CALVERT HALL COLLEGE HIGH SCHOOL

SENIOR CAPSTONE PROJECT

MCMULLEN SCHOLARS PROGRAM

Kernel Security and Protection: An Investigation Into User Freedom Through a Toy Operating System

---

Author:
Jason WALTER

Mentor:
Mr. Christopher TADDIKEN

January 2021

Contents

1 Models of Security and Protection

1.1 Security and Protection Goals

Security, in the context of an operating system kernel, is the assurance that system resources may only be accessed by those who have the proper rights to them. Its specific goals and implementation are often varied, and the exact issues that it confronts have changed substantially throughout the history of computing. The concerns of security have shifted from merely separating users to incredibly elaborate mechanisms that ensure each user has proper access rights. (Tanenbaum 612). In this way, the goals of operating system security often coincide with the goals of system protection, which enforces rules for the use of resources in a computer system (Silberschatz 612). Both concepts involve restricting access to given resources, with key differences. Security is focused on privacy and integrity with regard to malicious actors, while protection is focused on maintaining a usable system by preventing errant resource access.

Contemporary approaches to operating system kernel-level security and protection mainly include methods of restricting access to certain system resources based on user identity. Such mechanisms function within a multi-user paradigm in which a user need not correspond to a human individual. These approaches to security can mainly be derived from the concepts of the reference monitor, capabilities, process abstraction, memory protection, and privilege separation in general.

1.2 Reference Monitors and Capabilities

The reference monitor concept is an effective model for analyzing and constructing a secure system. It "defines the necessary and sufficient requirements for access control in a secure operating system," and an operating system must fulfill three distinct criteria in order to satisfy the model (Jaeger 17). The operating system must first serve as the sole mediator for secure operations. This ensures that there is a standard mechanism for the validation of privileges. Secondly, the complete reference monitor system must not be vulnerable to any type of meddling or outside tampering. Lastly, the reference monitor system must maintain a small enough size that its working status can be easily verified, assuring that the system is secure. Essentially, the reference monitor system

must serve as an independent access controller, ensuring that a given user or program has the necessary privilege to access a given resource. All requests for resources or security-sensitive operations must be independently verified by the reference monitor system. An operating system that does not provide this utility cannot necessarily be considered secure, as it would be lacking a tamper-proof method for actually enforcing any security policies.

This concept meshes well with the design of capability-based systems. Such systems inform the design of nearly all operating system security mechanisms today, even if they do not strictly adhere to their requirements. The defining characteristics of capability-based systems are that they "provide (1) a single mechanism to address both primary and secondary memory, and (2) a single mechanism to address both hardware and software resources" (Levy 3). Capability-based systems use an object-based approach to access control, and as such may be seen as a higher abstraction of the reference monitor concept. This means that they build upon the principles of the reference monitor concept in order to provide a broad mechanism that achieves its goals. In a capability-based system, every process and user is granted a set of capabilities that define their access rights to system resources, such as a restricted area of memory or a certain subset of files. The reference monitor concept and capabilities together inform the contemporary approach to kernel security through such measures as access control for read, write, and execute privileges in a file system as well as for process memory protection.

1.3 The Process Abstraction

In order to adequately implement either a reference monitor system or a capability-based system, it is necessary to keep track of each independent program running on a given computer system. The process abstraction is necessary to any comprehensive operating system as it allows for a distinct separation between different tasks being performed by the computer's hardware. Indeed, "[t]he most central concept in any operating system is the process: an abstraction of a running program" (Tanenbaum 83). An operating system is able to determine the resources that have been allocated to individual tasks using the process abstraction, a job which is integral to the implementation of both security and protection. Perhaps the most important function of security and protection related to the process abstraction is memory protection, which isolates the memory resources of each process. This mechanism prevents a process or user from reading or writing to memory that does not belong to them. This boosts security by preventing meddling in running programs or the reading of private data, and it facilitates protection by preventing processes from mistakenly overwriting another process' memory, which could result in catastrophic consequences for a system in run-time. Without

memory protection, data being stored by important system programs such as hardware device drivers could be randomly overwritten by a malfunctioning program, crippling the operating system. However, memory protection greatly restricts the ability of programs to work together by isolating processes completely. This separation then necessitates the implementation of message-passing systems, such as signals, pipelines, or shared memory spaces, independent of each process' allocated resources. Such solutions complicate the process of programming.

1.4 Separation of Privilege

Mechanisms such as memory protection accomplish a separation of privilege, with different users and processes maintaining their own domains of control. The concept of privilege separation is further explored in the construction of the kernel itself. The two dominating kernel paradigms are the monolithic kernel and the microkernel, each of which carries its own repercussions for security and protection, and for the ways in which privilege is separated. For monolithic kernels, all of the resources for an operating system are built into the kernel itself, leaving none of the responsibility for managing system resources to programs written and used by users, known as userspace programs. This hinders the ability of the kernel to manage privilege separation, as the kernel itself maintains full control over the entire system. In contrast, microkernel development seeks to minimize the size of the kernel as much as possible, limiting it only to necessary tasks such as the management of hardware drivers or setting up a basic system environment. Everything else is relegated to userspace, which allows for a more granular privilege separation. This in itself is a form of protection, as it prevents total system failure in the case of protection failing. If a monolithic kernel were to encounter a critical failure, the entire system would crash. A microkernel, on the other hand, would be able to isolate the failure and prevent a total crash because privileges would be strictly separated.

1.5 Internet Connectivity and Security

The role of increased internet connectivity in the development of security mechanisms must be noted in reference to contemporary operating system kernel security and protection. Although computer systems built before the advent of the internet almost never had to consider the repercussions of connecting to other systems, such connectivity is ubiquitous today; there are projected to be 34.2 billion global connected devices by 2025 (Congressional Research Service 2). Such connectivity, in combination with the multi-user paradigm utilized by most contemporary operating systems, has led to a

convoluted approach to security and protection which accounts for many external variables. For an internet-connected device, a "typical attack pattern consists of gaining access to a user's account, gaining privileged access, and using the victim's system as a launch platform for attacks on other sites" (Longstaff 5). This means that kernels must account for the privileges and capabilities of different users in order to prevent the escalation of privileges upon a system breach, which exacerbates the limits already imposed by protection mechanisms.

1.6 Conclusion

In general, security and protection measures in operating system kernels work to define who can access which resources and to prevent harmful access to those resources. Such measures are essential to maintaining a functional system in many contexts, especially when a computer is meant to be accessed by multiple users or when a computer carries sensitive information. As such, these measures are incredibly useful to businesses, governments, and any computer user who wants to keep their data safe and secure without having to pay any attention themselves. However, the implementation of such security and protection measures necessarily adds overhead to an operating system kernel and may negatively impact the ability of a user to use their whole computer system freely and effectively. The development of an operating system kernel should be informed by the security and protection measures necessary to fulfill its purpose.

2 Personal Computing Freedom

The design of the Capstone operating system is informed by the maximization of "personal computing freedom," which is defined as the ability of a user to interact with and modify all parts of a given computer system. This definition is derived from the characteristics of free software as suggested by the Free Software Foundation, namely: "the freedom to run . . . , to study and change . . . , and to redistribute copies [of the software]" (Stallman). Given that personal computing freedom necessitates the evaluation of a system in its entirety, it is necessary for most or all operating system security and protection measures to be absent in order for the freedom to be maximized. Such measures prevent users from accessing certain parts of their own systems, and only by removing them entirely may a computer system acquire personal computing freedom. Although such a proposition may at first seem radical and, admittedly, could easily result in a system completely useless to most possible users, the groundwork for such a system has already been established. This groundwork can be traced far back into the history of computing and owes itself to the free software movement as well as to the work of the communities surrounding free and open source software projects themselves (Free Software Foundation). Personal computing freedom also refers to the scope in which a machine is used. A computer used in a personal capacity must provide multimedia amenities to users in order to fulfill its purpose.

It is important to note that the principle of personal computing freedom extends not only to the interface between the system and the user, but also to the formatting and organization of the system's source code itself. For this reason, the principle of personal computing freedom may be related to such standards as the Linux kernel coding style, which, when editing code, "leads to ease of reading, lack of confusion, and further expectations that code will continue to follow a given style" (Love 396). Only a consistent coding style can permit a truly free personal computer, as muddled and convoluted source code makes it unduly difficult for a user to read and edit aspects of their system. The ability to read and understand a given system's source code also paradoxically increases the trust in and security of the system.

Despite the inherent contradiction in this claim, a truly free personal computer

would experience a peculiar increase in its security by eschewing security and protection measures. It would enable the user to trust and keep track of both the system's source code and its whole runtime environment. As such, a system that achieves personal computing freedom would address that famous observation from Ken Thompson's *Reflections on Trusting Trust*; namely, that you "can't trust code that you did not totally create yourself" (763). Forcing a user to run programs with unclear origins is a practice diametrically opposed to personal computing freedom. Informed by this, a truly free personal computer would avoid subversion by maximizing transparency, as to "use internal mechanisms within a computer system to protect sensitive information without demonstrable assurances as to the origins and effectiveness of the system components is contrary to a sound security practice" (Myers 10). In this way, although the basic and physical mechanisms of computer security may be abandoned, a truly free personal computer would involve a type of security not easily found in any contemporary commodity operating system. It would inspire trust and motivate its users to investigate its safety by rendering its source code accessible and well-documented.

In the event that such a free system were created, it would provide niche utility to the subset of computer users who desire complete control over every aspect of their system. Current commodity operating systems impose security or protection measures, or otherwise have convoluted internals that complicate the process of understanding and editing them. Such a system would also serve as an excellent teaching tool. It would allow a student to learn easily about how operating system software interacts with computer hardware, as there would be no limitations as to how the user might facilitate and meddle in those interactions. Most importantly, such a system would foster a great community, much in the style of the Linux kernel or other large free and open source projects. No matter the size of the community, the ensured freedom of the system would allow for quality modifications and updates that the community as a whole would be able to read and understand.

The Capstone operating system utilizes the principle of personal computing freedom in order to be such a system. It borrows from the free software aspects of contemporary commodity and research operating systems. Building upon this base provided by the Free Software Foundation and similar movements, it provides a computing experience that allows a single user to interact with and modify every part of the system by eschewing security and protection mechanisms in favor of a more permissive design, even if that design is unsafe. The Capstone operating system as such provides a transparent interface through which a user may perceive the workings of their computer as well as the foundation for teaching and a software development community.

3 Security, Protection, and Personal Computing Freedom

3.1 Security and Protection Reduce Freedom

The implementation of security and protection measures fundamentally reduces personal computing freedom, as the function of security and protection measures is to mediate access to resources that honest users could reasonably want to use (Gasser 9). An operating system kernel will therefore become more impossible to use as it employs more preventative measures that may frustrate users (Peterson 62). Advanced users are the most impacted by this relationship, as general users will usually have less need to circumvent the inconveniences caused by security and protection. It is therefore necessary to avoid the implementation of security and protection measures when developing an operating system kernel with personal computing freedom in mind.

3.2 The Reference Monitor Concept

Although it is useful for determining the security of a given computer system, the reference monitor concept is fundamentally opposed to personal computing freedom because it requires some part of the operating system to be tamper-proof. Rendering any portion of a computer system tamper-proof is the most extreme protection measure, and a total prevention of access and modification is not compatible with the principle of personal computing freedom. However, this classification only applies to software systems. Hardware systems are generally tamper-proof by nature and should pursue protection whenever possible because they are immutable and the lack of protection could render a hardware system useless. Software systems, on the other hand, can be easily modified in order to achieve personal computing freedom. Capability-based systems, by extension, also violate the principle of personal computing freedom. Although providing a standard system to access resources is not inherently restrictive, introducing access rights limits the ability of a user to fully interact with their machine.

3.3 Memory Protection

Memory protection, likewise, prevents a user from completely accessing their machine. By fencing off areas of memory through software controls in order to protect processes from errant memory access, memory protection privileges the machine itself over the user controlling the machine. In this way, strictly enforced memory segmentation is contrary to the principle of personal computing freedom. Although there is definite utility in crafting strong mechanisms for preventing system failure, it is not possible to prevent users from reading or writing to certain portions of memory without sacrificing freedom. The drawbacks of allowing complete access to every process' memory space are easily alleviated by the practice of good programming, which prevents accidental changes to important memory regions. However, methods of memory protection could remain compatible with the principle of personal computing freedom if they were only designed to protect processes from other processes. In this manner, users could elect to override the protection in their own programs while still maintaining effective protection against critical system failure.

3.4 Privilege Separation

Just as with memory protection, the separation of privilege does not automatically lead to a reduction in personal computing freedom. For this reason, there is no inherent difference between the freedom provided by a monolithic kernel or a microkernel, although they necessarily differ in the implementation of access to the computer's resources. In theory, a microkernel offers greater possibilities in personal computing freedom because it relegates many essential services to userspace. This allows the user to control their computer with greater granularity in a way that is expressly sanctioned by the kernel. A monolithic kernel, conversely, offers a simpler interface for accessing system resources and interacting with hardware because it consists of just one runtime. Laudable personal computing freedom can be achieved using both designs or even completely different designs, although each design will achieve it in different ways.

3.5 The Multi-User Paradigm

Finally, a multi-user approach to operating system design reduces personal computing freedom drastically. Dividing a computer system into distinct resource areas based on various users' access rights reduces the ability of any given user to fully interact with the computer system and adds needless overhead to operating system kernel. Multi-user design discards the needs of a single user in favor of maximizing the ability of several users to work together, ignoring the ideal of a personal computer in

the process. Separating the resources of users also entails running code to ensure that each user's files, processes, and memory are partitioned according to some predefined scheme at any given time, reducing the efficiency of the computer system. The security concerns that come with preventing privilege escalation between users also contribute to needlessly convoluted design in multi-user systems, which further reduces personal computing freedom.

3.6 Conclusion

This analysis indicates that an operating system kernel that conforms to the principle of personal computing freedom cannot implement security and protection measures. Although the hardware that the kernel runs on might implement these measures, further security and protection mechanisms only serve to impede the user. Memory protection is the most egregious violator of the principle of personal computing freedom, as it works in tandem with the multi-user paradigm to partition the running system into distinct resource areas that a single user may not interact with or edit. An operating system kernel that is meant to conform to the principle of personal computing freedom should be developed with care. Any rule that prevents the user from accessing or modifying any part of the system is in violation of the principle.

4 Security, Protection, and Freedom in Practice

4.1 Introduction

The security and protection paradigms introduced by various actual operating systems have varied effects on personal computing freedom, with commodity operating systems in particular having notably poor freedom. The biggest factors influencing this are the multi-user paradigm, closed source code, and convoluted design, in addition to a general disregard for ease of use. However, it is also necessary to investigate the methods of experimental or research operating systems in order to understand the frontiers in operating system kernel security and protection in relation to personal computing freedom.

4.2 UNIX

The UNIX operating system has had an immense impact on the design of operating systems in general, and stood at the forefront of multi-user systems (Bach 3-4). Many of its kernel design decisions have directly impacted or carried over to the design decisions of the Linux kernel, which is one of the most important and influential pieces of software today (Love 3-4). The UNIX design philosophy, which emphasizes a style of coding that facilitates personal computing freedom, also remains prevalent in the free and open source software community. However, it is necessary to investigate how the specific features of the UNIX operating system negatively impact this freedom.

The multi-user paradigm of the UNIX operating system has a disastrous effect on the freedom of the system as a whole. Personal computing freedom necessarily dictates a focus on a single user who has complete control over the system. Although a system built around multi-user interactions offers great utility in many circumstances, it is directly opposed to the principle of personal computing freedom. It requires the partitioning of the system into resource areas only accessible by certain users, which may not even correspond to the actual human users of the computer system. The convoluted system of permissions that necessarily follows greatly restricts the ability of any given

user to interact with the computer's software and hardware in any meaningful or complete way, as most functions and processes will be rendered inaccessible to them. Even a "superuser" with complete access to the system must operate within the boundaries imposed by a multi-user structure, and introduces another layer of complexity to the design and security of the system.

4.3 Windows 7

The Windows 7 operating system dominated the desktop operating system market before the introduction of its successors (Silberschatz 693). Although it was by far the most popular of the commodity operating systems targeted towards single users in its time, it failed to provide any semblance of personal computing freedom, largely as the result of its proprietary and closed-source code base and convoluted design. These features were the natural result of the operating system's status as a commodity marketed by the Microsoft Corporation and demonstrate how the commodity form is harmful for users when applied to foundational software like an operating system. Closed source code is fundamentally opposed to the principle of personal computing freedom because it prevents the user from understanding or modifying their system. However, the commodity form necessitates closed source code because the Microsoft Corporation cannot make money if a team of good-natured tinkerers can access their source code and distribute a modified and free version of Windows. This principle of profit is also the force behind the convoluted design of the Windows 7 operating system. In order to protect access to information about Windows internals, the operating system has purposefully undocumented functions that can be found by users, but have unknown properties (Russinovich 66). The operating system is designed from the bottom up to be hostile to any users.

4.4 Hydra, SPIN, and Xok

The Hydra, SPIN, and Xok operating systems are all systems designed for the purposes of experimentation or research. Each of them takes a unique approach to kernel design with distinct effects on personal computing freedom, and can be used as excellent reference points to inform the design of a more free operating system. Hydra emphasizes the separation of software policy and mechanism, while the SPIN and Xok systems are designed to grant the power to modify the kernel's extension infrastructure to userspace applications in order to improve modularity. A synthesis of these approaches can greatly improve personal computing freedom.

The main goal of the Hydra operating system is to definitively separate policy and mechanism. This approach, pioneered by the Hydra system, has since been adapted more generally to such software as Linux device drivers, which provide an interface between the Linux operating system and peripheral hardware devices. To this end, "[a]n important goal of the Hydra system is to enable the construction of operating system facilities as normal user programs" (Levin 132). This aim grants near-total control over the machine to the user by providing to them the ability to define how the operating system interacts with hardware. To clarify the distinction between policy and mechanism, "[p]olicies are (by definition) encoded in user-level software which is external to, but communicates with, the kernel ... Mechanisms are provided in the kernel to implement these policies" (Levin 132). This separation of policy and mechanism renders the kernel protection mechanism for the Hydra system "a passive one," which is of great interest to the endeavor of improving personal computing freedom while also maintaining a stable system that does not crash itself (Levin 139).

The SPIN operating system takes a very similar approach to this issue, with a greater emphasis on directly swapping out extension infrastructure for user-defined services. In particular, "SPIN provides an extension infrastructure, together with a core set of extensible services, that allow applications to safely change the operating system's interface and implementation" (Bershad 267). This extension infrastructure is a framework that allows a program to modify the operating system to maximize efficiency for particular tasks. In much the same way as the Hydra system, the SPIN system allows users to define kernel-level functions for allocating system resources. Once loaded, these functions "integrate themselves into the existing infrastructure and provide system services specific to the applications that require them" (Bershad 268). In much the same way as the Hydra system, this arrangement allows for a great amount of personal computing freedom by enabling the user to control how the kernel interacts with the hardware, and not forcing the user to abide by whatever security and protection measures the kernel has in place.

The Xok operating system builds upon the foundation of the Hydra and SPIN systems by operating as an exokernel, which aims to "give as much safe, efficient control over system resources as possible" (Engler 26). This kernel structure, as a departure from traditional monolithic kernel and microkernel styles, allows for perhaps the greatest possible conformance to the principle of personal computing freedom by addressing the most serious problem of older kernel styles, namely, that "only privileged servers and the kernel can manage system resources" (Engler 13). The Xok system and the exokernel architecture address this issue "by giving untrusted application code as much safe control over resources as possible, thereby allowing orders of magnitude more programmers to innovate and use innovations, without compromising system integrity"

(Engler 13). This system as such gives the user direct access to the relationship between their computer's hardware and the operating system software, maintaining a passive protection mechanism in much the same way as the Hydra system. The Capstone operating system uses the ideals of the Xok system as a guide, despite being limited to a monolithic, statically-linked kernel.

4.5 The Temple Operating System

Departing from the realm of research operating systems, the Temple operating system in many ways aims to act as a truly personal computer, meant to be operated by only a single user. Although its development has been halted due to the untimely demise of its sole developer, it achieves mostly positive results in conforming to the principle of personal computing freedom. According to its developer, "[t]he two most sacred and defining features of TempleOS are being ring-0-only and being identity-mapped" (Davis). This means that there is no separation of privilege between kernel space and userspace, which increases freedom by eliminating that restrictive method of privilege and resource separation. The identity-mapping of memory refers to the virtual addressing of memory in the operating system being the same as the physical addressing of memory in hardware, which allows for greater freedom by simplifying the representation of hardware in software.

The Temple operating system functions as notably free to the user as the result of its ring-0 and identity-mapped characteristics, but its own unique programming language and convoluted development goals detract from its accessibility and utility as a truly free personal computing system. Users are more able to control their system by having greater and more direct access to hardware, without the limitations of privilege separation. Both of these features are bolstered by the operating system's focus on a single user, allowing only one user to have complete access to their computer in its entirety.

However, the Temple operating system fails to be completely free because of its convoluted and nonsensical design goals. Its own unique programming language, Holy C, excludes programmers who do not spend the time to learn it, obfuscating how the operating system functions. This confusion is compounded by the system's focus on being a divine instrument to facilitate communication between a user and God. As such, the Temple operating system fails to function as a system that conforms to the principle of personal computing freedom because it is not designed with ease of control in mind.

5 The Capstone Operating System

5.1 Goals of The Capstone Operating System

The Capstone operating system facilitates the creation of a completely free personal computer, building upon the legacy of systems before it and learning from their mistakes. It provides complete and transparent access to the hardware through its kernel design and prioritizes the freedom of a single user. As a result of this, there are no security or protection measures in place. The kernel code follows the standards for the Linux kernel coding style and is well documented. This makes the code easy to read and modify, in line with the principle of personal computing freedom. In addition, the Capstone operating system is licensed under the GNU GPL Version 3, rendering it free software. Any user may edit and redistribute it so long as they maintain its licensing.

The scope of the Capstone operating system is limited because of the relatively short timeframe of the Capstone project as well as the inexperience of the developer. For this reason, some operating system features that tend to be ubiquitous, such as a file system or graphical windows, have not been integrated into the current build. The focus of the Capstone operating system is primarily to demonstrate the minimum possible computer system that conforms to the principle of personal computing freedom.

5.2 Kernel Design

The Capstone operating system kernel is monolithic, with all routines being managed within a single runtime. Although such a design complicates the internals of an operating system as it grows larger, it allows the relatively meager source code of the Capstone operating system to be simplified and managed more easily. This design also encourages users to experiment with tweaking the source code or directly writing their own routines to expand the functionality of the operating system, which is an aid to personal computing freedom. A microkernel design would be more conducive to personal computing freedom given a project of a larger scope, but a monolithic design is more than sufficient for the size of the Capstone operating system kernel.

5.3 Architecture and Accessibility

In order to maximize accessibility, the Capstone operating system is written entirely in C and Intel x86 NASM assembly. The C programming language is favored in every case because it is more readable and commonly known, but some portions of the operating system, such as the initial boot sequence and functions to interact with data ports, must be written in assembly. This design choice improves upon the standard set by the Temple operating system, which achieved a great deal of personal computing freedom but was largely inaccessible as a result of its unique programming language, Holy C. Such an approach also allows the Capstone operating system to borrow from the portability of UNIX, building upon the aspects of the UNIX design that offer the most personal computing freedom.

The Intel x86 platform was chosen for development because it is well-documented and boasts complete legacy support on many systems. This maximizes the portability of the Capstone operating system and allows development to focus on only one hardware paradigm, rather than diluting the focus of the project by including additional hardware support. In addition, the Intel platform integrates incredibly well with generic hardware, which speeds up development time and allows for testing on a greater range of devices.

5.4 Memory Representation

Memory in the Capstone operating system is organized into a single address space, rather than distinct memory segments. To borrow from the Intel 80386 Programmer's Reference Manual, this means that "the applications programmer sees a single array of up to 232 bytes (4 gigabytes)" (23). This flat model of memory organization is in contrast to the segmented model, which utilizes unique features of the Intel processor in order to create isolated memory segments each used for their own purpose. The flat model allows developers and users alike to have greater freedom in deciding how memory is organized on the system and makes it easier for a user to view and arbitrarily modify generic memory. Such a model also allows the Capstone operating system to borrow the concept of identity-mapping from the Temple operating system, ensuring that the representation of memory seen by the users of the Capstone operating system is as close as possible to the physical memory actually being used on the memory hardware. Although a segmented memory model offers the ability to use a 64 terabyte logical address space rather than a 4 gigabyte logical and physical address space as well as the ability to associate individual processes with distinct segments and switch more easily between them for multi-tasking, these benefits are overshadowed by the complicated structure

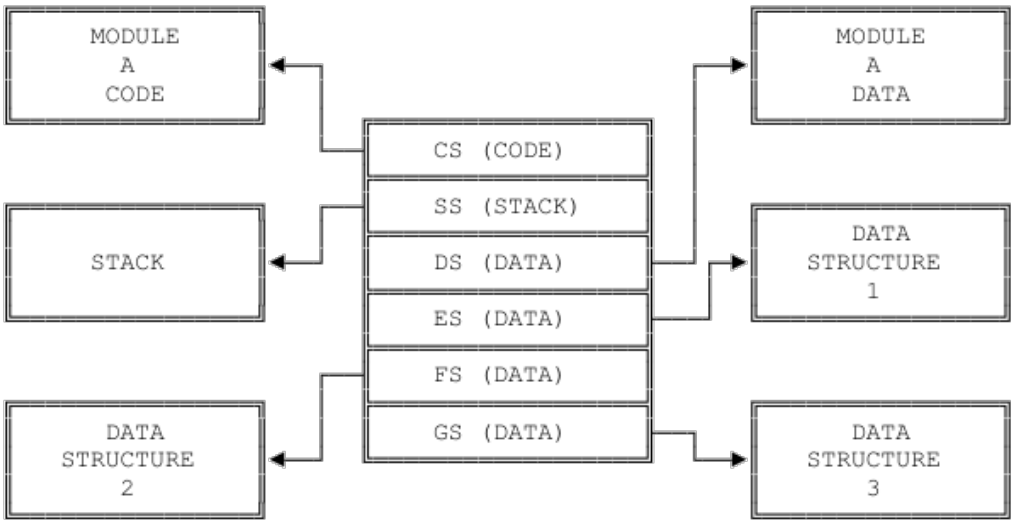

FIGURE 5.1: *Intel 80386 Programmer's Reference Manual*, page 32

involved with segmentation. The representation of memory that it entails is not conducive to the principle of personal computing freedom. It can be seen from Figure 5.1, which demonstrates the use of memory segmentation, that the discrete partitioning of code and multiple sets of data complicates the user's ability to easily access memory.

5.5 User Accommodations

Users of the Capstone operating system are able to access a universal function key, which is arbitrarily populated before compile-time. This key allows a user to assign a unique user-function name to any function within the operating system's source code so that it can be referenced and called from the operating system's terminal. These functions in general act as the normal terminal apps that can be found on any system. They take a list of arguments from the terminal along with an argument count. However, this framework is powerful because it allows a user to easily modify and reuse any system function as a user-function, which opens the entire operating system up as an opportunity for development and interactive use. As an example, a user could take the function for reallocating memory to a given pointer, crealloc(), and apply a simple edit to allow it to take string arguments before converting them to integers. This would provide to the user the ability to resize any allocated block of memory in use from the command line, which is a power conducive to the principle of personal computing freedom. Functions that require no arguments, such as t_clear(), the function that clears the screen, can be utilized as a user-function with no modification. The fexec() function is called from the terminal to execute user-functions, and is defined as such:

```
void fexec(void *func, int argc, char **args)
{
    void (*pfunc)(int, char **) = func;
    pfunc(argc, args);
    return;
}
```

In addition to these system features, the Capstone operating system also has basic multimedia functionality. By enabling timer interrupts, the operating system is able to use the PC motherboard beep speaker in order to build a music library. This library supports polyphony up to three notes and total rhythmic freedom. The current build of the Capstone operating system comes bundled with music functions for "Happy Birthday" and "Megalovania" by Toby Fox. Multimedia functionality is important for personal computing freedom because it defines a personal computer. Most personal uses of a computer rely on multimedia, and the multimedia capabilities of the Capstone operating system provide a sound base for expansion.

5.6 Dynamic Memory Allocation

The Capstone operating system allocates memory dynamically by maintaining a linked list of free regions of memory. When a program requests for memory to be allocated, that list of free regions is searched for a region that is the same as the requested size. If this search fails, the first larger memory region is split and a region of the requested size is created. Memory regions that are freed are attached to the very beginning of the list. There is no memory protection beyond marking regions of memory used by the processor as perpetually allocated. This memory allocation scheme causes a unique form of memory fragmentation as the operating system is used. Normal memory fragmentation occurs when there is enough free memory to satisfy a request, but allocated memory separates free regions of memory and prevents them from being used. In contrast, this scheme causes fragmentation in the form of free memory regions that are actually located next to each other reducing into smaller and smaller fragments, such that there eventually might not be a large enough region to satisfy a request. However, Such an issue is easily fixed by trivial defragmentation as exemplified through this function:

```
void defragment(void)
{
    struct free_hop *p = free_origin.fw;
    struct free_hop *cmp = p;

    /* search for adjacent hops, then collapse them together */
    while (p != NULL) {
        while (cmp->fw != NULL) {
            if (cmp->fw == (struct free_hop *)((uint8_t *)p + p->size)) {
```

```
10                  p->size = (p->size + cmp->fw->size);
11                  cmp->fw = cmp->fw->fw;
12              }
13
14              cmp = cmp->fw;
15          }
16
17          p = p->fw;
18          cmp = &free_origin;
19      }
20
21      return;
22  }
23  }
```

The Capstone operating system's first memory allocation scheme segmented all memory into a series of 256-byte chunks and was only able to allocate adjacent series of entire chunks. This scheme was incredibly inefficient both in conservatively utilizing the scarce memory resource and in quickly handling allocations. It was necessary to maintain an array of chunk headers at the very beginning of free memory in order to track allocated memory. Memory could be allocated more efficiently by reducing the standard chunk size, but this would lead to a longer array of headers at the beginning of free memory, reducing the amount of memory available to the machine. Similarly, a greater number of total chunks would lead to more time spent searching for free memory. This scheme allowed the Capstone operating system to initially implement features requiring memory allocation, but it was also necessary to be shed.

The current memory allocation scheme improves greatly upon this design. It allows for the efficient allocation of resources as well as for quickly finding free memory regions to allocate. It should be noted that this way of allocating memory is more complex than the original, and may not be entirely accessible to new programmers. The doubly linked lists and pointer arithmetic utilized can be confusing to inexperienced users. Developers pursuing operating system design should explore multiple different memory allocation schemes in order to realize the benefits and drawbacks of each.

5.7 Conclusions

The Capstone operating system, as a minimal example, embodies and justifies the principle of personal computing freedom. Its open source code is written in an approachable and readable style and allows users to view and edit the functionality of their system. This code is well-documented and accessible to new programmers, which

allows the operating system to serve as a teaching tool for operating system development. By extension, the code is capable of fostering a robust development community through its openness to outsiders. Its flat memory model provides an approachable memory structure and allows users to survey the entire system runtime without restraint and to define their own methods of memory organization with ease. Despite its small size, the Capstone operating system demonstrates the potential to be a free operating system alternative given a continued development cycle.

The limitations of the Capstone operating system include its lack of a file system, its lack of a clear process structure and multi-processing, and its restricted modularity. Without a proper file system, all programs must be compiled into the kernel before it is loaded onto hardware, which makes it impossible to edit any part of the operating system's code during runtime. This also prevents a user from storing data, which limits the possibilities of the Capstone operating system for personal computing. The lack of a process structure also means that the Capstone operating system is necessarily single-tasking, only able to run a single program at a time. Although such a design is workable for general computing tasks, it is generally favorable for the user to be able to run a program in the background or switch between multiple programs running at once. The Capstone operating system is not modular in that there are not easy replacements for different system functions. Currently, the user is only able to choose a single scheme for organizing and allocating memory. Any user is certainly free to write their own scheme, but it is necessary for the Capstone operating system to provide options in order for it to be truly free.

A more generic limitation of the Capstone operating system is its lack of a graphical mode. The operating system currently operates in text mode, making use of the VGA text mode graphics found in older BIOS computers. Newer systems, however, have abandoned support for this mode. As such, the Capstone operating system is unable to boot on newer hardware, despite otherwise providing generic Intel hardware support. The addition of a graphical mode will allow the Capstone operating system to boot on most Intel machines from the past three decades, and will also provide new possibilities for programming the operating system.

The Capstone operating system is a promising introduction to the possibilities of emphasizing personal computing freedom. An operating system that conforms to the principle of personal computing freedom is certainly possible. Indeed, many contemporary commodity operating systems do so in one way or another, and research and hobby operating systems have further demonstrated the feasibility of such an endeavor. By eschewing security and protection measures, the Capstone operating system demonstrates the potential to conform to the principle of personal computing freedom and

synthesize all the most relevant aspects of the UNIX, Xok, Hydra, and Temple operating systems.

A Capstone Operating System Code

The Capstone operating system is a living operating system still under development at the time of writing. The complete source code of the Capstone operating systemm can be found at https://www.github.com/walterj21/capos. Note that the Capstone operating system is free and open source software licensed under the GNU General Public License version 3.

Kustaa Nyholm's tiny format strings implementation is used throughout the source code of the operating system. The arrangement of Megalovania by Toby Fox for the PC motherboard speaker is licensed under CC BY-NC-SA 3.0.

Bibliography

Bach, Maurice J. *The Design of the Unix Operating System*. Prentice Hall / Bell Telephone Laboratories, 1986.

Bershad, Brian N., et al. "Extensibility, Safety, and Performance in the SPIN Operating System". *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Colorado) (1995): 267–284. https://doi.org/10.1145/224056.224077.

Davis, Terry. *The Temple Operating System*. The Internet Archive, archived 3 September 2013. https://web.archive.org/web/20130703141455/http://www.templeos.org:80/.

Engler, Dawson R. *The Exokernel Operating System Architecture*. Doctoral thesis, Massachusetts Institute of Technology, 1998. https://hdl.handle.net/1721.1/16713.

GNU General Public License. Free Software Foundation, 2007. https://www.gnu.org/licenses/gpl-3.0.html.

Gasser, Morrie. *Building a Secure Computer System*. Van Nostrand Reinhold, 1988.

Intel 80386 Programmer's Reference Manual 1986. Intel Corporation, 1987. https://css.csail.mit.edu/6.858/2014/readings/i386.pdf.

The Internet of Things (IoT): An Overview. Congressional Research Service, 2020. https://crsreports.congress.gov/product/pdf/IF/IF11239.

Irvine, Cynthia E. *The Reference Monitor Concept as a Unifying Principle in Computer Security Education*. Naval Postgraduate School, 1999. https://hdl.handle.net/10945/7200.

Jaeger, Trent. *Operating System Security*. Morgan & Claypool, 2008.

Kemerlis, Vasileios. *Protecting Commodity Operating Systems through Strong Kernel Isolation*. Doctoral thesis, Columbia University, 2015. https://doi.org/10.7916/D89C6WZ6.

Levin, R., et al. "Policy / Mechanism Separation in Hydra". *ACM SIGOPS Operating Systems Review* vol. 9, no. 5 (1975): 132–140. https://doi.org/10.1145/1067629.806531.

Levy, Henry. *Capability-Based Computer Systems*. Digital Press, 1984. https://www.homes.cs.washington.edu/~levy/capabook/.

Longstaff, Ellis, et al. *Security of the Internet*. Carnegie Mellon University, 2017. https://resources.sei.cmu.edu/asset_files/SpecialReport/1996_003_001_496597.pdf.

Love, Robert. *Linux Kernel Development*. 3rd ed. Addison Wesley, 2010.

Myers, Philip. *Subversion: The Neglected Aspect of Computer Security*. Master's thesis, Naval Postgraduate School, 1980. https://www.hsdl.org/?view&did=440858.

Peterson, A. P. "Counteracting Viruses in an MS-DOS Environment". *Information Systems Security* vol. 1, no. 1 (1992): 58–65. https://doi.org/10.1080/19393559208551318.

Roch, Benjamin. *Monolithic Kernel vs. Microkernel*. Vienna University of Technology, 2004. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.89.9877&rep=rep1&type=pdf.

Russinovich, Mark, et al. *Windows Internals Part 1*. 6th ed. Microsoft Press, 2012.

Silberschatz, Abraham, et al. *Operating System Concepts*. 9th ed. Wiley, 2014.

Stallman, Richard. "Why Open Source Misses the Point of Free Software". Visited on 08/22/2020. https://www.gnu.org/philosophy/open-source-misses-the-point.html.

Tanenbaum, Andrew S. *Modern Operating Systems*. 3rd ed. Pearson Prentice Hall, 2008.

Thompson, Ken. "Reflections on Trusting Trust". *Communications of the ACM* vol. 27, no. 8 (1984): 761–763. https://doi.org/10.1145/358198.358210.