Tilda Lerner

5/3/2024

Arch Linux Technical Report

The Linux kernel is a fascinating study in the vast potential of user control over hardware through intuitive yet highly customizable operating systems. In my years studying computer science, I have heard people talk about Linux on several occasions. When first trying my hand at programming a Raspberry Pi, I found it could only run lightweight systems, for which Linux was perfect. In fact, the Raspberry Pi OS is Linux-based. As such, when given the opportunity for researching operating systems, I wanted to delve into the deep realm of Linux.

In researching this paper, it became evident to me that the primary appeal of Linux is its versatility. Any user can install any series of packages to essentially create their own custom environment. The open-source nature of the Linux kernel makes every default option entirely open to customization. This has allowed developers to essentially create any operating system for whatever project they have.

The wide breadth of Linux “distributions” is indicative of the system’s wide potential. There are hundreds of various versions of Linux, referred to as distros, which work towards different goals. Some environments, like Linux Mint and Ubuntu, provide full functionality right after install, mimicking more standard desktop environments like Windows and macOS. Some distros, like Puppy Linux, were designed with the goal of fitting on extremely lightweight hardware, so that one can work with low-spec devices.

The distro often touted as being “the best” is Arch Linux. Developers online often take pride in using this system, and I’ve seen many respectable developers using their own Arch setup. While I have tried my hands at experimenting with more mainstream distributions like Ubuntu, Arch has always fascinated me, as every installation of Arch Linux I have seen seems entirely unique. This range of customization, alongside its reputation, has always been significantly appealing to me. As such, I wanted to use this paper as an opportunity to understand why Arch is so often revered as one of the strongest Linux distros, and how much it relates back to the original Linux kernel.

An interesting note to make about Linux distros is the amount that is still built up on top of the Linux kernel. This is particularly evident when we examine the Arch Linux documentation, and find writing that explains how “since Linux 2.6.23, the default scheduler is CFS, the ‘Completely Fair Scheduler’. The CFS scheduler replaced the earlier ‘O(1)’ scheduler.” (sched, Arch Linux Manual) It quickly becomes evident in reading statements like these that, while Arch presents its own features at a more surface-level, a lot of the foundational decisions were taken from the design of the Linux kernel. This scheduling policy is elaborated on later in the documentation:

> "CFS's design is quite radical: it does not use the old data structures for the runqueues, but it uses a time-ordered rbtree to build a "timeline" of future task execution, and thus has no "array switch" artifacts (by which both the previous vanilla scheduler and RSDL/SD are affected). [...]
>
> CFS also maintains the rq->cfs.min_vruntime value, which is a monotonic increasing value tracking the smallest vruntime among all tasks in the runqueue. The total amount of work done by the system is tracked using min_vruntime; that value is used to place newly activated entities on the left side of the tree as much as possible." (CFS Scheduler, The Linux Kernel)

This scheduling approach is rather impressive, using a red-black tree to mimic an ideal multitasking CPU, which, according to the Linux kernel documentation, "is a (non-existent :-)) CPU that has 100% physical power and which can run each task at precise equal speed, in parallel, each at 1/nr_running speed." (CFS Scheduler, The Linux Kernel) Much like how we discovered in class, there is always an ideal which we strive for but is technically impossible. The incorporation of powerful data structures like red-black trees creates a speedier scheduling environment.

In researching these facets, it is also clear that Linux strives to fulfill every motivation put forth for important features. The completely fair scheduler accounts for both fairness and speed, which were essentially the two complications related to every concept we covered in class. When studying processes and threads, I found that Linux boasts a rather standard toolset for handling them. In fact, it gets acknowledged in The UNIX Time-Sharing System paper from UC Berkeley. They acknowledge how "processes and threads are also quite standard on Linux." (The UNIX Time-Sharing System, Dennis M. Ritchie & Ken Thompson) Essentially, everything we seemed to have studied in class was covered by the Linux kernel.

There are many other similarities between Linux and UNIX. In the paper, they write that "the major programs available under UNIX are: assembler, text editor based on QED, linking loader, symbolic debugger, compiler for a language resembling BCPL with types and structures (C), interpreter for a dialect of BASIC , text formatting program, [...] and per-muted index program." (The UNIX Time-Sharing System, Dennis M. Ritchie & Ken Thompson) All of these aforementioned features are foundational to the development of the Linux kernel, as they are all core mechanics provided in every system.

According to the Linux documentation, “memory management in Linux is a complex system that evolved over the years and included more and more functionality to support a variety of systems from MMU-less microcontrollers to supercomputers.” In other words, memory management in Linux is meant to be flexible so that it can work in any environment, regardless of whether that environment is supported by a memory management unit. Linux is meant to be flexible and operable on any sort of hardware, regardless even of what components it comes with.

Reading the documentation on memory management feels like reviewing lessons learned in class, evidently because these policies were ultimately most applicable to any computing environment. The documentation describes how, “to avoid spending precious processor cycles on the address translation, CPUs maintain a cache of such translations called Translation Lookaside Buffer,” (Memory Management Concepts, The Linux Kernel) a concept which was also covered in class. It is evident therefore how the approaches taken by the Linux kernel to handle various concepts like memory management, are also deeply foundational in their influence.

Another interesting thing to note is the support the Linux kernel offers in terms of lack of hardware. A complex article provided by the Linux kernel documentation is on how the kernel operates on hardware that does not come with a memory management unit. They write that “the kernel has limited support for memory mapping under no-MMU conditions, such as are used in uClinux environments.”  (No-MMU Memory Mapping Support, The Linux Kernel) Nevertheless, the rest of the article is on the support available, which is surprisingly quite extensive. The other project mentioned, uClinux, was an offshoot of the Linux kernel developed specifically to work on hardware without MMUs. While it technically is not the Linux kernel itself, the fact that it was an offshoot of Linux is exemplary of the level of versatility strived for by the Linux project.

While there are plenty of shortfalls within the kernel, particularly here with no-MMU support, these shortfalls are addressed in the documentation, as they write, for example, that, “NOMMU mmap automatically rounds up to the nearest power-of-2 number of pages when performing an allocation. This can have adverse effects on memory fragmentation, and as such, is left configurable.” (No-MMU Memory Mapping Support, The Linux Kernel) Again, the ability to fully configure Linux to suit whatever a developer needs allows for a highly flexible environment, where adjustments can be made anywhere to create a potentially more efficient system.

What ultimately distinguishes Linux distributions are the foundational programs included within it, including the desktop interface and package manager. Arch Linux comes bare of any substantial program upon install, except for its package manager. Each distro uses its own package manager; Arch uses a manager called Pacman.

```
 .--.                  Pacman v6.1.0 - libalpm v14.0.0
/ _.-' .-.  .-.  .-.   Copyright (C) 2006-2024 Pacman Development Team
\  '-. '-'  '-'  '-'   Copyright (C) 2002-2006 Judd Vinet
 '--'
                         This program may be freely redistributed under
                         the terms of the GNU General Public License.
```

Package managers, like pacman, are often lightweight and highly configurable. Pacman allows the option of choosing which mirror to install from, even allowing the mirrors to operate locally. The user can also use this package manager to build and distribute their own packages, proving the versatility of the program. Something to note is that while each distribution has a different package manager, like apt, pacman, or whatever else, most of them operate very similarly. It ultimately comes down to a question of need, as distributions come with specific managers fit for their own systems.

A choice the user can make is in deciding the desktop interface. Some Linux distributions come entirely console-based, giving the option to the user as to what desktop environment they want to work out of. Arch comes devoid of any desktop interface, instead coming with a command line straight after install from which the user can install any variety of desktop interface programs. This approach allows for versatility with Arch being as complicated or bare-bones as the user needs.

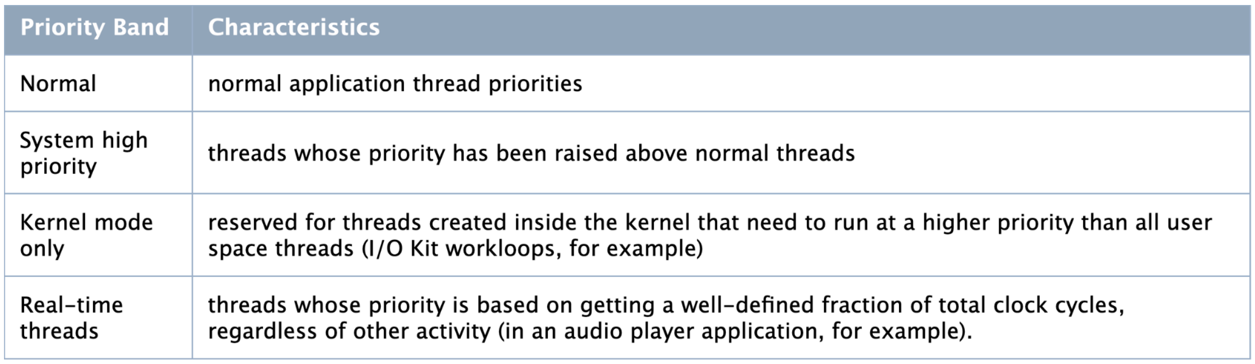
In researching the Linux kernel, I also became interested in MacOS and the XNU kernel, for two main reasons: the first being that I use MacOS as my day-to-day operating system; the second being that both Linux and MacOS seem to be based on Unix. XNU stands for X Not Unix. As I researched Linux, I grew curious about how it differed from my Mac. The most fascinating aspect of both XNU and Linux is in the scheduling of processes.

Finding documentation on MacOS, however, is significantly more difficult. A lot of documentation I have found is archived, referring to OS X, which was the old name for MacOS. As such, a lot of the writing I have found is likely outdated. Further research has revealed conflicting facts about MacOS; however, I could only find the old archived documentation as an official publication from Apple. The fact that Linux's documentation is so easily accessible is a testament to its open source nature. The ease with which any researcher can find any information on how the Linux kernel operates is what makes open source material so powerful.

Nevertheless, though archived, it is interesting to see the basis which OS X was built off. In particular, I found the comparison between the scheduler of the XNU kernel to that of the

Linux kernel quite interesting. The XNU kernel seems more preoccupied with hybridizing a myriad of approaches, writing in the documentation that, "OS X is based on Mach and BSD. Like Mach and most BSD UNIX systems, it contains an advanced scheduler based on the CMU Mach 3 scheduler." (Mach Scheduling and Thread Interfaces, Apple Documentation Archive) This approach feels quite different from the Linux documentation, which explicitly stated that their scheduler is the completely fair scheduler and not a hybrid of schedulers.

The OS X documentation also explicitly covers threading and the priority of various threads. The diagram below elaborates on the various priorities assigned to various threads of various imports:



| Priority Band | Characteristics |
|---|---|
| Normal | normal application thread priorities |
| System high priority | threads whose priority has been raised above normal threads |
| Kernel mode only | reserved for threads created inside the kernel that need to run at a higher priority than all user space threads (I/O Kit workloops, for example) |
| Real-time threads | threads whose priority is based on getting a well-defined fraction of total clock cycles, regardless of other activity (in an audio player application, for example). |

(Mach Scheduling and Thread Interfaces, Apple Documentation Archive)

When combing the Linux kernel documentation, it is surprisingly difficult to find documentation on threads. As previously stated, Linux does provide thread support; however, I would assume that, since so much of the kernel is built on foundational concepts covered in class, threading should be considered to operate standard to every other facet of multiprogramming.

While ultimately confusing and daunting, this paper has proved the complexity of documentation for topics matching in complexity, like operating systems. The Linux documentations are rich in information, much of it built on concepts covered in class. It is apparent in researching Arch that it provides an environment as close to the rules provided by the kernel as possible. Strangely enough, Arch Linux feels quite UNIX-adjacent, in the sense that UNIX and Linux are quite similar in their goals and approaches, and Arch is essentially the Linux kernel with added functionality. There is certainly a lot more to study, even just in the realm of Linux.

Sources:

CFS Scheduler, The Linux kernel: https://docs.kernel.org/scheduler/sched-design-CFS.html

Memory Management Concepts, The Linux Kernel: https://docs.kernel.org/admin-guide/mm/concepts.html

Mach Scheduling and Thread Interfaces, Apple Documentation Archive: https://developer.apple.com/library/archive/documentation/Darwin/Conceptual/KernelProgramming/scheduler/scheduler.html

No-MMU Memory Mapping Support, The Linux Kernel: https://docs.kernel.org/admin-guide/mm/nommu-mmap.html?highlight=mmu

sched, Arch Linux Manual: https://man.archlinux.org/man/sched.7.en

The Unix Time-Sharing System, Dennis M. Ritchie & Ken Thompson: https://dsf.berkeley.edu/cs262/unix.pdf