_whence=SEEK_END`.
- the field `l_len` indicates the length in bytes of the locked portion.
- the field `l_pid` is managed by the `fcntl` function that sets the lock, being used to store the PID of the process owning that lock.
  Remark: it only makes sense to query this field when the `fcntl` function is called with the `F_GETLK` parameter.

9 / 19

The fcntl system call for file locks

The interface of the function `fcntl` ([5] – one of them, the one for locking):

```
int fcntl(int fd, int cmd, struct flock* p_flock)
```

- *fd* = the open file descriptor on which the lock is placed
- *p_flock* = the address of the `flock` structure that defines that lock
- *cmd* = indicates the setting mode, which can take one of the values:
  - `F_SETLK` : allows you to put a read or write lock on the file, or remove one already placed (depending on the type specified in the `flock` structure).
    Remark: in case of failure due to conflict with another lock already set, the variable `errno` is set to `EACCES` or `EAGAIN`.
  - `F_SETLKW` : blocking call for aquiring a lock, *i.e.* it waits (*i.e.*, the function does not return) until the lock can be placed. Possible reason for waiting: attempting to lock an area already locked by another process.
  - `F_GETLK` : allows you to extract information about a lock on the file.
- the value returned is `0` for successful locking, or `-1` in case of an error.

10 / 19

The fcntl system call for file locks (cont.)

Remarks:

- In order to be able to put a read or write lock on a file descriptor, it must have been previously opened for reading or writing.
- The lock is automatically removed when the process that put it closes that file, or finishes its execution.
- Removing (unlocking) a segment from a previously locked larger portion can result in two locked segments.
- The `l_pid` field in the `flock` structure is updated by the `fcntl` function.
- Locks are not inherited by child processes.
  The reason: each lock has in the associated `flock` structure the PID of its owner (*i.e.*, the process that created it), and the child processes have, of course, PIDs different from that of their parent.
- In `Linux` there are two other interfaces that offer locks on files ([5]):
  - the function `flock` $\rightarrow$ see the documentation for details: `man 2 flock`
  - the function `lockf` $\rightarrow$ see the documentation for details: `man 3 lockf`
- There are also two useful lock commands: `flock` and `lslocks` ([6]).

Demo (2): An example of exclusive access to a file

We can rewrite the previous program by adding the use of write locks to “inhibit” concurrent access to the file: see the program `access_v2.c` ([2]).

“Reset” the file `fis.dat` with the following line of text: `aaaa#bbbb#cccc#dddd#eeee`

Then we run in parallel (“simultaneous”) two instances of this program, by command:

```
UNIX> ./access_v2 1 & ./access_v2 2 &
```

What will be the contents of the file after the execution of this command ?

This time, no matter how many executions are made, the desired result will always be obtained:

`aaaa1bbbb2cccc#dddd#eeee` or `aaaa2bbbb1cccc#dddd#eeee`

Remark: In the above program the lock is put with a non-blocking call (*i.e.*, with the parameter `F_SETLK`). We can also use a blocking call, *i.e.* the `fcntl` call will not return immediately, but will wait until it manages to lock the file.

See the program `access_v2w.c`

Simultaneously running two instances of this program, we will find that we get the same result as in the case of the non-blocking version.

Demo: for more detailed explanations, see [SecondDemo] presented in lab support #7 (in romanian).

Features of file locks

- Important: the write locks (*i.e.*, those with type `F_WRLCK`) are *exclusive*, and the read locks (*i.e.*, the ones with the type `F_RDLCK`) are *shared*, in the sense CREW ("Concurrent Read or Exclusive Write").
  In other words: at any time, for any part of a file, at most one process may hold a write lock on that portion (and then no process can hold a read lock at the same time), or multiple processes may have read locks on that portion (and then no process can hold a write lock at the same time).
- Important: file locks are advisory, not mandatory!
  In other words: the proper operation of write locks is based on *cooperation between processes* to ensure exclusive access to files, *i.e.* all processes that want to access mutual exclusive a file (or part of a file) will have to use write locks for that access.

  Otherwise, for example, if a process writes a file directly, or a portion of a file (and it has the necessary permissions), its write call will NOT be prevented by a possible write or read lock placed on that file, or that portion of the file, by another process.

  Also the read locks are *advisory*, *i.e.* we can write (and read) the file (if we have the necessary permissions) while another process has a read lock on that file.

13 / 19

Demo (3): Illustration of the advisory character of file locks

Here is a justification for the previous remark about the *advisory* nature of file locks:

"Reset" the file `fis.dat` with the following line of text: $\boxed{\texttt{aaaa\#bbbb\#cccc\#dddd\#eeee}}$

and then run the next command:

```
UNIX> ./access_v2 1 & sleep 2 ; echo "xyxyxy" > fis.dat
```

What will be the contents of the file after the execution of this command ?

Answer: after the execution of this command, the `fis.dat` file will contain the line of text: $\boxed{\texttt{xyxy1y}}$, which shows us that the overwriting of the `echo` command in the file took place within 5 seconds time frame of the `access_v2` instance blocking the file!

* * *

```
UNIX> ./access_v2 1 & sleep 2 ; cat fis.dat
```

By running this command, we will notice that the `cat` command reads successfully, even though the file was locked at the time of reading.

Demo: for more detailed explanations, see the last part of [SecondDemo] (in romanian).

14 / 19

Demo (4): An example of *optimized* exclusive access to a file

Important note: the second version of the demo program (both non-blocking and blocking variants) is not optimal:

Basically, the two processes (*i.e.*, the two instances of the program executed in parallel) do their job *sequentially*, one after the other, and not in parallel, because only after the end of that process that was the first who managed to put the lock on the file, the other process will be able to start to do its job (*i.e.*, scrolling through the file and replacing the first '#' character encountered).

* * *

This observation suggests that we can *improve the total execution time* by allowing the two processes to run truly concurrently, and for this we need to lock only one character (*i.e.*, the first position in the file we are encountering the '#' character on) and only for the minimum time required to overwrite it, instead of blocking the entire file, for the whole time – from the beginning to the end of program execution.

Demo (4): An example of *optimized* exclusive access to a file (cont.)

The third version, with lock on a character and minimum duration of locking:

Implementing this optimization: the program will have to do the following – when it encounters the first '#' character in the file, it locks it (*i.e.*, exactly one character) and then rewrites it: see the program `access_v3.c` ([2]).

In this case, what do you think will be the contents of the file after the completion of the parallel execution of two instances of this version of the program ?

* * *

Remark: the solution idea applied in the `access_v3.c` program is not entirely correct, in the sense that the expected result will not always be obtained, because between the moment of detecting the first position of a '#' character in the file and the moment of the successful lock there is the possibility that that '#' character to be overwritten by the other instance executed in parallel !

Note: just to force the occurrence of a situation that causes an unwanted result, we introduced in the program that `sleep(5)` call between locking the '#' character and rewriting it.

How can this shortcoming of the `access_v3.c` program be remedied ? $\rightarrow$ $\rightarrow$ $\rightarrow$

Demo (4): An example of *optimized* exclusive access to a file (cont.)

→ → This shortcoming of the `access_v3.c` program can be corrected as follows:

After locking the caracter, we will check again if that character is indeed '#' (because in the meantime it may have been rewritten by the other parallel instance of the program) and if it is no longer '#', then the lock must be removed and the search for the first '#' character encountered in the file must be resumed.

v4 → *Homework*: add this fix to the `access_v3.c` program.

* * *

Solution: if you are unable to add this fix yourself, you can look here: `access_v4.c` .

Demo: for more detailed explanations about this more efficient version of the demonstration program, see [ThirdDemo] presented in lab support #7 (in romanian).

* * *

Additionally, see examples of using the `lslocks` command to observe active locks at various times in the parallel job's execution, presented in [SecondDemo] and [ThirdDemo] in lab support #7.

Application: implementing a binary semaphore

How could we implement a binary semaphore using file locks ?

A possible implementation could use the following ideas:

Initializing the semaphore would be done by creating a regular file by a *supervisor* process (this can be any of the cooperating processes that will use that semaphore, or it can be a separate process). The new file will have a unique name to identify the semaphore within the group of cooperating processes.

This *supervisor* process will initially write 1 arbitrary byte to the file (the length of the file is not important, as we will write lock the entire file to “simulate” a binary semaphore).

The `wait` operation will consist of write locking the file, with a blocking call (*i.e.*, using `F_SETLKW` in the `fcntl` call).
The `signal` operation will consist of unlocking the file.

Homework: implement in C a binary semaphore based on the ideas above and write a demonstration program to use the semaphore thus implemented to ensure mutual exclusion of a critical section of code (for “inspiration” in writing the demo program, review the synchronization problems discussed in the theoretical courses #5 and #6).

Bibliographical references

Mandatory bibliography

[1] Chapter 3, §3.2 from the book "*Sisteme de operare – manual pentru ID*", by C. Vidrașcu, UAIC Publishing House, 2006. This is available as ebook at the address:

- `https://edu.info.uaic.ro/sisteme-de-operare/SO/books/ManualID-SO.pdf`

[2] The demo programs from this presentation can be downloaded from:

- `https://edu.info.uaic.ro/sisteme-de-operare/SO/lectures/Linux/demo/flock/`

[3] The online support lesson associated with this presentation:

- `https://edu.info.uaic.ro/sisteme-de-operare/SO/support-lessons/C/en/support_lab7.html`

Additional bibliography:

[4] Chapter(s) 55 from the book "The Linux Programming Interface : A Linux and UNIX System Programming Handbook", by M. Kerrisk, No Starch Press, 2010.

- `https://edu.info.uaic.ro/sisteme-de-operare/SO/books/TLPI1.pdf`

[5] POSIX API : `man 2 fcntl` , `man 2 flock` and `man 3 lockf` .

[6] Documentation of commands for locks: `man 1 flock` and `man 8 lslocks` .