CS 343 Operating Systems, Spring 2022
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 for NK (and abstractions) 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 a simple "character device" abstraction. Next, you will develop a more complex device driver for a sophisticated device (a graphics processing unit or GPU), which needs to implement a "GPU device" abstraction.

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

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 Piazza. 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.

Now build it:

```
server> cd [assignment-directory]
server> make clean
server> make -j 8 isoimage
```

[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. We describe how to build QEMU in Piazza post.

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.

Note: you should never need to `make clean` in NK. Just calling `./run` should be sufficient to rebuild all necessary files and start NK.

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.

Important files for the parallel port part of the lab:

- `include/nautilus/chardev.h`: header file describing the character device abstraction, which your simple driver will provide.
- `src/nautilus/chardev.c`: implementation of character device abstraction layer. This includes a test command that you will use (but not modify).
- **`src/dev/parport.c`**: stub source code for the parallel port driver (about 450 lines). You will add code to this.

Important files for the GPU part of the lab:

- `include/nautilus/gpudev.h`: header file describing the GPU device abstraction, which your driver will provide.
- `src/nautilus/gpudev.c`: implementation of GPU device abstraction layer. This includes a test command that you will use (and modify).
- **`src/dev/virtio_gpu.c`**: stub source code for the Virtio GPU device driver (about 1750 lines). You will add code to this.

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 by driving it high and then low 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]

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()`.

[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.

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 clearing the strobe bit in the control register to zero and writing it, then setting it to one and writing it, then clearing it to zero again and writing it.
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`. The actual test that is occurring can be found in the `handle_chartest` function in `src/nautilus/chardev.c`.

Interrupts Once you have basic output working, you will also start seeing interrupts. You're not done! You need to figure out how to use the interrupts so that the code is efficient and that `wait_for_attached_device()` stops spinning. You'll need to manage shared state in both the interrupt handler and in the `read_write()` function to do so.

[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.

Producer/consumer Another thing your interrupt handler should do is wake up any threads that might be waiting on the condition that the device becomes READY. You can do this with `nk_dev_signal()`. This may remind you of condition variables. If you're curious, you can find the corresponding wait on the condition in `src/nautilus/chardev.c` (which calls to `nk_dev_wait()`). And, yes, this is also an instance of the producer-consumer problem.

Common problem: understanding PMIO For Port-Mapped I/O like the parallel port, it's common to make a copy of the register contents in memory, configure/modify it, and then write it back to the device. The copy in memory is not the register, however. Consider this code:

```
ctrl.val = inb(CTRL_PORT(s));
ctrl.bidir_en = 1; // active low to enable output
outb(ctrl.val, CTRL_PORT(s));
```

Here we are first reading the control register into a variable in memory. On the next line we modify the variable in memory. And on the third line we write the variable back into the control register on the device. Until we finish line three, the device does not know about our changes.

Common confusion: interrupts happen quickly in QEMU In a real parallel port, operations take many CPU cycles to complete. A real parallel port is thousands of times slower than a CPU. However, the parallel port in QEMU reacts pretty much instantaneously because it is emulated in software, and does not attempt to provide accurate timing. As a consequence, students will see that interrupts happen almost immediately after initiating a request, and wonder what the point of being interrupt-driven is, when a polling loop would be similarly efficient. In reality, on real hardware, the polling loop would spin thousands to millions of times. The other QEMU-emulated devices have similar unrealistic behavior.

8 Virtio GPU device

The goal of this part of the lab is for you to implement a more complex device driver for a more complex abstraction. You will be build a driver for Virtio GPUs that implements the GPU device abstraction in NK.

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 a lot of stub code as the starting point for your driver.

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.

[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`

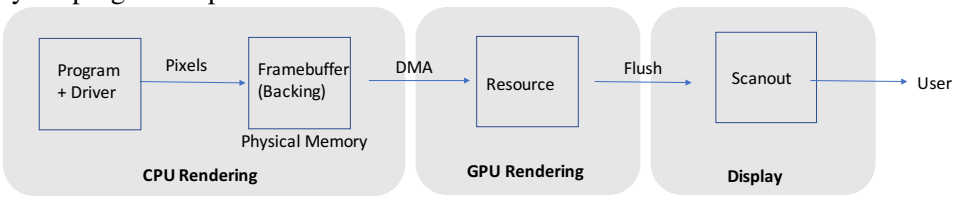
8.1 Understanding the device

A Virtio GPU device is a Virtio PCI[5] device. Most of the details of this have been abstracted away for you. However, to control a Virtio GPU, you will often need to make use of the functions that are in code that all Virtio PCI devices share, which you can find in `include/dev/virtio_pci.h`.

A Virtio GPU device exposes its registers via Memory-Mapped I/O (MMIO), and `virtio_pci.h` provides atomic load and store functions to read and write them correctly. The MMIO registers are pointed to by the `common` field of `struct virtio_pci_dev`) after the device has been set up.

The more interesting thing about the Virtio GPU on top of PCI is that it mainly operates via DMA, both for any non-trivial command and for data (such as 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:



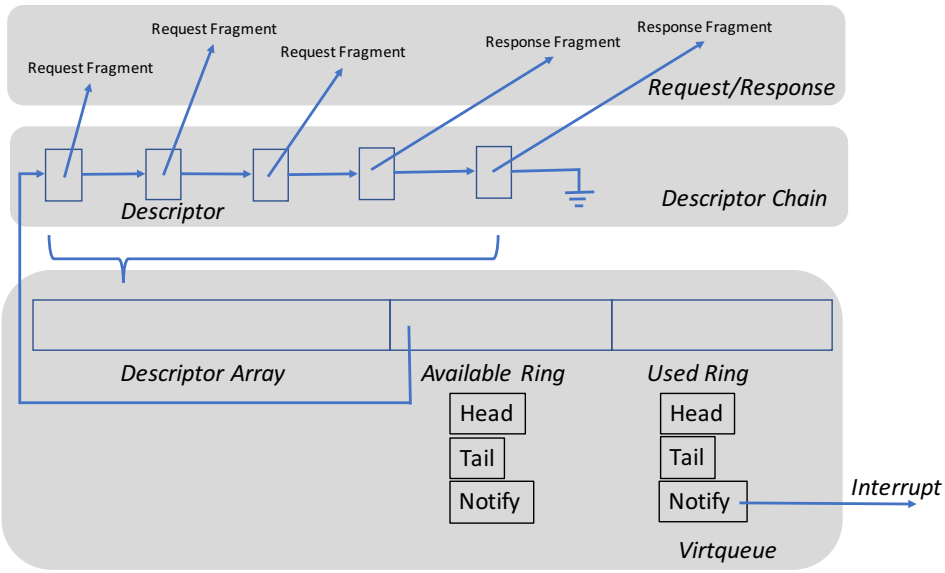
The system writes into a framebuffer (an array of pixels) located in normal memory. This framebuffer, also called backing storage, is associated with a resource on the GPU. The GPU copies pixels from the framebuffer to itself (to the resource) using DMA. It renders the pixels of the resource and makes 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.[6]

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 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 except for extra credit) 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:

[5] It is connected via a PCI bus, the most common high-speed interconnect.

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



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 in `src/dev/virtio_gpu.c`. These implement request/response handling for 1 and 2 fragment requests with single fragment responses.

9 Task 2: Mode switch and SW/HW pipeline setup

The goal of this task is to make it possible to switch from the default text mode of NK to a graphics mode supported by the Virtio GPU device. Another goal is to get a first look at the code and the abstraction.

Test code You can always run our GPU test code by first running NK using the `run` script, and then, at the NK shell running the following command:

```
gputest virtio-gpu0
```

This will run a small set of tests against the GPU device named `virtio-gpu0`. In the default configuration, this will be your device.[7] You can see the test code in the function `handle_gputest` within `src/nautilus/gpudev.c`. You can add your own tests there as well. The default code does the following: (1) It asks for the current mode of the device. (2) It asks for the list of possible modes and prints them out. (3) It chooses the first graphics mode and switches to it. (4) It clears the screen to black and then draws a bunch of random filled boxes composited (bit blitted) in random ways. (5) It clears the screen to purple and the draws a bunch of random lines. (6) It copies boxes of pixels around the screen with different compositing (bit blitting). (7) It returns to the initial mode.

If you run `gputest`, you'll note that it prints a lot of errors, and nothing happens. This is because much of the driver is unimplemented. The device is discovered, initialized, and registered, but does not do anything in operation. You will change this.

You need to start by implementing the `set_mode` function within `src/dev/virtio_gpu.c`. This involves sending the device a series of commands to set up the pipeline described in Section 8.1. There are comments in the code to help guide you. Also note that the function `reset` does the converse: it issues a set of commands to the device to tear down the pipeline and switch it back to text mode.[8]

As part of your `set_mode` code, you will initialize the frame buffer with some image. At this point, however, nothing will appear on the screen.

10 Task 3: Flushing the SW/HW pipeline to see your graphics

The next step to take with the GPU is write the code that causes the monitor to actually show your graphics. No matter what interesting image you put into the framebuffer, you need to get the GPU to run the pipeline described in Section 8.1 before it exists anywhere other than in main memory. Also, in NK's GPU device abstraction, there is no guarantee that any drawing becomes visible to the user until after `nk_gpu_dev_flush` function is called.

To "flush" the graphics to the display you must first transfer the framebuffer data to the GPU resource over DMA. Then another command to the GPU will send the display data to the monitor. You should implement the `flush` function in `src/dev/virtio_gpu.c`. There are comments to help you out.

Once you have the `flush()` function working correctly, you should be able to see the initial contents of your framebuffer appear on the screen.

[7] If you are interested, you can change the properties (resolution, color depth, etc) of the device by changing the QEMU command line in the `run` script.

[8] Note that a great deal of initialization of the device has already been done for you before you reach the `set_mode` function. If you are curious, you can begin at the `virtio_gpu_init` function, which is called on device discovery at boot time.

11 Task 4: Simple drawing

The next step is to implement simple drawing operations such as drawing pixels, lines, and polygons. Drawing commands affect the framebuffer. The user may not see the results until after `flush` is called.

We recommend that you begin by building `graphics_draw_pixel`. An important thing to learn is the relationship between a coordinate within the graphics space, the memory offset within the framebuffer at which the corresponding pixel is, and the offsets within the pixel for its various channels. This relationship will show up again and again in other drawing commands.

Note the `nk_gpu_dev_pixel_t` union that is defined in `include/nautilus/gpudev.h`. Pixels can be referred to as a 32-bit value or as an array of 8-bit values: red, green, blue, and alpha channels. The ordering of the channels in the pixel are dependent on the GPU configuration, however, so you may want to use the `NK_GPU_DEV_PIXEL_SET_RGBA` macro to initialize a pixel. You can ignore the alpha channel (pixel transparency), as it has no effect with this GPU.

Next, proceed to `graphics_draw_line`. You can invent your own line drawing algorithm, or use one of the various existing ones. Bresenham's line algorithm is a common choice. Note that a device driver should not using floating point arithmetic (and, in fact, this is completely disallowed in most kernels). As a consequence, you have to think about and implement the algorithm using only fixed point arithmetic or rational arithmetic (the arithmetic of ratios of integers). Be sure to test lines drawn diagonally, in any stroke direction, and figure out what to do when drawing off the edge of the screen.

Finally, build `graphics_draw_poly`, which draws a polygon using an array of its vertices.

To test your implementations, you can use the code in `handle_gputest` within `src/nautilus/gpudev.c`, which you can extend as you would like. Remember that this code uses the abstract GPU interface `nk_gpu_dev_....` These functions are automatically linked to your device driver when you use them with a GPU device that is handled by your driver.

You may be wondering at this point why these and other graphics functions are part of the GPU device abstraction. The idea is that an advanced GPU might provide these functions directly in hardware. By having them in the interface, it makes is possible for a driver to pass them along to such hardware.

12 Task 5: Box-related drawing

The next functions to implement are the three box-related functions. A `nk_gpu_dev_box_t` describes a rectangular region. The box-related functions allow colors or images to be drawn to the framebuffer. They also enable copying portions of the framebuffer to a new position in the framebuffer.

The simplest of these functions is `graphics_fill_box_with_pixel`, which "paints" (fills) the box with a pixel. This is a good place to start, and you should be able to finish the basics quickly.

The main complexity is that you will have to understand what *bit blitting* is. Blitting combines two pixel values into a single output pixel value. Commonly, a pixel already in the framebuffer and a new pixel are combined to create a new value for the framebuffer pixel. Combining operations include AND, OR, XOR, ADD, DIVIDE, etc. and there is also a COPY operation that just replaces the existing pixel with the new one. It important to note that arithmetic in a blit should be *saturating arithmetic*, not standard modular arithmetic. So, for example, if you ADD "+1 red" to a pixel with the maximum red channel value of 255, the result is that it stays at that maximum red channel value, *not* that it goes to the minimum red channel value of 0.

One use of bit blits is to overlay a transparent image, such as a video game sprite, onto a background. This can be done with two images, a mask which is applied to the background with AND, and a sprite that

is applied with OR. Another use case is to show window movement. The window can be XORed over the background to display it. Then, when it comes time to stop displaying it (the user having slightly moved the window, say), it is simply XORed a second time to restore the original background pixels. For more information see: `https://en.wikipedia.org/wiki/Bit_blit`.

The next step is `graphics_fill_box_with_bitmap`, where the idea is that instead of painting with a single pixel, you are painting with an image (the bitmap). The blitting operators still apply. In a more advanced 2D GPU, this operator is typically provided in hardware, and is a cornerstone of what the GPU does. This operation makes the sprite example given above much easier than doing it pixel-by-pixel.

Note that you will have to think about what to do if the bitmap is smaller than or larger than the box into which you are drawing it. If the bitmap is larger than the box, it should be clipped to the size of the box. If the bitmap is smaller than the box, it should be repeated (tiled) until the box clips a repeat of it.

Finally, implement `graphics_copy_box`. This is similar to `graphics_fill_box_with_bitmap`, but the bitmap is a part of the framebuffer. This operator makes the window movement example given above much easier than doing it pixel-by-pixel.

Same as before, to test your implementations, you can use the code in `handle_gputest` within `src/nautilus/gpudev.c`, which you can extend as you would like. Remember that this code uses the abstract GPU interface `nk_gpu_dev_...`.

Note that at its heart, an `nk_gpu_dev_bitmap_t` is just an array of pixels, each of which is just a `uint32_t`. You can therefore embed bitmap images into your code very easily. Indeed, if you have external file containing a bitmap in the required format, with the filename `image.img`, you can include it like this: First, create an assembly source file, `src/dev/image.S`, like this:

```
.global image_start
image_start:
.incbin "image.img"
```

This includes the contents of `image.img`, labeled as `image_start`. Next, include this file in your compilation process (modify the `Makefile` in the same directory). Finally, in your C code, you can now do the following:

```
extern int image_start;  // refer to the label in C, type doesn't matter

int my_func() {
   nk_gpu_dev_bitmap_t *my_bitmap = (nk_gpu_dev_bitmap_t*)&image_start;
```

To create an `image.img` file you can use `~cs343/HANDOUT/bitmapify.pl` which takes in normal image files and outputs them in the specified format. Then you can place the assembly source file in `src/dev/` and place the `.img` file in the root of the repo.

13 Task 6: Enforce clipping

Normally, when you draw, the drawing is limited to the bounds of the framebuffer (`frame_box` in our device state) and thus the screen. An attempt to modify a pixel that is at a coordinate that is outside of these bounds is ignored. *Clipping* extends this concept to allow you to restrict drawing to any arbitrary box. One example use of clipping is within a windowing system, where it is used to restrict a thread from drawing outside of its window. For more information on clipping, see
`https://en.wikipedia.org/wiki/Clipping_(computer_graphics)`.

The function `graphics_set_clipping_box`, which is where you should start, is pretty simple. It just needs to copy the requested clipping box to the device state—in fact, there is already a field for just this purpose, `clipping_box`.

The challenge here is to enforce clipping in the functions you have previously written. No drawing command should affect a pixel outside of the specified clipping region. The enforcement of this policy will need to be added to all of the code you have previously written. While there obviously many simple implementations of clipping, a *good* implementation will modify boxes or coordinates so no drawing is *attempted* outside of the selected area.

You should test your implementation by adding additional code to the `gputest` that demonstrates the creation of a clipping box and that subsequent drawing operations are indeed clipped properly.

In a more advanced 2D GPU, clipping would be done by the GPU itself, in hardware.

14 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 the 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`, `src/nautilus/gpudev.c` and any other files you are changing, for example for embedded bitmaps.

We will test your Parallel Port code by running `chartest`. We will test your GPU code by first running your `gputest` code and then testing with our own. Make sure that all portions of the GPU tasks are demonstrated in your `gputest` code. Feel free to animate it.

The breakdown in score will be as follows:

- 20% Task 1—Functional and sensible implementation of parallel port driver. It must pass `chartest` successfully.
- 15% Task 2—Functional and sensible implementation of Virtio GPU mode switch and SW/HW pipeline setup in `set_mode`.
- 15% Task 3—Functional and sensible implementations of `flush`. Your framebuffer should display correctly on the screen.
- 20% Task 4—Functional and sensible implementations of basic drawing functions, specifically `graphics_draw_pixel`, `graphics_draw_line`, and `graphics_draw_poly`. You should have corresponding test code in `handle_gputest`.
- 20% Task 5—Functional and sensible implementations of the box-related drawing functions, specifically `graphics_fill_box_with_pixel`, `graphics_fill_box_with_bitmap`, and `graphics_copy_box`. You should have at least one embedded bitmap that you draw to the screen. You should have corresponding test code in `handle_gputest`.
- 10% Task 6—Functional and sensible enforcement of clipping for the previously implemented functions. You should have corresponding test code in `handle_gputest`.

We may supply you with a test application, such as a game, that uses the various features you will implement to work correctly. If we do this, we will provide it as a patch to you.

15 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 us with a plan. We can give you more details and make this less daunting. Some possible extra credit concepts are the following:

- Implement text drawing in 2D graphics mode. Note that this is already part of the interface described in `gpudev.h`. Note `nk_gpu_dev_font_t`, which is a bitmap of 0/1 pixels for each character.
- Implement the hardware mouse cursor. Virtio GPU can give you control over the shape and location of the mouse cursor through transactions. `gpudev.h`'s abstraction includes such a feature. Be aware that the mouse code is pretty finicky and may not work if the mouse is not currently captured by the QEMU window, and may have issues with FastX.
- Implement clipping *regions*. These are arbitrary collections of pixels what the display is clipped to, rather than a box. One use of clipping regions is to support drawing into a partially occluded window, for example a window with another window positioned to be partially over it. There is already room in the interface for this concept. A typical implementation might use a *quadtree* data structure.
- Port a graphical user interface (GUI) to the `gpudev` abstraction. At some point, a port of μGUI was made to NK for use with an older GPU driver (`src/dev/vesa.c`), so we know this can be made to work pretty cleanly.