

CS 343 Operating Systems, Winter 2022

Getting Started Lab

1 Introduction

In this lab you will get, build, and run an operating system kernel, plus attach a remote debugger to it. This lab must be done individually. Any clarifications or revisions will be posted to Campuswire.

The purpose of this tiny lab is to make sure that you have everything set up so that you can do class labs. **If you are having problems, please post questions to Campuswire so we can assist you.**

2 Task 1: Remote display

You can work on this lab on any modern Linux system, but we strongly suggest you do this lab on our class server `moore.wot.eecs.northwestern.edu` first.¹

We need you to have remote display capability from the server. There are several ways to do this, including FastX², VNC, or X11. We will post more details about this on Campuswire. When you have this set up, you'll be able to do something like this:

```
client> ssh -Y you@server # NOT for FastX
server> emacs &
```

with the result that a new text editor window pops up on your client. Note that the first line is not needed for FastX, only X11. In FastX, just open a terminal window in your FastX browser session (click on Red Hat / Activities, then select terminal icon) — this is `server>`.

Note: in order for some of the following commands to execute properly, we assume you are using the `bash` shell on the server. If you aren't (and by default you are not), you can start it by running

```
server> bash
```

3 Task 2: Setup

We will describe the details of how to access the lab repo via Github classroom in lecture and on Campuswire. You will use this information to clone the assignment repo using a command like this:

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

¹For students who definitely want to work on their own machines, we will give guidance on Campuswire.

²This is the preferred way to set things up for this quarter. See <http://it.eecs.northwestern.edu/info/2020/09/14/info-labs-fastx.html>

It is very important that you use the ssh URL (and not the https URL) for this and other labs in the class.³ This will give you the entire codebase and history of the Nautilus kernel framework (“NK”). This is an actively developed research tool among several institutions, including Northwestern.

4 Task 3: Build it

To start, you will need to configure the kernel:

```
server> cd [assignment directory cloned from Github]
server> cp configs/cs343-base-config .config
```

To compile the kernel and produce a bootable disk, do the following in your assignment directory:

```
server> source ENV
server> make -j 8 isoimage
```

The end result of this should be a file `nautilus.bin`, which is the kernel, and `nautilus.iso`, which is the bootable disk. Run these commands and **capture the results** (see [Section 7](#)):

```
server> ls -ltr | tail -5
server> md5sum nautilus.bin
```

5 Task 4: Run it

To run the kernel in emulation, execute:

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

Note: if `source ENV` fails. You probably forgot to start a bash shell. See [Section 2](#).

You should see a new window pop up, with content that looks like a computer booting. You will also see a bunch of output (boot messages) show up in the terminal where you ran `./run`.

Eventually, the new window will go blue and present you with a prompt that says `root-shell>` This is the command prompt of an NK shell. Type the following commands:

```
root-shell> cpuid 0 3
root-shell> mem fd520 128
```

The first of these commands shows information about your processor. The second dumps out a part of the system firmware (the BIOS). **Save the results** (you will see everything duplicated in the terminal, where it is easy to copy and paste).

Feel free to play with it more. Type `help` to see commands. Note that this is not a Linux shell. This is a minimal interactive interface that is running *within* NK. You can do anything with full privilege and very easily nuke the kernel. Try `int 0 23` for an easy panic.

³The ssh url will look like `git@github.com:...` You will need to set up ssh public key authentication for your github account—and you should learn how to do this in any case. Why no https? Github no longer supports pushes to an https URL, which is needed to hand things in in this class. If you make a mistake and use the https URL, you don’t need to panic—we can help you fix this after the fact, but it’s a lot easier to not have to fix it.

6 Task 5: Run it with a debugger

In this final task, you will attach `gdb` to do remote debugging of the kernel. First, while `gdb` uses a default port to attach its debugger to, if every student in the class uses the default port, they will all conflict with each other and the operation will fail. In your assignment directory run:

```
server> ./mynumber_cs343.pl
```

That will provide you with a unique number `N` based on your username that you should use for the following GDB commands.

Start `NK` using the `run` command. While in the blue window, type `CTRL-ALT-2`. This will switch to a black screen with a `(qemu)` prompt. This is a command interface for the emulator on which the kernel is running. Run

```
(qemu) gdbserver tcp::N
```

where `N` is your unique number. Now type `CTRL-ALT-1`. This will switch you back to the blue window (`NK`). Now, in a separate window on the server, attach to it:

```
server> gdb nautilus.bin
(gdb) target remote localhost:N
```

where `N` is your unique number. Finally, run the following `gdb` commands and capture what they show:

```
(gdb) info threads
(gdb) bt
(gdb) x/s 0xfd7fd
```

The first command shows the hardware threads of the emulated environment on which the kernel is running. The second shows the the stack trace for the currently selected hardware thread. The last command dumps out memory at the given address as a string. This string is within the system firmware (BIOS).

7 Submission and grading

To hand in your work, create a file called `STATUS` in the repo you have checked out from Github classroom. Place the outputs we told you to capture in Tasks 3, 4, and 5 into this file. Now add the file to your repo, commit it, and push:

```
server> git add STATUS
server> git commit -m "Done!"
server> git push
```

That's it! We will look at your `STATUS` file to confirm you've successfully completed these steps.

The purpose of this tiny lab is to make sure you have everything in place for future labs. We will help you do this if you have issues. The grade for this tiny lab is therefore all-or-nothing.

8 Other stuff you can do if you are curious

If you got this far, you are done. You don't need to do anything else. If you'd like to learn more about this codebase and related concepts, keep reading.

By default, the build infrastructure⁴ for NK hides the details of the commands it is running. You can see them by adding `V=1` to the command line, like this:

```
server> source ENV
server> make V=1 clean
server> make V=1 isoimage
```

NK is a statically configurable codebase, meaning that the developer can choose which pieces are included, how various features are configured, and more. You can play with the configuration like this:

```
server> source ENV
server> make clean
server> make menuconfig
```

This will show you text-based menus that allow you do change the configuration. Once you are done, you can run

```
server> source ENV
server> make isoimage
```

to build the codebase for your new configuration.

In this class, we will run NK in a specially built version⁵ of the QEMU system emulator that is installed on moore. You can also run QEMU on your own machine by installing QEMU. After you install QEMU, you will run it as in the `run` script. QEMU creates an emulated machine with whatever options you ask for (what kind of CPUs, how many, how much memory, different I/O devices, etc) from its command line. The `run` script is just using some baseline options.

You can also run NK under a virtual machine monitor, like VMWare or VirtualBox—the `nautilus.iso` file is a bootable CD.

If you'd like to live dangerously, you can also try out NK on physical hardware. It should be able to boot on most PC hardware, though multiple processors and complicated memory systems can challenge it, and the static configuration has to be compatible with the hardware. One way to do this is to burn `nautilus.iso` onto a physical CD and then boot from that. Another is to write `nautilus.iso` onto a USB stick (it needs to overwrite the entire stick, not just be a file on the stick), and then boot from that.

If you look at `README.md` in your NK directory, you will see other information. This is displayed more pleasingly on the repo page on github.

The structure of the codebase is like this:

- This is a kernel for x64 machines that is written mostly in C, with a bit of x64 assembly. There is also assorted glue code for C++ and Rust, plus a few other research languages. The idea is to make it possible write kernel components in those languages.

⁴It is called KBuild, and is the infrastructure designed for the Linux kernel. It's used in numerous other codebases as well.

⁵Our version is "specially built" because we want it to not change/be updated, and because of the graphics part of the Device Driver Lab, which requires particular components in QEMU to work.

- The include directories parallel the src directories. So, for example, `include/nautilus/timer.h` is the header file that includes the timer abstraction, while `src/nautilus/timer.c` is the implementation of that abstraction.
- The 2nd-level within `src` largely reflects major components: `dev`=device drivers; `fs`=file systems; `gc`=garbage collectors; `nautilus`=core kernel; `net`=networking; `rt`=language runtimes; `test`=test code.
- Architecture-specific elements are (mostly) in `src/arch/[arch]/...`
- Assembly is (largely) in `src/asm`, though there are other `.S` files in the codebase, plus inline assembler.