The Nanvix Operating System

Pedro H. Penna

August 20, 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.

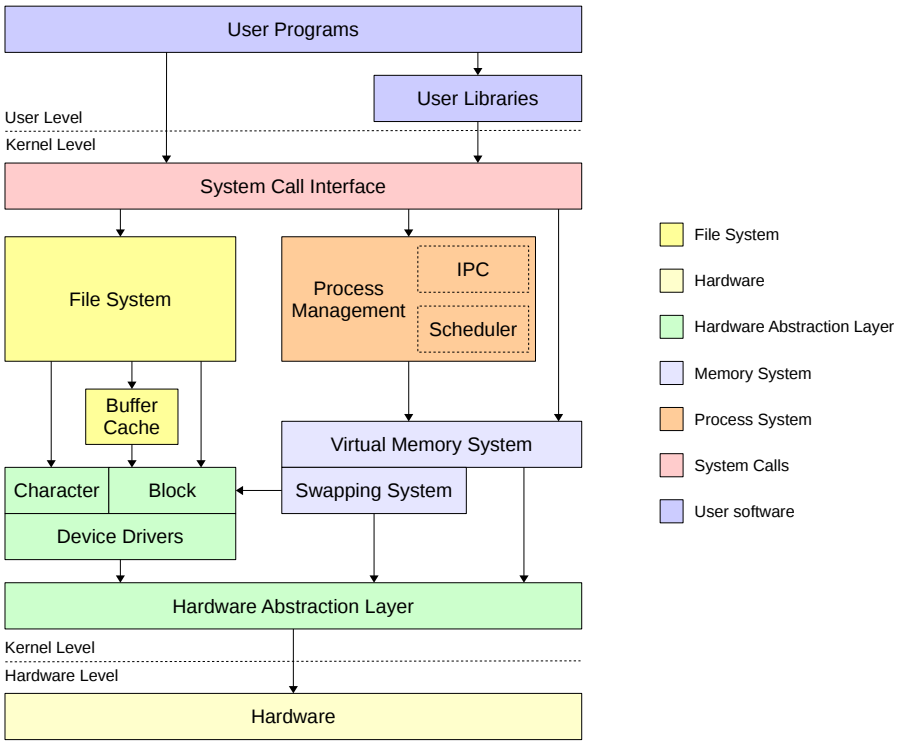


Figure 1: Nanvix architecture.

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. Finally, processes synchronize their activities using `sleep()` and `wakeup()` primitives, and communicate with one another through pipes and signals.

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 [**IEEE:08**]. 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 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()`.

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

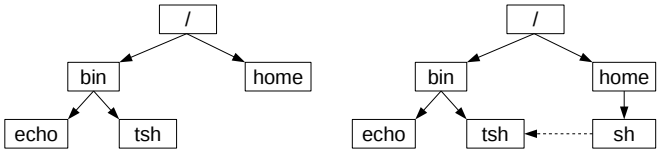
Table 1: Most relevant system calls that are present in Nanvix.

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

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



(a) Before. (b) After.

Figure 2: `link()` system call.

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. Nanvix is a preemptive system, and 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. Additionally, the hardware should also offer a mechanism to enable and disable interrupts. The kernel uses this feature to achieve mutual exclusion on critical regions, and thus avoid race conditions.

2 Process Management

In this section, we take a close 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 offered 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.

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. Nevertheless, 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 his 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).

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

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.

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 tree 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, offering them a mechanism to control which processes are more priority over others.

Once the scheduler choses 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

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

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

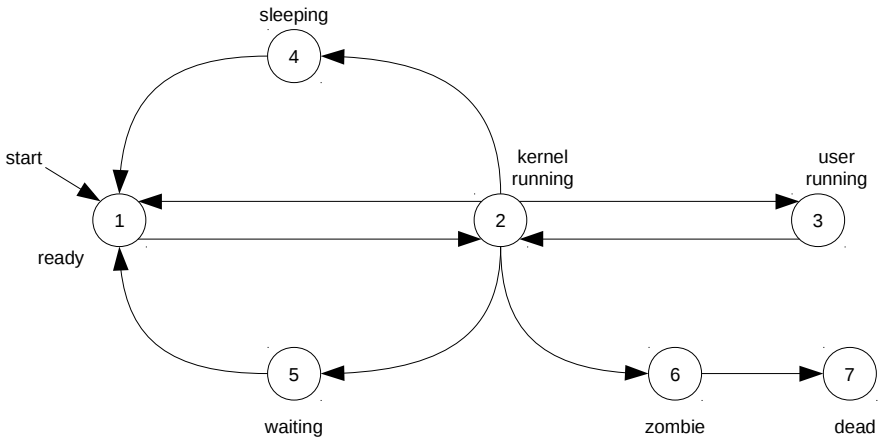


Figure 3: States of a process in Nanvix.

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 system to create a new process that is an exact copy of the calling 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. 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.

First the kernel searches for an empty slot in the process table, to store there all the information about the new process. Then, a new address space for the child 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 ??. 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 system to actually kill the process and release all resources that are assigned to it.

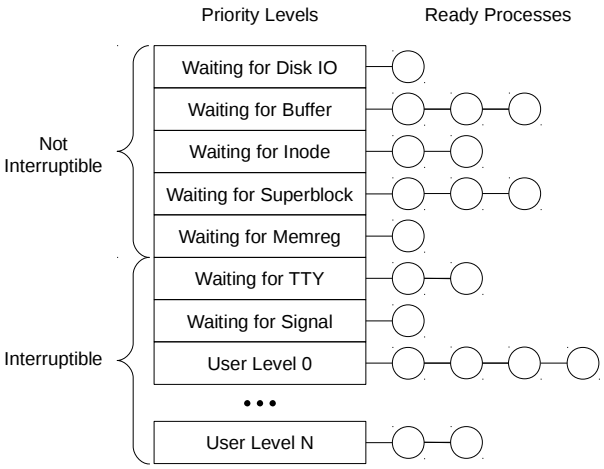


Figure 4: Scheduler queues.

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 it has no way to wipe off these information. The process then 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. At this time, 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 a set of synchronization primitives, so that processes can work together without messing up each other's job. In the current version of the system, three primitives are available: `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 under this situation to avoid race conditions in the kernel. The former puts the calling process to wait in a chain; whereas the later causes all processes that are sleeping in a process chain to be moved to the ready-to-execute scheduling queue. These routines support two different sleeping states: one that is interrupted 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.

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

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

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

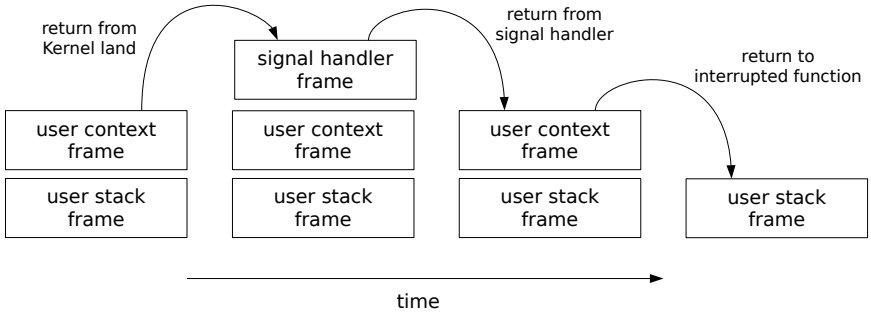


Figure 5: Signal stack.

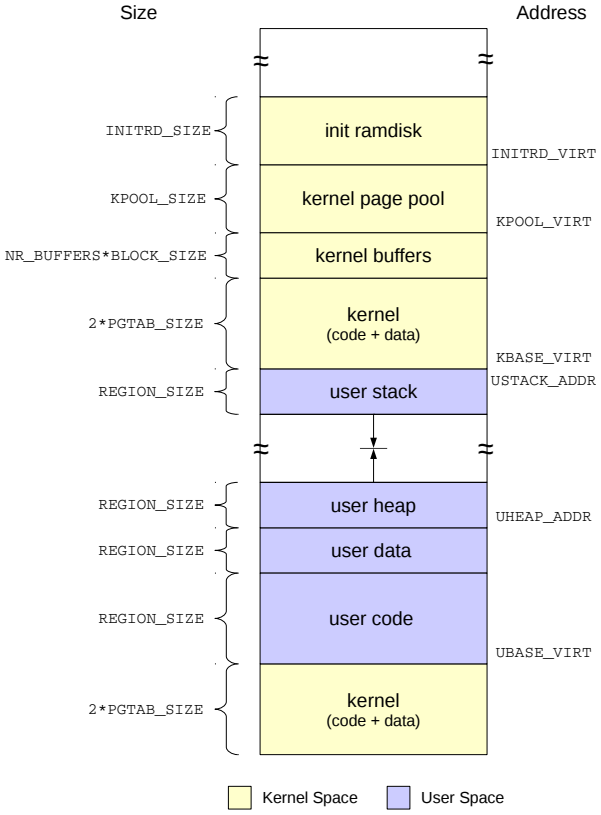


Figure 6: Virtual memory layout of a process in Nanvix.

placed to two different locations: identity mapped at the bottom of the memory, and virtually linked just after the user memory. Some operations performed 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 a 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 overall 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 seen 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 techinique that is used to enable virtual memory (*ie.* segmentation and paging). This abstraction is presented in Figure (??) and briefly discussed bellow.

From a design perspective, a memory region is indeed a collection of page tables[4], with some extra metainformation about the region itself. A table stores pointers to underlying page tables, which limits the maximum size of a region. The owner and group IDs, and access permissions fields are used to determine what processes can read or write to the memory region. Finally, the flags field indicate the growing direction of the region, wether or not the region can be swapped out to disk, and wether or not the region is shared among several processes.

The memory management subsystem maintains memory regions in a global table and exports a well defined interface for dealing with them. Some routines provided by this interface are listed in Table 3 and briefly detailed next. The `allocreg()` allocates a new memory region, by searching an empty slot in the memory regions table and properly initializing it. Conversely, `freereg()` frees a region table by releasing underlying page tables and erasing the corresponding

[4]Nanvix has been formally designed to paging based systems, but the memory region concept equally applies to segment-based or hybrid-based machines.

Table 3: Interface for dealing with memory regions.

Routine	Brief Description
`allocreg()`	Allocates a memory region
`attachreg()`	Attaches a memory region to a process
`detachreg()`	Detaches a memory region from a process
`freereg()`	Frees a memory region

entry from the memory regions table. The `loadreg()` routine is used to load a newly created region with some data from a binary file. This routine internally invokes raw I/O procedures exported by the file system to actually find the corresponding inode and perform the read operation. Finally, 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 the underlying page tables to the page directory of a process.

3.3 The Virtual Memory System

3.4 The Paging System