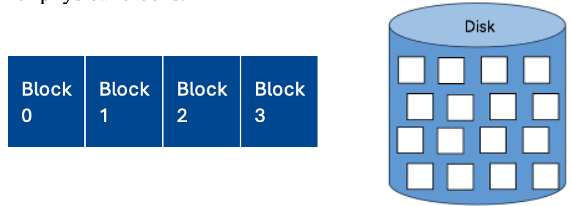
### **Contributions**

All members in the group contributed equally to all questions.

### **Question 1: Allocation Methods. ELIN**

Recall the allocation methods in File System Implementation and the logical-to-physical address mapping. Assume a file system uses 512 bytes for logical blocks and also 512 bytes for physical blocks.



Assume free device blocks are tacked by a bit vector that is used for implementing the free-space list, and ‘1’ means the device block is free. In this exercise, **First Fit** is used when requesting a new block from the free-space list.

{00111100111111111000000101010}

Given the following list of files:

{File0, 1024 bytes}, {File1, 500 bytes}, {File2, 128 bytes}, {File3, 4096 bytes}

(a) Use the contiguous allocation strategy, specify the logical-to-physical address mapping for each file. Hint: the output should look similar to the following. You may also list the updated free-space list after allocating each file.

|  |  |  |
| --- | --- | --- |
| File name | Start (physical block) | List of physical blocks |
| File0 | 2 | 2, 3 |
| File1 | 4 | 4 |
| File2 | 5 | 5 |
| File3 | 8 | 8-15 |

Updated free-space list: {00000000000000001000000101010}

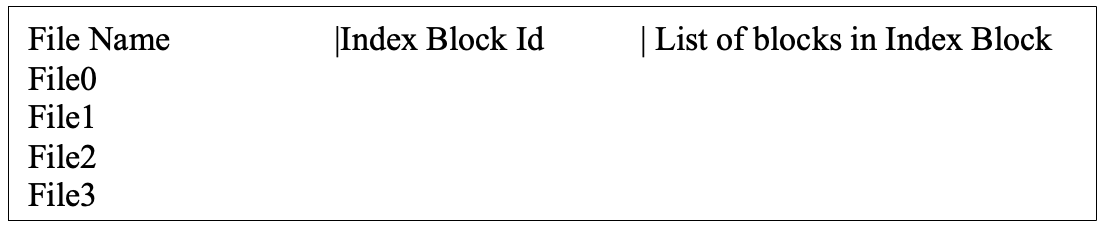
(b) Use the linked allocation, assume the block pointer needs 12 bytes. Specify the logical-to-physical address mapping for each file. Hint: the output should look similar to the following.

With a pointer of size 12 bytes, the blocks have an available size of 500 bytes.

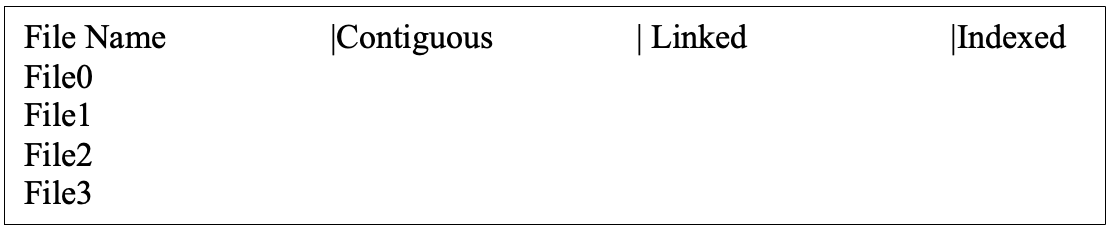
{00**1111**00**1**11111111000000101010}

|  |  |  |  |
| --- | --- | --- | --- |
| File name | Start (physical block) | Linked list of physical blocks | Logical-to-physical mapping |
| File0 1024 | 2 | 2, 3, 4 |  |
| File1 500 | 5 | 5 |  |
| File2 128 | 8 | 8 |  |
| File3 4096 | 9 | 9, 10, 11, 12, 13, 14, 15, 16, 23 |  |

(c) Use the indexed allocation, specify the logical-to-physical address mapping for each file. Hint: the output should look similar to the following.



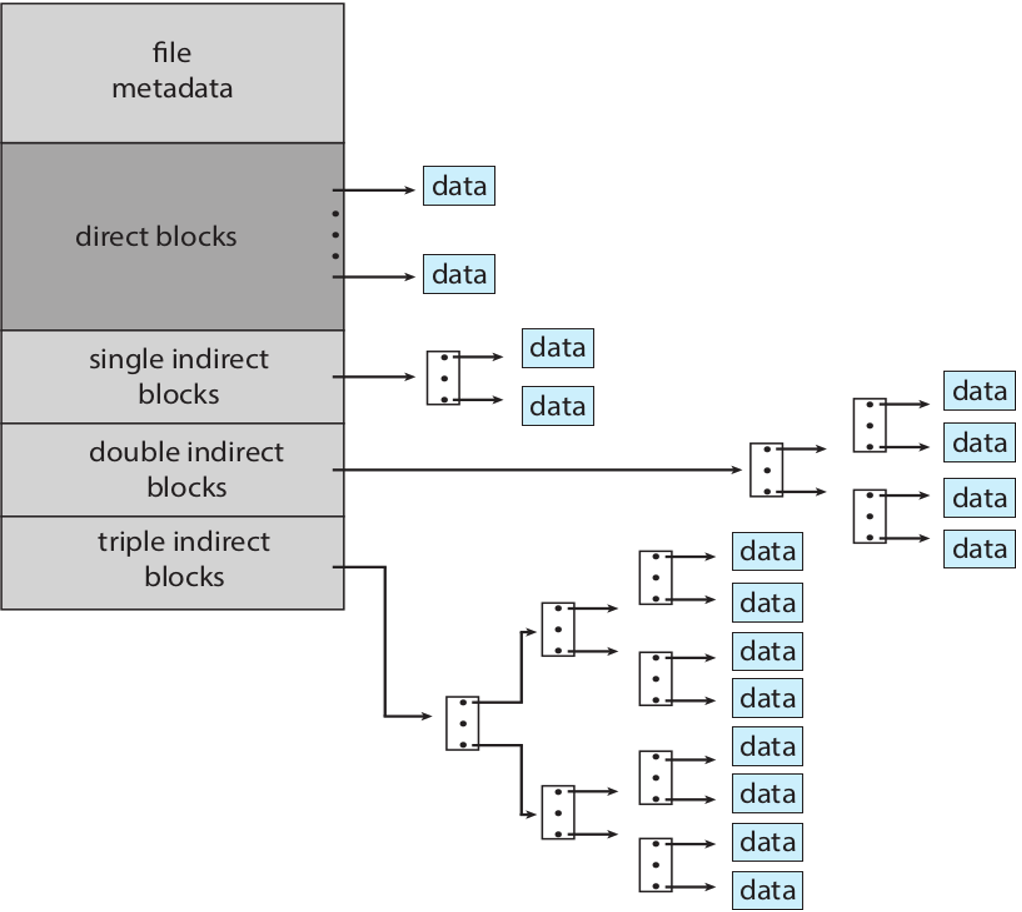
(d) Calculate the fragmentation ratio for each file using the equation = (File Size) / (Allocated Physical Blocks \* Block Size). Hint: the output should look similar to the following.



(e) If we are currently at logical block 2 of File 3 and want to access its logical block 4. For each allocation method, how many physical blocks must be read from the disk? Please briefly explain why.

### **Question 2: File System Structures. lydia**

Recall that file control block (FCB) contains (1) file metadata such as file ownership, timestamp of last modified, and (2) information of file content such as data blocks or pointers to data blocks. In such a file system, each file has a per-file FCB to represent the file. In UNIX file systems, a FCB is implemented as an inode entry as illustrated below.



UNIX file systems use Indexed Allocation with Combined scheme. 15 pointers to blocks are kept in the file’s inode. Assume that each data block on the disk is 8 KB in size, and a pointer to a disk block requires 8 bytes. Assume the inode is already copied in memory.

* The first 12 pointers directly point to data blocks.
* The 13th pointer points to a single index block, which contains a list of pointers to data blocks.
* The 14th pointer points to a double indirect block, which contains a list of pointers to single index block.
* The 15th pointer points to a triple indirect block, which contains a list of pointers to double index block.

Answer the following questions:

1. Explain and calculate the size of data that can be reached with one disk I/O operation?

Each data block on the disk is 8KB in size. With one disk I/O operation all data blocks which the pointers directly point to can be reached. Therefore the size of data that can be reached is 8\*12=96KB.

1. Explain and calculate the size of data that can be reached with two disk I/O operations?

The 13:th pointer points to a single index block which contains a list of pointers to data blocks. The data block is 8 KB and a pointer requires 8 bytes. This means the index block contains 1024 pointers. The data accessible with two disk I/O operations is therefore 1024\*8KB=8MB.

1. Explain and calculate the size of data that can be reached with three disk I/O operations?

A double indirect block contains 1024 pointers to single indirect blocks, and each one of those pointers contains 1024 pointers to 8 KB data blocks. This makes the total data reachable with three disk I/O operations 1024\*1024\*8 = 8 GB.

1. Explain and calculate the size of data that can be reached with four disk I/O operations?

A triple indirect block contains 1024 pointers to double indirect blocks, which each has 1024 pointers to single indirect blocks, each pointing to 1024 data blocks. This makes the total data reachable from four disk I/O operations 1024\*1024\*1024\*8 = 8 TB.

1. What is the maximum size of a file that can be stored in this file system?

The maximum size of a file that can be stored in this file system is the sum of 1-4, which is about 8.008 TB.

### **Question 3: File Directories ERIK**

Recall that acyclic-graph directories support files to be shared. Soft and Hard links can be used for implementation. In this experiment, you will create hard and soft linked files and investigate their behaviors under file modification and deletion.

First, create a new file called file\_original.txt and put some random text in it. Now, create a soft link to the newly created file using the following command

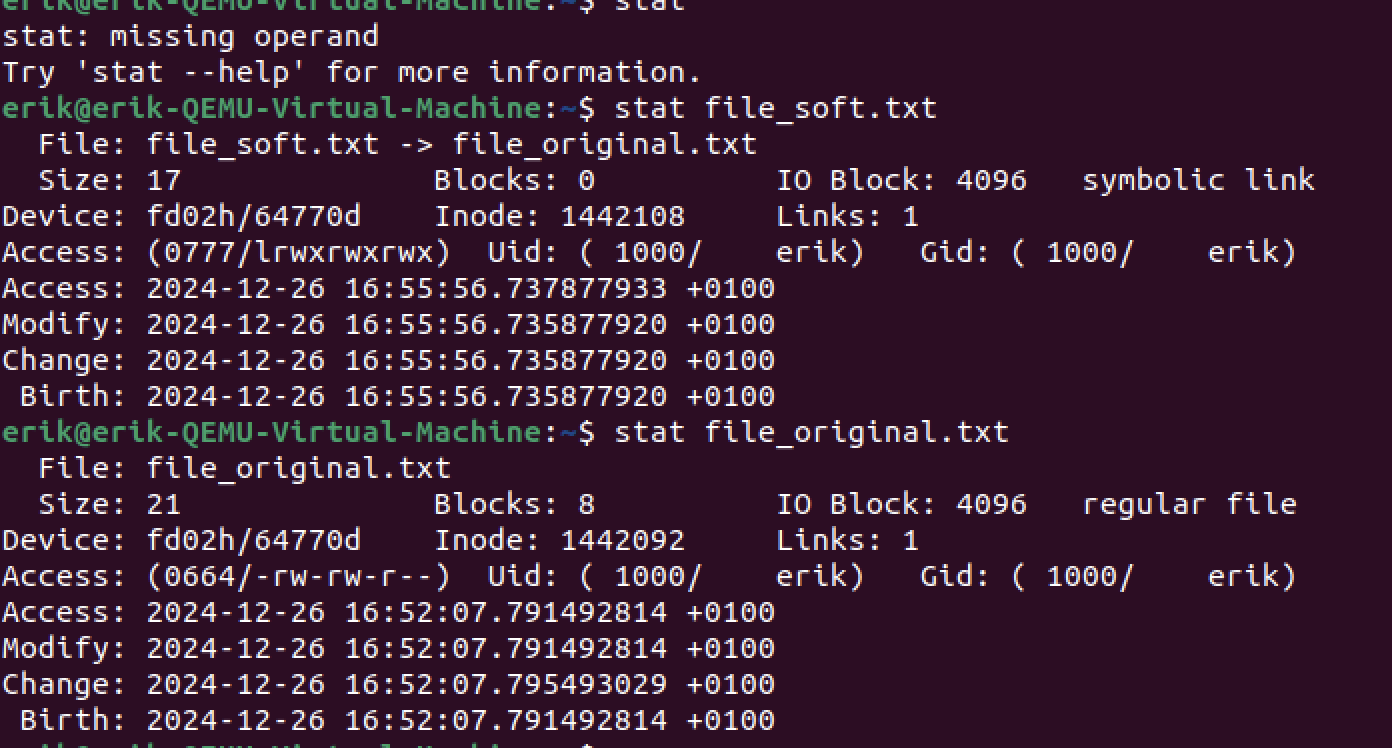
'ln -s <source file> <target file>'

(a) Specify the command did you use to create a soft link that creates a new file called ‘file\_soft.txt’ pointing to ‘file\_original.txt’.

‘ln -s file\_original.txt file\_soft.txt’

(b) Which command as introduced in the lecture can be used to find the inode number of a file? Are the inodes the same for ‘file\_soft.txt’ and ‘file\_origin.txt’?

The command introduced in the lecture is ‘stat <file>’. These are the results from running the command on both ‘file\_soft.txt’ and ‘file\_origin.txt’.



As one can see the inodes are not the same for the two. This is because a soft link is a reference or pointer to the original file, and thus is a completely different file storing the address of the original file and not the contents themselves.

(c) Now edit the contents of ‘file\_soft.txt’ by appending some random text to it. Have the contents of ‘file\_origin.txt’ been altered as well?

When adding contents to the file ‘file\_soft.txt’ the contents of file\_origin.txt change as well.

(d) Delete ‘file\_origin.txt’ with command ‘rm’. Can you still edit ‘file\_soft.txt’? Explain why.

No. This is because a soft link does not store any actual data, but rather acts like a pointer or reference to the file path of the original text file. Once the file has been deleted the soft link points to a non-existent file, and therefore the contents of the file can no longer be edited.

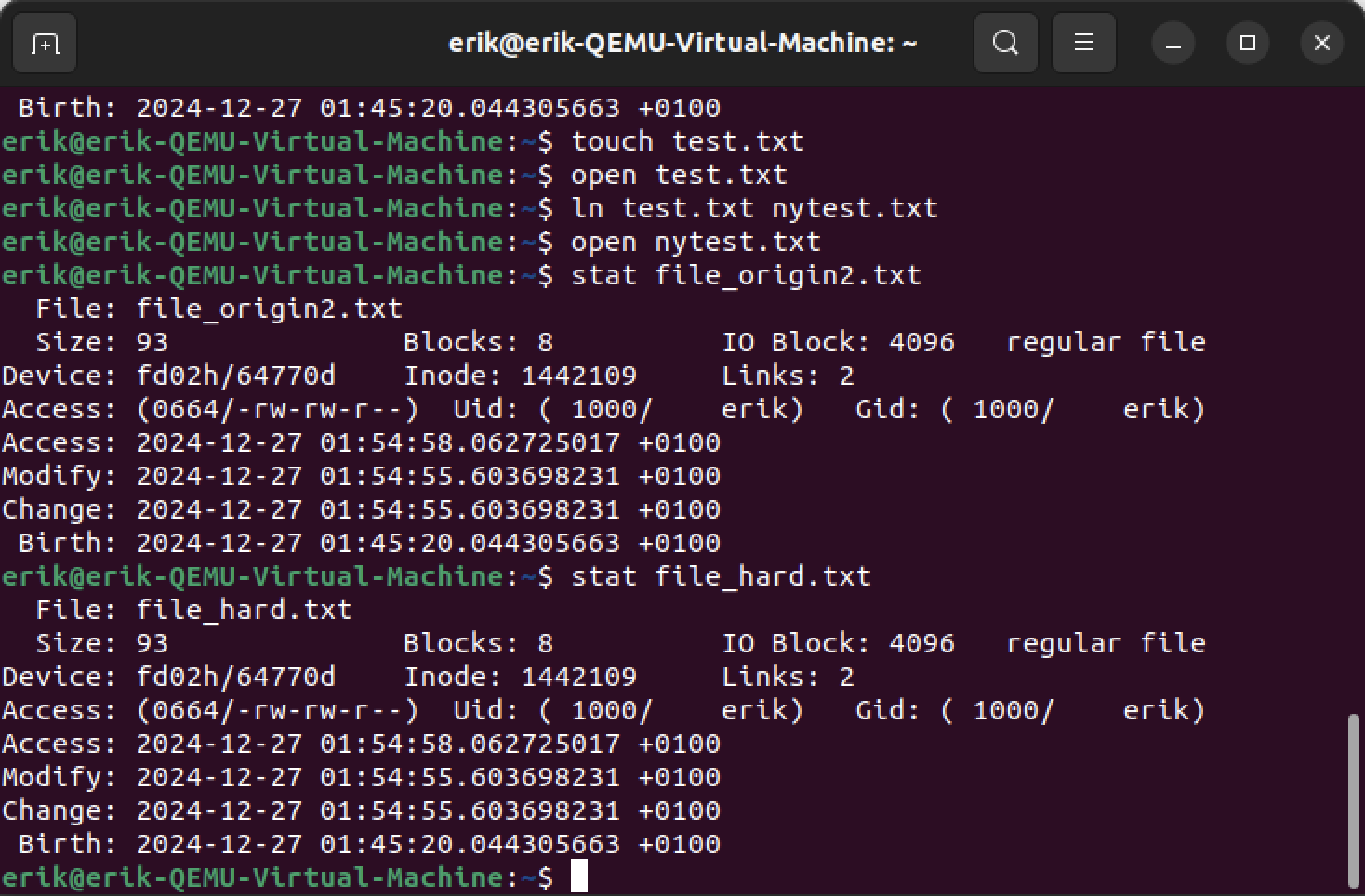
Now, create ‘file\_origin2.txt’ and put some random text in it. Create a hard link to the newly created file using the following command

'ln <source file> <target file>'

(a) Specify the command did you use to create a hard link that creates a new file called ‘file\_hard.txt’ pointing to ‘file\_origin2.txt’.

Ln file\_origin2.txt file\_hard.txt

(b) Are the inodes the same for ‘file\_hard.txt’ and ‘file\_origin2.txt’?

The inodes for the two files are the same, as seen below.

(c) Now edit the contents of ‘file\_hard.txt’ by appending some random text to it. Have the contents of ‘file\_origin2.txt’ been altered as well? Are the inodes for ‘file\_hard.txt’ and ‘file\_origin2.txt’ still the same? Explain why.

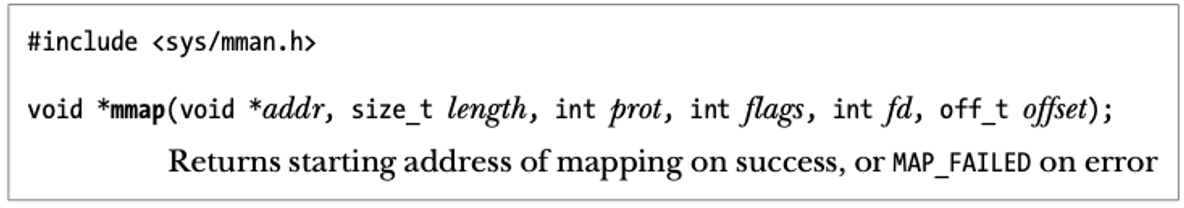
When editing the contents of ‘file\_hard.txt’ the contents of ‘file\_origin2.txt’ have been altered as well. This is because the hard link makes two files pointing to the same set of data on the disk. When changing the data from one of the files, it changes the underlying data on the disk - and thus both files will be updated to show the same text.

(d) Delete ‘file\_origin2.txt’ with command ‘rm’. Can you still edit ‘file\_hard.txt’? Explain why.

Yes, this is because the two files share the same inode, which contains a counter for amount of linked files. Unlike a soft link, the underlying data on the disk is not deleted until the amount of linked files is 0 meaning deleting the original file file\_origin2.txt will not delete the data on the disk if there is another linked file.

### **Question 4: Memory-mapped Files KRISTOFFER**

This is a programming assignment in C. You will create a simple C program that creates file-backed memory mapping in the virtual address space using *mmap().* You may use the skeleton code in the lecture node ‘Memory-mapped Files’ and submit your final code.

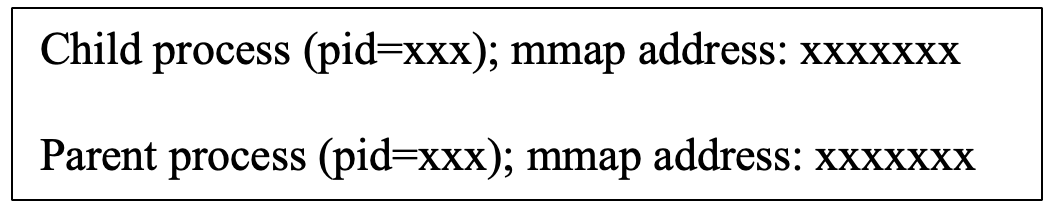


First, create a file named ‘file\_to\_map.txt’ of 1MB size. Which Linux file utility command did you use to create such a file? Hint: this step can be performed outside your code.

**truncate -s 1M file\_to\_map.txt**

In your code, use fork() to create a child process. Both child and parent process will now use *mmap* to create a memory mapping to file ‘file\_to\_map.txt’. For mmap(), child and parent processes will use shared mapping, i.e., MAP\_SHARED when mapping the file.

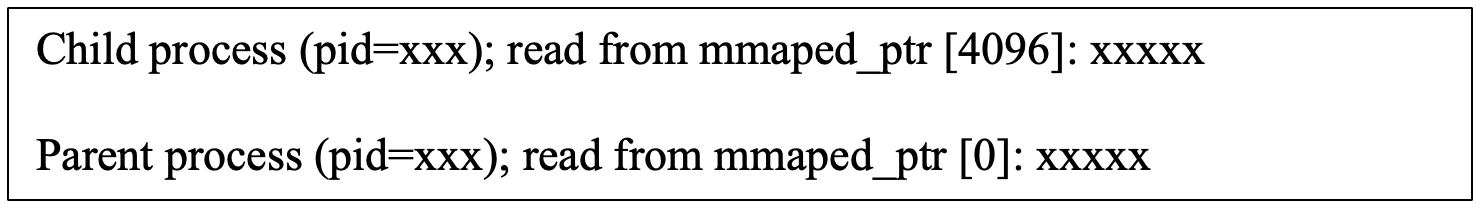
Run your code and report the screenshot showing the virtual address (stored in a variable named *mmaped\_ptr*) returned from *mmap*. The output should look like the following

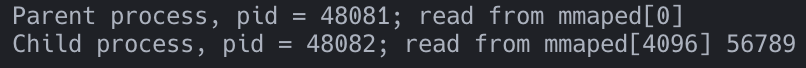




Next, both child and parent processes will try to write to the file-backed memory region. The child process will write ‘01234’ to location starting from mmaped\_ptr[0]’ and then read five characters from mmaped\_ptr[4096]. The parent process will write ‘56789’ to location starting from mmaped\_ptr[4096] and then read five characters from mmaped\_ptr[0].

Report the screenshot showing the characters read by child and parent processes, respectively. The output should look like the following

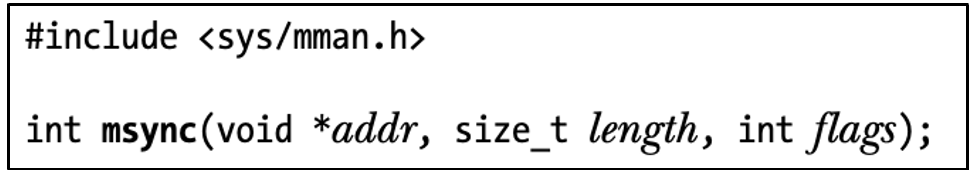




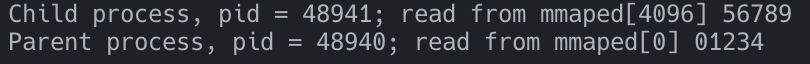
Run your program 5-10 times. Do you always observe the same output from step (b)?

The process ID will differ between runs, but roughly the same thing will always occur. One process will read nothing from the file as the other process has not yet written to the given memory location. This is also due to the function “*clear\_file()*” in the code which ensures the file is empty at runtime each time, in order to simulate a new file each time. If this were removed the first process would instead read every time except for the first time as the results from the last couple of runs would persist.

Recall that *msync()* forces data to be written to the disk and ensures the updates become visible to other processes read from the file.



How to modify your code so that a process only read from the file after the other process has issued msync()? Run your program 5-10 times and report the screenshot showing the output.



After also using *msync()* and a semaphore to wait for the other process to finish writing both processes will always read the correct value. Interestingly the processes will also swap order, as the one writing first will also issue the finished flag first and thus the other one will read first.