Assignment 4

Question

Objective of this programming assignment is to use mmap() call and observe page-fault using the ‘sar’ command.

A big file (about 8GB) should be created using the `fallocate` command. This big file should be written with a single integer value (say X) at a specific offset (say F). Both the integer value and the offset should be generated using a random function. Please do remember this random function should generate a quantity anywhere between 0 and 8G.

The above big file should also be mapped in the virtual address space using mmap() call. Once it is mapped, the data should be read from the same specific offset (F). Now, if the data read is X`; then verify that X and X` are the same. In case of verification failure, an error message is to be printed. Note that, the offset value F can be anywhere between 0 and 8G.

This sequence of writing and reading data to/from a specific offset and also verification should be put in a while loop to go forever.

In another terminal execute the command `sar -B 1 1000` to observe for the page fault. This command should be started before the above program is put under execution. So, one can observe that the page faults are increasing, once the above program starts executing.

OUTPUT - The output of the program and the `sar` command should be pasted as a comment at the beginning of the program file as indicated by the guidelines.

Solution

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

#define FILE_SIZE (8LL * 1024 * 1024 * 1024) // 8GB

void generate_random_offset_and_value(off_t *offset, int *value) {
    // generates two 32-bit random numbers and combine them to form a 64-bit number
    *offset = (((off_t)rand() << 32) | rand()) % FILE_SIZE;

    // makes sure the offset is a multiple of sizeof(int)
    *offset -= *offset % sizeof(int);
    // generates a random value between 0 and 999999
    *value = rand() % 1000000;
}

int main() {
    srand(time(NULL)); // seeds the random number generator with the current time

    // allocates the file size on the disk
    if ((system("fallocate -l 8G big_file.txt")) == -1) {
        perror("Could not allocate the file size");
        exit(1);
    }

    // O_RDWR to open the file for both reading and writing
    // O_CREAT to create the file if it doesn't exist
    // O_TRUNC to make sure that if the file exists and is writable, it will be
cleared when opened.
    int fd = open("big_file.txt", O_RDWR | O_CREAT | O_TRUNC, (mode_t)0600);
    if (fd == -1) { // checks whether the file is created
        perror("File could not be created or opened or properly truncated");
        exit(1);
    }

    // creates a memory mapping of the file
    int *map = (int *)mmap(NULL, FILE_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd,
0);
    if (map == MAP_FAILED) {
        perror("mmap");
        exit(1);
    }

    while (1) {
        off_t offset;
        int value, read_value;

        // generates a random offset and value
        generate_random_offset_and_value(&offset, &value);

        // writes the generated value to the file at the specific offset using pwrite
        ssize_t bytes_written = pwrite(fd, &value, sizeof(value), offset);
        if (bytes_written != sizeof(value)) {
            perror("pwrite");
            exit(1);
        }
        else{
            printf("Data written: %d at Offset: 0x%" PRIx64 "\n", value,
(uint64_t)offset);
        }

        // reads the value from the mapped memory at the same offset
        read_value = map[offset / sizeof(int)];
        printf("Data read: %d at Offset: 0x%" PRIx64 "\n", read_value,
(uint64_t)offset);

        // verifies whether the written value and the read value are the same
        if (read_value != value) {
            fprintf(stderr, "Verification failed: Expected %d, Read %d\n", value,
read_value);
        }
        else {
            printf("Verification successful.\n\n");
        }

        usleep(100000); // sleep for 100 millisecond before the next iteration
    }

    // unmaps the memory
    if (munmap(map, FILE_SIZE) == -1) {
        perror("Could not free the memory space");
        exit(1);
    }

    // closes the file
    if (close(fd) == -1) {
        perror("Could not close the file");
        exit(1);
    }

    return 0;
}
```

Explaination

This C program performs various file operations, including file creation, memory mapping, and data verification. The program is designed to create a large file, write random data to specific offsets within the file, and simultaneously read and verify the written data using memory mapping.

Here's a step-by-step explanation of the program:

1. **Header Files**:
    - The program includes several standard C libraries, such as `stdio.h`, `stdlib.h`, `fcntl.h`, `unistd.h`, `sys/mman.h`, `sys/stat.h`, `sys/types.h`, `string.h`, `time.h`, and `inttypes.h`, to provide access to various functions and data types.
2. **File Size Constant**:
    - The constant `FILE_SIZE` is defined to represent the desired size of the output file in bytes. In this case, it's set to 8 gigabytes (8GB).
3. **`generate_random_offset_and_value` Function**:
    - This function generates a random offset and a random value for writing to the file.

    - It combines two 32-bit random numbers to form a 64-bit offset within the file.
    - It ensures that the offset is a multiple of the size of an integer to align data properly.
    - It generates a random integer value between 0 and 999999 (6 digits).

4. **`main` Function**:

    - It seeds the random number generator with the current time to ensure different random values on each run.

5. **File Allocation**:

    - The program uses the `system` function to allocate space for the output file using the `fallocate` command. This command ensures that the file has the specified size.

6. **File Open and Truncate**:

    - The program opens the file (`big_file.txt`) for both reading and writing using the `open` function.
    - It uses `O_CREAT` to create the file if it doesn't exist and `O_TRUNC` to clear the file if it already exists.
    - If the file cannot be opened or created, it prints an error message and exits.

7. **Memory Mapping**:

    - The program creates a memory mapping of the file using the `mmap` function.
    - It specifies that the memory should be both readable and writable (`PROT_READ | PROT_WRITE`) and shared among multiple processes (`MAP_SHARED`).
    - If the mapping fails, it prints an error message and exits.

8. **Infinite Loop**:

    - The program enters an infinite loop to perform the following steps repeatedly:
        - Generate a random offset and value using `generate_random_offset_and_value`.
        - Write the generated value to the file at the specified offset using `pwrite`. It prints the offset and value.
        - Read the value from the mapped memory at the same offset.
        - Verify whether the written value and the read value match.
        - Sleep for 100 milliseconds before the next iteration.

9. **Memory Unmapping and File Closure**:

    - After the loop, the program unmaps the memory using `munmap` and closes the file using `close`.

10. **Verification**:

    - During each iteration, the program verifies that the value written to the file matches the value read from the memory mapping. If they don't match, it prints an error message.

This program demonstrates concurrent read and write operations on a large file while verifying the correctness of the data using memory mapping. It also showcases proper synchronization and error handling throughout the process.

Elaboration on the functions

3. `generate_random_offset_and_value` Function:

- The `generate_random_offset_and_value` function is responsible for generating two random values:
    - `offset`: This value represents the offset within the file where data will be written or read. It needs to be a valid offset within the file's size.
    - `value`: This value represents the random data that will be written to the file.
- The function combines two 32-bit random numbers to form a 64-bit offset. This is done by shifting the first 32-bit random number left by 32 bits and then combining it with the second 32-bit random number using a bitwise OR operation.
- To ensure that the offset aligns with the size of an integer (4 bytes), the function subtracts any remainder obtained when dividing the offset by the size of an integer. This alignment is important for writing and reading data in a way that avoids partial writes or reads.
- The `value` is generated as a random integer between 0 and 999999 (6 digits) using the modulo operator.

6. File Open and Truncate:

- After allocating space for the file using the `fallocate` command, the program attempts to open the file for both reading and writing using the `open` function.
- The `open` function is used with several flags:
    - `O_RDWR`: This flag indicates that the file should be opened for both reading and writing.
    - `O_CREAT`: This flag indicates that the file should be created if it doesn't already exist.
    - `O_TRUNC`: This flag indicates that if the file exists and is writable, it should be truncated (cleared) when opened.
- If the `open` function succeeds, it returns a file descriptor (`fd`) associated with the opened file. If it fails, it prints an error message and exits the program.

7. Memory Mapping:

- Memory mapping is a technique that allows a file to be mapped into the virtual memory space of a process. This allows the file's data to be accessed as if it were an array in memory.
- In this program, memory mapping is achieved using the `mmap` function:
    - `map` is a pointer that will point to the mapped memory.

    - `NULL` specifies that the kernel should choose a suitable address for mapping the file.
    - `FILE_SIZE` specifies the size of the mapping, which is the same as the size of the file.
    - `PROT_READ | PROT_WRITE` specifies that the memory should be both readable and writable.
    - `MAP_SHARED` specifies that the memory should be shared among multiple processes.
- If the `mmap` function succeeds, it returns a pointer to the mapped memory (in this case, an array of integers). If it fails, it prints an error message and exits the program.

- Memory mapping allows efficient access to the file's data, as any changes made to the memory are automatically reflected in the file, and vice versa. It simplifies the process of reading and writing data from/to the file.

These points are crucial in setting up the file and memory mapping for subsequent data read and write operations in the program. The memory mapping technique facilitates efficient data access and synchronization between file and memory operations.

Virtual Memory

Memory Mapping (mmap) is a technique used in many operating systems to create a mapping between a file or device and a region of virtual memory. This technique allows you to access files as if they were in-memory arrays, simplifying data I/O operations and providing several advantages. Let's explore how mmap works in the context of virtual memory:

1. **Virtual Memory Overview**:

    - Virtual memory is a memory management technique used by modern operating systems to provide an illusion of a larger and more contiguous memory space than is physically available.
    - Each process running on the system has its own virtual address space, which is divided into pages.
    - Virtual addresses are translated into physical addresses by the memory management unit (MMU) of the CPU.

2. **Memory Mapping (mmap)**:

    - When you use `mmap` in a program, you request the operating system to create a mapping between a file or device and a portion of the virtual address space of your process.
    - This mapping essentially associates a region of virtual memory with the contents of the file or device.

3. **Creating the Mapping**:

    - When you call `mmap`, you specify several parameters, including the starting address (or let the OS choose one), the size of the mapping, protection settings (read, write, execute permissions), and the file descriptor of the file or device you want to map.
    - The operating system searches for available virtual memory pages and maps them to the file or device.

    - These pages may not be contiguous in physical memory but appear as a contiguous block in virtual memory.

4. **Accessing Mapped Data**:

    - Once a mapping is established, you can access the data in the file or device by simply reading or writing to the corresponding memory addresses in your program.
    - This access is just like accessing any other variable in your program.
    - Under the hood, the OS handles translating virtual addresses to physical addresses when data is read or written.

5. **Advantages of mmap**:

    - **Efficiency**: Memory mapping can be more efficient than traditional file I/O because it reduces the number of data copies between user space and kernel space.
    - **Simplified I/O**: It simplifies file I/O operations by allowing you to treat files as memory buffers. You don't need explicit read and write functions.
    - **Shared Memory**: Memory mapping allows multiple processes to map the same file or shared memory region, facilitating inter-process communication (IPC).

6. **Write Operations**:

    - When you write to a mapped memory region, the changes are initially made in the process's virtual memory.
    - The OS tracks these changes and may update the corresponding pages in the file lazily (on-demand).
    - This means that writes may not immediately result in changes to the underlying file, and flushing or syncing may be required.

7. **Caveats and Considerations**:

    - Care must be taken with memory-mapped files, especially when writing data, to ensure data consistency and synchronization between processes if multiple processes are sharing the same mapping.
    - Mapping a large file entirely into memory can consume a significant amount of virtual memory, which should be managed carefully.

Swap Space

Virtual memory is typically implemented using a combination of RAM (Random Access Memory) and disk storage. Here's how it works:

1. **Physical RAM**:

    - Your computer has physical RAM, which is the actual memory hardware available for storing data and instructions that your CPU can access quickly.
    - RAM is volatile memory, meaning it loses its contents when the computer is powered off.

2. **Virtual Memory**:

    - Virtual memory is an abstraction provided by the operating system (OS) that extends the usable memory beyond the physical RAM.

   - It creates the illusion that your computer has more memory than it actually does.
   - Each process running on the system has its own virtual address space, which is divided into pages.

3. **Page File (Swap Space)**:

   - To implement virtual memory, the OS uses a portion of the computer's disk storage (usually referred to as the "page file" on Windows or "swap space" on Unix-like systems).
   - When physical RAM becomes scarce or when a process needs more memory than is available in RAM, the OS can swap data between RAM and the page file.
   - Pages of memory that are not currently needed can be temporarily moved to the page file, freeing up physical RAM for more important data.

4. **Memory Mapping**:

   - Virtual memory is managed through a combination of hardware (Memory Management Unit or MMU) and software (the OS).
   - When a program accesses a virtual memory address, the MMU translates it into a physical memory address. If the data is in RAM, it's retrieved quickly. If not, the OS fetches it from the page file.

5. **Benefits of Virtual Memory**:

   - Provides the illusion of a vast and contiguous memory space to each process, making efficient use of limited physical RAM.
   - Allows the OS to effectively manage memory, prioritizing processes and ensuring they don't exceed available resources.
   - Enables features like memory protection, which prevents one process from accessing the memory of another.

6. **Performance Considerations**:

   - Accessing data in RAM is significantly faster than accessing data from the page file on disk. Therefore, a well-managed virtual memory system aims to keep frequently used data in RAM and only use the page file when necessary.

In summary, virtual memory is a memory management technique that combines physical RAM with disk storage to provide the illusion of a larger and contiguous memory space to processes running on a computer. It allows efficient utilization of physical RAM and helps prevent processes from running out of memory, but it may involve slower disk access when data is swapped between RAM and the page file.

Writing in Memory Mapped File

Memory mapping, especially when writing to a mapped file, involves a mapping between a file on disk and a region of virtual memory in a process's address space. This mapping allows you to manipulate the contents of the file as if it were an in-memory array. When writing to a mapped file, there are several important aspects to consider:

1. **Mapping Setup**:

   - When you create a memory mapping using `mmap`, you specify a file descriptor and a range of virtual memory addresses that are associated

with the file's data. This range is where you will read from or write to the file.
   - The mapping can be established for both reading and writing (`PROT_READ | PROT_WRITE`) or for just reading (`PROT_READ`).

2. **Virtual Memory Pages**:

   - The virtual memory region that corresponds to the mapped file is divided into pages, which are typically 4KB in size. These pages serve as the unit of interaction with the file.

3. **Writing Data**:

   - When you write data to a mapped memory region, you are effectively changing the content of the virtual memory pages associated with the file.
   - The changes you make are initially made in your process's virtual memory space and are not immediately reflected in the actual file on disk.

4. **COW (Copy-on-Write)**:

   - Most modern operating systems use a technique called "copy-on-write" to manage memory mappings efficiently.
   - When you write to a mapped file, the operating system does not immediately modify the file on disk. Instead, it marks the affected pages as "dirty" or "modified" within your process's memory.
   - These pages are no longer shared with other processes that may have mapped the same file. Each process maintains its own copy of the modified pages.

5. **Flushing Changes**:

   - To persist the changes made to a mapped file, you typically need to explicitly flush the changes to disk. This is done using functions like `msync`.
   - The operating system may also perform automatic flushing under certain conditions, such as when memory is tight or during regular intervals.

6. **File Updates**:

   - Once you flush the changes to the mapped file, the file on disk is updated with the modified data.
   - This update can be done lazily, meaning the OS may delay the write operation to optimize performance.

7. **Data Consistency**:

   - When multiple processes share the same mapped file, care must be taken to ensure data consistency.
   - Inter-process communication mechanisms, such as semaphores or locks, may be needed to coordinate access to the shared mapping and maintain data integrity.

8. **Performance Benefits**:

   - Memory mapping provides performance benefits because it reduces the need for explicit read and write operations between user and kernel spaces.

    - It leverages the OS's virtual memory management to optimize data transfer between the process and the file.

9. **Resource Management**:

    - Proper resource management is crucial when using memory mapping. You should unmap the memory (`munmap`) and close the file descriptor when they are no longer needed to prevent resource leaks.