**Operating System**

**Lab Report 10**

**Hafiz Ahmad**

**19l-1316**

**Section-6A2**

Memory Mapped Files

**INTRODUCTION:**

Memory mapping a file is accomplished by mapping a disk block to a page (or pages) in memory. Subsequent reads and writes to the file are handled as routine memory accesses. Manipulating files through memory rather than incurring the overhead of using the read() and write() system calls simplifies and speeds up file access and usage. It shows memory mapping for two processes A and B. A memory-mapped file contains the contents of a file in virtual memory. This mapping between a file and memory space enables an application, including multiple processes, to modify the file by reading and writing directly to the memory. Creating a Memory Map: void \*mmap(void \*addr, size\_t len, int prot, int flags, int fields, off\_t off); ♣ addr: This is the address we want the file mapped into. The best way to use this is to set it to (caddr\_t)0 or NULL and let the OS choose it for you. If you tell it to use an address the OS doesn't like (for instance, if it's not a multiple of the virtual memory page size), it'll give you an error. ♣ len: This parameter is the length of the data we want to map into memory. This can be any length you want. (Aside: if len not a multiple of the virtual memory page size, you will get a block size that is rounded up to that size. The extra bytes will be 0, and any changes you make to them will not modify the file.) ♣ prot: The "protection" argument allows you to specify what kind of access this process has to the memory mapped region. PROT\_READ, PROT\_WRITE, and PROT\_EXEC, for read, write, and execute permissions, respectively. The value specified here must be equivalent to the mode specified in the open() system call that is used to get the file descriptor. ♣ flags: You'll want to set it to MAP\_SHARED if you're planning to share your changes to the file with other processes, or MAP\_PRIVATE otherwise. If you set it to the latter, your process will get a copy of the mapped region, so any changes you make to it will not be reflected in the original file—thus, other processes will not be able to see them. ♣ fields: This is where you put that file descriptor you opened earlier. ♣ off: This is the offset in the file that you want to start mapping from. A restriction: this must be a multiple of the virtual memory page size. This page size can be obtained with a call to getpagesize(). Return value: • mmap() returns a pointer to the mapped area. On error, the value MAP\_FAILED is returned. Munmap: • int munmap(void \*addr, size\_t len); • On success, munmap() returns 0. On failure, it returns -1. Header File to be included: #include

Diagram

Description automatically generated

**OBJECTIVES:**

• Learn and Understand the concept of Memory Mapped Files.

**Application:**

A memory-mapped file contains the contents of a file in virtual memory. This mapping between a file and memory space enables an application, including multiple processes, to modify the file by reading and writing directly to the memory.

**Issues:**

No issue found regarding this lab.

**Conclusion:**

In this lab we learn the concept of memory mapped files and how to read and write including multiple processes.