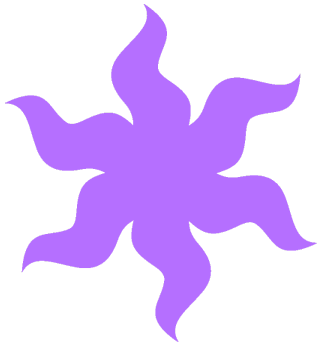

# Corona-X: A Modern Operating System

Designed By:
Tyler Besselman

Corona-X Overview
December 2016

# CONTENTS

# CHAPTER 1
# INTRODUCTION

## 1.1 INTRODUCTION

Corona-X is a next-generation Operating System (OS) which I have been designing for about a year. The system itself is of my own design, although I hope to meet people who may provide insight into the possible flaws and improvements which can be made to the system. This system is my third system design, and I have been designing systems for around three and a half years at this point. Corona-X is meant to support modern desktop and mobile platforms with 64-bit x86 and ARMv8 CPUs.

## 1.2 DESIGN PHILOSOPHY

The design philosophy used for Corona-X is reflective of my belief that most systems do a select group of things extremely well and another group of things not so well. Everything from the build system to the coding style to the functionality of the system as a whole has been designed using this philosophy.

Each component of the system has been designed by looking at implementations of similar components in both other Operating Systems and in research papers published on the topic. I attempt to isolate key features which make different components successful from both of these areas, and then synthesize a design which incorporates all of these features together into a single, seamless component design.

## 1.3 CURRENT STATUS

The current status of the project must be discussed with respect to two sub areas: the status of the system design and the status of the system codebase. This section will discuss both of these areas separately.

### 1.3.1 DESIGN STATUS

Currently, the majority of the kernel and privileged level of the system design has been completed enough to start coding. There is an experimental design for a new user-level API, but this has not reached a point which is ready to be implemented.

The kernel design, as it currently stands, is based in large part on two papers published at MIT during the 1990s. The first of these papers set forth a new kernel design known as an ‘exokernel,’ the goal of which was to revolutionize Operating System design by allowing applications much more direct access to hardware. The second of these papers extended the exokernel to function on Symmetric Multiprocessing systems, or machines with multiple processors or cores. The kernel design of Corona-X utilizes the same basic principles of the exokernel, but it also implements features detailed later in order to improve upon the failures and difficulties of the original exokernels.

The user-level system design, on the other hand, takes large parts of its design from other modern Operating Systems. It is an object-oriented, notification-based API. The object handling system is modeled after Objective-C and Swift from Mac OS X, but it also has principles taken from Windows’ C# Runtime. The handling of output displays between kernel and user level is of my own design. The user-level system design still needs a decent amount of design before it can be finalized.

### 1.3.2 CODE STATUS

The coding for the system was started in October of 2016. At this point, only the bootloader has been coded, as well as pieces of a kernel shared library. The codebase itself is influenced by many modern systems.

For instance, the system will utilize the framework-based shared library model as in Mac OS X and iOS. The repositories for the system code utilize the git version control system and are put together using Android’s ‘repo’ tool. The build system has not been entirely put together yet, but it will likely be a modification of Chromium’s gyp or gn tools building with ninja underneath.

The UEFI bootloader for x86_64 systems has been started. It is currently around 5,250 lines of code and runs on any UEFI enabled systems. The current thing which is being worked on is a full ACPI driver for the system. Once I have finished submitting applications to college and the only thing I have to worry about is school, I will begin programming the system kernel.

# CHAPTER 2
# ABOUT CORONA-X

## 2.1 PROJECT GOALS

The goals of the project are, in order of importance:

1. **SRP:** Achieve extreme **Security**, **Reliability**, and **Performance** through proper design and optimization of code.
2. **Functionality:** Guarantee that all components of the system function as intended. Additionally, guarantee that all components of the system function together to create a single, continuous experience.
3. **Documentation:** Provide entirely accurate documentation of both intended and actual functionality of every aspect of the System.
4. **Code Clarity:** Generate a system codebase which conforms to the project's given Coding Standards (External to this document). Ensure that code is clear in meaning and commented properly where necessary to enhance overall readability.
5. **Experiment:** Foster an environment in which risks can be taken; delve into unexplored areas of OS Development; change the world, and have fun with it!

## 2.2 JUSTIFICATION

Corona-X has been created for a few different reasons, chief of which is that designing an Operating System is the ultimate programming challenge. Designing and implementing a modern Operating System is no easy task, as there is virtually no area of computation which modern systems do not touch upon.

Besides the sheer challenge and fun of the project, I have noticed certain things in every OS I have ever used which I do not enjoy. With Linux, it's the lack of unity in the userland. With Windows, it's the proprietary nature of the system—this leads to the impossibility of validating whether the code is properly functioning. With Mac OS X it's the speed and again the proprietary codebase. With most BSD distributions, it's the necessity of building your own GUI. Corona-X attempts to address these problems and fulfill the goals set forth in the previous section of this document. Most modern systems have a lot of places where they can improve with respect to the goals of Corona-X.

# CHAPTER 3
# PRIVILEGED LEVEL

## 3.1 KERNEL DESIGN

As mentioned before, the kernel of Corona-X utilizes a lot of principles set forth in the MIT exokernel papers. These principles are modified in order to optimize for Security, Reliability, and Performance of the system proper. The modifications are taken from three different sources: another type of kernel design from CMU known as a microkernel, the original kernel design known as a monolithic kernel, as well as some optimizations of my own. The next three sections will address each of the three different kernel designs used to create the Corona-X kernel, as well as how their principles are utilized in Corona-X.

### 3.1.1 EXOKERNEL PRINCIPLES

The main exokernel principle is that the kernel proper simply ensures that hardware is securely accessed by applications. Exokernels do not expose any high level abstractions to user programs. Instead, they allow applications to define their own abstractions directly on the hardware of the system. These abstractions may be contained in so-called “library Operating Systems,” which can be used either as-is or modified to optimize for an application’s specific use cases.

The Corona-X kernel uses a modified version of this main exokernel principle. The modifications to this principle arise from two issues with it: it can be somewhat insecure without a lot of very complex programming and the vast majority of applications on a system will not likely need their own library Operating System. Because of this, the kernel of Corona-X implements applications as entirely unprivileged without any access to hardware, and it implements something like library OSes known as ‘servers’ in the same semi-privileged state as applications in a classical exokernel. Applications may then ‘link’ to a server which will provide abstractions optimized for their use case. This is the same as in a classical exokernel except with a thin layer of added security between applications and servers. If an application developer would like to have optimized abstractions for its use case and no proper server is currently running, the developer may choose to develop their own server to provide the abstractions they would like. This allows for extremely optimized applications running under the Corona-X kernel while still allowing most applications to just run under the standard userspace.

### 3.1.2 MICROKERNEL PRINCIPLES

The most notable microkernels are Mach, one of the original microkernels which was developed at CMU in the 1980s, and L4, the microkernel which proved that microkernels could operate as quickly as most other systems. The main idea of a microkernel is that the abstractions which are implemented in the kernel in most classical systems are instead implemented in userspace programs termed ‘servers.’ Each server in a microkernel system would be responsible for a different piece of the classical system. Servers in a microkernel may pass information between each other through so called message-passing, through which they call methods in other servers. The microkernel itself implements only the basic pieces required to support the servers in this scheme: Inter-Process Communication, Virtual Memory management, and process scheduling. This scheme is extremely reliable, for if one server of the many crashes, it may be restarted in order keep the system running without any major error.

The Corona-X kernel utilizes this principle in the servers which it runs. These servers act almost identically to microkernel servers, as they may communicate through message passing and remote procedure calls (directly calling procedures in other servers). Some principals discussed later help to optimize this server-based system for performance, but in its basic form, the Corona-X kernel implements microkernel servers for its exokernel library Operating System abstraction.

### 3.1.3 MONOLITHIC KERNEL PRINCIPLES

The monolithic kernel is the “classic” design for Operating System kernels. It is the oldest and most popular design for kernels. Virtually every UNIX system is a monolithic kernel based system. Monolithic kernels implement all Operating System abstractions inside the kernel itself. This is an extremely secure and generally pretty fast system, but if there is any fault in the kernel, the entire thing becomes extremely volatile and may crash (this is commonly known as a kernel “panic” in UNIX systems, or a “Blue screen of death” in windows).

This kernel design is least implemented in the Corona-X kernel. The only reason why it is utilized at all is for the performance and security which it can provide. The way through which principles of the monolithic kernel are utilized is that there are not nearly as many servers as in a classical microkernel. Instead, servers are separated based on where the majority of bugs may be found in classical systems. For instance, the large majority of bugs in most systems are found in the device drivers. For this reason, device drivers are

implemented in the Corona-X kernel separately from the rest of the system. Yet, servers which would be implemented separately in a microkernel, such as filesystems and user permissions, are combined in the Corona-X kernel.

### 3.1.4 CORONA-X PRINCIPLES

Corona-X also includes certain kernel principles which are unique. Without these key principles, the system would not be as optimized or as seamless as it is with them. The majority of these principles are very specific to programming the system and, as such, are out of the scope of this document.

One notable specific function which has been created specifically for Corona-X is called a temporary shared region. Temporary shared regions are areas of physical memory which may be shared between servers for the duration of the time they are communicating with each other. This allows for optimized message passing in a way that is impossible with microkernels (each server has a separate virtual memory space). This is just one example of the many optimizations which the Corona-X kernel has to allow the system to function quickly and efficiently without sacrificing security (only a specified region of memory is shared).

Another example of a Corona-X kernel-specific optimization is the implementation of message passing. Because the Corona-X kernel is more similar to an exokernel than a microkernel, message passing can be implemented entirely in absence of privilege checking or context switching, as both of those things can be implemented directly in servers as necessary.

The final example that will be given here of how Corona-X is optimized to satisfy the goals set forth for the project is the method through which new servers are started. In order to prevent allowing any code to arbitrarily get access to the hardware in the machine, all currently running servers must be prompted to see if they would support giving privilege to the new proposed server. Whether this means simply prompting the user to ensure that they are okay with allowing the server to run, parsing the binary code of the server for viruses, or verifying code signatures on the new server, it allows servers to be secured by all other servers currently running on the system. Therefore if even one server has a reason to believe that giving the new server hardware access would be detrimental to the system, access may be denied.

## 3.2 CORE SYSTEM LAYER

The main user interface and application layer in Corona-X is provided by the so-called “Core System Layer” (CSL). This layer provides a classic Operating System API with a modern twist. This layer is made of five separate servers in three categories. Each server was created with the idea of reducing any sort of domino effect spawned by bugs in a server. That is, the CSL is separated into groups based on where the majority of bugs in modern systems usually occur.

### 3.2.1 THE HARDWARE (IO) SERVERS

Two of the five servers in the CSL are designated hardware, or IO, servers. These two servers are responsible for device drivers for all of the hardware in the system. The reason why these servers are separate from the rest of the servers in the CSL is that device drivers are the components of systems which are the most prone to bugs. Separating these from other servers is vital to prevent those bugs from affecting any other parts of the system.

The first server includes drivers for things such as system power management, the system’s main interconnect, and the CPU itself. It also includes any interfaces to runtime services which may be provided by the system’s bootloader. This server is known as the core hardware server. If this server crashes, it won’t matter whether or not the rest of the system continues functioning properly. If there is a bug in one of these drivers, the system will crash regardless of where the code which includes the bug is physically located, as these drivers are integral to the proper function of the system.

The second server includes any and all other drivers for the rest of system hardware. This includes graphics chips, network interfaces, any ports on the system, keyboards, mice, virtual reality hardware, or anything else which may be connected to a system. This server is extendable, allowing for third party device drivers to be written and loaded into the system.

### 3.2.2 THE SYSTEM SERVERS

The third and fourth of the five servers in the CSL are designated the system servers. The first of these two servers is designated simply as *the* “SystemServer.” The second of the two server is designated as the “SystemUIServer.”

The SystemServer is in charge of the underlying abstractions which make the system unique. This includes a task and thread abstraction, a virtual memory management system, inter-task communication, a filesystem abstraction, cryptographic functionality, random number generators, and everything else included in a standard, modern system. All of these abstractions, in their current state, will be specified (to the high level scope of this document) in the chapter on the Core System Layer API (Chapter 4).

The SystemUIServer is the Window Server of Corona-X. This is one of the simplest CSL servers, as it simply in charge of outputting to the screen. This is the only server in the CSL which directly outputs to the screen; however, it does not handle any real abstractions. This server is directly messaged by applications using a temporary shared region in order to output information to the screen.

### 3.2.3 THE FAULT SERVER

The Fault Server is the final server in the CSL. It is by far the simplest server, as it is only responsible for printing and logging error messages on unrecoverable error.

## 3.3 CORONA-X AS A HYPERVISOR

Hypervisors and virtual machines are vital to the server hosting industry today. For instance, Windows Server 2016 offers a hypervisor named Hyper-V which allows the host to run multiple virtual machines on a single system. This practice is becoming extremely popular as the power of each individual system is becoming the so much more powerful than is necessary to run a single web server (or whatever type of server is attempting to be hosted).

Hypervisors are also vital to being able to run applications from different types of Operating Systems. For instance, running a Linux, Mac OS X, and a Windows system on a single computer through a hypervisor would allow for applications for any of those types of systems to be run simultaneously.

Corona-X functions in both of these ways. A single Corona-X kernel is able to run multiple instances of the CSL in order to create virtual machines without any of the overhead of the hypervisor scheme. Additionally, native servers which implement the API of a given system using the SystemServer API may be written in order to run native applications for a given system (again without the overhead of a virtual machine).

# CHAPTER 4
# CORE SYSTEM LAYER API

## 4.1 APPLICATION DEVELOPMENT UNDER THE CSL

In virtually every modern Operating System, the average user expects an abstraction known as an “application.” An application, in the user’s mind, is a program with a logo and a set of “windows” on the screen which perform a well-specified function. Because Operating Systems have all generally developed to function this way, the CSL does away with the much more classical view of applications as being extremely simple command-line tools running in a single thread of execution.

Corona-X’s SystemServer defines abstractions for applications based on groups of ‘tasks’ which each perform a single action. It also defines a method of creating a group of tasks and resources in a single file, known as a CAR Archive (the specification for this file format is external to this document). Each of these archives is the on-disk representation of a single application. It may include compression, be encrypted, and be digitally signed. The next sections will discuss certain abstractions provided by the CSL and move on to describe some high level abstractions used by the system’s application runtime.

## 4.2 CSL APPLICATION ABSTRACTION

Under the CSL, a CAR Archive holds a single application. Inside an application, there may be one or more tasks which may each include one or more threads. Each task then holds one or more threads. Each task has its own virtual address space and, by extension, its own program text, data, and stack. Each thread holds only a register state, but it shares the address space with its owning task. This system is not by any means revolutionary. In fact, it’s nearly identical to the system used in the XNU kernel. What is revolutionary is the elimination of tasks which are not within other applications and the method of grouping tasks further into applications.

When an application is launched, a set number of tasks is launched. The task system which is used is extremely reflective of the server system utilized in the microkernel system. Each task is meant to have a single function which it performs individually from every other task in the application. The application, then, is simply a group of tasks all performing separate but connected functions which presents an interface to the user based upon the application’s resources.

A note: Tasks within an application may have their own virtual memory map, but context switches may still be optimized and even entirely eliminated because of the exokernel-based design of the Corona-X kernel proper.

## 4.3 OBJECT RUNTIME

Object-Oriented programming has been standard in every API for years now. However, the core of many APIs are still written and implemented in a procedural style. The CSL throws this model out, implementing the entire API in an object-oriented style from the ground up. Every interface in CSL API will be exported in C, C++, and Objective-C simultaneously through the object runtime which it provides. Even in regular C, objects are implemented as pointers which can be passed to 'methods.' This unique runtime which CSL provides allows for the API to be exported in virtually any object-oriented language which can be written.

The runtime which is provided by the CSL provides an extremely abstract system of class registration which provides methods for registering new classes based on class maps; allocating, instantiating, and freeing objects and classes; and calling methods and accessing instance variables on objects and classes. This runtime is extremely flexible and allows for application development with the CSL API to occur in many different object-oriented languages.

## 4.4 NOTIFICATION SYSTEM

The application runtime of the CSL is based on a system of notifications. That is, different sources can spawn notifications and call specified handlers in tasks in the application in order to notify the application of changes of state. While an application is running, the main tasks generally should simply loop and persist the state. It is only when the user or system changes the state of the application through interaction that the application should actually process information. This is the reason why a notification-based processing system is perfect for the CSL: only when the user state changes is the application itself modified.

When applications are started, they are called with a map object which passes in some type of current system environment, application preferences, debugging information, and any other optional preferences. The application should then process any initial setup

based on this map, and enter its main looping section. Any time a notification is presented to the system, it is forwarded to the application for processing.

## 4.5 OUTPUTTING TO THE SCREEN

Outputting to the screen is done through the SystemUIServer. Applications simply request an area on the screen and then publish the data they would like placed into it whenever they would like the screen to be updated. This is very true to the exokernel principles in the system, as all screen processing is delegated to application-level libraries and may also be handled by the application itself if the application so chooses. This is one of the simplest pieces of the system, a fact which allows for extremely quick screen updates and, if well implemented, multi-threaded GPU access.

## 4.6 THE SYSTEM TREE

The System Tree is the name for how files are handled by the SystemServer. This tree handles a lot more than just files, however. In fact, every device and configuration value for the system is stored in this tree. It is in many ways a combination between the UNIX file abstraction, the BSD sysctl interface, and the Plan 9 filesystem interface. Each node in the tree has a name, children, and attributes which can be read and written.

The root node of the system is named '#'. Every other node on the system is, in some way, a child of this root node. One of the attributes on a node may the encoding to use for its children.

The system tree is meant to include a representation of every piece of data which would differ between two systems. That includes a list of physical hardware, disks, partitions, files on each partition, network interfaces, installed applications, screens, system configuration information, and anything else that can differ between systems.

The system tree has a lot of more complicated functions, including locking and jailing nodes on a system and methods of updating children based on changes in the parent. Each node may dictate which attributes are changeable, what encoding to use for the name, which children it exposes to where, and many other things. The system tree and the library to handle it resides in the SystemServer.

## 4.7 BACKWARD COMPATIBILITY

Backward compatibility is one area where this system can truly shine. The entire system API can be optimized into a single binary library file on disk. In order to keep applications backward compatible, the API file from the previous version simply needs to be included on the disk alongside the current version.

This allows any function in the API to be changed between versions however the designers decide. In order to save disk space in this scheme, the old API library can be optimized to only include those functions which have changed in the newer version. That way, the older versions will only be loaded when the old API needs to be loaded, but normally the new API can simply be loaded in full. This scheme also optimizes the speed of loading the entire API.

# CHAPTER 5
# CODE AND DOCUMENTATION

## 5.1 HOW TO ACCESS CODE AND DOCUMENTATION

The code and documentation for Corona-X can be found online. The code can be found at github.com here. These git repositories here are in google repo format. In order to clone them all, read the instructions in the main Corona-X repository.

Some documentation may be periodically uploaded to this folder on my website. Simply click the link to view the document in your browser. Updated versions of this document may be found here under the title “Corona-X Overview.”

# CHAPTER 6
# MOVING FORWARD

## 6.1 SHORT TERM GOALS

My short term goals for this project, in chronological order, are:

1. Finish implementing the ACPI driver in the bootloader and upload it to github.
2. Create a functioning (bootable) kernel and upload it to github.
3. Collect some basic statistics for system calls & context switches in the experimental kernel and compare them to some popular systems of today.
4. Write drivers for PCIe and CPU (Core IO Server drivers).
5. Find some other people who are interested in developing a new Operating System and get some help designing and implementing the system.

A lot of these can be done simultaneously. Number 5 will likely always be a goal of the project: I'm always looking for more help! College will probably be good for this, I'll likely meet a lot more people interesting in computer science.

## 6.2 LONG TERM GOALS

Tentative long term goals for the project are as follows:

1. Create a community around the system and its development.
2. Add support for the ARMv8 microprocessor architecture.
3. Implement the POSIX API and other System APIs
4. Create a centralized application store for the platform.
5. Add more frameworks and functionality to the system.