CS 343 Operating Systems, Winter 2020
Driver Lab: Interfacing with Hardware and the Rest of the Kernel

1 Introduction

The purpose of this lab is for you to engage with the challenges of device drivers. A device driver needs to closely interact with specific hardware that is not a processor nor memory. At the same time, it needs to provide a useful abstract interface for the rest of the kernel. Developing a driver is a relatively common task in kernel development, and being able to design a good abstraction that can bridge the hardware/software interface is a generally useful skill. You'll work within the NK kernel, but what you'll learn in developing drivers and abstractions for NK will generalize to other kernels, notably Linux.

You will start by developing a very simple device driver for a very simple device (a parallel port), a device driver that needs to interact with the kernel using an existing abstraction. Next, you will develop a driver for a sophisticated device (a graphics processing unit or GPU), and design and possibly implement an abstraction for similar devices.

You may work in a group of up to three people in this lab. Clarifications and revisions will be posted to the course discussion group.

2 Setup

You can work on this lab on any modern Linux system, but our class server(s) have been specifically set up for it.[1] We will describe the details of how to access the lab repo via github classroom in lecture and on the discussion group. You will use this information to clone the assignment repo using a command like this:

```
server> git clone [url]
```

This will give you the entire codebase and history of the Nautilus kernel framework ("NK"), just as in the Getting Started Lab. As before, you may want to use `chmod` to control access to your directory.

Unlike the Getting Started Lab, the code for this lab is in a separate branch. You now need to get to the relevant branch and configure it:

```
server> cd [assignment-directory]
server> git checkout driverlab
server> cp configs/cs343-driverlab-config .config
```

Now build it:

[1] For students who are using your own machine, the primary additional requirement in this lab compared to the Getting Started Lab is that you have a modern version of QEMU, since older versions, including the default version that ships with Ubuntu, have bugs in their support for the GPU device you will use in this lab. On the class servers, we have built QEMU 4.4.1 from scratch, and validated that it works. If there is interest, we will describe how to build QEMU in the discussion group.

```
server> make clean
server> make -j 8 isoimage
```

You will want to recall your setup from the Getting Started Lab for doing remote display. In this lab, you will need to have remote display functional. You can now boot your kernel:

```
server> source ENV
server> ./run
```

The `run` command will execute the emulator (QEMU) with a set of options that will "install" both devices this lab will use in the emulated machine. The emulated machine will boot NK, and if all is successful, you will see a blue screen with a red prompt, just as in the Getting Started Lab. Remember that the shell you are talking to is within the kernel itself.

3 Files

While NK is tiny compared to the Linux, Darwin, or Windows kernels, it does have several hundred thousand lines of code spread over over more than a thousand files. Therefore it is important to focus on what is important for your goals. As with any significant codebase, trying to grok the whole thing is either impossible or will take far too long. Your strategy for approaching the code has to be adaptive. In part, we are throwing you into this codebase to help you learn how to do this.

Here are some important files for this lab:

- `include/dev/parport.h`: stub header file for the simple driver. You may add code to this.
- `src/dev/parport.c`: stub source code for the simple driver (the parallel port). You will add code to this.
- `include/nautilus/chardev.h`: header file describing the character device abstraction, which your simple driver will provide.
- `include/dev/virtio_gpu.h`: stub header file for the complex driver (the GPU). You may add code to this.
- `src/dev/virtio_gpu.c`: stub source code for the complex driver. You will add code to this.
- `include/dev/virtio_pci.h`: header file for the framework within which your complex driver will fit. The framework solves many problems for you.
- `src/arch/x64/init.c`: the boot code that will instantiate your device drivers (you will not need to change this file).

Additionally, you will add these files:

- `include/nautilus/gpudev.h`: This is where your abstraction for GPUs will go, similar to how `include/nautilus/chardev.h` defines the abstraction for character devices.
- (optionally) `src/nautilus/gpudev.c`: This is where the implementation of your abstraction will go, should you decide to do this extra credit.

4 Devices, device drivers, and abstractions

Providing and managing access to hardware devices, particularly for I/O, is a core thing that any kernel does. There are literally thousands of different devices in existence that can be attached to a modern x64 machine through a wide range of buses and other mechanisms. A device driver is code within the kernel that provides a software interface to a specific device. The device driver code does this by directly interacting with the device hardware. The software interface provided by the driver is abstracted by the driver or other software so that devices that are similar at the conceptual level can be treated the same by the rest of the kernel, even if their device drivers are very different.

To make this more concrete, consider networking. The common Intel 82576 is a chip that implements a gigabit Ethernet network interface. On a typical machine, the chip is likely placed somewhere on a PCIe bus tree whose root is a processor. This means that in order to talk to it, we will also need to understand PCIe. The 82576 is part of a large family of similar chips from Intel that have more or less the same programmatic interface, an interface often called "e1000e". If your kernel has an e1000e driver that can talk to such chips over PCIe, you have the basic capability to send and receive Ethernet packets.

We could tell the same story about the common BCM5720 chip from Broadcom, its interface to PCIe, its variants, and its "tigon3" programmatic interface. The tigon3 driver is quite different from the e1000e driver, yet both are at heart about sending and receiving Ethernet packets. Do we really want the network stack to actually differentiate sending an Ethernet packet via an e1000e, tigon3, or any of the other of the dozens of kinds of Ethernet chips/boards?

Clearly that would be horrific. Instead, we implement an abstraction on top of the drivers. For example, a general abstraction of sending/receiving Ethernet packets might be layered on top of all these drivers. Then the network stack could use the abstraction instead of using the drivers directly. How to create a useful, general, and widely-implementable abstraction is a challenge because part of any implementation of the abstraction must span from software to hardware. We cannot change the hardware to suit the abstraction.

A device driver is very subtle code to write. In addition to the concurrency that we have seen before, we now also need to worry about races and other interactions with a piece of hardware we do not fully control. Devices are not processors, yet still can manipulate memory and timing, and typically produce interrupts as well. Modern devices are often implemented to use ring buffers (similar to your previous lab), but now either the producer or consumer is in the hardware of the device itself.

5 What a device does

A hardware device operates independently of the processor, but provides an interface that kernel code running on the processor (i.e., the device driver) can interact with. The interaction goes both ways.

5.1 Model and interface

We typically think of the device as comprising a *state machine* that is driven by *events* coming from the driver and from internal and external sources. Typically, the device exposes some of its *registers* for manipulation by the device driver. Note that these are at all not the same thing as the registers of the processor. In order to read and write the device's registers, they must be made visible to the processor. On x64, there are two ways to do this:

- *Port-mapped I/O (PMIO):* Here, the registers are accessed via a special "I/O" address space using special instructions called `in` and `out` instructions. The parallel port uses PMIO.

- *Memory-mapped I/O (MMIO):* Here the registers are mapped into the regular physical memory address space and are accessed via instructions like `mov`. However, because this "memory" is not real memory (for example, it cannot be cached and reads/writes must not be reordered or discarded by either the compiler or hardware), it is critical for the device driver author to understand how to get exactly the memory operation behavior they want to see at the device. The GPU uses MMIO.

The hardware state machine can also create events for the processor. This is typically done by *interrupts*. Interrupts force the processor to switch to kernel mode and start executing at a well-defined entry point, typically in the device driver. Both of your devices raise interrupts.

We often want to transfer data from/to memory to/from the device. There are essentially two ways to do this:

- *Programmed I/O (PIO):* Here the software (the device driver) reads/writes memory and writes/reads a device register directly. The parallel port uses PIO.

- *Direct Memory Access (DMA):* Here the software (the device driver) tells the device where in physical memory the data is coming from/going to (i.e., the software gives the hardware a pointer), and then the device reads/writes memory accordingly. DMA is usually implemented as *gather/scatter*, meaning that a single unit of data can be fragmented across multiple disjoint chunks in memory. The GPU uses DMA.

On a sophisticated device, like the GPU, DMA is also used for more complex control of the device than would be sensible using the registers alone. A common abstraction is a *producer-consumer queue* (usually implemented as a ring buffer like in a previous lab) in which the producer or consumer is the hardware device. Often, what is queued is a *gather/scatter list* for further DMA. The driver creates a higher-level command by building a linked list in memory, then it queues a pointer to the head of this list in a ring buffer. The device dequeues the pointer using DMA, and then uses DMA to walk to linked list to recover the command. It then executes the command, which might involve further DMA, and might result in an interrupt.

5.2 Lifetime

When the system is turned on, the device will start in a known initial or *reset state*. In this state, it is essentially "off", except for *advertisement*, in which it makes its existence visible by indicating where its registers are mapped in memory or the I/O space. The device driver can then discover it, initialize it as needed, and interact with it. If no device driver exists, the device remains "off". We are ignoring other aspects of modern devices here, such as sleep/resume and hot-plugging.

6 What a driver does

The purpose of a device driver is to make a hardware device useful. Typically, the combination of the driver and the device implement an abstract interface within the kernel.

6.1 Lifetime

A device driver has the following lifetime.

- *Discovery:* The driver tries to find devices it is compatible with using device advertisements.

- *Initialization:* The driver interacts with the device to place it into a suitable state for normal operation.
- *Registration:* The driver makes the device visible to the rest of the kernel through some abstract interface.
- *Operation:* The driver manages requests coming from the rest of the kernel, and from the device, to provide the service of the abstract interface to the rest of the kernel. This where the device driver spends most of its time.
- *Teardown:* The driver gets the device back to its reset state and unregisters it from the abstract interface.

Some devices or kernels will have additional elements, including *Hot-plug* (the user can install and remove devices at will—think USB), and *Sleep/Wake* (the kernel can tell the driver to put the device and itself to sleep sleep to save power).

The particulars of there steps are highly dependent on the device itself, as well as the abstract interface the driver must provide to the rest of the kernel.

6.2 Driver frameworks

Many devices, even of radically different kinds, will share implementation commonalities beyond the very low level ones described above. Kernels may provide factored driver frameworks to take advantage of this commonality to reduce the amount of code that a driver author needs to write and debug. For example, the GPU device you will be using in this lab is simultaneously a GPU device, a Virtio device, and a PCI device with MSI-X interrupts. A great deal of the code for Virtio and PCI+MSI-X is just reused. What is new is the "GPU" side.

7 Task 1: Parallel port character device driver

The simplest I/O device on a PC-derived machine is the "parallel port". This still exists on all modern machines, but there is typically no connector to attach anything to it. Fortunately, QEMU can easily connect a parallel port to a terminal or file so we can see what it outputs. The concept of a parallel port is this: you feed it a byte, and it outputs the 8 bits of the byte via 8 wires on the connector. If you want the attached device to notice that the output byte has changed, you can also "strobe" a different wire to essentially "clock" the data to the attached device. Technically, a parallel port can also input 8 bits at a time, but we will not use that capability or other special features like EPP/ECP, nybble mode, etc.

In this lab, we will use only the legacy "first parallel port" ("LPT1") that dates all the way back to the first IBM PC from 1981 (where it was first used to connect to a printer, but soon was used almost like how USB is used today). This first parallel port, if it exists, is at a well known place, which makes discovery easy. We have constructed the discovery and initialization code for you.

The parallel port driver (`include/dev/parport.h` and `src/dev/parport.c`) is implemented within NK's character device abstraction (`include/nautilus/chardev.h`). The purpose is to show you how a driver for a specific kind of device can fit within a more general abstraction, allowing the kernel to use the driver without being aware of its internals.[2] Later, we will ask you to design an abstraction for

[2]Another character device (or chardev) within NK is the serial port driver. A serial port does input/output a bit at at a time. Using the NK shell command "chartest" you can do test I/O from/to any character device, including serial port and parallel port devices.

GPU devices.

The stub parallel port driver we have given you has detailed comments. We will only go through the high level theory of operation of the parallel port device here. If you are reading the code, start with `nk_parport_init()`.

The parallel port device has three one-byte-wide registers (control, status, and data) that are used by the driver to interact with it. These registers are accessed using PMIO as described earlier. The NK functions `inb()` and `outb()` execute the relevant instructions.

As you can guess, the control registor allows us to configure and control the device. At startup, we use this register to set the device into output mode, reset the attached "printer", and arrange to have every acknowledgement of the receipt of a byte by the printer to raise an interrupt that vectors to the handler within the driver. That is, when the attached printer has the current byte you are sending to it, it will interrupt you to let you know.

The status register provides considerable information, but the main thing we will care about is the busy bit. If the printer is busy, we will wait for it before outputting the next byte to it. Note that for electronics-level reasons, the busy bit is *active low*, meaning that true is represented by zero, while false is represented by one. Several other bits in the control and status registers are similar.

When you write the data register, the device will output the byte written, while when you read it, it will attempt to input data and give it to you. One of the bits on the control register determines the direction of data flow that is enabled.[3]

We expect you to write the code necessary to allow the parallel port drive to successfully handle the use of the `chartest` shell command for outputting data to the port: the `status` and `write` chardev interface functions need to be functional. Your implementation should use interrupts instead of polling. To put this into context, our implementation adds about 30 lines of code to the stub implementation you already have.

To write a byte, you will take the following steps:

1. Poll the status register, looking for the attached printer to be ready. When you implement interrupt handling, this loop will execute only once.
2. Use the control register to set the device to output mode.
3. Write the data byte to the data register.
4. Strobe the attached printer. You will do this by setting and then resetting one of the bits in the control register.
5. An interrupt will then occur when the attached printer has acknowledged your byte.

Timing is important on real hardware. This device is thousands of times slower than a processor. As a consequence, it is often useful to insert delays. One function to do so is `io_delay()` , which is about a 1 microsecond delay.

To test your implementation, you will run NK using the `run` script. Then you will issue the command `chartest parport0 w 26` from the NK shell. If you are successful, the letters of the alphabet will appear in the file `parport.out`.

[3] On the original parallel port hardware getting this wrong would damage the hardware or even start a fire. Sadly, this is not possible with QEMU.

8 Task 2: Virtio GPU device initialization and test

The goal of this part of the lab is to initialize a complex device, a GPU, and make it do something interesting. You will configure the GPU to switch from text mode to 2D graphics mode, then draw something on the screen, and finally switch back to text mode. You will do this during the kernel boot process because we do not yet have an abstract interface for GPUs on which to base operation. You will design such an interface in the next task.

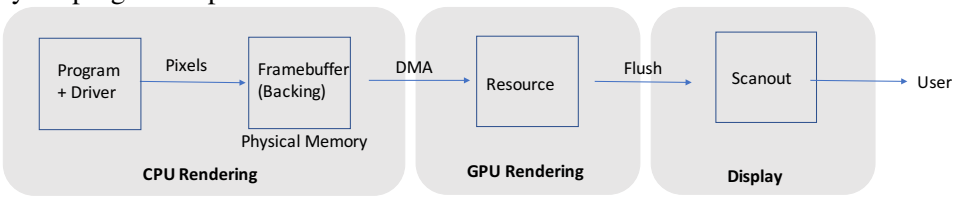
The GPU you will target is a Virtio GPU device. Virtio is a widely-used interface standard for creating *paravirtualized devices* within virtual machines monitors (like QEMU, KVM, Xen, VMware, Parallels, VirtualBox, etc) and interfacing to such devices from a guest OS. The latter is what you are doing here. The detailed Virtio specification, including for a wide range of device types is given elsewhere.[4] **DO NOT BE DAUNTED.** You will be working in our pre-existing frameworks for Virtio devices and PCI devices. There are also two other Virtio drivers (for block devices (disks) and network devices) already in the codebase, and we have provided you with stub code as the starting point for your vrtio-gpu driver. Finally, at this point, we are not asking for you to develop a complete driver, but just to build just enough of a driver so you can draw something on the screen.

While Virtio is an interface to virtual hardware, it is quite similar to the interfaces typically exposed by modern physical hardware. What you will learn and be exposed to in this lab is widely applicable to most devices today, particularly high throughput devices. If you'd like to look at the stub code, the place to start is the `virtio_gpu_init()` function.

8.1 Understanding the device

The Virtio GPU device exposes its registers via MMIO. In the stub code, the `common` field within `struct virtio_pci_dev` points to where they are mapped. We also provide atomic load and store functions to interact with them. The more interesting thing about the Virtio GPU device is that it mainly operates operates via DMA, both for any non-trivial command, and for data (the pixels that will appear on the screen.

The following figure shows the high-level conceptual model of this device—how data flows from pixels you draw in your program to pixels on the screen:



The system writes into a framebuffer (an array of pixels) located in normal memory. This framebuffer, also called backing storage, is associated with a resource on the GPU. The GPU copies pixels from the framebuffer to itself using DMA. It renders the pixels of the resource and makes them ready for display. The display data is flushed to a "scanout", which is an abstraction for a monitor. The scanout makes the rendered screen of pixels visible to the user.[5]

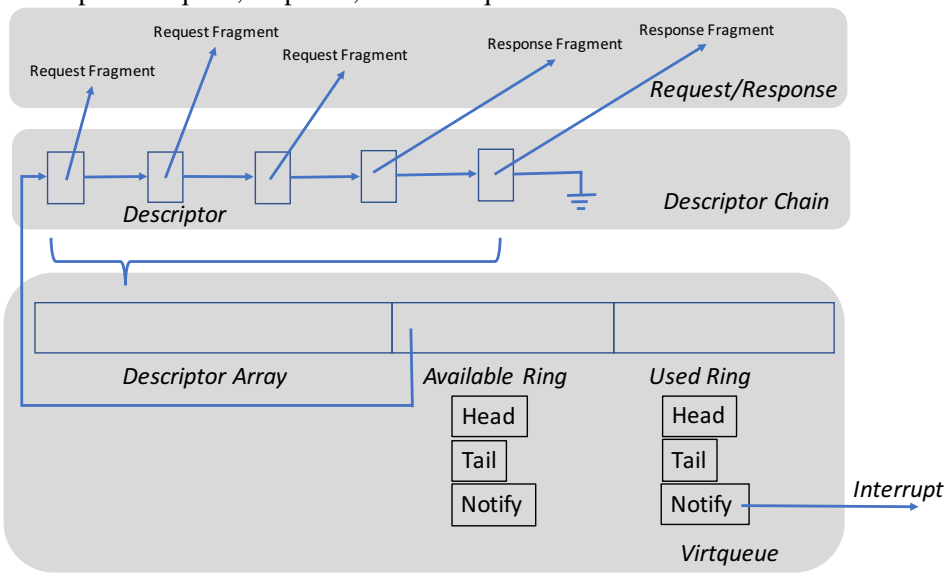
To set up and trigger this pipeline from our driver, we need to issue requests to the device and receive responses from it. This is also done via DMA using a general abstraction called a virtqueue. Virtqueues

[4] We are using the Virtio 1.1 specification, public review draft 01, dated December 20, 2018, `https://docs.oasis-open.org/virtio/virtio/v1.1/csprd01/virtio-v1.1-csprd01.html`

[5] We are using this device in its simplest, unaccelerated 2D mode.

are used by all the different Virtio devices—they are a common framework. Different Virtio devices differ based on how many virtqueues they have and the specific requests that can be issued to them. For the Virtio GPU device, one virtqueue is used for configuration and triggering of the pipeline, while a second virtqueue (which we will not use) is for controlling a hardware mouse cursor that is overlaid on the screen.

The relationship of a request, response, and a virtqueue is shown here:



To command the device to do something, we allocate both a *request* and a *response*. These can be fragmented (and sometimes must be) across non-contiguous chunks of memory. We then fill out the request as needed for the specific command we are sending to the device. At this point, we assemble the request and response fragments into a linked list called a *descriptor chain*. Each node in this list is a *descriptor*—it describes the fragment (its starting address, length, whether it is writeable, etc), and contains a pointer to the next descriptor in the list. While request/response fragments are allocated in the ordinary way, descriptors are allocated from a *descriptor array*, within the virtqueue itself. The descriptor chain is a scatter/gather list.

The virtqueue is the data structure that is shared with the hardware. We read/write it directly, while the hardware reads/writes it using DMA. It has three components: the descriptor array we have previously mentioned, the *available ring*, and the *used ring*. The available ring is a producer/consumer ring buffer in which the driver is the producer and the device is the consumer. The used ring is a producer/consumer ring buffer in which the device is the producer and the driver is the consumer.

We issue a command to the device by pushing a pointer to the head of the descriptor chain (the index of the first descriptor in the chain) into the available ring, much as in the producer/consumer lab. But how does the device know we did this? We write a special register called the notification register to inform the device that we have changed the contents of the virtqueue.

At some point, the device pulls our descriptor chain pointer from the available ring via DMA, and reads the descriptor chain via DMA. It next reads the request fragments, again with DMA, by walking the descriptor chain. Now it has gathered our request, and it begins to execute it. When it is done, it writes the response into the response fragments (scattering), again all with DMA. Next, it pushes the pointer to the head of the descriptor chain (the index of the first descriptor) into the used ring (DMA again). This indicates the command is complete. But how does the driver know of this push to the used ring? The device notifies the driver by raising an interrupt, which simply indicates that the device has changed the virtqueue.

For the purposes of this lab, much of the required functionality in the driver, albeit with polling, has

already been implemented for you. Please see the `transact_rw()` and `transact_rrw()` functions. These implement request/response handling for 1 and 2 fragment requests with single fragment responses.

8.2 What to do

We have taken care of device discovery, virtqueue setup, interrupt setup, and even issued the first command to the device to show you how this is done. You should look at the code in `src/dev/virtio_gpu.c`. You will be adding to `test_gpu()`, which is copiously commented. You will also note that there are a wide range of structure definitions (and declarations of requests and responses). Your goal here is to show some sort of coherent image on the screen for a few seconds at boot time. Think of the Apple or Windows boot screen for inspiration. The following are the conceptual steps you would take to do this:

1. *Enumerate Displays:* In this step, you would find all available scanouts (monitors) and pick one. This step is already written for you.
2. *Create Resource:* Here you are asking the GPU to internally create a resource for rendering. This resource will cover the entirety of the scanout you chose in the previous step. You will also need to pick the pixel format for the resource.
3. *Create Framebuffer:* Here you would allocate space for the framebuffer you will be using. This step is written for you.
4. *Draw Image:* Here you would fill out the framebuffer pixels with the image you want to put on the screen.
5. *Associate Resource and Framebuffer:* In this step, you would tell the device that you want to create a tie between the resource you created in step 2 and the framebuffer you allocated in step 3.
6. *Associate Resource and Scanout:* Here, you will command the device to make a connection between the resource of step 2 and the display from step 1. At this point, you have assembled the pipeline we saw earlier. Since the pipeline is now active, you will probably see the QEMU screen switch to graphics mode and change size.
7. *Transfer Pixels:* Now you will command the device to transfer pixels from your framebuffer (step 3) and its resource (step 2). More DMA!
8. *Flush To Scanout:* This command will tell the device to render from the resource (step 2) to the monitor (step 1).

After these steps, your image should be visible. Note that the code we provide resets the GPU back to text mode as the last thing it does. You probably want to introduce a delay loop so that you can appreciate your displayed image for more than a split second.

9 Task 3: GPU device abstraction

It look a lot to get an image on the screen at boot time, but currently this is the only thing your driver can do. Wouldn't it be nice if code anywhere in the kernel could use your driver to do graphics whenever it would like? It would be even better if it could use *any* graphics driver to do graphics without being concerned about the details of the driver (or the device). You saw something similar in the first part of the lab, when

you implemented your parallel port within NK's character device abstraction and were able to use test code (and even a shell command) that was totally independent of your driver. What is the right abstraction for GPUs?

In addition to your new Virtio GPU driver, there is another simple graphics driver in NK, within the files `include/dev/vesa.h` and `src/dev/vesa.c`. VESA is an ancient standard for extremely simple GPUs that most hardware GPUs still sort of support. The model is quite simple: you can ask the GPU what modes (resolution, color depth etc) it supports (on its single monitor). There are a range of standardized modes. You can tell it to switch to the mode you want. Instead of having a separate resource and framebuffer like the Virtio GPU, it has only a framebuffer. Furthermore, the framebuffer is on the GPU itself. To draw on the screen, you ask the GPU for the physical address of the framebuffer, and then draw pixels directly on it. These immediately are displayed by the attached monitor. NK's VESA driver assumes only a single VESA GPU is present and that it's attached to a single monitor. You can see how the VESA driver is used by looking at the `vesa_test()` function.

In addition to the character device abstraction from the first part of this lab, you area also welcome to look at the block device (e.g. hard drives) and network device (e.g. Ethernet cards) abstractions in NK, these are in `include/nautilus/blkdev.h` and `include/nautilus/netdev.h`.

9.1 What to do

Your goal is to design a GPU device abstraction that takes into account the differences between how dfferent GPU devices operate or might operate. You can use the Virtio GPU device and the VESA GPU device as examples. You can assume that we only care about unaccelerated 2D GPUs. The abstraction should also take into account its user. How would you like to be be able to do graphics? For example, does the `vesa_draw_pixel()` function make any sense? The abstraction must also take into account the burden placed on the device driver author. We do not want to have a gloriously powerful abstraction that no one can implement. You should also consider the difficulty of implementing the abstraction itself, independent of the device drivers.

You will define your abstraction in a new file `include/nautilus/gpudev.h`. The definition should mostly be comments that explain the theory of operation behind the abstraction for both the device driver developer and the user of the abstraction. You can use the other abstraction header files as examples of the level of detail we are expecting. Your header file should compile. We are not asking you to implement your abstraction.

10 Grading

Your group should regularly push commits to github. You also should create a file named `STATUS` in which you regularly document (and push) what is going on, todos, what is working, etc. Your commits are visible to us, but not to anyone else outside of your group. The commits that we see up to deadline will constitute your hand-in of the code. The `STATUS` file should, at that point, clearly document that state of your lab (what works, what doesn't, etc).

In addition to your `STATUS` file, you should regularly push your work within `src/dev/parport.c`, `src/dev/virtio_gpu.c`, `include/nautilus/gpudev.h` and any other files you are changing.

Grading of this lab will be by interview and demo. After the deadline, the TAs will make arrangements to talk to each group. The group will need to demo their drivers and answer questions about them, as well as to answer questions about their proposed GPU abstraction.

The breakdown in score will be as follows:

- 20% Task 1—Functional and sensible implementation of parallel port driver. It must pass `chartest` write commands.
- 50% Task 2—Functional and sensible implementation of enough of the Virtio GPU driver to be able to show images. The demonstration should show a boot splash screen for NK.
- 30% Task 3—Sensible and justified GPU device abstraction that can be compiled (it can fail to link). Much of the value here is in the comments and thought process for your abstraction. There is no right answer.

11 Extra Credit

We will allow up to 20% extra credit in this lab. If you would like to do extra credit, please complete the main part of the lab first, then reach out to the instructor and TAs with a plan. Some possible extra credit concepts are the following:

- Animate your splash screen. You can do this using a variety of methods. One is simply to change the framebuffer, and then push the pixels through the pipeline using the commands you already have. Another is to use hardware scrolling.
- Implement the hardware mouse cursor. VirtioGPU can give you control over the shape (image) of the mouse cursor, and allow you to position it where you would like. Have the hardware mouse cursor move around your screen.
- Implement your GPU device abstraction layer. You have definite it in your header file. Implement it! You can add some shell commands to launch tests.
- Reimplement your Virtio GPU driver to conform to your GPU device abstraction layer instead of to the generic device abstraction layer.
- Port a GUI to your driver or your abstraction layer. At some point, a port of μGUI was made to NK for use with VESA. It is feasible to get it to work.