

# CS169 Programming Assignment 1: Kernel 1

## Drivers and Initialization

---

*Assignment Out:* Monday, 22 Sept. 2008  
*Assignment Due:* Monday, 29 Sept. 2008 (11:59 pm)

---

## 1 The Assignment

Before reading this, make sure that you read the other handout on general assignment information.

The following is a list of the features which you will be adding to your version of Weenix. This assignment can be overwhelming at first, so we have decided to split it into two sub-assignments as opposed to giving you two weeks and letting you do your thing. This way, you should be spending appropriate time developing the lowest-level device IO in the first week and then building upon your code in the following assignment in the second week. However, the assignment is not complete until the entire kernel assignment works properly. We will ask you to demo your progress at the conclusion of this first week to ensure that the device drivers work properly.

In this section we present the steps required to get your kernel initialized and your drivers running.

- Initializing the kernel
- Low-level device drivers
  1. Terminal and line discipline code.
  2. Disk
  3. Memory Devices

Each of the functions is initially empty save for the macro `NOT_YET_IMPLEMENTED`. This macro will print out a message if you hit a function that you have not yet written. To see a list of all the functions that remain to be implemented, type `make nyi` in your root directory. Be sure to examine the stubs for each function carefully, the best documentation is found in the comments for each function. The Weenix Kernel Hacker's Guide may provide useful information.

## 2 Booting

Most of the boot process is handled for you by the simulator and TA code. The simulator sets up the machine context and calls the `kern_boot` function, which initializes the TA-provided support systems, and calls your `bootstrap` function in `boot.c`. Note that `kern_boot` calls `tvb_init`, which sets up the device drivers for the system. You will need to implement these configuration functions, which are explained later.

At this point we are still in the boot process, which means that we don't have a context in which to properly execute. We cannot block, and we cannot execute user code. The goal of the `bootstrap` function is to ultimately set up the first kernel thread and process, and to have it call the `idleproc` function within this new thread. Before you can do this, you will have to write code for processes, threads, and the scheduler (covered in the next assignment). For now, you'll be writing all of your test code in this half-booted state. When you've completed the drivers, you can progress to Kernel2 and upgrade to a multi-threaded kernel.

Functions you need to write in `boot.c`:

- `void bootstrap(void)`

## 3 Device Drivers

At this point you are generating building blocks for a functional kernel. For your kernel, you need only worry about terminal devices, disk devices, and some simple memory devices.

### 3.1 C Polymorphism in action! (Sort of...)

As you'll notice, term devices and disk devices are built on top of two device types: block devices and byte (or character) devices. As you look through the code and the structure definitions, you might notice that all of the more specific devices contain within them their more generalized types of devices (parent devices). These are not pointers but occupy memory that is part of the struct. This is by design; we need the more generic types to follow the specialized types contiguously in memory. Armed with this, by taking a term or disk struct we can access the struct fields of the parent type since the parent itself is a field of the child struct. Moreover, we can go from the parent to the child by using memory offsets. By taking the address of a parent (block or byte device), we can subtract the size of the rest of the term or disk struct and get the address of the specialized type. There is a macro provided called `CONTAINER_OF`. Look in the term device driver for examples of its use.

## 3.2 Terminal Device Driver

Each terminal device possesses a `term_device` structure. This structure contains all the state information and synchronization primitives necessary to operate a simple serial terminal. This portion of the code is difficult to get right, so be glad that it's given to you! Though you're not responsible for writing it, you should be very familiar with its contents. You should understand the simulator device documentation, which is in the simulator guide.

Make sure that you understand the difference between the low-level terminal driver and the higher level line discipline code. The device driver is responsible for the lower level calls that actually interact with the simulator. The read/write semantics that would cause text to appear in a terminal window are higher level and would deal with the `tty`, not the terminal.

You are given the low-level terminal code. Your job is now to write the higher level line discipline code that will read and write from the terminal. Your code should support echoing of characters and newlines.

Functions written for you in `drivers/term.c`:

- `void term_device_add(sim_devconfig_t *conf)`
- `void term_intr(int ipl, sim_context_t *ctx, void* info)`
- `void term_init()`
- `void tstart_provide_char(tty_device_t *tty)`
- `void tstart_provide_char(tty_device_t *tty)`
- `void term_block_io(tty_device_t *tty)`
- `void term_unblock_io(tty_device_t *tty)`

## 3.3 TTY and Line Discipline

`tty`<sup>1</sup> is the high-level terminal device driver for Weenix. This driver will sit atop the low-level driver given to you in this assignment and will provide the usual terminal semantics for read and write.

As you experienced when programming your shell, `read()` on a terminal blocks the calling thread until a newline from the user is encountered. In contrast to this, the `term_provide_char` given to you in `term.c` returned as soon as it had any input. Other functionality that Linux provided for you when you implemented your shell were echoing of characters and the ability to backspace.

---

<sup>1</sup>You'll get a cookie (read: nothing) if you can tell us what "tty" stands for.

Here you will wrap a high-level driver around the low-level driver to implement these features. The tty code will defer all of the read/write calls to its current line discipline, since this is what manages all of the read/write semantics. All of the character processing is done by the line discipline.

The driver you are writing for this assignment will eventually be accessed through the filesystem, specifically in the files `/dev/tty0`, `/dev/tty1`, ... To use the driver, you will need to open it with `do_open()`, which takes the same arguments as `open()`. Reading and writing are done with `do_read()` and `do_write()` respectively. However, because we do not yet have a filesystem, we will test these devices by explicitly getting tty pointers from the `dev_byte_lookup` function. Then you can explicitly call the the tty versions of the read/write system calls.

Functions you will need to write in `tty.c`:

- `int tty_provide_char(tty_device_t *tty, char *c)`
- `void tty_receive_char(tty_device_t *tty, char c)`
- `int tty_device_register(tty_device_t *tty)`
- `int tty_read(byte_device_t *dev, int offset, void *buf, int count)`
- `int tty_write(byte_device_t *dev, int offset, const void *buf, int count)`

Functions you will need to write in `ttydisc.c`:

- `int ttyld_attach(tty_device_t *tty)`
- `int ttyld_read(tty_device_t *tty, void *buf, int len)`
- `int ttyld_write(tty_device_t *tty, const void *buf, int len)`
- `void ttyld_receive_char(tty_device_t *tty, char c)`
- `int ttyld_provide_char(tty_device_t *tty, char *c)`
- `int ttyld_echo(tty_device_t *tty, const char *buf, int len)`

### 3.4 Disk Device Driver

A `disk_driver` structure is associated with each disk device. This structure contains information about what threads are waiting to read or write from the disk, the current location of the disk head, the disk IPL, the configuration of the disk, any necessary synchronization primitives, and the corresponding device in the simulator. To simplify writing the disk driver you can assume that disk blocks are page-sized. You should use `SIM_PAGE_SIZE` rather than an absolute number, though page sizes in Weenix are 4k. It would probably be a good idea to use `alloc_page` or

`SIM_PAGE_SIZE` when you try to test the driver. The simulator guide will have helpful information on DMA.

Functions you will need to write in `disk.c`:

- `void disk_init()`
- `void disk_device_add(int i, sim_devconfig_t *conf)`
- `void disk_intr(int ipl, sim_context_t *ctx, void *info)`
- `int disk_read_at(block_device_t *bdev, char *buf, int len, int loc)`
- `int disk_write_at(block_device_t *bdev, char *buf, int len, int loc)`

### 3.5 Memory Devices

You'll also be writing some memory devices, which will not really be necessary until the Virtual File System is developed. Still, these fit in well with the other device drivers, so you'll be asked to implement a data sink and source.

If you've played around with a Linux/UNIX machine, you might be familiar with `/dev/null` and `/dev/zero`. These are both files that represent memory devices. Writing to `/dev/null` will always succeed (the data is discarded) and reading any amount of data from `/dev/zero` will always return as many 0's as read. The former is a data sink and the latter is a data source. The low level drivers for both of these will be implemented in `memdevs.c`.

Functions you will need to write in `drivers/memdevs.c`:

- `void memdevs_init()`
- `int null_write(byte_device_t *dev, int offset, const void *buf, int ct)`
- `int null_read (byte_device_t *dev, int offset, void *buf, int ct)`
- `int zero_read (byte_device_t *dev, int offset, void *buf, int ct)`