CS2106: Introduction to Operating Systems

Lab Assignment 4 (A4)

Implementing Zero-copy File Operations

IMPORTANT

The deadline of submission through Canvas: **15th April, 2023, 11.59 PM Saturday**

The total weightage is 8% (+2% bonus):

- Ex1 (4%): Zero-copy read & write [Optional Demo Exercise]
- Ex2 (1%): Repositioning the file offset
- Ex3 (2%): Zero-copy file transfer
- Ex4 (1%): Readers-writers synchronization
- Bonus (2%): Per-page synchronization

You must ensure the exercises work properly on the SoC Cluster.

1 Introduction

1.1 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:[1]

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

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.

1. 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.
2. Data is copied from the kernel buffer into the user buffer, and the read system call returns. The return from the call causes 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.
3. 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.

[1] file and socket are two file descriptors.

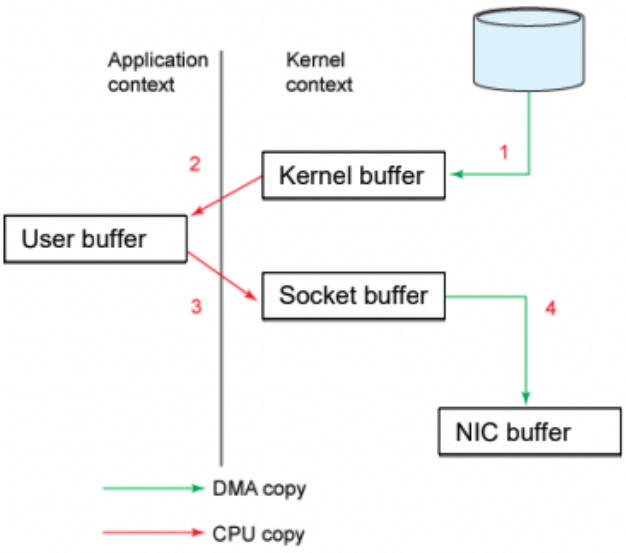

Figure 1

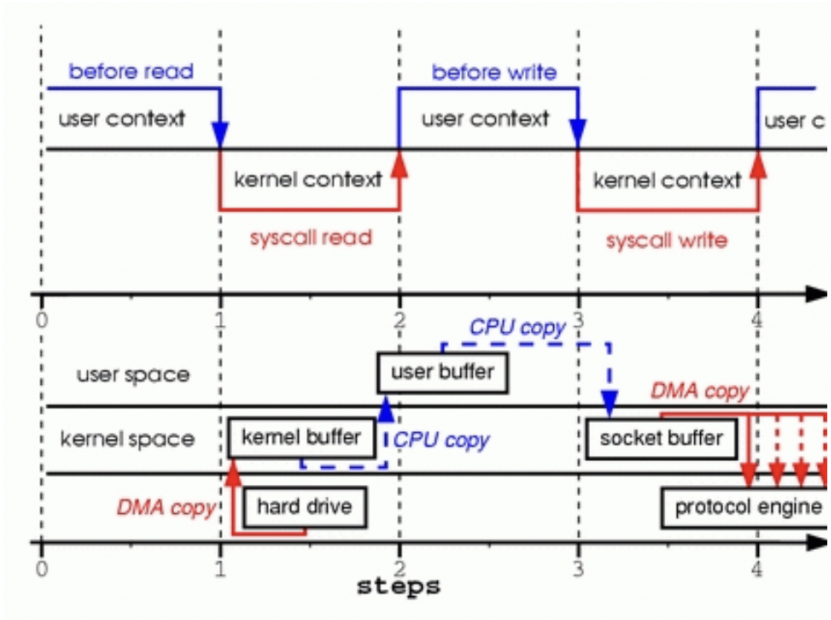

Figure 2

4. 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.

1.2 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 develop a new library with a 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.** Refer to the references[2] and [3] to help you understand `mmap`. DO NOT PLAGIARISE.

A Creating a Zero-copy IO Library

We provide a `Makefile` to make the process of compilation easier. Running `make` in the main folder 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[4]:

   ```
   -L<directory_path_containing_libzc_io.so> -lzc_io
   ```

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.[5] Oth-

[4] `<directory_path_containing_libzc_io.so>` would be the directory containing the library
[5] https://tldp.org/HOWTO/Program-Library-HOWTO/shared-libraries.html

erwise, if the loader cannot locate `libzc_io.so`, it won't be able to execute the programs that use it. 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
```

You may execute the runner using the command below[6]:

```
LD_LIBRARY_PATH=. ./runner
```

[6] https://www.gnu.org/software/bash/manual/html_node/Environment.html

2 Exercises in Lab 4

The goal of this lab assignment is to produce a zero-copy IO library. All function and data structures names are prefixed by `zc_`. The library uses a data structure called `zc_file` (defined in `zc_io.c`) to maintain the information about the opened files and help in the reading and writing operations. You are required to use it and add any information needed to maintain the information about the opened files into this data structure.

Some info you may want in `zc_file` is suggested in the comments of the skeleton code. These include size of file, **offset** that keeps track of the current position for read/write operations, its pointer in the virtual memory, its file descriptor, and relevant mutexes (for ex4 & bonus).

For ex1 to ex3, 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 ex4 and bonus exercise. For all exercises, you may assume that there is **no concurrent opening of the same file** (the file is opened at most once at the same time, and the file is not modified outside the runner).

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 some 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. You are also encouraged to implement your own program to test the library.

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 opened files. This requirement can be achieved by packing in `zc_file` with all the necessary information about an opened file.

2.1 Exercise 1A: Zero-copy Read [1% + 1% demo or 2% submission]

You are required to implement four library calls to open/close and perform zero copy read from a file.

```
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.
You can use `fstat()` to obtain information (if needed) regarding the opened file.

```
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. You can use `msync()` flush copy of file in virtual memory into file.

```
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 number 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. Instead, the user can simply use the data from the returned pointer.
Your `zc_file` structure should help you keep track of a `offset` in the file. Once `size` bytes have been requested for reading (or 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)
```

This function is called when a reading operation on file has ended.
It is always guaranteed that the function is paired with a previous call to `zc_read_start`.

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

You might find two possible approaches for implementing this exercise:

1. `mmap`/`munmap` in `zc_open`/`zc_close` (recommended)
2. `mmap`/`munmap` in `zc_read_start`/`zc_read_end`

and `mremap` whenever necessary. Useful flags to look into include:

1. `mmap`: `PROT_READ` / `PROT_WRITE` / `MAP_SHARED_VALIDATE`
2. `mremap`: `MREMAP_MAYMOVE`

2.2 Exercise 1B: Zero-copy Write [1% + 1% demo or 2% submission]

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.
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 written. 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. See ftruncate.

```
void zc_write_end(zc_file *file)
```

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. You can check out `msync()` to help you with flushing.
It is always guaranteed that the function is paired with a previous call to `zc_write_start`.

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

2.3 Exercise 2: Repositioning the file offset [1%]

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. (Please refer to Appendix B for a simple example on this.)

Repositioning the file offset should have the same semantic behaviour as lseek system call.

2.4 Exercise 3: Zero-copy file transfer [2%]

You are required to implement the following library call:

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

This function copies the content of `source` into `dest`. It will return 0 on success and -1 on failure. You should make use of the function calls you implemented in the previous exercises, and should not use any user buffers to achieve this. Do `ftruncate` the destination file so they have the same size.c

2.5 Exercise 4: Readers-writers Synchronization [1%]

Exercises above assumed that the operations on the same file would be issued in sequence. In ex4 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.

You need to make sure that your `zc_read_start`, `zc_write_start` 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.

While multiple readers can read at the same time, ensure that the offset variable of the file is protected and multiple `zc_write_start` or especially `zc_read_start` access and increment the offset variable one at a time. For eg. if two threads read 10 bytes each, with initial offset = 0, one of the threads should read the first 10 bytes, the other the next 10 bytes, and the final value of offset should be 20.

Note: if you are attempting bonus exercise, two new functions will be added to the library, they will not be tested for exercise 4. However, all functions in the library will be tested in the bonus exercise.

2.6 Bonus Exercise: Per-page Synchronisation [2%]

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_offset` on page N will block until all readers have ended their read. You can use `sysconf()` to get configuration information such as page size.

We also allow the specifying of an offset explicitly instead of using the tracked offset since multiple threads may want access at different offsets. We add two new functions:

```
const char *zc_read_offset(zc_file *file, size_t *size, long offset)
```

Similar to previously implemented `zc_read_start`, except we use a specified `offset` instead of the offset we are tracking.

The function returns the pointer to a chunk of `*size` bytes of data from the file, offset by `offset` bytes from the start of the file.
If the file after offset contains less than `*size` bytes remaining, then the number of bytes available should be written to `*size`.

```
char *zc_write_offset(zc_file *file, size_t size, long offset)
```

Similar to previously implemented `zc_write_start`, except we use a specified `offset` instead of the offset we are tracking.

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` at offset `offset` bytes from the start of the file.
File size might change when information is written to file.

Files should be able to be read with either `zc_read_start` or `zc_read_offset`, and written with either `zc_write_start` or `zc_write_offset`.

`zc_read_end` and `zc_write_end` should still be called at the end of each read and write operation respectively (each read/write will be paired with one subsequent `zc_read_end`/`zc_write_end`).

Note: we will not accept a separate file for your bonus part. Make sure that your attempt doesn't corrupt other exercises. Before attempting, you are recommend to make a copy of your implementation, or use a version control system, such as git.

3 Submission through LumiNUS

Your current directory should have the following files:

```
check_zip.sh
zc_io.c
```

You will check and zip your work by executing `check_zip.sh` once you have the folders and files as above. This is to avoid any folder archiving errors.

```
$ chmod +x ./check_zip.sh
$ ./check_zip.sh E0123456 # (replace with your NUSNET id file name)
    or
$ ./check_zip.sh E0123456_E0123457.zip
```

Use your NUSNET ID 'E0xxxxxx', instead of your student number. Use capital 'E' as prefix.

The script checks the following:

a. You provided a valid NUSNET ID starting with a capital 'E'.
b. Your current directory contains all the folder and files that needs to be zipped. The folder structure follows the structure presented above.
c. Each exercise can be compiled properly.

During execution, the script prints if the checks have been successfully conducted, and which checks failed. Successfully passing checks ensures that we can grade your assignment.

You can check your file structure matches the above format by using the following command:

```
$ unzip -l E0123456.zip          % replace with your zip file name
```

Please ensure that you follow the instructions carefully. Deviations will be penalized.

E0123456.zip will be created containing a top folder with the files listed above. Upload the created **zip file** to the "Student Submissions Lab 4" folder on Canvas. Note the deadline for the submission is **15th April, 2023, 11.59 PM Saturday**.

A 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 section if it's not your cup of tea.

A dynamic or shared library is created with the purpose of being linked at runtime 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.

A.1 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.

A.2 Linking a dynamic library to an executable

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` or `libfoo.a`.

A.3 Execute a dynamically linked library

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.

A.4 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
```

B Seek beyond the end

> POSIX allows seeking beyond the existing end of file. If an output is performed after this seek, any read from the gap will return zero bytes. Where supported by the filesystem, this creates a sparse file.

The code snippet below writes a 17-bit string into a new file, then seek beyond the end of the file and write something else there.

```c
#include <stdio.h>
#include <unistd.h>

const char *filename = "example.txt";

int main() {
    truncate(filename, 0);

    FILE *pFile;
    pFile = fopen(filename, "wb");
    fputs("This is an apple.", pFile);
    fseek(pFile, 32, SEEK_SET);
    fputs(" sam", pFile);
    fclose(pFile);
    return 0;
}
```

The content of the generated `example.c` is as below:

```
00000000  54 68 69 73 20 69 73 20  61 6e 20 61 70 70 6c 65  |This is an apple|
00000010  2e 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000020  20 73 61 6d                                       | sam|
00000024
```