

**OPERATING SYSTEMS**

**BEEC3453**

**SEMESTER 1**

**SESI 2021/2022**

**LAB 3: MEMORY MAPPING**

<b>NO.</b>	<b>STUDENTS' NAME</b>	<b>MATRIC. NO.</b>
1.	NOOR SHAFINA BINTI ABDUL GHANI	B081910293
2.	NORISA SHAFIKA BINTI MOHD GHANI	B081910039
3.	NUR ALYSHA BINTI NORAZIZAN	B081910155
4.	RAHMAN KAZI ASHIKUR	B081910450

<b>PROGRAMME</b>	3BEEC
<b>SECTION GROUP</b>	/ S1/1
<b>DATE</b>	15/11/2021
<b>NAME OF INSTRUCTOR(S)</b>	1. NOOR MOHD ARIFF BIN BRAHIN
	2.

<b>EXAMINER'S COMMENT(S)</b>	<b>TOTAL MARKS</b>

Rev. No.	Date	Author(s)	Description
1.0	30 JAN 2019	1. Shamsul Fakhar 2. Noor Mohd Ariff	1. Update to new UTeM logo 2. Update faculty's name 3. Change "course" to "programme" 4. Remove verification stamp

## 1. LEARNING OUTCOMES

1. Differentiate the functionality among various kinds of OS components.
2. Manipulate OS theories to solve basic functional problems.
3. Perform lab and present technical report in writing.

## 2. REQUIREMENTS

1. PC with Linux Ubuntu installed (or any other POSIX-compliant system)
2. Knowledge of C programming language

## 3. SYNOPSIS & THEORY

POSIX environments provide at least 2 ways of accessing files. There's the standard system calls `open()`, `read()`, and `write()`; but there's also the option of using `mmap()` to map the file directly into virtual memory. The programmer can then access the file directly through memory, identically to any other chunk of memory resident data. It is even possible to allow writes to the memory region to transparently map back to the file on disk.

`mmap()` is great if you have multiple processes accessing data in a read only fashion from the same file, which is common in server systems. `mmap()` allows all those processes to share the same physical memory pages, saving a lot of memory.

`mmap()` also allows the operating system to optimize paging operations. For example, consider two programs; program A which reads in a 1MB file into a buffer created with `malloc`, and program B which `mmap()`s the 1MB file into memory. If the operating system has to swap part of A's memory out, it must write the contents of the buffer to swap before it can reuse the memory. In B's case any unmodified `mmap()` pages can be reused immediately because the OS knows how to restore them from the existing file they were `mmap()`d from. (The OS can detect which pages are unmodified by initially marking writable `mmap()` pages as read only and catching seg faults, similar to Copy on Write strategy).

`mmap()` is also useful for inter process communication. You can `mmap()` a file as read or write in the processes that need to communicate and then use synchronization primitives in the `mmap()` region (this is what the `MAP_HASSEMAPHORE` flag is for).

#### 4. PROCEDURE

1. In terminal, open a text editor.
2. Save the file as "mytextfile.txt" on your desktop.
3. Type the words below into "mytextfile.txt":

This text file will be mapped into computer memory.
---

4. Close "mytextfile.txt".
5. Create a new file named "mymemorymap.c". Write the code below:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    int fd;          struct stat
    mystat;
    void *pmap;

    fd = open("mytextfile.txt", O_RDWR);

    if(fd == -1)
    {
        perror("Error opening input file!");
        exit(1);
    }

    if(fstat(fd, &mystat) < 0)
    {
        perror("fstat returns an error!");      close(fd);
        exit(1);
    }

    pmap = mmap(0, mystat.st_size, PROT_READ|PROT_WRITE,
    MAP_SHARED, fd, 0);
```

```
// continued on next page -->
```

```
if(pmap==MAP_FAILED)
{
    perror("mmap");
    close(fd);
}

strncpy(pmap, "That", 4);

close(fd);
```

```
    return 0;
}
```

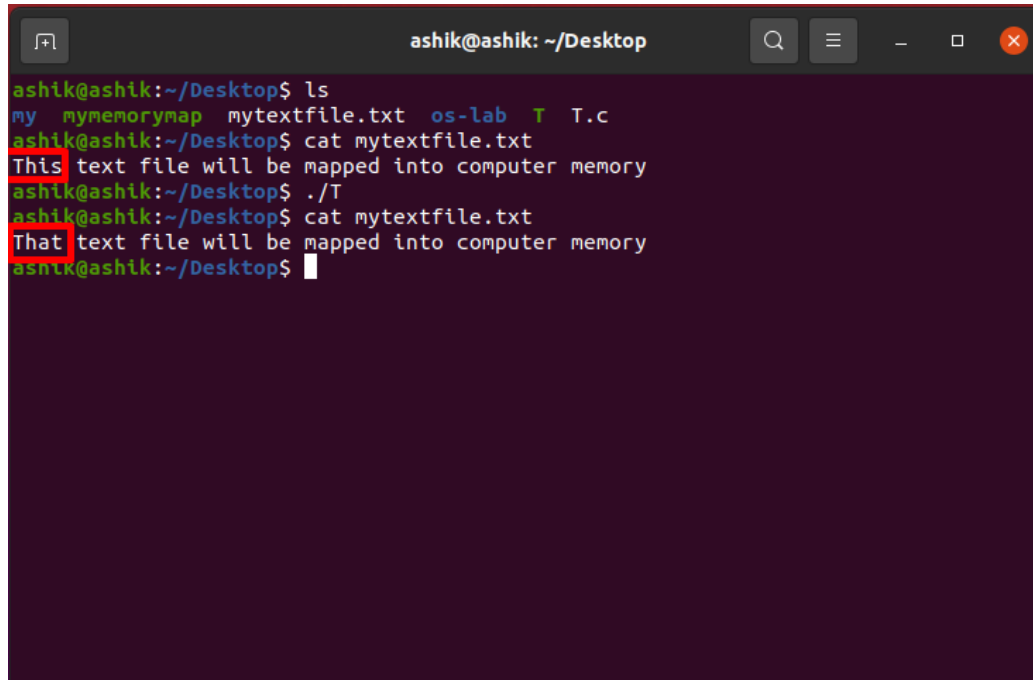
6. Compile the code.
7. In terminal, enter the following command to read the content of the text file:

```
cat mytextfile.txt
```

Run the program.

Read the content of the text file again.

8. Observe the difference in the output before and after the program is run.



A terminal window titled 'ashik@ashik: ~/Desktop' showing the following commands and output:

```
ashik@ashik:~/Desktop$ ls
my  mymemorymap  mytextfile.txt  os-lab  T  T.c
ashik@ashik:~/Desktop$ cat mytextfile.txt
This text file will be mapped into computer memory
ashik@ashik:~/Desktop$ ./T
ashik@ashik:~/Desktop$ cat mytextfile.txt
That text file will be mapped into computer memory
ashik@ashik:~/Desktop$
```

The output shows that after running the program (./T), the content of mytextfile.txt changed from 'This' to 'That'.

After running the program that using the command cat mytextfile.txt, we gain read the content of the text file we see mytextfile.txt file contain change “this” to “that” at the beginning of the text sentence.

## 5. QUESTIONS

1. To call mmap() users need to supply 6 parameters to the function. Briefly describe what each of these parameters are.

```
void * mmap (void *addr,
             size_t len,
             int prot,
             int flags,
             int fd,
             off_t offset);
```

addr = address

This option specifies the mapping's recommended starting address. If no other mapping exists, the kernel will choose a page boundary close and construct the mapping; otherwise, the kernel will choose a new address. If this parameter is NULL, the kernel is free to insert the mapping wherever it wants.

len = length

The number of bytes to be mapped is specified here.

prot = protect

This option is used to limit the types of access that are allowed. This argument could be a logical 'OR' of the flags below. PROT\_READ | PROT\_WRITE | PROT\_EXEC | PROT\_NONE is a PROT\_READ | PROT\_WRITE | PROT\_EXEC | PROT\_NONE. The permissions on the content are read, write, and execute access kinds.

flags = flags

This argument is used to control the nature of the map. As example, MAP\_SHARED where it allows all other processes that are mapped to this object to share the mapping. The file will be updated with any changes made to the mapping region. Then, MAP\_PRIVATE when this flag is set, no other processes will be able to see the mapping, and no changes will be written to the file. Next, MAP\_ANONYMOUS / MAP\_ANON where it is used to create an anonymous mapping. The term "anonymous mapping" refers to a mapping that is not linked to any files. This mapping is used to extend the heap as a basic primitive. MAP\_FIXED where the system must be forced to use the exact mapping address supplied in the address when this flag is set. If this isn't possible, the mapping will be unsuccessful.

fd = filedes

This is the descriptor for the file that needs to be mapped.

offset = offset

This is the offset from the beginning of the file mapping. The mapping connects (offset) to (offset+length-1) bytes for the file open on filedes descriptor in simple terms.

2. Why do we set the flag as MAP\_SHARED?

```
pmap = mmap(0, mystat.st_size, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
```

The flag set as MAP\_SHARED, enables the mmap subroutine to grant read or execute permission for the file open for reading and write access if the file was opened for writing. Therefore in this case, the file is opened for reading and writing.

3. What are the advantages of using mmap()?

- Enables memory access which is highly advantages through pointers arithmetic and memcp
- Accessing bigger files is made easier.
- Reading from and writing to a memory-mapped file avoids the extraneous copy that occurs when using the read( ) or write( ) system calls, where the data must be copied to and from a user-space buffer.
- The data is shared among all processes when numerous processes map the same object into memory. Private writable mappings have their not-yet copy-on-write pages shared, whereas read-only and shared writable mappings are shared in their entirety.

4. What are the disadvantages of using mmap()?

- mmap is costly to use
- The writing of mmap code is often complex.
- Inside the kernel, there is overhead in constructing and maintaining memory mappings and associated data structures.
- The memory mappings must fit within the address space of the process. A high number of different-sized mappings in a 32-bit address space might fragment the address space, making it difficult to discover large free contiguous sections.



- The size of memory mappings is always an integer number of pages. As a result, slack space is "wasted" as the difference between the size of the underlying file and an integer number of pages.

## 6. DISCUSSIONS & CONCLUSION

(What have you learned from this experiment?)

From this laboratory session, we had learn how to use memory mapping, map the file directly into virtual memory using mmap where we can access the file directly through memory, identically to any other chunk of memory resident data. Even it possible to allow writes to the memory region to transparently map back to the file on disk. We observe from output on main text file before and after the command cat mytextfile.txt change the text **this** to **that**. So, we could change anything by using the mmap in memory region as the mmap functioned to map files or devices into memory.

As a conclusion, from this laboratory session we have learnt about how to differentiate the functionality among various kinds of OS components. Then, we also able to manipulate OS theories to solve basic functional problems and perform lab and present technical report in writing. Thus, the objectives of this laboratory session is achieved.