SYSTEM AND DEVICE PROGRAMMING

solved theory exercises from past exams

File system and IO management (exercises related to chapters 12, 13 and 14)

1. Consider a file system supporting concurrent access to the same file by multiple processes.

   Which operations does the operating system have to perform to implement an open()? And a close()?

open()
it receives the file name as a parameter, it must search the file, open it in the requested mode and return the corresponding file descriptor (handle/pointer, depending on the operating system):
1) the file name is searched using the directory data structure (in memory and/or disk);
2) if successful, the search returns the File Control Block (copy of the system-wide open-file table, already present due to a previous search/open or generated now);
3) a new entry in the per-process open-file table (which refers to the FCB in the system-wide) is created, and the function returns a pointer/handle or index (file descriptor) to this entry.

close()
it must close the file, removing the related entries in the open file tables, if they are no longer useful:
1) the corresponding entry in the per-process open-file table is deleted;
2) the reference counter in the system-wide table is decreased, if this becomes 0, then the entry is deleted also from this table.

   Which data structures (file management tables) have to be accessed in order to implement a read () and/or write ()?

Access is performed to
1) per-process open-file table: the corresponding entry has the pointer to the read/write position in the file and the access mode;
2) system-wide open-file table: the entry in this table is needed, among other things, to convert from a logical address in the file to a physical one (block number and offset in the block).

   What are the *system-wide open-file table* and the *per-process open-file table*? Why are both needed in an operating system instead of just one?

The tables contain all the necessary information in memory to manage files correctly.
Both are necessary because a file can be opened concurrently (in different ways), both in the context of a single process (possibly with multiple threads) and/or by multiple processes.
The common information, in the system-wide, includes a copy of the FCB and synchronization primitives to handle/synchronize shared accesses and more.
The information in the per-process includes the pointer to the file and the access mode (e.g. read/write)

2. Consider a kernel buffer used as a transition area for file blocks in transit between disk and user memory: data are thus transferred (for example through a `read(fd,addr,size)`, with `addr` and `size` representing the destination in user memory) with one extra step: from disk to kernel buffer (for a size `size`) and then from buffer kernel to the user destination `addr`.
   Why can the kernel buffer be beneficial, while forcing an extra step in RAM?

R1 *The main advantage is related to having decoupled the work in the USER memory with respect to disk access. In systems having paging and/or swapping it is thus possible to enable swapping out of an entire user process waiting for I/O, or of a page involved in this I/O, as the user buffer is not "blocked" by waiting for I/O. The advantage of the double buffer over the single buffer is also having a pipelined behavior, in which it is possible to transfer from kernel memory to user and at the same time from disk to kernel memory.*
A further advantage of the kernel buffer can be the cache-like behaviour, that is, the kernel buffer could be already filled in advance, to prevent the process from waiting for I/O.

ATTENTION: Note that the advantage is NOT to avoid the user process doing the I/O. The USER process HAS NO PRIVILEGES, so it does not operate the IO directly. Both with buffer and without buffer, the I/O is carried out by a system call, using an appropriate driver (a KERNEL task): in one case the driver works on user memory (and blocks it) in the other case on kernel buffer.
Nor does it make sense in this context to talk about interrupts, DMA, CPU or other.

In a paged system, can the `size` parameter be arbitrary, or must it be a multiple of the block or page size?

R2	The `size` parameter is arbitrary, as `read()` is a function at the user level, which has no direct dependence on pagination strategies and/or file system implementation. Not only can `size` be arbitrary, but the `addr` starting address is not necessarily aligned to a page start.

Suppose you use a "double" buffer.
[Explanation (read() is explained, the write() will be dual): while one of the two buffers (let's call it "kernel", as it works for disc-to-kernel transfer) is involved in a transfer from disk, the other buffer (let's call it "user", as it is involved in kernel-to-user memory transfer) can be used (provided that it was previously loaded with data from disk) to transfer data to the destination in user memory. After each operation the roles of the two buffers are exchanged].
Assuming you want to read sequentially a 200KB file, how many bytes with transit (overall) on the data bus in the two cases (single and double buffer)? For disk readings, consider using DMA. In the case of double buffers, is the number of bytes transferred halved compared to the single buffer (*motivate the response*)?

R3	The double buffer can only speed up operations (thanks to a higher level of parallelism/pipelining), but the number of bytes transferred is exactly the same in the two cases (single and double buffer: the location of the data in the kernel buffer is the only difference). Altogether, the 200KB will transit once on the bus during the transfer in DMA from disk to buffer kernel. The transition from buffer kernel to user memory is instead a copy from RAM to RAM (different source and destination addresses). In the case of a data transfer managed by the CPU, the data will transit twice on the data bus (from RAM to CPU and from CPU to BUS). It is therefore a question of reading 200KB from RAM and writing them. 600KB in total (200KB + 2 * 200KB). *NOTE (out of the course scope): DMA could used for the transfer from RAM to RAM as well, so the transit could be done with a single transit on the bus, but only on condition that a fetch-and-deposit DMA controller is used. That kind of DMA allows (in two bus cycles, one for reading and one for writing) to manage single RAM-RAM operations.*

3. Consider the two types of synchronization, related to I/O operations: synchronous and asynchronous. Explain the main differences between the two I/O types. Then define blocking and non-blocking I/O: are they synonyms od synchronous and asynchronous? Are there any differences (between blocking/non-blocking and synchronous/asynchronous)? (If not, why? If yes, which ones?)

R1	In a very concise way, we can say that: • blocking and synchronous I/O are equivalent: the process that performs I/O awaits its completion, in a wait state; • non-blocking I/O allows the process to continue, asynchronous I/O is in fact NON-blocking, with the addition of techniques that allow you to manage (later) the completion of the I/O. These techniques are: `wait` functions (depending on the operating system) that allow you to wait for the completion of the I/O, or "*callback*" functions, automatically called by the system, upon completion of the I/O (attention: these are functions that must be written by the user)

Can synchronous I/O be in polling or should it be in interrupt? (answer the same question for the case of asynchronous I/O).

R2	Polling and interrupt are two different ways to manage an I/O device. Obviously polling is less efficient, with rare exceptions. However, this is a problem internal to the drivers (kernel modules) and therefore independent of the user process. Basically, implementing a read/write system call synchronously or asynchronously can be done with drivers that work both in polling and interrupt mode.

How can a process benefit from asynchronous I/O? In case of asynchronous I/O, is it possible to write a program with instructions, subsequent to the I/O, which depend on the data involved (e.g. read) during the I/O?

R3	The process can benefit as it can execute other instructions while the I/O is in progress. It is therefore a possible form of concurrency. During the I/O, the process cannot use the data involved. If it must do it, it is necessary to synchronize (and wait) on the related (asynchronous I/O) wait operation.

4. Consider the problem of managing a block I/O request made using DMA. What do we mean, in this context, by the term "*cycle stealing*"? Why is the DMA transfer advantageous compared to the programmed I/O? Imagine you transfer 40KB of data from disk to RAM memory, how many bytes transit on the RAM data bus in the two cases (transfer to DMA and programmed I/O)? (Motivate the answer). If you want to use a buffer in the kernel memory for the I/O, how does the double buffer differ from the single buffer and why can it be advantageous?

R (The main aspects that a correct answer should contain are briefly highlighted)

- “Cycle stealing” is the subtraction (with CPU wait) of BUS cycles to the CPU, while the DMA has control of the RAM access BUS
- For two main reasons:
  - because DMA transfers pass data "directly" between I/O and RAM (without going through the CPU, with twice the number of operations);
  - because during a DMA data transfer, the CPU can do other things (increasing the degree of multiprogramming).
- The number of Bytes that transit on the bus is 40KB in DMA and 80KB (the 40KB pass twice, as they must pass through the CPU) in the case of programmed I/O.
- The double buffer is a technique in which while one buffer is being written the other (previously filled) can be read in parallel. It therefore allows a form of pipelining. With the single buffer solution implies that one of the two (the writer or the reader) must wait for the completion of the other operation.
  (ATTENTION; we speak here of double kernel buffer, not of duality buffer kernel and/or user buffer)