Lab 5 – Implementing Zero-copy File Operations

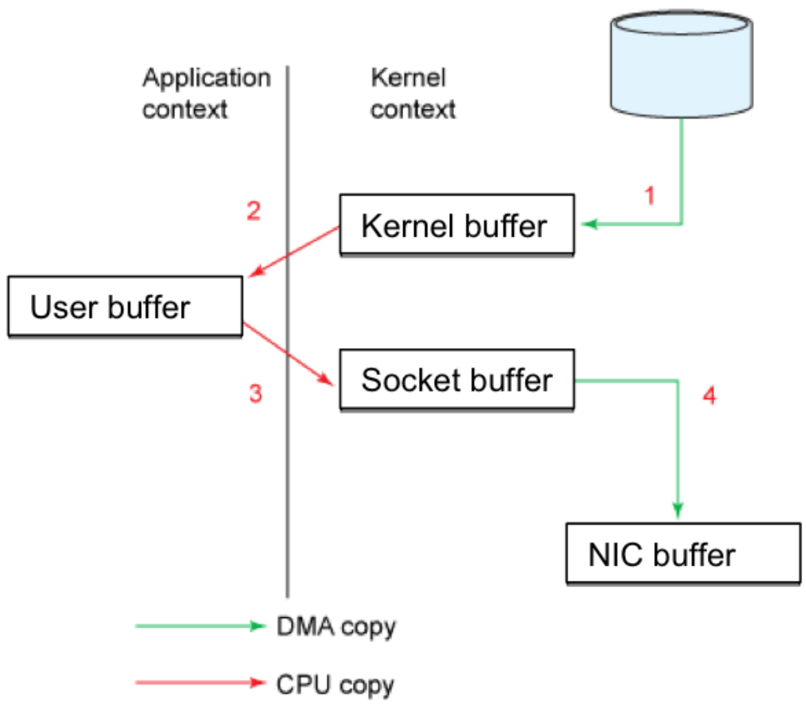
Section 1. Background

Consider the scenario of reading from a file and transferring the data to another program over the network. This scenario describes the behaviour of many server applications, including Web applications serving static content, FTP servers, mail servers, etc. The core of the operation is in the following two calls:

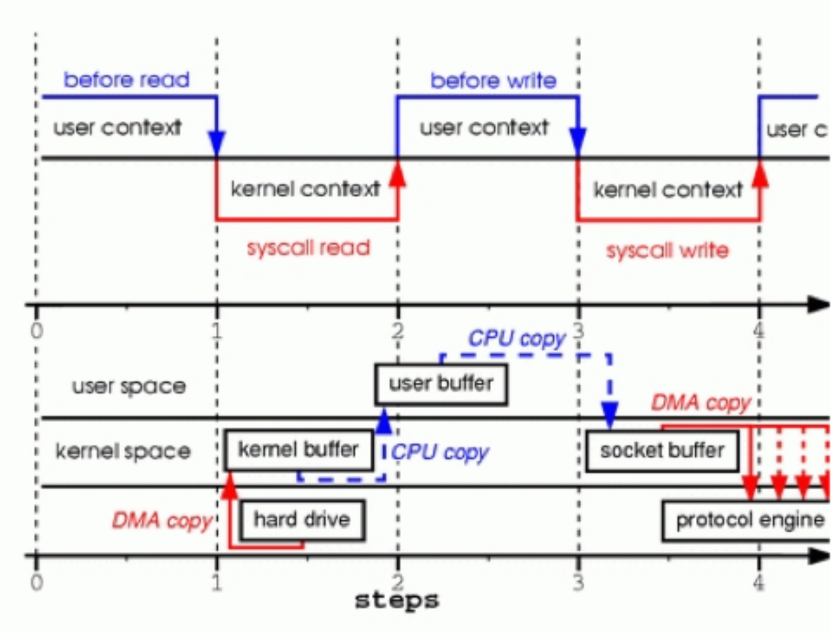
```
read(file, user_buffer, len);
write(socket, user_buffer, len);
```

*`file` and `socket` are two file descriptors.

Figure 1 shows how data is moved from the file to the socket.



Behind these two calls, the data has been copied at least four times, and almost as many user/kernel context switches have been performed. Figure 2 shows the process involved. The top side shows context switches, and the bottom side shows copy operations.



Step one: The read system call causes a context switch from user mode to kernel mode. The first copy is performed by the DMA (Direct Memory Access) engine, which reads file contents from the disk and stores them into a kernel address space buffer.

Step two: Data is copied from the kernel buffer into the user buffer, and the read system call returns. The return from the call caused a context switch from kernel back to user mode. Now the data is stored in the user address space buffer, and it can begin its way down again.

Step three: The write system call causes a context switch from user mode to kernel mode. A third copy is performed to put the data into a kernel address space buffer again. This time, though, the data is put into a different buffer, a buffer that is associated with sockets specifically.

Step four: The write system call returns, creating our fourth context switch. Return from write call does not guarantee the start of the transmission. It simply means the Ethernet driver had free descriptors in its queue and has accepted our data for transmission. Independently and asynchronously, a fourth copy happens as the DMA engine passes the data from the kernel buffer to the protocol engine. (The forked DMA copy in Figure 2 illustrates the fact that the last copy can be delayed).

As you can see, a lot of data duplication happens in this process. Some of the duplication could be eliminated to decrease overhead and increase performance. To eliminate overhead, we could start by eliminating some of the copying between the kernel and user buffers.

Section 2. Overview and Technical Details

Your task in this lab is to implement **zero-copy read and write operations** that would eliminate the copying between the kernel and user buffers. You will provide a new library with a new set of library calls that allow a user to:

- Open a file
- Read from the file without using a user buffer
- Write to the file without using a user buffer
- Reposition within the file
- Close the file

The user directly uses the kernel buffer provided by the library calls to read and write data.

Your implementation should not call `read` and `write` system calls or other library calls that wrap around `read` and `write` system calls. Calling `read` and `write` would involve some type of duplication of buffers. You should use the `mmap` system call in your implementation.

Creating a Zero-copy IO Library

We provide a `Makefile` to make the process of compilation easier. Running `make` in the main folder of the assignment compiles the source codes and produces the library **libzc_io.so**, as well as the `runner` and `demo` executables. You can then use `runner` to test your code against a set of tests and check whether your code is working as expected.

Steps in the `Makefile`:

1. Compile **zc_io.c** source files into **libzc_io.so**. Note that only **zc_io.c** will be considered for grading and your code should only be added into that file.
2. Compiles **runner.c,** linking the **zc_io** library. In general, if you want to link the library when compiling your code, you must specify this during the compilation of your code by adding:
   `-L<directory_path_containing_libzc_io.so> -lzc_io`
   `* <directory_path_containing_libzc_io.so>` would be the directory containing the library
3. You can run `make clean` to remove the files that were produced during compilation.

Before you can successfully execute `runner`, you must set the environment variable LD_LIBRARY_PATH to prompt the loader to search for libraries in the current directory as well when starting programs. Otherwise, if the loader cannot locate **libzc_io.so**, it won't be able to execute the programs that use it. You can set LD_LIBRARY_PATH as follows:

- Call **call_runner.sh** script. This script sets the LD_LIBRARY_PATH and then calls the runner. You may have to set 'execute' permission on the **call_runner.sh** script by running (one time only):
  ```
  $ chmod +x call_runner.sh
  ```

  After this, you can compile and run the **runner** after each change in your code:
  ```
  $ make
  $ ./call_runner.sh
  ```
  Note that **call_runner.sh** creates a new process that sets the `LD_LIBRARY_PATH` and then calls **runner**. However, the variable is set only for that process and thus the modification is not persistent.

If you don't set up your LD_LIBRARY_PATH accordingly, you will encounter this error when trying to execute the runner:

```
./runner: error while loading shared libraries: libzc_io.so:
cannot open shared object file: No such file or directory
```

Dynamic Libraries

Here we describe how to create and use dynamic libraries. Note that you can solve the assignment without this knowledge, and you may skip this subsection if it's not your cup of tea.

A dynamic or shared library is created with the purpose of being linked at run-time by other programs. The library can be linked by many programs at the same time, despite having only one instance of it loaded in memory - this can greatly reduce the memory consumption.

On Unix-like systems, dynamic libraries have the extension .so, from (dynamic) shared object, whereas their counterparts on Windows have the extension .dll, from dynamic-link library. Our focus will be on dynamic libraries for Unix-like systems.

Creating a dynamic library

To create a dynamic library, the `-fPIC` flag must be used during compilation. PIC stands for Position Independent Code and ensures that the generated machine code does not require to be located at a specific virtual memory address in order to work properly. This allows multiple processes to share the library code because they can map it anywhere in their own virtual address space without affecting the proper functionality of the library.

Let's say we want to create a dynamic library from our source file, `foo.c`. As you may expect, the first step is to compile it:

```
$ gcc -Wall -fPIC -c foo.c
```

Next, we have to turn the resulting object file `foo.o` into a shared library, which we shall call `libfoo.so`. To do so, we run:

```
$ gcc -shared -o libfoo.so foo.o
```

The `-shared` flag allows us to create a shared object that can later be linked by other files to form an executable.

Using a dynamic library

To allow `bar.c` to use functionalities defined in `libfoo.so`, we have to link `libfoo.so` during the compilation of `bar.c` using the `-l` option. In addition, we also have to specify where the library is located in the system using the `-L` option. Thus, the compilation command will look similar to this:

```
gcc -L/path/to/directory/containing/foo -o bar bar.c -lfoo
```

GCC assumes libraries to be starting with **lib** and end with **.so** or **.a**, thus `-lfoo` will look for `libfoo.so`.

The last step is to inform the loader (i.e., the part of the OS that's responsible for loading programs and libraries) that it should be looking in `/path/to/foo` as well when searching for libraries during the program loading. This can be done by adding to `/path/to/foo` to the LD_LIBRARY_PATH environment variable. Earlier, we instructed you to add ., i.e., the current directory, to the LD_LIBRARY_PATH, so whenever you try running the **runner** or **demo**, the directory from which you are running the command will also be searched for libraries.

Miscellaneous

`ldd`: The `ldd` command prints the dynamic libraries required by a program. You can test it on the **runner** executable before and after setting up LD_LIBRARY_PATH!

```
$ ldd runner
```

You may see where different libraries are mapped in a process' address space using $ `cat /proc/<PID>/maps`.

Section 3: Implementing the Assignment

The goal in this lab assignment is to produce a zero-copy IO library. All the function names and data structures names are prefixed by `zc_`.
The library uses a data structure called `zc_file` to maintain the information about the open files and help in the reading and writing operations.

This `zc_file` structure has been defined for you in `zc_io.c`. You are required to add any information needed to maintain the information about the opened files into this data structure.

Please note that multiple files can be manipulated at the same time. As such, you should avoid using global variables to maintain the state of the open files. This requirement can be achieved by packing in `zc_file` all the necessary information about an open file.

For exercises 1 to 3, you may assume that the operations on the same file will not be issued concurrently (i.e. you do not need to be concerned about synchronization). We will change this assumption in exercise 4.

The provided runner implements a few testcases on reading and writing a file using the `zc_io` library. It is not exhaustive but will catch most common errors. If your implementation is correct, the runner will run successfully. Otherwise, it may segmentation fault, or print a "FAIL" message with the reason of the failure.

Exercise 1: Zero-copy Read

You are required to implement four library calls to open/close and perform zero-copy read from a file. Additionally, you are required to update your `zc_file` structure.

`zc_file *zc_open(const char *path)`
Opens file specified by path and returns a `zc_file` pointer on success, or `NULL` otherwise. Open the file using the `O_CREAT` and `O_RDWR` flags.

`int zc_close(zc_file *file)`
Flushes the information to the `file` and closes the underlying file descriptor associated with the `file`. If successful, the function returns `0`, otherwise it returns `-1`. Free any memory that you allocated for the `zc_file` structure.

`const char *zc_read_start(zc_file *file, size_t *size)`
The function returns the pointer to a chunk of `*size` bytes of data from the `file`. If the file contains less than `*size` bytes remaining, then the numbernumber of bytes available should be written to `*size`.

The purpose of `zc_read_start` is to provide the kernel buffer that already contains the data to be read. This avoids the need to copy these data to another buffer as in the case of `read` system call. The user can simply use the data from the returned pointer.

Your `zc_file` structure should help you keep track of the offset in the file. Once `size` bytes have been requested for writing, the `offset` should advance by `size` and the next time when `zc_read_start` or `zc_write_start` is called, the next bytes after `offset` should be offered. Note that reading and writing is done using the same offset.

`void zc_read_end(zc_file *file)`
The function is guaranteed to be always paired with a previous call to `zc_read_start`. This function is called when a reading operation on `file` has ended.

Reading from a file using the `zc_io` library call should have the same semantic behaviour as observed in `read` system call.

Exercise 2: Zero-copy Write

You are required to implement two library calls that allow writing to file:

`char *zc_write_start(zc_file *file, size_t size)`
The function returns the pointer to a buffer of at least `size` bytes that can be written. The data written to this buffer would eventually be written to `file`.

The purpose of `zc_write_start` is to provide the kernel buffer where information can be written. This avoids the need to copy these data to another buffer as in the case of `write` system call. The user can simply write data to the returned pointer.

Your `zc_file` structure should help you keep track of the offset in the file. Once `size` bytes have been requested for writing, the `offset` should advance by `size` and the next time when `zc_read_start` or `zc_write_start` is called, the next bytes after `offset` should be offered. Note that reading and writing is done using the same offset.

File size might change when information is written to file. Make sure that you handle this case properly.

`void zc_write_end(zc_file *file)`
The function is guaranteed to be always paired with a previous call to `zc_write_start`. This function is called when a writing operation on `file` has ended. The function pushes to the file on disk any changes that might have been done in the buffer between `zc_write_start` and `zc_write_end`. This means that there is an implicit flush at the end of each `zc_write` operation.

Writing to a file using the `zc_io` library call should have the same semantic behaviour as observed in `write` system call.

Exercise 3: Repositioning the file offset: zc_lseek()

You are required to implement one library call that allows changing the offset in the file:

```
off_t zc_lseek(zc_file *file, long offset, int whence)
```

Reposition at a different `offset` within the `file`. The new position, measured in bytes, is obtained by adding `offset` bytes to the position specified by `whence`. `whence` can take 3 values:

- `SEEK_SET`: offset is relative to the start of the file
- `SEEK_CUR`: offset is relative to the current position indicator
- `SEEK_END`: offset is relative to the end-of-file
  (The `SEEK_SET`, `SEEK_CUR` and `SEEK_END` values are defined in `unistd.h` and take the values 0, 1, and 2 respectively.)

The `zc_lseek()` function returns the resulting offset location as measured in bytes from the beginning of the file or `(off_t)-1` if an error occurs.

`zc_lseek()` allows the file offset to be set beyond the end of the file (but this does not change the size of the file). If data is later written at this point, subsequent reads of the data in the gap (a "hole") return null bytes (`\0`) until data is actually written into the gap.

Exercise 4: Readers-writers Synchronization

Exercises 1, 2, 3 assumed that the operations on the same file would be issued in sequence. In exercise 4 we lift this assumption and allow multiple reads and writes to be issued at the same time for the same instance of an open file.

Note that that on operation is considered to take place between the time `zc_*_start` was called and until `zc_*_end` has completed.

You need to make sure that your `zc_read`, `zc_write` and, `zc_lseek` executed on an open file follow the following rules:

- Multiple `zc_read` operations can take place at the same time for the same instance of the `zc_file`.
- No other operation should take place at the same time with a `zc_write` or `zc_lseek` operation.
- All operation issued while `zc_write` or `zc_lseek` is executing would block waiting to start. They would start only once the `zc_write` or `zc_lseek` ends.

In other words, you should solve the readers-writers synchronization problem when multiple operations are issued at the same time for the same instance of an open file. You are not required to ensure that your solution is starvation-free.

Exercise 4B: Per-page Synchronisation

Instead of locking the whole file to achieve your synchronization, lock the file per-page instead. That means that, for example, if one or more threads have started a read from page N, then another thread that calls `zc_write_start` on page N will block until all readers have ended their read.

For this bonus exercise only, you can modify the function definitions found in `zc_io.h` and `zc_io.c` according to your needs.

Exercise 5: Zero-copy file transfer

You are required to implement the following library call:

```
int zc_copyfile(const char *source, const char *dest)
```

This function makes a copy of `source` into `dest`. You should not use any user buffers to achieve this.
You can use additional system calls.