The Nanvix Operating System

Pedro H. Penna

August 6, 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 both, novices and experienced 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 gone 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 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 points. First, several 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-architected and reliable 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 the 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 level; and (ii) resources can be shared among users, fairly and concurrently. The userland, the top layer, is where all 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.

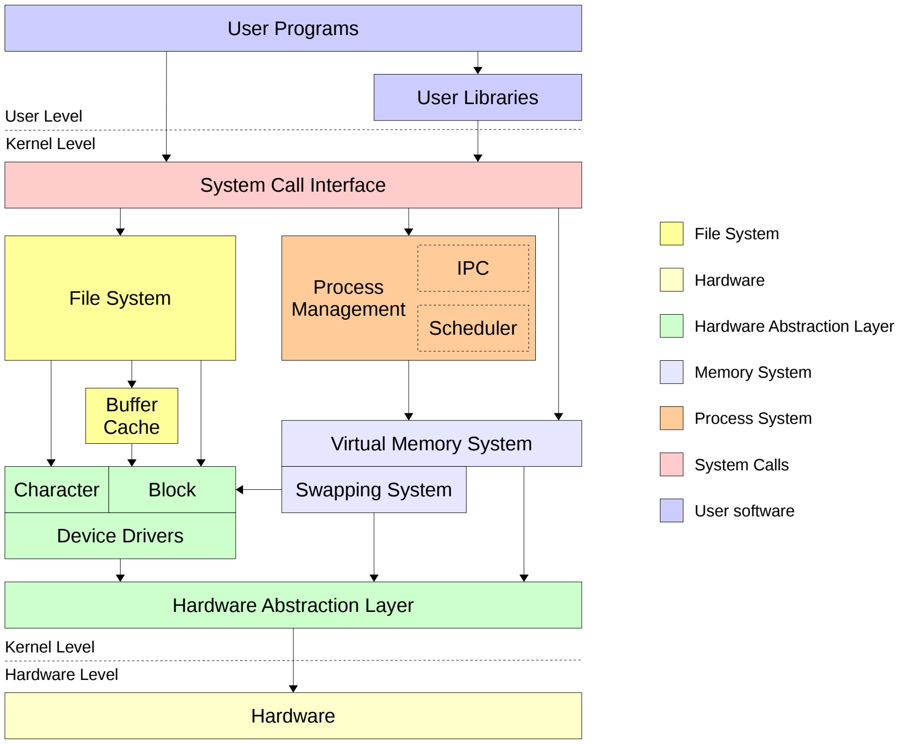


Figure 1: Nanvix architecture.

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, destruction, 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. Finally, processes many synchronize their activities using semaphores, and communicate with one another through pipes and shared memory regions.

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 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 pleasant 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.

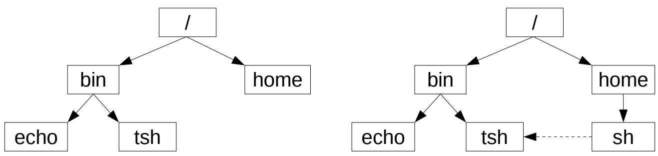
1.2 System Services

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 [??]. The most relevant system calls that are present in Nanvix are listed in Table 1. In the paragraphs that follow, we briefly discussed each of them. For further information about system calls in Nanvix, refer to the 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 it. 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, stack and data segments, 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()`, 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 the required producer-consumer synchronization being 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 technique.

Second, the hardware shall support interrupts, because Nanvix a preemptive system. More precisely, the hardware shall provide some clock device that would generate interrupts at regular time intervals. The scheduler completely relies on this feature, and the system would not even boot without it, getting stuck on the idle process. Additionally, the hardware should also offer a mechanism to enable and disable interrupts. The kernel uses this to achieve mutual exclusion on critical regions, and thus avoid race conditions.