REPORT.

AndromacheOS:

An Exo-Monolithic Kernel Hybrid Design with Modularity

CSE233

Mahros M. El-Qabasy[1], Mostafa M. Bakr[2], Sherif M. Haredy[3], Youssef B. Kaiser[4]

Galala University, Egypt

Prof. Amr Hefny

—

December 26, 2025

I.D.: [1]223106831
I.D.: [2]223100891
I.D.: [3]223107334
I.D.: [4]223102042

ABSTRACT.

Contents

1 Introduction

2 Booting the Kernel

So God created mankind in his own image,
in the image of God he created them;
male and female he created them.

— Genesis 1:27.

2.1 ELF Header

We first have to create our own kernel image. The executable is in an ELF format; this means it has to follow that header format, along with a `mutliboot2`-compliant header. This can be found in our `include/boot/head.S` file.

```
.set MAGIC, 0xe85250d6
.set PROTECTION_MODE, 0
.set HEADER_LENGTH, 32
.set CHECKSUM, -(MAGIC + PROTECTION_MODE + HEADER_LENGTH)

.section .multiboot
.balign 8
header_start:
    .long MAGIC
    .long PROTECTION_MODE
    .long HEADER_LENGTH
    .long CHECKSUM

    .balign 8
    .short 1
    .short 0
    .long 8

    .balign 8
    .short 0
    .short 0
    .long 8
header_end:
```

The following are the header contents:

1. `MAGIC` encodes the ELF magic number.
2. `PROTECTION_MODE` is ISA-specific, but for `x86` it refers to the protection ring; the kernel runs in ring 0, and thus is set to 0.
3. `HEADER_LENGTH` is the header size in bytes.

4. `CHECKSUM` is the sum of aforementioned variables (c.f. code above).

The rest refers to flags, CPU information, and other miscellanea that are out of scope for this report.

2.2 Kernel Header

We define our own kernel header to check if the kernel was loaded onto memory correctly; if it was not, a "panic" occurs. The kernel header structure definition can be found in `include/boot/boot.h`.

```
struct start {
        uint32_t magic;
        uintptr_t kernel_entry;
        uint32_t flags;
        uint32_t checksum;
        uint32_t hgr_mem;
        uint32_t cmdline;
} __attribute__((packed));
```

It is important prior to describing the contents of the header that, we address the C11 attribute `__attribute__((packed))`; but first, one has to understand what it means to access memory from the compiler's point of view: if one wishes to "read" a variable from memory location A that is stored as the type `int`, they must first determine how many bytes it is; if they attempt to read from A a size larger than `int`, they might get appended "garbage" at the end of their returned value; if they likewise attempt to read a size smaller than `int`, they might get less than what they asked for; this means, one has to know *exactly* how large the size of a variable is, before they attempt to access its memory location[†]. It should be relatively easy in a statically-typed language like C, one might think, but in reality, the C memory model says nothing about a specific size that compilers or systems have to conform to; in reality, the opposite is what takes place: the C language, and compilers adhere to the System ABI in the manner that it determines memory allocation. Now, lets assume that the compiler knows exactly what size it is, but unfortunately has to, for whatever reason, store 4 bytes in A location; first, the location has to at least 4 bytes or more of memory; second if the location is `0x10000000` (for example), it is naturally aligned with that size, but if it is `0x10000001`, it is not evenly divisible by 4 despite the location being "long enough"; this is known an **unaligned memory access**.

The compiler is not all-knowning, but it knows that sometimes our structures are not always naturally aligned with a specific memory location, so it "pads" our structure by a number of bytes (sometimes bits) to resolve any potential unaligned memory accesses. The `__attribute__((packed))` attribute overrides this, forcing the compiler to remove

[†]This does not happen in reality, as this would throw an exception if it were truly let to happen. The CPU (in `x86` AMD ABI) simply stalls and "stitches" the data together.

that padding.

The following are the kernel header contents:

1. `magic` encodes the kernel's magic number, which is "ANDR" encoded in ASCII characters.
2. `kernel_entry` is the physical location at which our `kernel_main` starts.
3. `flags` is the tty flag for whether to enable a serial terminal or not. If `0x00000000`, then it is disabled, otherwise `0x00000001` disables it; the latter is the default.
4. `checksum` is `0x414E4452 + kernel_entry + 0x01`; it is used to verify the aforementioned.
5. `hgh_mem` defines the higher memory region into which the kernel switches post-boot, i.e., `%eip` jumps to that location, and the CPU starts executing that instruction; it is set to `0x1000`.
6. `cmdline` is another flag which controls whether the tty is serial or not. It has a value of either `0` or `1`; it is set to `0` by default.

3 Process Control

4 Scheduler

5 Inter-Process Communication

6 Do not Panic Rule

7 Further Implementation Notes

8 References & Contributions Table