The 3 Tier Kernel Design

A modern, secure and fast rejection of the status quo in kernel design

revision 1.0, 27th of April 2025

Written by Logan Russell
(https://github.com/GBX9570)

A brief overview of the 3TK

The 3TK is a kernel design based around speed, stability, and security. It employs various approaches to achieve these goals:

- Hardware-based ring separation (using all rings in the x86 architecture, including ring 1 and ring 2)
- Distinct separation of drivers from kernel space and user space
- Strict control of what different things can do and access
- A new approach to syscalls in User Space, circumventing the micro-kernel approach of an IPC and the monolithic approach of direct access to the kernel using the SCD (Syscall Communication Driver)
- Minimal driver overhead outside of the kernel
- Enforced driver sandboxing
- No legacy baggage
- Security *by default* – not an afterthought
- A way to restart without ever needing to power off

This 3TK was designed with a clean slate in mind – not to replace the designs of Microsoft's NT Kernel, the Linux kernel or the Darwin kernel. This is ideal for advanced hobbyist operating systems, and is especially useful for RTOS's (real time operating systems) that could see significant benefit from a stable, secure and fast kernel. More on this at the end.

This design aims to remove the age old problem of monolithic v. micro-kernel. Micro-kernels, of course, are ridiculously stable – at the expense of performance. Monolithic kernels are famous for their speed, but infamous for their instability. The 3TK is designed to take the best of both worlds and some new takes on system engineering to make a far improved – albeit, complex – approach to kernels.

Warning – This design is not only very complex in many ways, it incorporates features that are poorly documented (eg, using ring one and two). This isn't for the faint of heart, and you should be very well versed in operating system development before attempting this. Here be dragons!

The Design

The 3TK follows a strict design, separated into 4 operational tiers and 3 security tiers.

Operational Tiers

Ring 0 – Kernel mode

- Contains all kernel processes, code, hardware management, etc
- No limitations
- Contains the bare basics for driver operation and user space

Ring 1 – Driver Space 1 (DS1)

- Close to Ring 0, not much overhead for jumping into kernel space if necessary
- Used for drivers that require access to the hardware with very little performance overhead (eg, GPU drivers)
- Contains the SCD, which is necessary for using syscalls in userspace

Ring 2 – Driver Space 2 (DS2)

- Closer to Ring 0 than userspace, but far away enough to be a bottleneck (by design)
- Used for drivers that aren't particularly important to separate them from critical drivers in Ring 1 (eg, USB drivers)

Ring 3 – User Space

- Far away from Ring 0, very slow for communication to the kernel
- All user processes (like programs, desktop environments) would run here

Security Tiers

Tier 1 – Restricted

- Drivers in this tier can only run inside DS2, and are only given absolutely necessary permissions (for example, a serial driver would run in Tier 1 and ONLY have permission to the serial port and any dependencies of that, nothing more)

Tier 2 – Intermediate

- Drivers in this tier can only run inside DS1, and are given many more permissions than Tier 1 drivers. They may need direct hardware access, to manipulate memory, etc etc. These drivers will be permitted to access these things, with strict measures to ensure they are not abused.

Tier 3 – Unrestricted

- This tier runs within kernel mode. Only very specific, very specialised drivers may run within Tier 3. Something like a CPU or chipset driver that directly interacts with the systems most critical components may run in Tier 3, but something like a GPU driver or serial driver will be restricted to their respectful tiers.

Security tiers define what drivers can and can't do; operational tiers dictate where drivers can operate.

Driver Isolation

Drivers are not isolated in the traditional sense, ie, being separated from the rest of the system and being forced to work within constraints. Rather, they are isolated based on strict and frequent auditing. A piece of software is given strict rules, and they must follow them. For example:

Software type: driver
Tier: Ring 2, Tier 1
Name: Generic VGA text Framebuffer Driver (80x25)
Permissions: Direct memory manipulation within set boundaries
Allowed memory addresses: 0xB8000 through 0xB9000

This driver would be allowed to manipulate through 0xB8000 to 0xB9000 with no issues. If the driver attempted to manipulate eg, 0xB7999 or 0xB9001, then a General Protection Fault (https://en.wikipedia.org/wiki/General_protection_fault) exception will be called, resulting in a fatal error. The kernel can decide how to handle this, and whether or not it should kill and restart the process.

If a driver tries to use permissions outside of its granted ones, it will call the kernel to do this. This will alert the *PRW (Permissions Regulation Watchdog)*.

The role of the PRW is to determine whether or not a driver is over-reaching its limits (eg, trying to use a permission that it isn't allowed to). For example, if a driver developer forgot to request access for syscalls, then the PRW would allow the syscall to be ran and continue normal operation. If the driver suddenly attempted to manipulate the memory of another process (and it somehow didn't cause a GPF), the PRW will stop this. If the PRW stops a process, the driver will immediately be restarted. If the driver attempts to use the permission again, it will be shutdown completely and flagged in the systems logs. Second infringements are ALWAYS treated as malicious attacks, not as mistakes – anything after the second one will cause the driver to be treated like malware and be blocked from running.

The Permissions Regulation Watchdog Setup
The PRW essentially acts in the same way as white blood cells in the blood stream. They are constantly active, but don't really do much until required. Anything drivers do is seen by the kernel, for example, if a driver had this code:

```
#include "libs/print.h"

int main(void) {
    printf("Hello, Kernel World!");
    return 0;
}
```

This is how it would be relayed to the kernel:

```
libs/print.h used
int main declared
printf used – within permissions?
Driver authorised to use print, continuing
```

But what if the driver wasn't authorised to use print (or anything else)? This is where the PRW comes in. The kernel send the PRW to check on the driver. If it is something non-critical, like print, the PRW would likely just allow it. But in this case, we will assume that print is somehow extremely critical. The driver would firstly be restarted and monitored. This would be logged in the kernel logs. If print is not called again, the

PRW will return to an idle state. If print is called again, then the PRW will kill the process and log it in the kernel logs.

Syscall Communication Driver (SCD)

The common method of communicating with the kernel or servers in a micro-kernel is using an IPC. This can be thought of as essentially a pipe, where messages are passed in encrypted messages and decrypted on the other end. While this is secure, it is slow and relatively expensive in overhead. Monolithic kernels, on the other hand, directly interact with the kernel from user space – which is extremely fast, but creates the possibility of abuse (ie, a piece of software repeatedly and maliciously accessing the kernel directly causing fatal errors).

Here comes in the *SCD*. The SCD is a privileged driver in DS1 that can take input from both User Space and DS2. Essentially, the program accesses a shared memory buffer to query whether or not it can use a syscall.

Common syscalls like `printf()` are authorised by default, without querying the kernel. However, more critical syscalls (like process termination, hardware interrupts, etc) will be queried. If the kernel determines that the program should not be using the requested syscall, the PRW will be launched and investigate as usual, regardless of whether its a program or driver in this case. If the kernel determines the program should be using the requested syscall, it will relay that back to the SCD and the SCD will relay that to the User Space.

It is worth noting 'syscalls' are not really syscalls with this approach – to prevent overhead from context switching and passing arguments, the SCD unlocks the User Space to use functionally identical functions. However, these are not by any circumstances what functions the kernel would use.

This concludes the necessary parts to be a 3TK compliant kernel. Below are nice-to-haves, but not required.

Hot-Reboot process

In mission-critical cases, a full restart of the system could be catastrophic. That is why the 3TK supports the HRP (hot reboot process). When called (for example, in the case of an exception or update to the kernel), everything except for the bare basics are gradually shut down when it is safe. Every program saves its data and memory into a custom file type called .hrpr (hot reboot process recovery), which contains everything the program may have been doing in an encrypted file. The kernel, upon the HRP being called, shuts down to only a basic GDT, basic paging, a file system driver and basic serial output. Once all has been recovered or files have been updated, the kernel begins the boot init system and essentially starts up as if the system had just been powered on. Custom tasks cannot be ran in this state as the kernel is extremely vulnerable (ie, no protections whatsoever on memory, programs, etc).

POSHix

poSHix is a compatibility layer for 3TK kernels with POSIX/Unix programs. While implementation may vary between kernel, it essentially does the same thing; allow the end-user to use Unix scripts on a wildly incompatible kernel. It should essentially translate POSIX/Unix syscalls into the syscalls of the kernel, translate filesystems, provide APIs, etc etc.

UDAPI

The UDAPI (Unified Driver API) is essentially a tool that allows drivers to work independent of kernel as long as it is:

- 3TK compatible
- the driver uses only syscalls, no kernel or OS specific functions

Essentially, it provides a translation sheet that translates the syscalls of one kernel into a universal version. For example, one kernel may use:

```
#include <print.h>

int main(void) {
    printf(“Hello, World!”);
    return 0;
}
```

Where as another kernel uses

```
#include <print.h>
```

```
int main(void) {
    kprint(“Hello, World!”);
    return 0;
}
```

The UDAPI would translate `kprint` and `printf` into something like ‘uprint’, making code look like this:

```
#include <udapi.h>

int main(void) {
    uprint(“Hello, World!”);
    return 0;
}
```

Behind the scenes, this is still `printf` and `kprint`, but the driver doesn’t know any different.

Q&A

Why not just use a micro-kernel?
Micro-kernels are extremely secure, and that is commended. However, the IPC, servers, encryption, user space drivers, etc etc makes it extremely slow. The 3TK tries to be as secure as possible like a micro-kernel while keeping the speed of a monolithic kernel
Why not just use a monolithic kernel?
Monolithic kernels are great for unimportant tasks that don’t require extreme stability (ie, a desktop computer). While monolithic kernels have come a long way since the Windows 9x and Classic macOS days, they are still far from as stable as micro-kernels. A general-purpose operating system, in reality, probably doesn’t need what the 3TK offers – but no one would turn down stability *and* speed.
Why use the x86 hardware tiers instead of software protection? Does that not hinder adoption on platforms without rings?
The reason for using rings is that in software protection, there is always an exploit. There is always some way that the protection will be broken. If you need to manually request access from the kernel to do anything outside of your ring (which any malware would need to do to really affect anything), it can immediately be stopped by built-in hardware protections as opposed to software protections that could be bypassed

by any small mistake. Unfortunately, this does restrict the 3TK to x86 platforms – however, it could likely be emulated (of course, at the expense of some security).

When will Windows (or macOS or Linux) adopt this?
Good question, tl;dr – they won't. The long answer is different for each OS

Windows
Windows has used the NT kernel since 2001 on the consumer side and 1993 on the commercial side. All drivers and programs for ~24 years at this point have been written expecting:

- Ring 0 accesses
- The NT kernels unique design
- Windows security features

To suddenly replace the NT kernels 32 year legacy with the 3TK would be ludicrous, as every single process would need to be rewritten majorly. It would, essentially, lead to most developers abandoning windows
macOS
This is an interesting one. macOS already employs some security features like sandboxing and a semi-micro-kernel design – however, the Darwin (macOS/iOS/tvOS/etc) kernel is made of many different kernels (Mach, XNU, various parts of FreeBSD and the original BSD) and to implement the 3TK would require everything ever written for macOS to be completely redesigned as changing 3 different kernels (that weirdly enough culminate not into an operating system but one kernel) would be a huge effort – and a futile one, as they are all so incompatible that they would essentially be new kernels at the end.
Linux
Linux is probably the most likely contender of the three – it is completely FOSS, modular, secure (with things like SELinux, and is secure by itself anyway) and can be edited to work with the 3TK. However, it is not without its challenges.

One of Linux's primary points is 'Don't break User Space' – something that has been said since the very beginning. Essentially, this says to always stay compatible with old programs if possible. The 3TK would severely damage that, as it would require an entirely new approach to kernel and user space interaction, driver interaction, and program interaction. Most Linux programs, if converted, would no longer work as intended. Then, there is the kernel. The kernel is extremely modular – however, it is designed to be monolithic – much like the NT kernel. All drivers ever written were written to run in Ring 0 kernel mode, and to force them all to run in Ring 1 or 2 would have the same pitfall of doing the same to the NT kernel – no drivers (or kernel modules, for that matter) would work anymore.

However, what about hobbyist operating systems? They assume blank slates, and as long as they are not focusing on compatibility with Linux or something else, they could very much implement this design. They would benefit from the strict security and speed, while not worrying about the legacy of old programs. Unfortunately, the complexity of the design may scare many developers away.

Real Time Operating Systems (RTOS) may benefit from this, however. They are often designed as micro-kernels for stability, but speed is always a problem. RTOSs are found commonly in anywhere from vehicle dashboards to spacecrafts – and a fast, stable and low requirement kernel (other than programming knowledge, of course) would be groundbreaking.