Keyronex Operating System Internals

Release 1.0

NetBSD User

Jul 19, 2023

CONTENTS

Note: Keyronex (and this book) are both very early in development.

CHAPTER

ONE

CONTENTS

1.1 Introduction

This book is intended to outline the design of the Keyronex operating system and discuss its implementation.

1.2 Virtual Memory Manager

1.2.1 Overview

The virtual memory (or VM) subsystem is the centrepiece of Keyronex. It provides for applications to use more memory than is physically available in the system, to treat resources (e.g. files) other than main memory as if they are memory, to implement the Posix *fork()* function efficiently, and to provide protection and sharing of memory between programs.

Note: "Paging" (and "virtual memory") are sometimes used online as simple synonyms of "virtual address translation", the process by which a virtual memory address is translated to a physical address, or to the management of page tables alone. "Paging" traditionally refers to movement of pages of data between backing store and main memory, while virtual memory refers to the abstraction described in the first paragraph of this section. These are the senses in which these terms are used here.

Keyronex' VM subsystem has several responsibilities. These include:

Resident page management:
: VM provides for physical page frames to be allocated as needed for various uses. It implements a page replacement policy to determine when to page-out a disk to backing store, such as the backing file for mapped files or the swapfile for anonymous memory. A cleaning policy is also implemented to write back changed pages pages of mapped files to disks regularly so that the risk of data loss is reduced. Virtual copy optimisation is also provided.

Address space management
: An abstract, machine-independent description of the virtual memory state of each process is kept, including the layout of its address space and the inheritance and memory protection associated with regions of that space.

Physical mapping
: The abstract representations maintained by VM must be translated to forms required by the platform's memory management unit. This involves creating and maintaining hardware-specific page tables to describe the mapping of resident pages, updating these in response to changes in state such as page-ins or out, and ensuring the TLBs are appropriately flushed, especially on multiprocessor systems.

The Keyronex virtual memory manager is principally derived from the design of NetBSD's UVM.

1.2.2 Data Structures

A number of data structures play a role in the system. This is an overview of the major players:

vm_page_t

Represents a physical page of usable memory. These are placed on queues according to their use and state. Queues include the free queue, wired queues, and others - queues (and `vm_page_t` in greater depth) are described in the section on paging. The set of all `vm_page_t`s is collectively called the Resident Page Table (or RPT).

Hint: A `vm_page_t` is analogous to a `struct page` in GNU/Linux or a PFN database entry in NT.

If it is being used as a page to store part of anonymous memory, it holds a pointer to the `vmp_anon` it belongs to; if it is being used to store data belonging to a regular VM object, e.g. part of the vnode of a regular file on a filesystem other than tmpfs, it stores a pointer to the VM object it belongs to and the offset of this page within that object. In either of these cases, it also contains a `pv_map` list head, which stores, for each mapping of the page, the `vm_map_t` into which it is mapped and its virtual address within that map.

Note: When talking about a "page owner", this refers to the *vmp_anon* or the *vm_object_t* to which it belongs, as described above. Page ownership is irrelevant if a page does not belong to either of these, e.g. in the case of kernel wired memory. Anything which says "the page owner must be locked" is ignored in that case.

---

vm_page_t`s also carry a status enumeration. This That bit is guarded by the page queues lock. Some other fields of a `vm_page_t are guarded by its owner.

vmp_anon

A virtual page of anonymous memory. It may link to a `vm_page_t` if it is resident in memory. If it has been swapped out, it instead stores an identifier by which its contents can be retrieved by the swap pager, called a *drum slot*. It also stores a reference count used in copy-on-write logic, and a mutex lock.

It is hoped that `vmp_anon`s will themselves become pageable in the future, which will help to dissociate availability of virtual memory from physical memory; currently availability of virtual memory is indirectly constrained by available physical memory, because used virtual memory must be described by `vmp_anon`s.

vm_amap

A map the anons associated with an anonymous VM object or the anonymous part of a *vm_map_entry*. The map is a 3-level trie, with each level one page in size. A pointer to the 1st level is stored in the `vmp_amap`, which stores pointers to 2nd levels, which itself stores pointers to 3rd levels, with the 3rd levels storing pointers to *vm_anon*s. The three-level system ensures that memory use is minimised for sparse regions of anonymous memory.

`vm_object_t` and *vmp_objpage*

These are mappable VM objects. Some are embedded in vnode structures. There are three kinds, and for each kind, different elements of a union of fields are used:

- Anonymous objects are backed by the swapfile. They are embedded in vnodes only in the case of *tmpfs*. They store *vm_amap* in the union.
- Regular objects must be embedded in vnodes; they are backed by the *getpage*/*putpage* routines of the vnode they are embedded in. They store a head of an RB tree of *vmp_objpage* structs in the union.
- Device VM objects refer directly to areas of physical address space, e.g. a framebuffer. They may not be embedded in vnodes. They store a start and size physical memory address fields in the union.

All three also carry a mutex lock.

A *vmp_objpage* is allocated for each resident page in a regular *vm_object_t*. These store linkage for the RB tree, the page number of the object they describe, and point to the *vm_page_t* where the data is resident.

`vm_map_t` and `vm_map_entry`

An abstract representation of a virtual address space that comprises an RB tree consisting of *vm_map_entry* ` structures. A special *kmap* ` contains the kernel's mappings, which are mapped into all processes but not accessible in user mode. *vm_map_t*s carry a mutex lock.

Each *vm_map_entry* is an entry in the address space map, with member fields including the base address and size of the virtual region it describes, the current and maximum memory protection of the region, and its inheritance; whether on fork it is virtual-copied or shared.

These entries may refer to a *vm_object_t*, and if they do, they include a page-aligned offset into the object. Faults in the virtual address region described by the map entry are handled by the underlying *vm_object_t*, except in the case of write faults on anonymous-on-vnode mappings; these are described later in the section on fault handling.

The has_anonymous flag is set if the entry refers to either process-private anonymous memory or a copy-on-write mapping of a *vm_object_t*. If the flag is set, the entry also has a *vmp_amap* ` associated with it.

1.2.3 Page management

In a virtual memory system, main memory is treated as the cache of secondary storage, so the virtual memory manager tries to ensure that a large amount of memory is used at all times as a cache, as unused memory is memory wasted. The technnique by which this is done in Keyronex is called paging - the movement of pages of data to and from the backing store with which that page is associated. Pages are associated with backing store according to which VM object they belong to; if they belong to a regular VM object, their backing store is a file or block device, while if they belong to anonymous memory, their backing store is the swap space.

Note: In the future Keyronex might support user-defined VM object types as well as the built-in regular and anonymous objects. A custom pager would be required for these to carry out page-in and page-out. One use-case would be to allow for a single-level store for the Oopsilon programming environment.

Pages are paged-in from their backing store in response to page faults, and paged out according to a page replacement policy. Keyronex uses a simple global page replacement algorithm, the FIFO second-chance approach, a variant of the general category of Not Recently Used page replacement algorithms in the same family as Clock. This involves maintaining two queues of pageable pages: active and inactive.

The page daemon

The page daemon, a kernel thread named *vm_pagedaemon*, is responsible for maintaining the page replacement policy. It maintains high and low watermarks for number of free pages and number of inactive pages, and spends most of its time sleeping on an event.

The event is signalled under two conditions:

- there has been a request to allocate a physical page, but the number of pages on the free queue is less than the low watermark for the free page queue; or
- greater than 75% of main memory is in use, and the number of pages on the inactive queue is less than the low watermark for the inactive queue.

The page daemon will wake up and calculate new watermarks for the inactive queue; these aim to keep around 33% of pageable pages - that is, the sum of the number of pages on the active and inactive queues - on the inactive queue. If the number of pages on the inactive queue is less than that of the low watermark, the page daemon will move pages from the tail of the active queue to the head of the inactive queue until the inactive queue high water mark is reached. Pages carry used bits to determine whether they have been accessed or not; this bit is reset when the page is moved to the inactive queue (this may involve a TLB shootdown; see the Physical Mapping section).

If the number of free pages is below the free page low watermark, the pagedaemon will now also take pages from the tail of the inactive queue and check their used bit. If it is set, the pages get a second chance - they are replaced to the head of the active queue. Otherwise, they are put back to their backing store. This is done by invoking the relevant *pager* according to the VM object to which the page belongs. For regular VM objects, the vnode pager is used, while for anonymous VM objects, the swap pager is used.

After the pager has completed the put back to backing store, the page is placed on the free queue. This process will continue in a loop until the number of pages on the free page queue reaches the high watermark.

Todo: describe what happens when no pages can be put back to backing store anymore, e.g. when pagefile space is exhausted.

Pagers

Pagers are reponsible for carrying out the actual paging-in and paging-out of pages; page-in requests are generated by page faults, while page-out requests are generated by the page daemon. The interface is simple - just a page-out function to put a page to backing store, and a page-in function to retrieve a page from backing store.

Page-in

Page-in takes two arguments - the *vm_page_t* to page in, and a *drumslot_t* - this is only passed for page-ins for anonymous memory, it's irrelevant for other object types. The fault handler will have allocated a new page already, and have inserted it into the owner, setting the busy bit so that any further page faults will wait on an event which will be signalled when the page-in is complete. The page fault handler unlocks the address space in which which the fault occurred, the owner object, and the page queues before calling the pager. The page-in routine must now carry out any I/O necessary to bring the page into memory, after which the busy bit can be unset. The fault handler now returns with the "retry" status code, causing the fault to be restarted.

Note: Dropping the locks requires page faults to be restarted after page-in, but it has a major advantage: when two threads share an address space map, it allows page faults on separate pages to be handled simultaneously, since the map remains unlocked. For vnode VM objects there are an additional two advantages, which come about because of vnode VM objects relying on just one lock, rather than the one-lock-per-anon of anonymous memory:

1. It allows for the pagedaemon to simultaneously page-out other pages of that object.
2. It allows other threads to simultaneously page-in other pages of that object.

Page-out

Page-out also takes only one argument, the *vm_page_t* to page out. The page daemon will have set the busy bit of the page and unmapped it from all physical maps in which it is mapped. Note that the object is unlocked at the time of making the page-out request. The pager can then do any I/O necessary to put the page to its backing store, after which it can lock the owner and update its state appropriately - that is, remove and deallocate the associated *vmp_objpage_t* from the RB tree of its owning *vm_object_t*, or set the owning *vmp_anon*'s state to 'nonresident' and set its drumslot appropriately.

Important: Page fault code paths have a "top-down" lock ordering (they lock the address space, then the object, then for anonymous mappings also the *vmp_anon*, then the page queues) while the page daemon code path has a "bottom-up" lock ordering (it locks the page queues, then the *vmp_anon* or *vm_object_t* the page is owned by.)

This lock ordering violation is dealt with by having the page daemon simply do a try-lock of the owner object; if the object cannot be locked, the page is put back on the queue and left for later. In a low-memory situation, fault handlers will be waiting with all locks released on an event which is signalled when memory is plentiful again, so even in the worst-case scenario the page will eventually be paged out.

Allocation

Pagers may need to allocate memory themselves to carry out page-out even under low-memory conditions. This is why the low watermark for free pages is set to a number higher than zero. When that low watermark is reached, most page allocation attempts will sleep until an event is signalled indicating that there are sufficient free pages to proceed again. Pagers do not make these sleepable allocations.

1.2.4 Anonymous mappings

Anonymous mappings supporting copy-on-write semantics are implemented efficiently with reference-counting. The core principle is that a `vmp_anon` with a reference count greater than 1 is always mapped read-only, and if there is a write fault at an address which is represented by that `vmp_anon`, it must copy the `vmp_anon` and its underlying page before mapping it read-write.

Todo: as well as the example below, fully detail the logic in an anonymous fault?

Consider a region of anonymous memory newly allocated in a process with PID 1. There are no `vmp_anon`s yet because they are lazily allocated on first fault. PID 1 writes to the first page of region; a `vmp_anon` is allocated with a refcnt of 1. PID 1 also writes to the 2nd page, and the same logic is followed.

Now PID 1 *fork()*s into PID 2. PID 2 is given a new anonymous map entry for that region with a copy of the `vmp_amap` of that of its parent's equivalent VM object. The copy refers to the same `vmp_anon`s, but the copying process has incremented the reference count of the 1st and 2nd `vmp_anon` as they are now referenced by two `vmp_amap`s. The copying process has also made all the old writeable mappings of these pages read-only again.

PID 2 now writes to the 2nd page of the anonymous region. The fault handler finds the corresponding `vmp_anon` and notices that its refcnt is 2. As this is a write fault, it must copy the `vmp_anon` and its underlying page. After copying it,

it releases its reference to the `vmp_anon` that was shared with PID 1, and maps the new copied `vmp_anon`'s underlying page read-write. The same thing would happen if PID 1 had tried to do the write.

1.2.5 Anonymous-on-vnode mappings

A special case of mapping is used for `MAP_PRIVATE` `mmap()`'s of a vnode. A *vm_map_entry* is created holding both a *vm_object_t* pointer to the vnode VM object, and also the *has_anonymous* flag.

Fault handling for this case is modified with respect to handling for faults in simple anonymous memory. A read fault will first check for a `vmp_anon` that already exists, but if there is none, it will instead ask the *vm_object_t* to map the page for the faulting address read-only into the process' address space.

In the case of a write fault, the page for the faulting address in the vnode object will be copied into a new page allocated which will be associated with a `vmp_anon` and placed in the anonymous-on-vnode *vm_map_entry*'s `amap`. This is then mapped read-write into the faulting process' address space, and copy-on-write has been achieved.

It should be noted that this is *asymmetric copy-on-write*; that is, once an anonymous-on-vnode mapping is established, if the *vm_object_t*'s pages are changed by writes into a shared mapping of that object, or by any other means, e.g. file I/O, then this does not cause those pages to be copied.

This means that unless and until there is a write fault within the anonymous-on-vnode mapping range, causing the page in the *vm_object_t* to be copied, the contents of the range may change.

1.3 I/O

1.3.1 Storage Stack

In this example storage stack, there is a VirtIODisk (*viodisk0*) at the root of the stack, which interfaces directly with the hardware (and - not pictured - below it would be a PCIDevice.) An attached Disk (*dk0*) provides the standard conveniences. Attached to *dk0* is a VolumeManager, which has detected two GPT partitions and created and attached two Volumes (*ld0s1* and *ld0s2*).

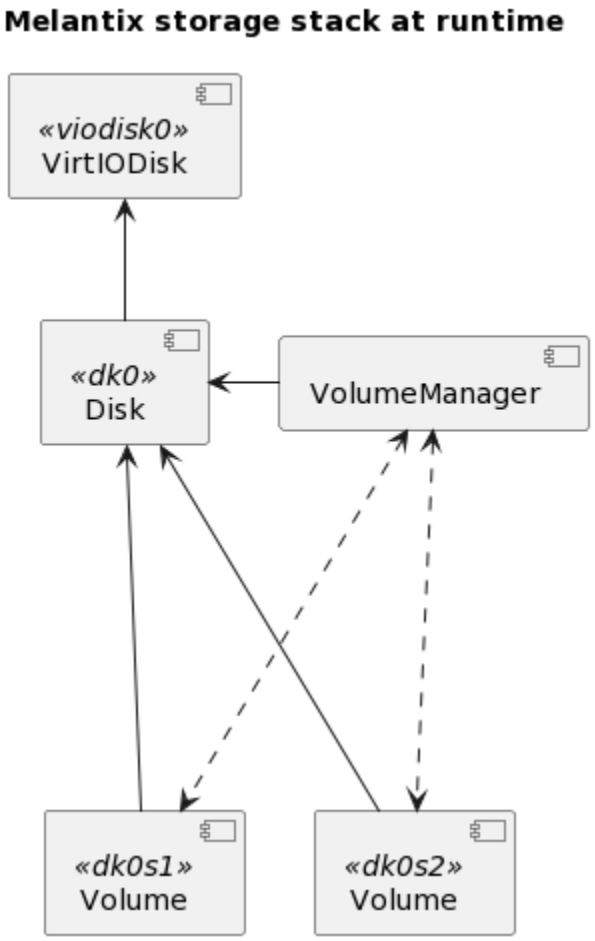

Fig. 1: A storage stack that might be created for a VirtIO disk.