Assignment Report: CSC3150 Assignment3

Name: Wang Jiaju, Student ID: 121090544

Your report should follow the template with the following section structure.

No page limitation

1 Introduction [2']

The assignment is centered around enhancing the xv6 operating system, a simple Unix-like teaching OS, with the implementation of 'mmap' and 'munmap' system calls. These calls are pivotal for memory sharing among processes and mapping files into process address spaces, facilitating the use of memory-mapped files. This technique allows files to be treated as part of a program's memory, thus improving efficiency and ease of file manipulation. My work involved coding these system calls to achieve memory mapping and unmapping functionalities, adhering to the specifications and utilizing the provided virtual machine setup for development and testing.

2 Design [5']

The design of the program commenced with understanding the requirements and limitations of xv6 concerning memory mapping. The primary challenge was to allocate virtual memory space dynamically for file mapping, respecting the constraints of the xv6's memory management system. The design included:

mmap System Call Implementation: Crafting a mechanism to map files into the process's virtual address space. This involved identifying suitable regions within the virtual address space where files could be mapped without conflicting with existing mappings.

munmap System Call Implementation: Developing a corresponding method to unmap previously mapped files, ensuring the release of allocated resources and proper cleanup of the process's address space.

Error Handling and Edge Cases: Incorporating robust error handling to manage scenarios such as invalid file descriptors, non-page-aligned addresses, and requests exceeding the process's address space limits.

Integration with Process Lifecycle: Ensuring the mapped regions were correctly handled during process creation, execution, and termination, including the duplication of mappings on process fork.

Next, I will analyze the files we have rewritten

proc.c:

Explanation of code snippet line 325:

This part of the code is part of the process creation (fork), and is specifically used in the process of copying the parent process to the child process. Its main purpose is to copy the virtual memory area (VMA) of the parent process to the child process. Each VMA contains information about the virtual memory area, such as its starting address, length, and the file associated with it (if any). The code logic is as follows:

Iterate over each VMA of the parent process. Copy the VMA information of the parent process to the corresponding VMA of the child process.

If this VMA is associated with a file (meaning this part of the virtual memory is file mapped), then the reference count of the file needs to be increased to ensure that the file is still available in the child process and its life cycle is managed correctly.

Explanation of code snippet on line 372.

This part of the code is part of the process exit (exit) and is used to clean up the virtual memory resources occupied by the process that is about to exit. The logic is as follows:

Iterate over each VMA of the process.

For each VMA, use the uvmunmap function to unmap its mapped virtual memory region from the process's page table, and possibly free those pages. This is a necessary cleanup step to ensure there are no memory leaks.

Parameter 1 (the last parameter) indicates whether the physical memory page is released when unmapping. This is usually desired because process exit means it no longer needs any resources.

Generally speaking, these two pieces of code are a key part of the operating system kernel in process life cycle management, ensuring the correct management of virtual memory, including copying the necessary memory mapping when the process is created, and releasing all occupancy when the process exits. virtual memory resources.

proc.h:

Add fields about the ‘vma‘ structure in lines 86 and subsequent lines of code in order to describe and manage the characteristics of each virtual memory area in detail. The following is the role and meaning of each field:

uint64 addr: Indicates the starting address of the virtual memory area. This is the location of the region in the process's virtual address space.

int len: The length of the area, usually in bytes. This specifies the memory range starting from 'addr'.

int prot: Specifies the protection permissions of the memory area. These permissions, which can be any combination of read (R), write (W), and execute (X), define what operations a process can perform on that memory area.

int flags: Flags of the memory area, used to further specify the attributes of the area, such as whether it is shared, whether it can be mapped by files, etc.

int fd: If the virtual memory area is mapped from a file, the 'fd' field stores the file descriptor. This allows a process to map the contents of a file directly into its virtual address space, enabling efficient file access.

int offset: If the memory area is a file mapping, ‘offset‘ represents the offset within the file where the mapping starts. This is the byte offset from the start of the file to the start of the map.

struct file *f: Pointer to the file structure associated with this virtual memory area. If the area maps file contents, this pointer is very important because it allows the system to manage the details of the file mapping, including access to the file contents.

By including these fields in the VMA, the operating system is able to effectively manage the virtual memory of the process, including but not limited to:

File Mapping: Allows a process to directly map part or all of a file on disk into its virtual address space, supporting efficient file access and sharing.

Memory Protection: Ensure that a process can only access its virtual memory area in a specified way (such as read-only, executable, etc.) to improve system security.

Virtual Memory Management: Record the memory usage of each process in detail to facilitate operations such as memory allocation, recycling, and swap (swap).

sysfile.c:

These two system call functions 'sys_mmap' and 'sys_munmap' are used to provide memory mapping (memory mapping) and unmapping (unmapping) functions within the operating system. They allow a process to map or unmap the contents of a file or device to an area in its virtual address space. This is a very important feature in modern operating systems for efficient file access and memory management.

he role of 'sys_mmap':

'sys_mmap' maps files to the virtual address space of the process, allowing the process to access the file content as if it were accessing memory, thereby improving the efficiency of file operations.

It supports multiple mapping types, including shared mapping ('MAP_SHARED') and private mapping ('MAP_PRIVATE'), allowing memory to be shared between different processes or creating copy-on-write memory areas.

This function also handles different protection permissions such as read ('PROT_READ'), write ('PROT_WRITE') and execute ('PROT_EXEC'), controlling the process's access to the mapped area.

When mapping a file, you can specify an offset within the file, allowing the process to map a specific portion of the file rather than the entire file.

The role of 'sys_munmap':

'sys_munmap' is used to cancel the mapping relationship previously established through 'sys_mmap' and release the virtual address space occupied by the mapping area.

If the mapping area is a shared mapping and a write operation is performed, 'sys_munmap' will write the changes back to the underlying file to ensure data consistency.

Demapping can reclaim memory areas that are no longer needed and effectively manage the virtual address space of the process.

trap.c:

Page faults usually occur when a process attempts to access a virtual memory address that is not currently mapped to physical memory, such as accessing a lazy-loaded memory page or an address space that exceeds the current memory map.

In this code, page faults in two situations are specially handled: 'r_scause() == 13' or 'r_scause() == 15', which represent read (load) or execution (instruction fetch) operations respectively. page fault. The steps to handle page faults are as follows:

Get the virtual address where the page fault occurred: Obtain it through 'r_stval()' and save it in the variable 'va'.

Check address validity: Make sure that the error address is not caused by accessing memory outside the process's address space. If the address exceeds the process's address space or is below the process's stack pointer, it will jump to the error handling logic.

Find the corresponding virtual memory area (VMA): Traverse the VMA list of the process and find the VMA containing the error address 'va'. If found, this means that the address is valid, but is not currently mapped into physical memory.

Allocate physical memory and map: Use 'kalloc()' to allocate a physical page, clear it, and then map the newly allocated physical page according to the attributes of the VMA (such as protection permissions) and file mapping information to the virtual address where the page fault occurred. Here the 'mapfile()' function may be an assumed function used to load content from a file into allocated physical memory.

Update page table: Use the 'mappages()' function to map the virtual address 'va' to the newly allocated physical memory, and set the correct page table entry flags (such as read, write, and execute permissions).

Error handling: If any step fails (such as memory allocation failure or page table update failure), the allocated resources will be released and the process will be marked through the 'setkilled()' function so that it can be terminated.

For EXART CREDITS(Fork Handle): To implement the correct handling of 'mmap()' behavior when calling 'fork()', you mainly need to consider the following steps:

Copy the virtual memory area (VMA):

Traverse the VMA list of the parent process: For each VMA in the parent process, create a new VMA instance and copy it to the VMA list of the child process. This includes the replication map's starting address, length, permissions, mapping type ('MAP_PRIVATE' or 'MAP_SHARED'), and the associated file descriptor.

Increment the file reference count: For each copied VMA, if it is associated with a file (via a file descriptor), 'filedup()' needs to be called to increment the file object's reference count. This ensures that the file is valid in both the parent and child processes, and that the file is not accidentally closed when either process ends.

Handle page errors:

Find the corresponding VMA: When a page fault occurs, the error handling logic needs to find the corresponding VMA based on the error address.

Load the file content into memory: For 'MAP_PRIVATE' mapping, if it is the first time to access the page, the corresponding content needs to be read from the file into the newly allocated physical page. For 'MAP_SHARED' mappings, this step may be slightly different, as modifications may need to be reflected back to the file.

Mapping physical pages to virtual addresses: Once a physical page is allocated and the file contents are loaded, the system needs to map this physical page to the virtual address that triggers the page fault.

Implement 'munmap()' support for 'fork()':

Decrement the file reference count: If the unmapped region is associated with a file, you need to decrement the file's reference count. If the reference count reaches 0, the file is closed.

Update VMA list: Remove the corresponding entry from the child process's VMA list and release any associated resources.

3 Environment and Execution [2']

The program's development and testing environment was a provided virtual machine configured with the necessary tools and setup for xv6. The execution of the program involved several steps:

Compilation: Utilizing the 'make' command to compile the modified xv6 source code, integrating the new system calls.

Running xv6: Starting the xv6 operating system using the 'make qemu' command to boot into the modified OS environment.

Testing with mmaptest: Executing a provided test program, 'mmaptest', to validate the correct functioning of the 'mmap' and 'munmap' functionalities. The test cases covered various scenarios, including mapping files with different permissions, testing shared and private mappings, and handling unmapping operations.

4 Conclusion [2']

This assignment provided a comprehensive experience in operating system development, focusing on memory management and file handling mechanisms. The implementation of 'mmap' and 'munmap' system calls required a deep understanding of xv6's memory management, process lifecycle, and file system interaction. Through this assignment, I gained valuable insights into how operating systems manage memory and files, the challenges of designing system calls that interact closely with the OS kernel, and the practical aspects of testing and debugging within an OS development environment. This knowledge is not only fundamental to operating system concepts but also applicable to various areas of computer science and software engineering.