CS 194-24 Lab 0: Introduction

Palmer Dabbelt

January 23, 2013

Contents

The purpose of this assignment is to familiarize you with the development enviornment that you will be using for the remainder of this class. Note that while this assignment **must be completed individually** you will be working in groups for the vast majority of this class. You should form your groups as soon as possible.

1 Virtual Machine Configuration

In order to minimize the pain of setting up a development enviornment we have provided you with a virtual machine image that contains all the tools you will be using in this class. You will be doing all of the development for this class inside of the provided virtual machine.

It's probably best to have a little overview of the enviornment you're about to setup before delving into the setup. You'll be using a VMWare virtual machine as your development enviornment. This VM contains all the tools that will be necessary to complete this class, you'll be spending all of your time inside that virtual machine.

In order to test the kernel modifications you will be making for this class you will be using a second VM, this one based on QEMU. This QEMU VM lives inside the VMWare VM, just like every other tool used in this class. The QEMU VM allows you to boot a kernel containing your modifications without risking the filesystem on your workstation. Figure 1 shows how these two virtual machines are nested.

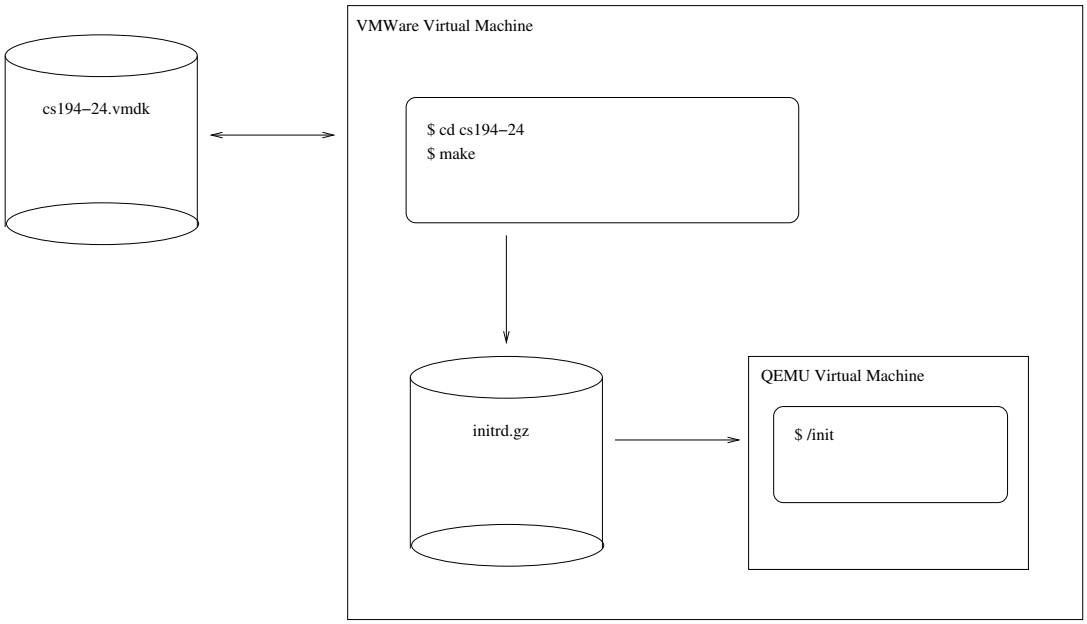

Figure 1: Virtual Machine Layout

1.1 Installing VMWare Workstation

While the virtual machine image we've provided is in a standard format, we can only support running it inside of VMWare Workstation 9. VMWare is already installed on the lab machines designated for this course. It's recommended that your setup VMWare on your personal machine as well.

Obtain the VMWare installation package that cooresponds to your operating system (note that the program will be called "VMWare Fusion" on Mac OSX) and install it on your computer. The Windows download site for VMWare is here

`http://www.vmware.com/products/workstation/overview.html`

while the Mac OS download site is here

`http://www.vmware.com/products/fusion/overview.html`

The EECS department provides VMWare Workstation 9 licenses to all students registered for this class. Unfortunately, we haven't gotten this setup yet so you'll have to apply for a 30-day trial for now. You can apply for a trial key online

`https://my.vmware.com/web/vmware/evalcenter?p=vmware-workstation9`

1.2 Configuring Your Virtual Machine

Download the provided VM image from the class website. As the image is quite large, it's probably best to do this over a wired connection. The EECS department provides wired AirBears connections that will allow you to quickly fetch the image to your laptop. You can obtain the VM image from the following URL:

`http://www.eecs.berkeley.edu/~palmer.dabbelt/cs194-24-vm.zip`

The virtual machine should come almost entirely pre-configured. The machine will boot to a graphical login, at which point you should be able to login using the username "`user`" and the password "`user`". At this point all of your work should be done within VMWare.

2 Linux

The bulk of your programming assignments in this course will consist of hacking on Linux. In this section you'll learn how to build and boot a Linux kernel.

2.1 Building a Linux Kernel

Boot the provided virtual machine and login to the user account. Inside your home drive there will be a `cs194-24` folder which contains all the source code you will need for this class (aside from what you will be writing for assignments, of course).

The Linux sources we've provided you have been almost fully configured. Linux uses its own build configuration tool, so in order to familiarize yourself with it your first assignment will be to configure a build with debugging symbols enabled. You can bring up the Linux configuration utility by running

```
$ make -C linux menuconfig
```

A menu of configuration options should appaer. In order to enable debugging symbols, you'll first need to navigate to the "`Kernel hacking`" submenu. You then need to enable "`Kernel debugging`" and "`Compile the kernel with debug info`". If you pay close attention, you'll notice that the option for "`Compile the kernel with debug info`" only appears after you enable "`Kernel debugging`". This happens because the debug info option depends on the debugging option.

It can often be tricky to discover the correct set of dependencies to enable in order for an arbitrary option to appear in the kernel configuration menu. Luckily there is a search feature. You can bring up the search dialog by pressing the "`/`" key. By searching for "`DEBUG_INFO`" you can both the location of that particular configuration option as well as any dependencies it may have.

Exit from the configuration utility and save your changes (you will be prompted when you try and exit). You can build a kernel that uses your new configuration by running

```
$ make linux
```

2.2 Building a Busybox userspace

The above steps build an Linux kernel image, but that itself is not particularly useful. In order to actually use this kernel we need some userspace tools. We will be using the Busybox userspace tools in this class. Busybox is an implementation of the standard POSIX tools that are designed to be as compact as possible, and as such are commonly found in embedded devices. The Busybox project uses the same build system as the Linux kernel. We've already provided a Busybox configuration, so you can build Busybox by running

```
$ make busybox
```

2.3 Building an initrd

In order for the Linux kernel to be able to access the Busybox tools we need to provide a filesystem that contains those tools. In Linux terminology, this is known as an `initrd` (an INITialization Ram Disk). For this class, we'll be using a CPIO formatted archive as our initrd because Linux provides a program to generate properly formatted initrds. Just like everything before, the Makefile is capable of building an initrd for you.

```
$ make .lab0/interactive_initrd.gz
```

2.4 Booting Linux in QEMU

In order to test the operating system you've just built without having to reboot your development machine we're going to use yet another virtual machine. In this case we'll be using QEMU because it is simple to script and runs efficiently when nested inside of VMWare. In order to boot the Linux install you just build you can run

```
$ make
$ ./boot_qemu
```

After a few seconds a QEMU window should appear inside of which you should see Linux boot. After a few seconds, a shell will appear where you can interact with all the code you just built. The file `lab0/interactive_config` contains a list of all the tools availiable inside this interactive enviornment. You can add additional tools by following that pattern.

2.5 Debugging Linux with GDB

While it's possible to use GDB to debug a running Linux kernel, it is a bit quirky. Luckily, a script has been provided that works around most of these quirks for you. Running

```
$ ./boot_qemu --debug
```

will boot up your Linux kernel under QEMU with a GDB session attached to it. The GDB session will launch in a seperate terminal window which you can use to interact with Linux. At this point, using GDB to debug the kernel is very similar using GDB to debug a userspace application. If you are unfamiliar with GDB then now would be a good time to read the manual – GDB will prove to be an invaluable tool during this class.

3 Cucumber

For this class we will be using cucumber for automated testing of your kernel changes. In this section we will be walking you through implementing and testing a simple feature. In the spirit of Cucumber, we'll be trying to use test driven development for this class – this means we'll be writing the tests for each feature before we implement it.

It's important to note that we'll also be using some super-secret Cucumber tests for grading your assignments, so it's very important for you learn how to write your own tests. More information on Cucumber can be found in the one of the required books for this class.

Cucumber has already been insntalled as part of the provided VM image. We have provided a simple set of example Cucumber tests for your first assignment. You can run the provided tests by typing

```
$ cucumber --tags @lab0.0
```

3.1 Writing a Cucumber feature

Cucumber isn't particularly useful without any tests to run. Over the course of this class you will be adding many additional features to the Linux kernel, each of which will need to have at least one test. Cucumber organizes each feature into a single file that can contain multiple tests.

As a simple, illustrative feature we will be adding an extra version number to the Linux kernel. Linux prints out the version number when booting, so this will serve as a good demonstration of how to get data out of a VM.

We've already created the Cucumber file that cooresponds to this feature. The file resides at `features/lab0.0-extra_version.feature` inside your VM and contains the text shown in Figure 2.

```
@lab0.0
Feature: CS194-24 Version String

    Seeing as how we'll be running modified Linux kernels for this
    class, we're going to go ahead and tag our sources with an extra
    version number to distinguish them from the mainline.

    Scenario: Check Version
        Given Linux is booted with "--initrd .lab0/version_initrd.gz"
        Then the extra version should be "cs194"
        And Linux should shut down cleanly
```

Figure 2: A simple Cucumber feature

Cucumber doesn't inherently know anything about these human-readable strings, it instead just provides some pattern matching primitives that you need to fill out in order to provide meaning to each statement. These patterns are called "feature definitions". For this assignment, the feature definitions have been provided – they reside in the "`features/step_definitions`" folder. You should look at these files to get a sense of how Cucumber works.

This pattern matching code is written in Ruby and heavily uses regular expressions. If you're not familiar with either of those concepts you should read up on them, they'll be used through the class.

3.2 Implementing the feature in Linux

You should now be able to run `cucumber` and see the single test case fail. It's time to write some code that makes said feature actually work.

For this simple test case it's actually exceedingly easy: you only have to change a single line in the Linux Makefile. Simply set the "`EXTRAVERSION`" variable to "`cs194`". The following patch describes the change

```
diff --git a/linux/Makefile b/linux/Makefile
index fbf84a4..c60d949 100644
--- a/linux/Makefile
+++ b/linux/Makefile
@@ -1,7 +1,7 @@
 VERSION = 3
 PATCHLEVEL = 7
 SUBLEVEL = 1
-EXTRAVERSION =
+EXTRAVERSION = cs194
 NAME = Terrified Chipmunk
```

If you don't know the `diff/patch` format, you sould learn it now, it's going to prove exceedingly useful for this class.

At this point you should be able to run the test case and see it pass. You can run the test case by simply typing

```
$ cucumber
```

4 Git

For this class we will be using the git version control system, which is a common open source VCS. If you are completely unfamiliar with git now would be a good time to read some documentation. You'll want to me familiar with the terminology used by git as you'll be using git to submit your assignments this semester. We'll be using a fairly advanced git configuration here: you'll need to understand how multiple branches and multiple remotes work.

4.1 Repository Management

You will find that the provided `cs194-24` folder in the virtual machine's home directory is a git repository. There are three important repositories you'll need to know about for this class:

- `gitolite@ist-1.eecs.berkeley.edu:public.git`: The public repository for this class. New assignments will be distributed via commits to this repository. The git repository in your virtual machine already contains a remote setup called `public` for this repo.
- `gitolite@ist-1.eecs.berkeley.edu:student-private-storage/USERNAME.git`: Your personal repository, stored on the class's server. You are the only person (aside from class staff) who was read or write access to this repository. This is where you'll be submitting this assignment, and is a place to stage work before showing it to your group. This will be refered to as the `personal` repository in the future.
- `gitolite@ist-1.eecs.berkeley.edu:group/GROUP.git`: Your group's repository, stored on the class's server. Only your group members (in addition to class staff) have read or write access to this repository. This will be refered to as the `group` repository in the future.

All these branches and repositories are managed by a centralized Redmine server that you can login to using your university credentials. The management server is located at `https://ist-1.eecs.berkeley.edu/projects/`.

Login using your university username and password (your username is your email address without the domain). The remainder of this section will walk you through setting up your virtual machiane such that it is able to communicate with the class's Redmine server.

4.2 Branching Strategy

We will be using git to manage your assignment submissions for this class, which means the course staff will have access to your entire commit history. In order to keep things organized, we've decided to enforce a particular branching strategy. Your git repository should contain at least two branches, but you can have as many as you feel are useful.

- `master`: Your default development branch. Commits that exist solely on this branch will not be considered for grading. `master` is a good place to put code that works, but that you're not ready to have graded yet. For example: if you're working on an assignment and you've written some Cucumber features but haven't yet implemented them in Linux, `master` would be a good place to push that commit.
- `release`: The branch of commits to be graded. When you commit to this branch, a run of the autograder will be triggered. It's important to keep this branch clean both because we will be grading your commit history and because you don't want to overload the autograding server.

We're also going to ask that you provide merge messages when you merge to the `release` branch, which will make sorting out your solutions to each assignment much simpler. The following command will merge `master` to `release` while allowing you to type a merge message. A sane merge message might be something like "solution to lab 0".

```
$ git checkout release
$ git merge master --no-ff
```

Merging in this way will provide a single commit for each solution you expect to be graded, which will make tracking the results of your autograder runs much simpler. If you follow this merging strategy then you can get a list of all your submissions by running the following command.

```
$ git log release --merges
```

4.3 Git Configuration

Before you can actually submit your assignment, you'll first need to configure your git installation. You'll first need to inform git of your name and email address so it can include these in your commits. Be sure to use your real name and email address, as this is what will identify you to your group members in the next assignment. Input your name and email as follows:

```
$ git config user.name "My Name"
$ git config user.email "me@eecs.berkeley.edu"
```

You'll now need to generate a SSH keypair. SSH keypairs, like all public key encryption schemes, consist of two parts: the public key and the private key – if you're unfamiliar with how to use SSH public key encryption you should read up on it now. In SSH terminology, the public key will be named with a "`.pub`" extension. You can generate a SSH keypair by running

```
$ ssh-keygen
```

which will prompt you a few times. If you select all the defaults you'll end up with a private key located at `$HOME/.ssh/id_rsa` and a public key located at `$HOME/.ssh/id_rsa.pub`. It's important to keep your private key to yourself, but your public key is not sensitive.

Go to the class Redmine site and add your public key to the list of keys that have access to your account. That list lives inside your account settings (there's a link on the top-right after you login).

You'll now need to add your personal git repository (which lives on our Redmine server) as a remote of the repository that lives in your VM. On the class's Redmine server, find your project (there is a projects link at the top of the page, you project will reside within the "Students" project).

Find the repository URL on your project page (it's on the left side, about half the way down), and add it to the repository that lives inside your VM by running something like

```
$ git remote add personal gitolite@ist-1.eecs.berkeley.edu:student-private-storage/USERNAME.git
```

4.4 Submitting Changes for Grading

As previously mentioned, we will be using git to submit code in this class. As you now have a small feature implemented (the extra version number you added previously), now is a good time to test the class' submission mechanism.

You'll first want to commit your code to the master branch. Be sure to use an informative commit message as it will be visible to the course staff.

```
$ git checkout master
$ git add .
$ git commit
```

You now need to create a merge commit on your release branch. While this is overkill for a single commit, it will become useful later when your implementations consist of many commits

```
$ git checkout release
$ git merge master --no-ff
```

At this point you have everything committed to your local repository, all that's left to do is inform the Redmine server about your changes.

```
$ git push personal master
$ git push personal release
```

In a few minutes you should get an email that summarizes the results of your test run. If your results did not pass the test suite then you should fix your error now, otherwise continue on with the remainder of the assignment!

5 A simple Linux modification

In order to test your knowledge of the toolset (ie, before you go and write any real code), it's time to make a simple modification to Linux without step-by-step guidance. You'll first want to merge the `lab0.1` branch from the class's git repository into your repository

```
$ git fetch public
$ git checkout master
$ git merge public/lab0.1
```

This branch contains some test cases for a new feature you'll want to add to Linux. While the provided Cucumber tests should contain enough information to implement everything required for this assignment, some form of introduction is always useful. When you are done with the assignment be sure to merge and submit as described earlier in this document.

5.1 `init` with shared libraries

You may have noticed that we are building a staticly linked Busybox executable for inclusion in the initrd used to boot your Linux kernel. This is done in order to avoid needing to copy any shared libraries into the initrd, thus making the process of creating it significantly simpler.

Whenever I setup a new Busybox install, I invariably forget to set the "build a static binary" flag in the Busybox configuration file. When I then go to boot the machine, the Linux kernel panics because it cannot load the `init` process because of these missing shared libraries. While this panic is expected, the message attached to it is not even remotely helpful

```
[ 0.790125] Kernel panic - not syncing: No init found.  Try passing
 init= option to kernel. See Linux Documentation/init.txt for
 guidance.
```

As you can see, the kernel message suggests that no `init` exists. This error message is actually incorrect: a `init` exists, the kernel is just unable to load it. A significantly more informative panic message would be something like

```
[ 0.801769] Kernel panic - not syncing: /init: unable to load library
 or interpreter
```

For this assignment you must modify the Linux kernel such that it produces a proper error message for this failure condition. The kernel will still panic, as this really is an unrecoverable condition, but it will at least provide the user with a correct error message.

6 Fish

The remainder of this assignment consists of a proper lab. As most of the design work has been done for you, you will not be required to write any supporting documentation (aside from your git commit history) for this lab.

For this assignment you will be implementing a system call that allows you to draw an animation on the screen. You will be required to understand system calls, memory-mapped IO, some Linux scheduling perimitives, and the Linux build infastructure.

You'll want to start out by merging the `public/lab0.2` branch into your repository. In addition to testing and build related code, you'll find the following source files to be relevant:

- `lab0/fish_compat.h`: Defines a compatibality layer that allows the same code to be used inside the kernel and the userspace test harness.
- `lab0/fish_impl.{c,h}`: The core implementation of the Fish drawing API. The vast majority of the code you're going to write will end up here.
- `lab0/fish_syscall.{c,h}`: Contains a single function `fish_syscall()`, which allows you to emulate system calls (with a function call) when running in userspace, and allows you to actually make system calls when running in the kernel. You must implement this without using the C library's `syscall()` functionality.
- `lab0/sys_fish.c`: Kernel space implementation of the Fish system call. You'll have to implement the two stubbed out functions in here, but not anything else.
- `lab0/test_harness.c`: This code interfaces between Cucumber and the rest of the code you'll write for this assignment. Any modifications to this file will be overwritten by the autograder, but if you want to support your additional tests by modifying this file then that's fine.

6.1 Userspace Implementation

Kernel development tends to be significantly more difficult userspace development, so it's best to test your code in userspace whenever possible.

For this assignment you have been provided with a test harness that aims to let you share exactly the same code between your kernel implementation and your userspace implementation. In order to make this work you'll want to be sure to take a look at `lab0/fish_compat.h`, which contains the code that facilitates this sharing. Specifically, you'll want to use `kmalloc()` and `kfree()` instead of `malloc()` and `free()` to allocate memory.

6.1.1 The Fish API

The system call you'll be implementing provides an API that can be used to draw simple animations to the screen. In order to support the required functionality while being minimally invasive to the Linux kernel, every function availiable to userspace will be multiplexed onto a single system call – this is a pretty standard practice, see the `prctl()` system call for a real world example.

You will be making extensive use of the blink structure in this assignment, which is shown below

```
struct fish_blink
{
    /* The position of this character on the screen */
    unsigned short x_posn;
    unsigned short y_posn;

    /* The given location will iterate between these two characters. */
    char on_char;
    char off_char;

    /* The period of the iteration. */
    unsigned short on_period;
    unsigned short off_period;
};
```

You'll have to implement the following functions, all of which return 0 on success and -1 on failure:

- `long fish_clear(void)`: Clears the entire screen. This is used to overwrite any prior drawings that were on the screen. It also serves to clear any remaining boot text from Linux's framebuffer, which will interfere your animation. When text is cleared it must be set to grey text on a black background. Note that you must clear text by writing out spaces, not by modifying the text such that it is invisible.

- `long fish_add(struct fish_blink *to_add)`: Takes a pointer to a blink structure that contains all the information required to describe how a single screen location is animated. When the `fish_add()` call is made, the `on_char` will be written to VGA memory at the location specified by `x_posn` and `y_posn`. After `on_peroid` ticks have fired, `off_char` will be written to the screen for `off_peroid` ticks. This must then cycle between `on_char` and `off_char`, which allows for some simple animation – we'll refer to this as a blinking location from now on.

  It's important to note that changes to the blink structure do not propogate backwards to calls to `fish_add()`. In other words

  ```
  blink_struct.on_char = 'a';
  fish_add(&blink_struct);
  blink_struct.on_char = 'b';
  ```

will show "a" on the screen instead of showing "b".

Attempting to add to a location twice is an error.

- `long fish_remove(short x, short y)`: Takes a screen position of a location that should no longer be part of the animation – effectively this entirely undoes `fish_add()`. After this call the cooresponding location should be blank and should not blink until a call to `fish_add()` is made again. Attempting to remove a location that does not have any blinking character is an error.
- `long fish_find(struct fish_blink *to_find)`: This takes in a blink structure and only looks at its position fields. It then searches the animation for a blinking character at that location, if that exists it copies the {on,off}_{char,period} from the animation into the given structure. Searching for a location that has no matching structure in the animation is an error.
- `long fish_sync(short fx, short fy, short tx, short ty)`: This synchronizes two locations on the screen. Two locations are synchronized when they are always on at the same time and always off at the same time, their `on_char` and `off_char` fields should not be changed. The location designated by the first cooridnate pair should overwrite the location designated by the second cooridnate pair. Attempting to sync from or to a location that is not blinking is an error.
- `long fish_start(int i)`: Enables the tick source designated by `i`. In this assignment you will have two tick sources: a periodic interrpupt from a hardware timer and a manually triggerable interrupt that comes from a system call, but your solution should be generalizable to an arbitrary number of sources.
- `long fish_stop(int i)`: Disables the tick source designated by `i`.
- `long fish_tick(int i)`: If the tick source designated by `i` is enabled, then count one tick for every blinking location on the screen. Otherwise simply return: calling `fish_tick()` on a disabled tick source is not an error.

6.1.2 VGA

The VGA standard has evolved from the original display hardware present in the first IBM PC, which means it's ubiquitous on modern PC hardware. There is a very complete VGA programming reference availiable online at `http://www.osdever.net/FreeVGA/vga/vga.htm`.

For this assignment we'll be using the same text-mode configuration that Linux uses, so you don't need to worry about messing with the VGA configuration registers. You do, however, need to set the color bits properly for each character – you can find those settings in the above reference.

The text mode VGA memory layout is actually quite simple. The VGA framebuffer is mapped by your system's memory controller directly into the processor's address space, which means you can write to the screen using regular store instructions. Code is provided to map VGA memory such that it is accessibly by your code, it's mapped to `vga_mem`.

The VGA framebuffer is 80 characters wide and 25 characters high. Each character is represented by 16 bits. The first 8 bits represent the character to be displayed, and the remaining 8 bits are flags that control how the character is to be displayed – hopefully Figure 3 is illustrative of this.

6.1.3 Userspace Testing

At this point your should have sufficient code written such that your code will work when run from the userspace test harness. Booting your kernel with `.lab0/user_initrd.gz` will allow you to interact with the test harness. In order to prevent the Fish code from trashing the terminal you use to interact with QEMU, it's recommended that you boot the kernel with

```
$ ./boot_qemu --initrd .lab0/user_initrd.gz --stdio
```

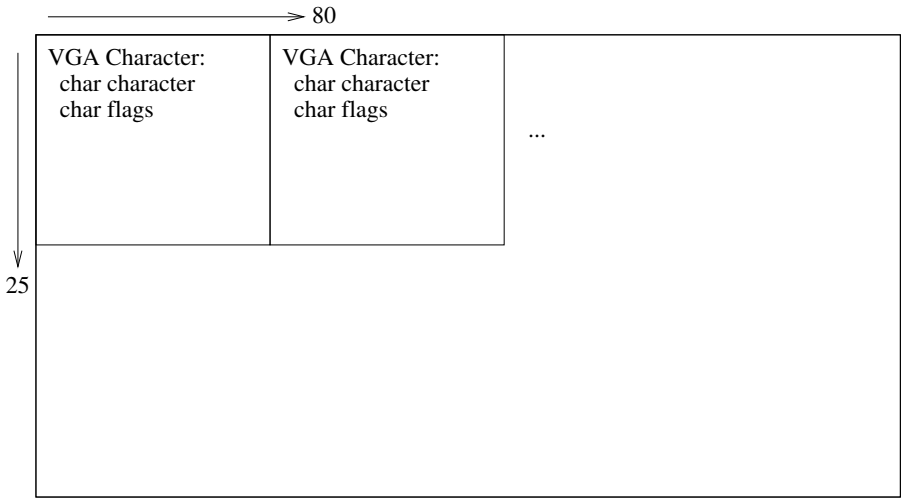

Figure 3: VGA Memory Mapped IO

The included "`demo`" command is probably the most useful test for your implementation. The demo runs the following set of commands

- Clear the screen
- Add `/frame0.txt` and `/frame1.txt`
- Tick for 10 seconds
- Change a blank character for a character that toggles between `I` and `M`, out of sync with the rest of the fish
- Tick for 10 seconds
- Sync the `I/M` character with the rest of the fish
- Tick for 10 seconds

which is a fairly solid test of the fish system calls. You should, of course, go ahead and write proper Cucumber tests for this functionality – that's how we'll be grading you!

6.2 Kernel Implementation

Once you have an implementation that works in userspace, it's time to make your code work in the kernel. Assuming you don't have any bugs, this should be a pretty simple matter. That said, there are a few common bugs you're going to have to worry about:

- The in-kernel code gets ticks via an interrupt, which means they're asynchronus. You have to take this into account when you update your internal blink location database.
- You have to make sure to properly use `copy_to_user` and `copy_from_user`.
- You'll now be using system calls instead of function calls.

6.2.1 Kbuild

We've setup `lab0/Makefile.mf` to copy the relevant sources into the kernel source tree, but we haven't told the kernel how to build them yet. You'll need to modify both `linux/drivers/gpu/vga/Kconfig` and `linux/drivers/gpu/vga/Makefile` in order to make this happen. While there is a bunch of relevant information `linux/Documentation/kbuild`, you'll probably be fine with just following the pattern present in those files.

After informing Linux about your new configuration option, you'll need to update your kernel config. If you simply run

```
$ make -C linux
```

it will prompt your for any new configuration options – in this case, the one you just added. You'll want to enable this new feature, otherwise it won't be particularly useful.

6.2.2 System Calls

While you code should now build, it won't actually do anything yet. In order to allow the test harness to communicate with your new driver, you'll need to implement a bit of system call infastructure.

First, you'll want to add an entry for `sys_fish` to the x86 system call table. This table is located at `linux/arch/x86/syscalls/syscall_64.tbl`. By adding this entry to the system call table Linux will know to call `sys_fish` when a system call of the number your choose comes in. You'll want to be sure to assign a unique number for `sys_fish` and to duplicate that number into `fish_impl.h`.

Next, you'll need to implement `sys_fish`. All you'll need to do here is to call the correct function from `fish_impl.c`, so it should be pretty straight-forward. You'll want to be sure to handle any invalid Fish function types by returning -1 here.

Finally, you'll want to implement the `fish_syscall` function. For this class you will not be allowed to use glibc's `syscall()` function, which means you'll have to write some inline assembly at some point.

6.2.3 Scheduling a Tasklet

In order to draw animation we will need to interface with some sort of real-time clock so your code can flip frames at a consistant rate. While most hardware that needs periodic interrupts contains some sort of internal timer, this adds some unintended functionality to the VGA driver so you'll have to piggyback off another periodic interrupt.

The first thing you'll need to do is find such a source of periodic interrupts. A hint: `sched_clock_tick()` (inside `linux/kernel/sched/clock.c`), if you break on that function a backtrace will show you where your interrupts are coming from.

Once you've found your source of interrupts, you'll need to add some code to update the screen. You don't want to directly add a call to `fish_tick()` into the timer interrupt path because writing to VGA-mapped memory can be very slow. Instead, you should schedule a taskelet during the timer interrupt that does all the long running work. You should be able to read all about tasklets in your Linux book, but the general idea is simple: rather than running a slow process during an interrupt handler, you instead create a schedulable entity that runs that slow process.

`fish_tasklet` is already defined inside `lab0/fish_syscall.c`. First, you'll want to add code to update the screen inside this tasklet. Next, you should add code to schedule the tasklet inside the timer interrupt that you found above. You can schedule `fish_tasklet` by running

```
tasklet_schedule(&fish_tasklet_struct);
```

Note that you'll need to declare the tasklet first by adding the following anywhere in the same C file you schedule the tasklet from

```
extern void fish_tasklet(unsigned long);
static DECLARE_TASKLET(fish_tasklet_struct, fish_tasklet, 0);
```

6.2.4 Kernel Testing

At this point, the provided demo program should run inside the kernel in exactly the same manner as it does in userspace. You can test your in-kernel Fish implementation by running

```
$ ./boot_qemu --initrd .lab0/kernel_initrd.gz --stdio
```

which behaves in the same manner as the userspace test harness – again, the `demo` command is a good first test.

6.3 Submit

You should submit your code by merging to the release branch and then pushing to your personal repository. You can do this by running

```
$ git checkout release
$ git merge master --no-ff
$ git push personal release
```