1 Block Device I/O Scheduling

Vanilla x86

In assignment 2, we had to implement a Shortest Seek Time First I/O scheduler. The goal of the algorithm was to have the longest and most sequential read/write movement of the hard disk drive head in one direction at a time, and then reverse the direction of the head in the longest and most sequential movement. This algorithm was to be based upon the No-Op scheduler.

The details of this scheduler required us to merge the block I/O requests when the access was either part of a previous (and currently not dispatched) access or increase the access of the request that was the previous block to the block request in question.

In this class, our group was not able to correctly implement this algorithm within the time limit of the assignment. However, we were able to understand what needed to be accomplished in order for the algorithm to operate correctly.

Android x86

In doing a simple diff between the Vanilla x86 kernel and the android x86 kernel on the No-Op scheduler, it would appear as if the scheduler algorithms are not different between the two implementations of the kernel. Logically, this would make sense because access block devices on the different platforms would not need to be any different as the drivers here are very mature and there are not very many different ways to access a block device.

The reason why they chose not to change the implementation of these algorithms is that they are not dependent upon hardware specific drivers that dictate what they do. They are a way to manage access to the hardware and keep the access requirements in a way that makes sense for the device. They do not need to have different ways to access an individual block on some random block device, that is what hardware drivers are for.

2 Block Device Driver

Vanilla x86

In assignment 3 we implemented an block device in ram that also encrypted access to the storage. This implementation was a driver and required handlers to initialize the device, register the device, and conform to the abstraction type that the kernel requires that allows the system to have a ”simple” and standard way to see and access the device. We also had to use the kernel’s crypto API to encrypt and decrypt the access to the blocks on the fly.

In the implementation of this driver, we made it a module to allow for easy development, and also allowing the system to only load the module if it is needed. We also had to provide a way for the kernel to detect the size and geometry of the ramdisk so that it wouldn’t try to to overwrite a section of memory when accessing the storage space. It also needed a way to inform the system about the actual size and limitation of the device so that it could be partitioned and formatted allowing the use of the memory as storage. We also had to register the device with the system in order to address it in the way the kernel requires, such as the listing under /dev. we also had to implement a way for the kernel to unregister the block device, clean up and resources it required and used, and not crash the system when the driver (or theoretically the device) was removed from the system.

With the encrypting and decrypting the access of the device, the driver had to have a way to communicate with the use to allow the setting of a key for the encrypting of the data. The easiest way to allow on the fly key changing was by using the sys interface in the kernel. Once we had the problem on setting a key on the fly by the user, we had to find out how to easily and efficiently encrypt or decrypt the data being accessed on the device. Since there isn’t much documentation of the Linux Kernel crpyto API, the most logical way to do the encryption/decrpytion was to perform the action on each byte accessed.

Android x86

With the android kernel being a (mostly) Linux kernel, it would make sense for them to not modify the way in which the crypto API is implemented, as that would be breaking one of the major rules in computer science: Do not make your own crpyto API! You leave this to people who really know what they are doing when it comes to cryptography.

In the Android kernel, the developers also kept the module implementation because it provides a standard way to allow devices to be dynamically loaded after compile and runtime, which requires a standard way to interface with the kernel. And since the android kernel requires drivers in order to interact with hardware, leaving the module loading system unmodified is a logical choice.

In terms of block device drivers, the android kernel would want to implement this in the same way the vanilla kernel does, as it provides the most compatibility between platforms and devices. And since the android x86 kernel is on an x86 device just like the vanilla x86 kernel, keeping the implementations the same between platforms reduces the amount possible bug introductions and incompatibilities.

The android kernel provides the same way to register a device and report the size specifics of the device as well as the geometry of the device allowing for the partitioning, formatting, and usage of the block device.

3 USB Driver

Vanilla x86

In the last assignment, assignment 5, we had to write a usb driver for a usb missile launcher that controlled the movement of the launcher. The implementation of this driver required a way to register the device with the system through the standard usb spec, and a way to communicate with the device over usb protocol. When we had the basic usb driver outlined, and having used a usb sniffer to discover how the proprietary Windows driver communicated with the launcher, we coded in the specific communication process as well as the code(s) that the diver sends to the launcher to make it function.

In the specifics, the driver was setup as a module, that only needs to be loaded when the launcher is actually connected to the system. This driver requires a way for the system to access the hardware in a standard, defined way since the device is a usb device. The driver also has to provide a way to register with the system and also unregister with the system when the device is removed. It also has to have a way to read basic information from the device and also be able to handle the device being removed at any time and not crash the system.

Android x86

Since the android started with a low power constrained environment in mind, the android kernel has a lot of usb power usage enhancement features that try to minimize the amount of power that the usb device(s) and controllers use. This is also more than likely evident with the android x86 kernel, since to origin of the project had those goals in mind. And since Intel is also trying to use x86 devices in the mobile market, keeping these power enhancement tweaks might prove worthwhile in the grand scheme to keep conflicts down to a minimum.

Other that the power enhancement tweaks, the usb driver has the same style of implementation as it would with the vanilla x86 kernel would. The module loading system is the same, to keep the most compatibility possible. The android kernel also uses a very similar, if not the same, process by which usb devices register them selves with the system and provide the required information to allow the system to use them.

Since usb has a defined specification, there is not much that can change between different platforms in the way usb is used. The only changes are how efficient each type of function executes. Since the android x86 kernel and the vanilla x86 are very similar, the way in whic the usb driver is setup is essentially the same, for all intents and purposes.