The Nanvix Operating System

Pedro H. Penna

September 21, 2015

1 Introduction

Nanvix is an operating system created by Pedro H. Penna for educational purposes. It was designed from scratch to be small and simple, and yet modern and fully featured, so that it could help enthusiasts in operating systems, to learn about kernel hacking. The first release of Nanvix came out in early 2011, and since then the system has undergone through several changes. This paper details the internals of Nanvix 1.2. All previous and future releases are available at `github.com/ppenna/nanvix`, under the GPLv3 license.

In this section, we present an overview of Nanvix, starting with the system architecture, then presenting the system services, and finally discussing the required hardware to run the system. In later sections, we present a more detailed description of Nanvix.

1.1 System Architecture

The architecture of Nanvix is outlined in Figure 1. It presents a similar structure to Unix System V, and it has been intentionally designed to be so due to two main reasons. First, some successful operating systems, such as Aix, Linux and Solaris, are based on this architecture. Second, System V has earned Dennis Ritchie and Kenneth Thompson the 1983 Turing Award. These points indicate that System V is a well-structured system, thus serving as a good baseline design for a new educational operating system, such as Nanvix.

Nanvix is structured in two layers. The kernel, the bottom layer, seats on top of the hardware and runs in privileged mode, with full access to all resources. Its job is to extended the underlying hardware so that: (i) a more pleasant interface, which is easier to program, is exported to the higher layer; and (ii) resources can be shared among users, fairly and concurrently. The userland, the top layer, is where user software run in unprivileged mode, with limited access to the hardware, and the place where the user itself interacts with the system.

The kernel presents a monolithic architecture, and it is structured in four subsystems: the hardware abstraction layer; the memory management system; the process management system; and the file system. The hardware abstraction layer interacts directly with the hardware and exports to the other subsystems a set of well defined low-level routines, such as those for dealing with IO devices, context switching and interrupt handling. Its job is to isolate, as much as possible, all the hardware intricacies, so that the kernel can be easily ported to other compatible platforms, by simply replacing the hardware abstraction layer.

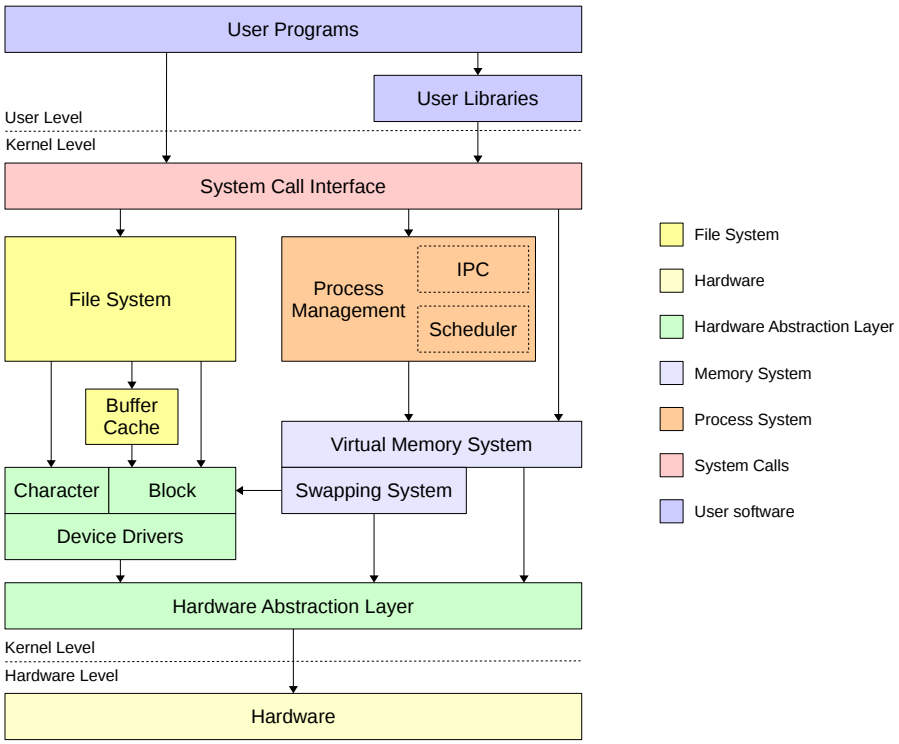


Figure 1: Nanvix architecture.

The memory management subsystem provides a flat virtual memory abstraction to the system. It does so by having two modules working together: the swapping and virtual memory modules. The swapping module deals with paging, keeping in memory those pages that are more frequently used and swapping out to disk those that are not. The virtual memory system, on the other hand, relies on the paging module to manage higher-level abstractions called memory regions, and thus enable advanced features such as shared memory regions, on-demand loading, lazy coping and memory pinning.

The process management system handles creation, termination, scheduling, synchronization and communication of processes. Processes are single thread entities and are created on demand, either by the system itself or the user. Scheduling is based on preemption and happens in userland whenever a process runs out of quantum or blocks awaiting for a resource. In kernel land, on the other hand, processes run in nonpreemptive mode and scheduling occurs only when a processes voluntarily relinquishes the processor. Additionally, the process management subsystem exports several routines that allow processes to synchronize their activities and communicate with one another.

The file system provides a uniform interface for dealing with resources. It extends the device driver interface and creates on top of it the file abstraction. Files can be accessed through a unique pathname, and may be shared among several processes transparently. The Nanvix file system is compatible with the one present in the Minix 1 operating system, it adopts an hierarchical inode structure, and supports mounting points and disk block caching.

The userland relies on the system calls exported by the kernel. User libraries wrap around some of these calls to provide interfaces that are even more friendly to programmers. Nanvix offers great support to the Standard C Library and much of the current development effort is focused on enhancing it. User programs are, ultimately, the way in which the user itself interacts with the system. Nanvix is shipped out with the Tiny Shell (`tsh`) and the Nanvix Utilities (`nanvix-util`), which heavily resemble traditional Unix utilities.

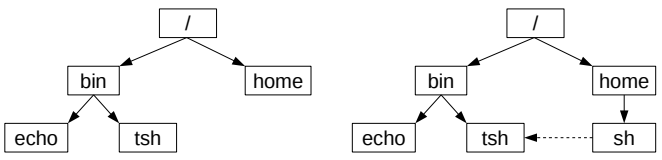
1.2 System Services

The main job of the kernel is to extend the underlying hardware and offer the userland a set of services that are easier to deal with, than those provided by the bare machine. These services are indeed exported as system calls, which user applications invoke just as normal functions and procedures. Nanvix implements 45 system calls, being the majority of them derived from the Posix 1 specification [**IEEE:08**]. The most relevant system calls that are present in Nanvix are listed in Table 1 and briefly discussed in the paragraphs that follow. For further information about all system calls in Nanvix, refer to the system man pages.

Files are high-level abstractions created for modeling resources, being primarily designed and used to provide a natural way to manipulate disks. One program that wants to manipulate a file shall first open it by calling `open()`. Then, it may call `read()` and/or `write()` to read and/or write data to the file. If the file supports random access, the read/write file offset may be moved through the `lseek()` system call. Finally, when the program is done with that file, it may explicitly close it by calling `close()`, and have its contents flushed to the underlying device.

Files are organized hierarchically, in a tree-like structure, and are uniquely identified by their pathnames. Programs may refer to them either by using an absolute pathname, which starts in the root directory; or by using an relative pathname to the current working directory of the program. Programs may change their current working directory by invoking `chdir()`. Alternatively, users can create links to files by calling `link()`, and then refer to the linked file by referring to the link (Figure 2). Links and files may be destroyed through the `unlink()` system call.

Every file has a owner user and a 9-bit flag assigned to it. These flags state what are the read, write and execution permissions for the file, for the owner user, the owner's group users and all others. If a program wants to perform any of these operations it must have enough permissions to do so. The file ownership and permissions may be changed through the `chown()` and `chmod()` system calls, respectively, and users can query these information by calling `stat()`.



(a) Before. (b) After.

Figure 2: `link()` system call.

Table 1: Most relevant system calls that are present in Nanvix.

Category	System Call	Description
File System	`chdir`	Changes the current working directory
	`close`	Closes a file descriptor
	`chmod`	Changes the file permissions
	`chown`	Changes the file ownership
	`ioctl`	Device control
	`link`	Creates a new link to a file
	`lseek`	Moves the read/write file offset
	`open`	Opens a file descriptor
	`read`	Reads from a file
	`stat`	Gets the status of a file
	`unlink`	Removes a file
	`write`	Writes to a file
Process Management	`execve`	Executes a program
	`exit`	Terminates the current process
	`fork`	Creates a new process
	`getpid`	Gets the process ID
	`kill`	Sends a signal to a process
	`pause`	Suspends the process until a signal is received
	`pipe`	Creates an interprocess communication channel
	`signal`	Signal management

Processes are abstraction of running programs and play a central role in Nanvix. Programs may create new processes by calling `fork()`, which creates an exact copy of the current running process. The primal process is called the parent and the new process is called child, and they have the same code, variables, opened files, and execution flow, differing only differ in their ID number. Processes may query about their ID by calling `getpid()` and may change their core through the `execve()` system call. When the process is done, it invokes `exit()` to relinquish all resources that it was using.

Processes may synchronize their activities using two approaches: through signals or synchronization primitives. In the former, the intend recipient process A first registers a callback function that will handle some specific signal, by calling `signal()`. If the process has nothing more to do than waiting for a signal to arrive, it may invoke `pause()` to block until such event happens. Later, another process may send a signal to A by calling `kill()`, therefore triggering the handler function in A. In the second approach, two processes may open a pipe, a dedicate communication channel, and effectively exchange data with one another. A process in one end of the pipe writes data to it, and in the other end a second processes reads data from the pipe, with all the required producer-consumer synchronization being transparently handled by the kernel.

1.3 Hardware Requirements

Nanvix has been primarily designed to target the `x86` architecture. Nevertheless, thanks to the hardware abstraction layer, it may be easily ported to other platforms. Still, the new platform shall meet some requirements to enable this smooth transition.

First, paging shall be somehow supported, since the memory management subsystem relies on this feature to enable virtual memory. Additionally, the hardware shall provide a protection mechanism that would point out whether a page fault has been caused due to a missing page or to a permission violation. Nanvix uses this information to easy the creation of new processes through the copy-on-write and demand paging techniques.

Second, the hardware shall support interrupts. Nanvix is a preemptive system, and the hardware shall provide some clock device that generates interrupts at regular time intervals. The scheduler completely relies on this feature, and the system would not even boot without it. Additionally, the hardware should also offer a mechanism to enable and disable interrupts. The kernel uses this mechanism to achieve exclusion on critical regions, and thus avoid race conditions in kernel land (chaos).

2 Process Management

In this section, we take a closer look in the process management subsystem. We start by first presenting the process structure itself; then highlighting how process scheduling and switching happens; and finally discussing the most important mechanisms exported by this subsystem.

2.1 The Process Structure

A process is an abstraction of a running program. It depicts the memory core, opened files, execution flow, access permissions, current state and every other relevant information about a program. Table 2 outlines the (simplified) structure of a process in Nanvix. The information about all processes is kept in a kernel table, named the process table, and it is globally visible to all subsystems.

Table 2: Simplified structure of a process in Nanvix.

Category	Field	Description
Context Switch Information	`kesp`	Kernel Stack pointer
	`kstack`	Kernel Stack
	`intlvl`	Interrupt Level
File System Information	`pwd`	Current Working Directory
	`ofiles`	Opened Files
	`tty`	Output Terminal Device
General Information	`status`	Exit status
	`pid`, `gid`, `uid`	Process, group and user IDs
Memory Information	`pregs`	Code, data, stack and heap segment regions
	`pgdir`	Page directory
Scheduling Information	`state`	Current state
	`counter`	Remaining quantum
	`nice`	Priority adjustment
	`priority`	Priority
Signal Information	`received`	Received signals
	`handler`	Signal handlers
Timing Information	`utime`	User CPU Time
	`ktime`	Kernel CPU Time

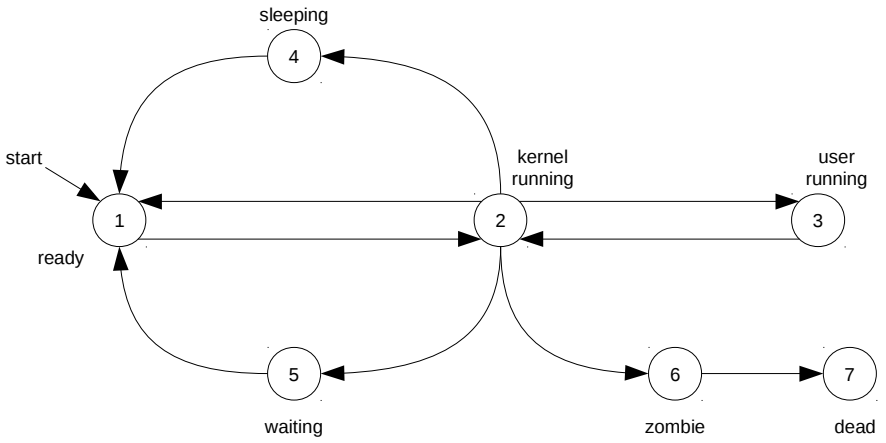


Figure 3: States of a process in Nanvix.

The process management subsystem maintains the process table: whenever a new process is created, a new entry is added to it; and when a process terminates, the corresponding entry is erased. However, it is worthy to point that each subsystem is in charge of maintaining their own fields in the process structure. For instance, the memory management subsystem is the one that fills up information regarding the segment regions and page directory, whereas the file system keeps track of the current working directory and opened files fields.

A process is created whenever a user launches a program or the system itself spawns a new daemon. After that, the new process goes through a complex lifetime before it terminates, as it is shown in Figure 3. Initially this new processes is ready to execute (state 1), and it is eventually selected to run by the scheduler. At this moment, the process resumes back its execution in kernel land (state 2) and jumps to userland (state 3). There, the process performs some computation, and it may either get preempted if it runs out of quantum time (state 4), or block waiting for a resource (state 5). Either way, the process later resumes its work, and loops back on these states. When the process finally gets its job done, or it somehow crashes, it terminates and becomes a zombie process (state 6). A zombie process awaits to gets all resources assigned to it to be taken away by the kernel, and then it is turned in a dead process (state 7).

2.2 Process Scheduling and Switching

Several processes may be active in the system, but only one can be running at a time[1]. The scheduler chooses which process to run next whenever the running process gets preempted or blocks. In user mode, that happens when the running process runs out of quantum and gets preempted by the kernel, or else when it issues a system call. In kernel mode on the other hand, scheduling occurs only when the running process voluntarily relinquishes the processor and goes wait for a resource to be released.

[1]Nanvix does not support multiprocessor systems, thus processes are indeed not running in parallel.

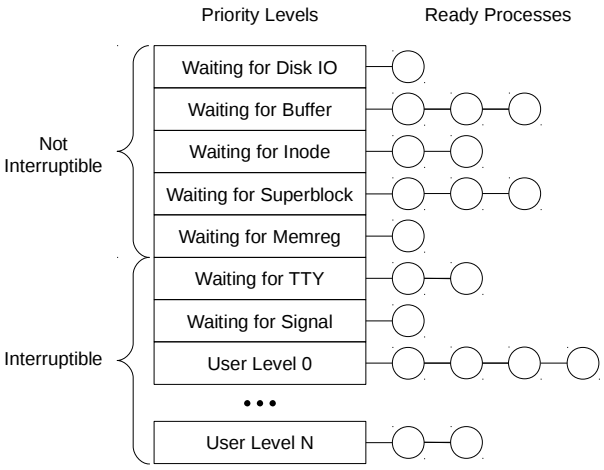


Figure 4: Scheduler queues.

The scheduler (Figure 4) uses a priority based criteria to choose the next process to run, with processes with higher priorities being scheduled to run before those with lower priorities[2]. Processes with equal priorities are chained in a queue and are selected in a round-robin fashion.

The priority of a process changes over the time and it is given by the sum of three numbers: base priority, dynamic priority and nice value. The base priority is assigned by the kernel itself and changes as the process sleeps and wakes up from resource queues, and leaves the kernel or enters it. The dynamic priority is increased by the scheduler as the process waits longer in the ready queue. Finally, the nice value is adjusted by the users, exporting them a mechanism to control which processes are more priority over others.

Once the scheduler chooses which process to run next, the process management subsystem performs the context switching operation. It first pushes the contents of all machine registers in the kernel stack. Then, it instructs the memory management unit hardware to switch to the address space of the selected process. Finally, the state of the machine registers are restored from the kernel stack of the selected process. The context switching operation is machine dependent and it is actually carried on by the hardware abstraction layer.

2.3 Process Creation and Termination

In Nanvix, processes are created through the `fork()` system call. This call instructs the process management subsystem to create a new process that is an exact copy of the calling process. The primal process is called parent and the new process is called child, and they have the same code, stack and data segments, opened files and execution flow, differing only differ in their ID number. The `fork()` system call is a complex and expensive operation, and in order to get it done the process management subsystem heavily interacts with the memory management system and the file system.

[2]Nanvix adopts the Unix's priority system, in which lower values mean higher priorities.

To create a process, the kernel first looks for an empty slot in the process table, to store there the information about the process. Then, an address space for the process is created: page tables are initialized, the kernel code and data segments are attached to the process' core, and a kernel stack for the process is created. After that, the kernel duplicates every memory region of the parent process and attaches them to the address space of the child process. In this operation, underlying pages are not actually copied, but they are rather linked and copied on demand. This technique, called copy-on-write, greatly speeds up the `fork()` system call, and it is covered in Section 3.3. Then, when the address space of the new process is built, file descriptors for all opened files are cloned, and every other information is handcrafted. Finally, the process management system marks the child process as new, so that the scheduler can properly handles this situation, and places it in the ready queue for later execution.

The process then performs some computation and undergoes through a complex lifetime, which we depict in Section 2.1. When the process finally finishes its job it invokes the `exit()` system call to terminate. This call orders the process management subsystem to actually kill the process and release all resources that are assigned to it.

To do so, several steps are involved. First, the kernel masks out signals and closes all files that are still opened. After that, all child processes of the process that is about to terminate are assigned to a special process, called `init`[3]. This ensures that no process becomes orphan, and thus become unreachable by signals (see Section 2.4). Then, the kernel detaches all memory regions from the dying process and marks it as a zombie. At this moment, the dying process still has an address space and a slot in the process table, and the kernel has no way to wipe off these information. Therefore the dying process hands out this task to the parent process, and sends a death of child (`SIGCHLD`) to it. This signal cannot be blocked or ignored, and it is eventually handled by the corresponding parent process. When that happens, the parent process destroys the address space of its zombie child process, and marks the corresponding slot in the process table as not used.

2.4 Process Synchronization and Communication

In Nanvix, processes may cooperate to one another to accomplish a common goal. To enable this, the kernel provides three synchronization primitives, so that processes can work together without messing up each other's job: `sleep()`, `wakeup()` and `wait()`.

When running in kernel land, processes often need to acquire and later release resources, such as slots in the system's tables and some temporary memory. The `sleep()` and `wakeup()` routines are used in these situations to avoid race conditions in kernel land. The former puts the calling process to wait in a chain; whereas the later causes all processes that are sleeping in a chain of processes to be moved to the ready-to-execute scheduling queue. These routines support two different sleeping states: one that is interrupitible by the receipt of signals, and another that is not. Thanks to this mechanism, users can interact with foreground processes, such as those that read/write to the terminal.

In userland, processes may coordinate their activities by calling the `wait()`

[3] `init` is a daemon process whose job is to spawn the logging processes.

system call. This call causes a process to block until one of its child process terminates. When such event happens, the parent process is awaken and the exit status of the terminated child is made available to it. This information can be later parsed by the parent process, so that it can known why exactly its child process has terminated and properly handle the situation.

These three mechanisms enable processes to synchronize their activities and friendly work together towards a common goal. Notwithstanding, processes also have to somehow be able to communicate with one another to exchange valuable information about their work progress. Nanvix offers two main mechanisms to do that: signals and pipes.

Signals are short messages that processes can send/receive asynchronously. They are mainly used to notify about the occurrence of interesting events, such as that a process has to be killed, a remote terminal has hangup and a breakpoint has been reached. The process management subsystem exports two system calls to deal with signals: `signal()` and `kill()`. The former allows a process to actually control how signals shall be handled. A process can either choose to ignore a signal, and not be notified at all about it, or else handle a signal, by registering a callback function that will do the job. The `kill()` system call on the other hand, allows a process to raise a signal, either to another process or to itself. Additionally to these system calls, the kernel also provides a system call, named `pause()`, which allows a process to wait for the receipt of a signal. Signals are implemented by having the kernel to instrument the user stack, when a signal is send. The kernel carefully inserts hooks to the registered signal handler in the stack, and when the process leaves the kernel, it transparently executes the signal handler function (Figure 5).

Signals are useful for exchanging small amount of information. However, if processes want to carry on more verbose conversations, they can use pipes instead. A pipe is a virtual file that serves as a dedicated communication channel between two processes. At one end of the pipe, a writer process puts data on the pipe, and at the other end a reader process takes data out from it. Processes can open pipes through the `pipe()` system call, and after that they can read/write data to it by using the `read()` and `write()` system calls, as they would normally do. In all these operations, data transfers take place entirely in memory, with the kernel borrowing a page frame from the kernel page pool and pinning it in memory. Additionally, the kernel handles the required producer-consumer synchronization with the help of the `sleep()` and `wakeup()` routines.

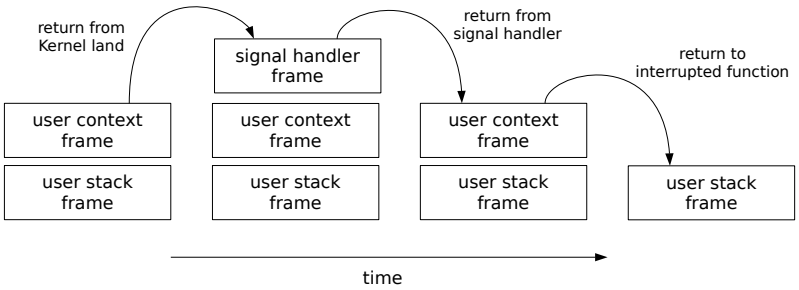


Figure 5: Signal stack.

3 Memory Management

In this section, we detail the internals of the memory management subsystem. We first present the memory layout of a process in Nanvix, then we introduce the memory region abstraction, and finally we detail the virtual memory and paging systems.

3.1 Memory Layout

The virtual memory layout of typical process in Nanvix is presented in Figure 6. The overall address space is divided in two great portions: one that is owned by the kernel and is shared among all processes; and a second one that is private and belongs to the process itself. The kernel is the only one that has full access to the address space of a process. While running in user mode, if a process touches any kernel memory, a protection fault is raised. This way, the kernel can isolate and protect itself against malicious software, thus increasing the system's security and reliability.

The kernel address space is carefully handcrafted so that it looks as transparent as possible to user processes. The kernel code and data segments are placed to two different locations: identity mapped at the bottom of the memory, and virtually linked just after the user memory. Some operations performed

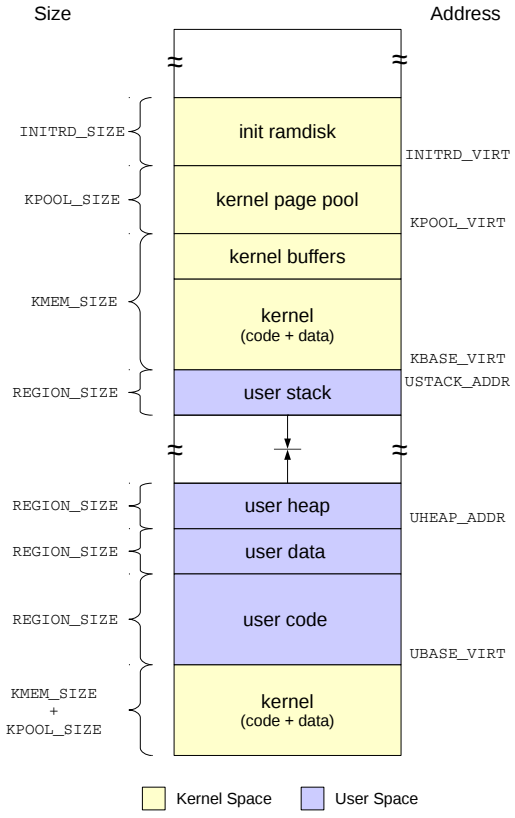


Figure 6: Virtual memory layout of a process in Nanvix.

by the kernel, like physical memory copy and context switching, require the virtual memory system to be shutdown momentarily. Placing the kernel code and data segments at the bottom of the memory, enables physical address access; whereas positioning the whole kernel just after user memory ensures that future enhancements on the kernel will not affect the user memory layout.

The kernel buffers, kernel page pool and init ramdisk are accessed exclusively through virtual addresses. The kernel buffer area provides the file system with some temporary memory to hold disk data. The kernel page pool feeds the kernel memory allocator with some memory that will indeed be used to build dynamic structures in the kernel, such as page tables. Finally, the init ramdisk is a static memory area that functions as an in-memory file system and stores configuration files and (eventually) dynamic loadable drivers, which are all used during system's startup.

The user memory portion occupies most of the virtual address space and it is divided in several regions. The user code and data regions are statically allocated and hold the text and static data segments of the binary file, respectively. The user stack and heap regions on the other hand, dynamically grow and shrink with the course of time. The former is used to store temporary variables and it is managed by the kernel's memory management subsystem. The heap region however, holds long-term variables and it is handled by the process itself. User libraries rely on the `brk()` system call to request the kernel to attach, or detach, some memory to the heap.

3.2 Memory Regions

In Section 3.1 we have briefly discussed that the user memory portion of a process is divided in four regions, namely code, data, stack and heap regions. A region is a high-level abstraction created by the memory subsystem to easy the management of user memory and abstract away the underlying technique that is used to actually enable virtual memory (*ie.* segmentation or paging). This abstraction is presented in Figure 7 and briefly discussed bellow.

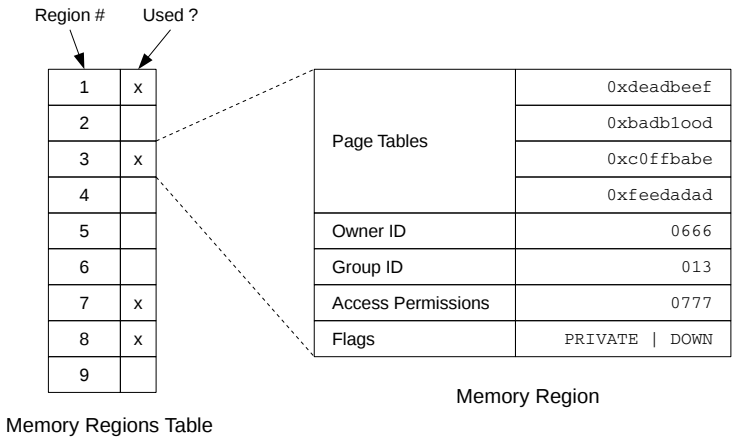


Figure 7: Memory region.

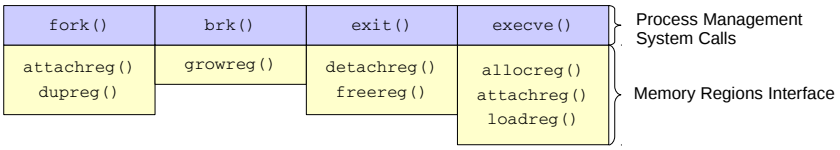


Figure 8: Memory regions interface.

From a design perspective, a memory region is indeed a collection of page tables[4], with some extra metainformation about the region itself. Pointers to underlying page tables are stored in a table, so that the virtual memory and paging systems can quickly retrieve low-level information about the pages that are assigned to a region. The owner and group IDs, and access permissions fields are used together to determine what processes can read or write to the memory region. Finally, the flags field indicate the growing direction of the region, whether or not the region can be swapped out to disk, and whether or not the region is shared among several processes.

The memory management subsystem maintains memory regions in a global table, namely the memory regions table, and exports a well defined interface for dealing with them. The process management subsystem heavily relies on the routines provided by this interface to implement the `brk()`, `exec()`, `exit()` and `fork()` system calls, as it is outlined in Figure 8. The `allocreg()` allocates a new memory region, by searching an empty slot in the memory regions table and properly initializing its fields. It is important to point out however, that underlying pages are not actually assigned to the memory region in this routine, but they are rather allocated on-the-fly, using a technique called demand paging (see Section 3.3). Conversely, `freereg()` frees a region table by releasing underlying pages and page tables, and erasing the corresponding entry from the memory regions table.

The `loadreg()` routine is used to load a region with some data from a binary file. This routine invokes raw I/O procedures exported by the file system to actually find the corresponding inode and perform the read operation. The `growreg()` procedure expands (or shrinks) a target region by a certain size, by allocating/freeing page tables accordingly. The `attachreg()` and `detachreg()` routines attach and detach a memory region to the address space of a process, respectively. They do so by effectively linking/unlinking underlying page tables to/from the page directory of a process. Finally, the `dupreg()` routine creates an exact copy of a target memory region. To speedup this task, the memory management subsystem uses a lazy copying technique known as copy-on-write, which e further discuss in Section 3.3.

3.3 The Virtual Memory and Paging Systems

While the memory management system exports the memory region interface to other system modules, internally it works with paging and swapping to enable virtual memory. The former technique delivers a flat memory abstraction, whereas the latter effectively enables the physical memory to be virtually extended.

[4]The memory region abstraction equally applies to segmentation.

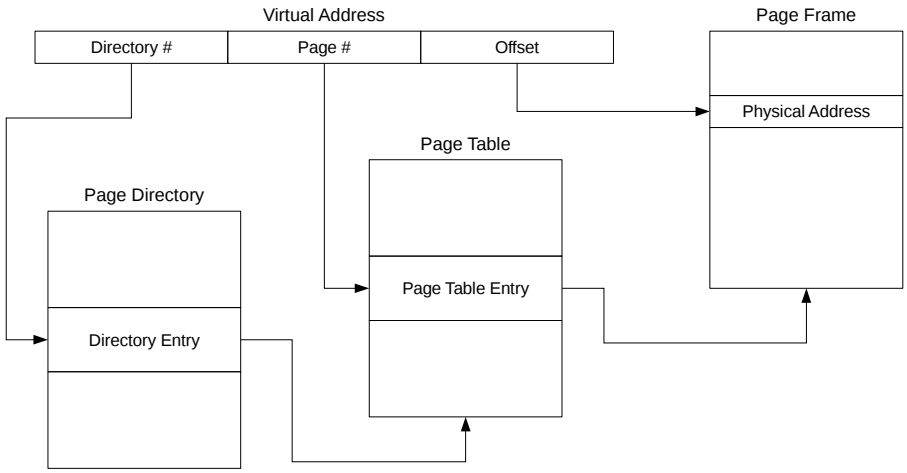


Figure 9: Two-level paging scheme used in Nanvix.

The paging system relies on a two-level page table that is depicted in Figure 9 and detailed next. Each process has a single page directory, and each entry in this directory points to a page table. Whenever a virtual address is generated, the memory management unit (MMU) breaks it in three portions. The upper portion is used to index the currently active page directory, and thus to find out the corresponding page table. Then, the middle portion is used to index the page table, and hence to discover the physical frame where the requested page is placed. Finally, the third portion is added to the address of the memory frame to build the ultimate target physical address.

Page directories and page tables of all processes are pinned down in kernel memory, and the paging system carefully bookkeeps the physical addresses of these structures. This way, the kernel can access any address space from whatever process is running, handle physical memory copying, and step away from live locks when performing swapping. Memory frames, on the other hand, are handled in a different way by the internal memory frame allocator. The paging system maintains a table, namely the memory frames table, where all information about in-memory pages are stored. When the system first startups, it marks all entries in this table as not used. Whenever a page is requested, the paging system searches this table for an empty slot and allocates the corresponding memory frame. If no slot is available, the swapping system is invoked to swap out an in-memory page to disk.

Aside from managing page directories and page tables, and dealing with page allocation, it is worthy to note that the paging system also implements two fundamental mechanisms: demand paging and copy-on-write. The former is extensively used in the `exec()` system call and works as follows. Whenever an executable file needs to be loaded to memory, the paging system does not copy any data from disk nor allocate any space in memory at first. Instead, it marks all corresponding entries in the page table as "not present, demanded paging". Later, when an attempt to access an address that falls in such kind of page is made, a page fault occurs and then the paging system allocates a frame to that page and loads it in to memory.

The second mechanism, namely copy-on-write, is heavily employed in the `fork()` system call to easy the overall task of creating a new process. In a nutshell, this system call creates a child process that is an exact copy of the calling process by duplicating every opened file descriptor and cloning the memory address space of the parent process. The former step can be completed in a handful of time as it comes down to iterate over the file descriptors table incrementing the reference count for each opened file. On the other hand, cloning the address space of a process can be a real time-consuming endeavor, since that involves copying a massive amount of data. When copy-on-write is enabled however, such operation is greatly speed up by having the copy operation to happen on demand, rather than immediately. To enable so, at first, the kernel marks writable pages of the parent process as "read-only, copy-on-write". Later, when a write attempt is made to a marked page, a protection fault is fired and the kernel performs the actual cloning, marking both pages as writable in the end.

So far, we have detailed the internals of the paging system, which delivers a flat memory abstraction to the whole system, but now it is time to turn our attentions to the swapping system. The ultimate goal of this later one is to keep in memory those pages that are more frequently used, and use some disk space to store those that are not. This way, it is possible to execute a process whose memory footprint is larger than the physical memory available. In other words, the physical memory is virtual extended, enabling the maximum memory footprint of a process to be bounded by the size of the virtual address space, rather than the physical one.

To achieve such goal, the swapping system relies on a First-in-First-out (FiFo) strategy to evict pages to disk. For each in-memory page, the swapping system bookkeeps the timestamp in which that page has been brought into memory. Whenever the swapper is invoked, the page with the oldest timestamp is evicted to the swapping area, in disk. The swapping system selects pages using a local-policy and implements no load control mechanism. On the other way around, when a page is brought in to memory from the swapping area, a shadow copy is kept there. This way, if no modifications are done to that page, no extra I/O operation to write the page back to disk needs to be performed.

4 File System

The file system is certainly the largest and most complex subsystem in Nanvix. It extends the hardware abstraction layer and exports through the file abstraction a uniform interface for dealing with bare resources. In this section we take a closer look in this subsystem. We first present the buffer cache, an internal module that speedups the performance of the file system, and then we move our discussion to the file system's layout, structure and internals.

4.1 The Buffer Cache Module

Most requests that arrive at the file system involve disk I/O operations. If, in order to handle each of them, the file system would have to issue an I/O operation, the system's performance would be awful. Therefore, to get around this problem, the file system buffers disk blocks in memory and also maintains a cache of most frequently used buffers there, as shown in Figure 10. In this

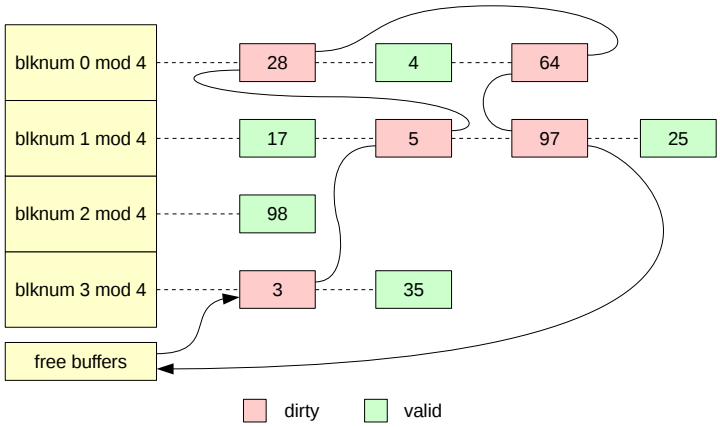


Figure 10: Buffer cache.

cache, all buffers are hashed in a table according to the number of the disk block that they currently hold, and buffers that are not in use are linked in a list structure. This way, any buffer can be quickly retrieved through the hash table, and the kernel can easily get a free buffer from the list, whenever it needs one.

The buffer cache module exports a set of routines to other subsystems to enable them manipulate buffers. To read a buffer from the cache, the `bread()` routine is provided. Internally, this routine searches the buffer cache for a specific block and returns a pointer to it. If the requested block is not in the buffer cache, its contents are read from disk to some buffer. What buffer will be be used to handle the request may come from two different sources. If the list of free buffers is not empty, the kernel claims a buffer from there and uses it in the operation. However, if this list is empty, a buffer from the cache is evicted to disk, according to a least-recently used policy, and then it is used to handle the `bread()` request.

After retrieving a block buffer from the buffer cache, subsystems may freely read/write data from/to the buffer. Nevertheless, it is important to point out that buffers are indeed shared resources, and thus to consistently operate these structures subsystems should carefully use locking mechanisms. To handle this situation, the buffer buffer cache module provides the `blklock()` and `blkunlock()` routines, to enable buffers to be locked and unlocked respectively. Finally, when a subsystem is done with a buffer, it calls `bwrite()` to release it. If no more processes are using that buffer, its contents are flushed to the underlying device and then the buffer is marked as "not used".

As a final remark, it is worthy to note two techniques that are supported by the buffer cache to further improve the system's performance. The first technique, namely delayed write, is outlined in Figure 10 and works as follows. As we detailed before, not-used buffers are linked in a list of free buffers, so that the kernel can easily get a buffer whenever it needs one. Note, however, that buffers that are not being used also appear in the hashing structure. The buffer cache does so in the hope that those buffers will soon be need, and thus tries to postpone writing as much as possible. The second technique, known

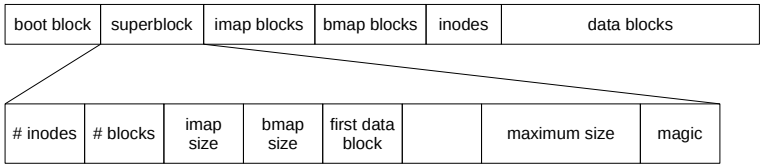


Figure 11: File system layout.

as asynchronous read, improves the overall system usage by having read operations to take place asynchronously, rather than synchronously to the process' execution flow. To enable this, the buffer cache handles a read operation as it would normally do, however, instead of scheduling a synchronous read operation in the disk driver, it schedules an asynchronous one and immediately returns. The process may then go compute something else, and when the content of the buffer gets valid, it is notified about this event.

4.2 File System Layout and Structure

Nanvix adopts an hierarchical file system based on inodes, which heavily resembles the original System V file system reported in [**Bach:86**]. Indeed, the Nanvix file system has been intentionally designed to be somewhat compatible with the Minix 1 file system, so that existing user-level utilities specifically designed to manipulate this file system could be used in early development phases[5]. The Nanvix file system is outlined in Figure 11 and detailed next.

The Nanvix file system works with 1 KB blocks and it is divided in two major zones, one that stores meta information about the file system itself and another one that actually holds user data. The very first block in the meta information zone stores the bootloader, a very small and special program that setups all machine dependent features and actually loads the kernel. Following the boot block, is the superblock, the place where information concerning the file system's peculiarities are kept. For instance, it holds how many data blocks and inodes the file system has, what is the maximum size for a file size and what is the file system version. After this block, there are the inode-map (imap) and block-map (bmpap) blocks, which are used for managing disk space; and the inode blocks, which store all inodes of the file system. Finally, data blocks hold user data, and are linked to the file system hierarchy through the inode structure.

In turn, an inode is a low-level abstraction of a file. It stores information concerning the file's type, owner, access permissions, current size, time of last modification, and blocks pointers, as it is shown in Figure 12. Data blocks are assigned using an hierarchical addressing scheme, where the first seven data blocks are directly addressed, and next ones are addressed via either double or triple indirections. The way that blocks pointers are actually used depend on the type of the file itself. If it is a regular file, then blocks pointers point to data blocks filled with user data. If the file is a directory, however, block pointers point to data blocks that store directory entries. Finally, if the file is a special file, the block pointers hold the minor and major numbers of the referred device.

[5]Since version 1.1 Nanvix is shipped with its own utilities for sake of portability.

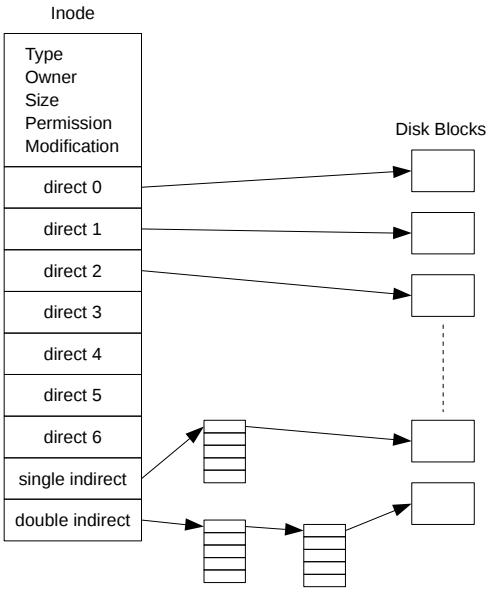


Figure 12: Inode Structure.

4.3 File System Internals

So far we have explored the buffer cache module and the file system layout. In this section, however, we turn our attentions to the file system internals. We first discuss how disk space is actually managed, then we introduce the interface that is exported for dealing with inodes, and finally we detail some kernel routines for manipulating files.

The file system manages disk space with the help of the `bmap` and `imap` blocks (see Section 4.2). These are indeed bitmaps that point out which data blocks and inodes are currently in use in the file system, respectively. Whenever an inode or a data block needs to bee allocated, the correspondent bitmap is sequentially searched for an inode/data block that is available. To speedup the search, the kernel maintains these bitmaps in memory and from time to time it writes them back to disk. Additionally, the kernel keeps track of the last allocated bit in each bitmap and uses bitwise operations to perform the search. This way, the first free bit in a bitmap can be find slightly fast, if the disk is not heavily fragmented, and search time is decreased by a factor of n, where n is the width in bits of the largest machine register. To allocate and free data blocks the kernel exports two routines, namely `block_alloc()` and `block_free()`; and conversely does the same for inodes, through the `inode_alloc()` and `inode_free()` routines.

Inodes are low-level abstractions for files, and are used to model regular files, directories and devices. The most relevant routines for manipulating such structures are presented in Table 3 and briefly discussed next. Ultimately, inodes are stored in disk, so that changes in the file system can persist over time. However, to further improve the file system's performance, the kernel caches in memory the most recently used inodes, in a similar way as it does with disk blocks (see Section 10). To get an inode from the inode cache, the file

Table 3: Interface for dealing with inodes.

Category	Routine	Description
Locking	`inode_lock`	Locks in inode
	`inode_unlock`	Unlocks an inode
Inode Cache	`inode_get`	Gets an inode
	`inode_put`	Releases an inode
	`inode_sync`	Synchronizes the in-core inode table
Traversal	`inode_name`	Converts a path name to inode
	`inode_dname`	Gets inode of the topmost directory of a path

system exports the `inode_get()` routine. If the requested inode is in the cache, a pointer to it is immediately returned. However, if the inode is not in the cache, the kernel reads it from the disk, first evicting not-in-use inodes using a least recently used strategy if needed. Conversely, the `inode_put()` routine is provided to release an in-memory inode, making available the correspondent slot in the inode cache. Finally, to sync the in-memory inodes with in-disk inodes, the file system provides the `inode_sync()` routine. This call iterates over the cache, writing back to disk every inode that is dirty.

The `inode_get()` routine is handful in situations where the serial number of the target inode is known in advance. For instance, when mounting a device or reading/writing data to a file that has been already opened. On the other hand, if the serial inode number is not known, but its pathname is, one can use either `inode_name()` or `inode_dname()` routines. The first one retrieves the inode that matches a given pathname, and the second when gets the inode for the topmost directory in the pathname. These routines work by having the pathname to be parsed and the file system hierarchy to be traversed, differing essentially in their stop criteria.

The last two routines exported by the file system, namely `inode_lock()` and `inode_unlock()`, are provided for locking purposes. The first one causes the calling process to block until it acquires exclusive access to a target inode, and the second routine releases the lock that a process owns over an inode and wakes up all processes that were awaiting for this event. Having a process to call `inode_lock()` before using and inode and `inode_unlock()` when it is done with it enables processes to consistently operate over inodes, thus promoting the file system's integrity.

On top of the inode interface, the kernel builds higher-level abstractions and routines. A table, namely the opened file descriptors table, is used to bookkeep all files that are opened in the system. Each entry in this table stores a pointer to the underlying in-memory inode and tracks the current read/write cursor in the file. The opened file descriptors table is heavily manipulated by the `open()` and `close()` system calls. For reading/writing to a file, the kernel provides `file_read()` and `file_write()`. These routines take as parameters a target inode and buffer, and transparently handle a read/write request by traversing the data-block tree in the inode structure. Conversely, to add and remove entries from a directory, the file system exports the `dir_add()` and `dir_remove()` routines.