Memory Mapped Files on MINIX

Claudiu-Dan Gheorghe and Andrei Faur
Politehnica University of Bucharest
{claudiu.gheorghe, andrei.faur}@cti.pub.ro

Abstract

Most modern operating systems provide support for memory mapped files, which allow users to treat files using pointer arithmetic, thus avoiding the usage of expensive system calls. MINIX 3 is a UNIX-like, recently developed operating system that recently got virtual memory support, opening the opportunity for memory mapped files. This article describes the design and implementation of memory mapped files on MINIX, and highlights the specific details of MINIX's microkernel particularities.

keywords: MINIX, mmap, memory mapped files

1 Introduction

Minix 3 is a recently developed operating system designed to be fault-tolerant, reliable and secure, with minimal intervention from the user. While prior versions of Minix have existed since 1987, Minix 3 is viewed as a complete overhaul, meant to be a full-fledged operating system and not just an educational tool. Development of Minix has seen renewed interest with the arrival of Minix 3, since it still lacks many of the features that modern operating systems have, such as virtual memory support which has been implemented only two months prior to the writing of this article. With the arrival of virtual memory, many new features have now become possible to implement, among them being memory mapped files.

Memory mapped files provide a mechanism by which a file can be accessed just like regular memory by mapping the file into a process's address space. Afterwards, the contents of the file can be accessed using pointer arithmetic, rather than read/write system calls.Thus they create transparency between files and the adress space of a process, letting applications treat files present on disk as a primary memory area. Memory mapped files are not a new concept and they are mentioned in the POSIX standard,being implemented in almost every modern operating system because they improve performance, ease of use and they create further concurrency opportunities. This article presents the implementation of this mechanism in MINIX 3.

2 MINIX operating system

Minix is an open-source, microkernel-based operating system first created by Andrew Tanenbaum in 1987. The latest version, Minix 3, aims to be an operating system that is "reliable, self-healing, multiserver UNIX clone". Having this in mind, the code running in the kernel has to be minimal, while the processes, memory and file-related tasks are delegated as device drivers running as separate user-space processes. Reliability and self-healing come from the ideea of implementing a part of the system called the reincarnation server. This server monitors other drivers and when they fail or malfunction in some way, they are shut down and replaced by a new copy. More than that, the reduced kernel size and restricted access to kernel functions and I/O ports improve the reliability of the operating system.

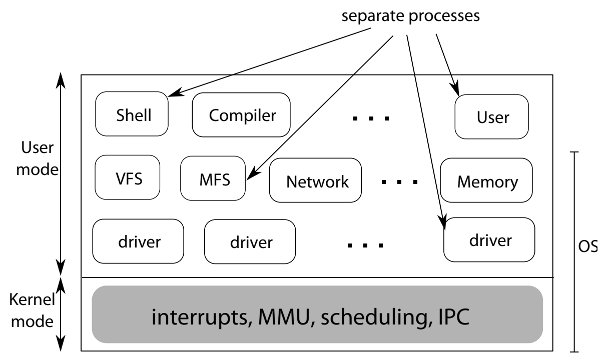

Figure 1: MINIX architecture

The architecture of Minix 3 is divided into four seperate layers, as it can be seen in Fig. 1. The bottom layer is the microkernel, which handles interrupts, scheduling and message passing. It is important to specify that all communications between processes are made with messages, which are handled by the kernel. Message handling requires that the kernel verify that the source and destination are valid and also find the exact location in physical memory of the send and receive buffers. In addition, the bottom layer also contains a clock task, which only interfaces with the kernel, and interacts with hardware that generates timing signals, plus a system task which contains the implementation of a set of privileged kernel calls to be used by drivers and server from the upper layers.

The upper three layers can be considered as being part of only one layer, because of the uniform way the kernel treats them. However, their functionality, and, more important, their access rights, make it possible to further divide the layer into the three we have in the picture above. For example, a server process never has to access the disk directly, so it does not have the privilege to make such an attempt, as oppsed to the actual disk driver which handles all such requests.

The second layer contains device drivers, each running as a separate process that controls different I/O devices. Note that these drivers do not have direct access to the I/O port space and cannot issue I/O instructions directly. The way they interact with the I/O is through the system task, by making special requests to read data from or write data to I/O ports on their behalf. The kernel's role is to check the authorization of the driver making the request.

The next layer contains server processes, and this is where all the operating system services reside. The most important of these are the process manager and the file system (FS). The process manager handles all system calls that involve starting or stopping process execution as well as signals that can potentially alter process states. In order to open, close, read and write files, processes from the upper layer have to send messages to the FS server. The FS server sends messages to the disk driver, which actually controls the disk. Another recently implemented server resides at this layer too, the virtual memory (VM) server. This and the FS server have been augmented with our implementation, in order to provide memory mapped file support.

The final layer is where all user processes reside, such as a shell, window manager, different utils. It is for the correct functioning of this layer that the bottom three layers have to work together.

3 Memory mapped files

Memory mapped files are a very useful technique in modern operating system, for two reasons:

- They improve performance by reducing accesses to incredible slow hard drives and especially to moving file cursor around the content of the file.
- They make possible for multiple processes to share the in-memory content of the same file, thus reducing the overall physical memory usage.

The best example is the shared library. Any executable needs at least one shared library as GLIBC, so when it will have to load it from the disk in a physical memory region. Imagine that tens of processes will need to do exactly the same thing, wasting lot of physical memory with duplicate data.

As it makes usage of both files and memory, implementing memory mapped files implies dealing with the MINIX's system processes concerned with this tasks. The latest MINIX version is 3.1.5, released on November 2009. Regarding memory management it provides Virtual Memory (available from 3.1.4) and regarding file systems it provides also Virtual File system (available from version 3.1.3). We will take a quick tour over these system processes (Virtual Memory server and Virtual File system server) to introduce the implementation details, especially because Virtual Memory is not documented at all.

3.1 The Virtual Memory (VM) server

VM is the system process that manages each processs virtual address space. So whenever we need to allocate memory, to create a mapping or any operation related with memory management we have to send an appropriate message to the VM server.

The distributed manner of the MINIX operating system lead to the existence of separate special processes (system processes) for managing user processes resources on behalf the kernel. Comparing to the Linux kernel, where running a system call caused a trap that switched the system from user mode to kernel mode, here a system call is translated to an appropriate message for a specific system process (server), which all it does is to listen for request messages and serve them. Also all the data structures kept by the kernel for each process to handle them are not centralized, so each system process has a specific global table of data structures that contain all the relevant information for each user process, identified by a numerical id, called endpoint number. For example, in the VM server the process info table is called vmproc and the data structure is struct vmproc, or in the VFS server is called fproc and the structure is struct fproc.

As the VM is not documented at all, we will shortly take a tour of the data structures and concepts used inside it.

The address space for each process is described in the struct vmproc data structure and it is organized as a simple linked list of regions. A region represents a virtual memory range and its described in struct vir_region data structure. It has a start address (virtual address), a length and it is annotated with specific flags. The virtual memory region is the basic unit used for mappings, so the flags include information relevant to the memory mappings. Each virtual region contains a sequence of other memory representation structure called physical region, described in struct phys_region data structure. For improving lookup speed for such regions, MINIX uses AVL tree for storing them.

The key for a physical regions is the offset from the start of virtual memory region, but there is also additional information stored which helps the system to step from virtual addresses to physical addresses. So, each physical region is linked to a parent virtual region using a many-to-one relationship, and its also linked to another data structure called physical block using a one-to-one relationship. The physical regions are also part of another data structure, which is a linked list of physical regions that reference the same physical block.

The physical block is described in the struct phys_region data structure. It contains a length which is the number of bytes of the contiguous memory area that it represents, a reference count which is the number of physical regions that reference this physical region, the first element of the linked list of physical regions discussed earlier, and the most important, a physical memory address. Having the intermediate representation of the memory as physical region and not directly as a physical block permits the system to easily share pages of memory between different physical regions, virtual regions and address spaces belonging to different processes.

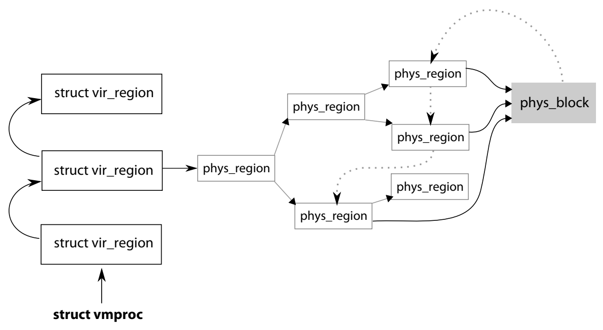

Figure 2: Virtual memory in MINIX

All the information from these data structure is kept only by VM server, but the data structure used for translating virtual addresses to physical addresses by the processors MMU is the page table. So any modification in the address space is made visible only after synchronizing the changes with the content of the page tables. MINIX has a hierarchical page table with one page directory of 1024 entries (10 bits) and is implemented only on the i386 architecture.

The free physical memory ranges are stored also in VM using a global AVL tree queried each time a number of memory pages are needed. All the memory allocations are deferred as much as possible in the same manner as they are in the Linux kernel, and they are managed by efficiently handling the memory pagefaults. From the VM server point of view, the pagefaults are also normal messages received from the kernel process.

The VM doesnt serve only user processes. As the Linux kernel needs a lot of data structures for keeping track of processes and other internal things, the system processes from MINIX (that include the VM itself) also do need them. The approach is merely the same as in Linux and a slab allocator is used.

The slab is a data structure used for handling memory allocations of small objects as a struct vir_region for example. What it is inefficient at memory allocations is not simply finding a hole in the free memory, but the initialization process. So this is the purpose of the slab: to maintain a pool of certain fixed size objects that are ready to be used without doing any initialization.

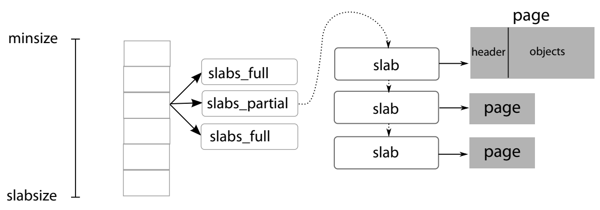

Figure 3: Slab allocator

The primary key for the slab allocator is the size of the object. It keeps a fixed set of slabs for a range between a minimum (now in MINIX we have e minimum size of 8 bytes) and a maximum size (SLABSIZE macro, defined as 60 bytes now) in a table that is indexed using as offset the size of object. Each slot from the slab table is described in struct slabheader and it contains pointers to the heads of three lists: LIST_FREE the list of free slabs, LIST_USED the list of partial free slabs, and LIST_FULL the list of slabs filled-up by objects. A slab itself spans over a single page of memory, and is composed from a header, described by struct sdh data structure, and the remaining part to the page size is used for storing data objects. The lists are implemented as a double linked list which makes for a slab to migrate from one list to another in constant time.

3.2 The Virtual File system (VFS) server

The VFS was introduced from the 3.1.2 release of MINIX and its presented in the Gerofis masters thesis [3], so we will just present a quick overview of the architecture. The VFS is an additional abstraction layer over the file system implementations. Due to the distributed and multi-server MINIX architecture, the VFS lives as a separate process and it interacts with the underlying file system implementations that are also separate processes for each mounted partition. The communication with the FS processes is performed synchronous and the VFS server waits until the response comes for each request. The MINIX VFS implementation is different compared with the monolithic implementations where the communication between different components of the operating system is done simply through functions calls, and especially because the MINIXs principles are reliability and security.

4 Design and implementation

The entry point for implementing memory mapped files by POSIX standard is the **mmap()** call, together with **munmap()** and **msync()**. As it is in Linux, the **mmap()** library call is translated in a dedicated system call.

In the current version of MINIX the **mmap()** and **munmap()** system calls exist and they are partial implemented in the VM server to support only anonymous mapping, used for memory allocation and for the address space mapping techniques offered by virtual memory. The implementation of memory mapped files involves a message passing session between the VM server and the VFS server and has also to deal with inconsistency problems that will further be explained.

4.1 Message flow

The system call for **mmap()** is routed to the VM server which creates a virtual region in the address space of the process on behalf the system call is made and annotates it with a special flag that makes this region observable as a file mapping - VM_MMF. Also additional information has to be stored in order to handle further requests, like the *inode* number. In the first place, the VM doesnt make further actions like sending request to VFS for loading data or mapping the physical blocks to pages and defers them similarly with the demand-paging mechanism. Anyway, the VFS must be notified that a process is mapping the certain file and at least the reference count from *struct filp* must be incremented.

The harder part will come when a page fault will be caught in the page fault handler. If the page fault is detected and it seems to be a file mapping, then a page is allocated and a new physical region is added inside the virtual region, mapping it to the physical block that contains the newly allocated page. Once we have this access-ready memory we send a message to the VFS server to load the needed content from the hard disk. The request is similar with a *read()* request. The message is sent synchronous and the VM waits until VFS finishes the request. The VFS obtains the *struct filp* structure associated with the open file by using the given file descriptor on behalf the process who made the file mapping. After that it issues a message to the process responsible for the mounted partition which contains the file for reading the content of the file in a given buffer. Suppose that the files content is not in memory, then it will be brought from disk to the file system implementation buffer cache. After that the content will be copied to the given memory address, and the responses are chained back till we are again in VM. Now we have files content needed for the user request.

By far anything looks fine, but the previous scenario works only with MAP_PRIVATE mappings. If we have shared mappings, we need to take additional actions. For that we need some way to keep a list of the user processes that mapped our file (identified by the *struct filp* structure and not by the file descriptor which is a per-process resource). The list will be stored in VM, but the link to it (a unique mapping id) will be stored in the *struct filp* structure. So when having a shared mapping, in the first step of creating the mapping, the VM must contact the VFS to obtain the mapping id from the *struct filp* structure and add the user process on behalf the mapping is made to the mapping lists. When a new page is allocated it must be mapped on the same offset in all the other processes that are mapping the same file. This way different address spaces regions will point to the same physical page.

At **munmap()**, the address space region is searched and it must be erased, after all the modified content of the file (pages with write access) has been written to disk. For MAP_SHARED mappings, the VM must firstly check the shared mappers list and delete the process on behalf the **munmap()** call is performed. Only if the list is empty after this operation the content should be sent to the disk. Also the reference count from struct filp must be decreased.

Also when creating a new mapping we use copy-on-write and we duplicate the content of virtual region from one of the shared mapping processes. Anyway, copy-on-write happens only for private mappings, because at shared mapping we dont need separate mem-

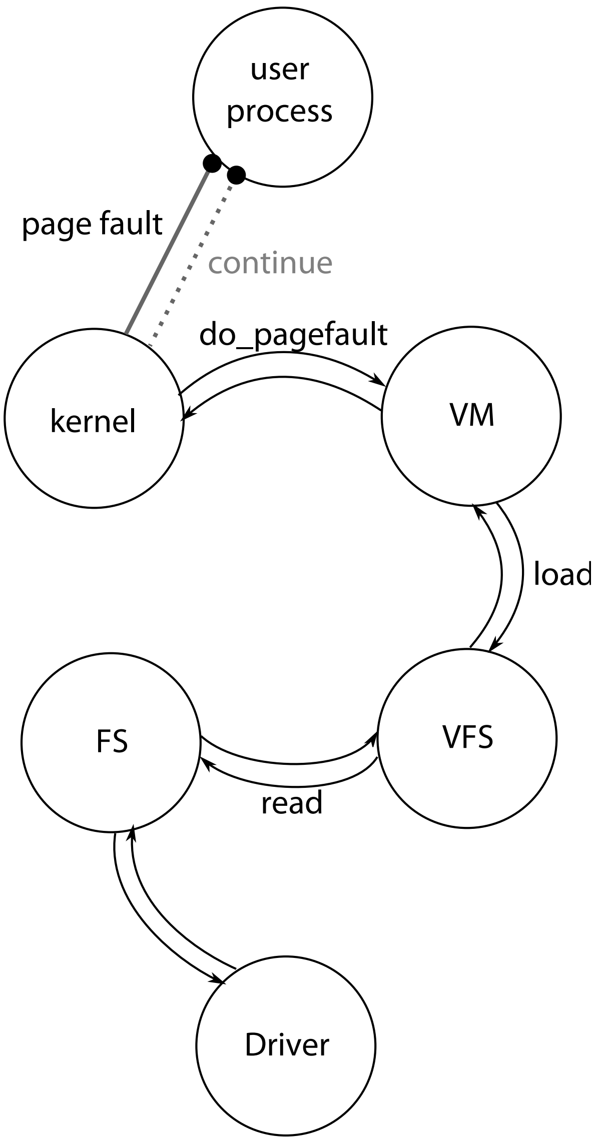

Figure 4: Messages sent on pagefault

ory pages.

4.2 Inconsistency concerns

Mapping files into memory brings inconsistency issues between the memory operations made on the mapped region and *read/write* system calls made on the same open file. In Linux these problems are gracefully solved using the Page Cache [2], but in MINIX we don't have it.

When *read/write* system calls arrive in the open file structure, if it appears to be mapped, we must see if the $[offset, offset + length]$ range where we should perform the request is included in the shared mapping region, and more than that, if there is a physical page allocated for this range, which means at least a READ memory access was performed in this range and the page was fetched from disk. This check must be performed in VM, and if it is true, we must perform the data copy directly using the found physical memory. If it is not, we must emit the request to block IO underlying layer, as normally. So basically is like using the mappings as a page cache, when mapping is present on the open file.

This should resolve conflicts between *read/write* calls and MAP_SHARED mappings. As it is stated on POSIX standard it is unspecified whether modifications to the underlying object done after the MAP_PRIVATE mapping is established are visible through the MAP_PRIVATE mapping, so eventually can be inconsistencies with this kind of mappings.

5 Related work

Memory mapping has become quite a common feature for most modern operating systems. The Single UNIX Specification details the exact behaviour and interface that the mmap system call should have. Linux and Solaris, among others, try to conform as much to this standard as possible, and our implementation tries that as well. While the exact details of the implementation differ between different OSs, it is important to provide a common interface so that userspace programs using it are portable.

For example, in Linux, memory mapped files are implemented using the Page Cache, where the kernel stores page-sized chunks of files. In a regular file read, the kernel must copy the contents of the page cache into a user buffer. When using file mapping, the kernel maps the program's virtual pages directly onto the page cache. This technique is not only specific to Linux, but to Windows and Solaris as well.

6 Conclusions and future work

In this paper we presented an implementation of memory mapped files support in the MINIX 3 operating system. This is a small step towards making MINIX 3 an operating system capable of providing all the features that recent operating systems provide. Our implementation only provides support for MAP_PRIVATE, and the next step is establishing and creating a solution ready to be integrated in the MINIX mainline repository.

References

[1] Andrew S Tanenbaum, Albert S Woodhull. *Operating Systems Design and Implementation.* Prentice Hall, 3rd edition, 2006. ISBN-10: 0131429388

[2] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel.* OReilly Publishing, 3rd edition, 2005. ISBN: 0-596-00565-2.

[3] Balazs Gerofi. *Design and implementation of the MINIX Virtual File system.* 2006