`faux_s`: a Fake Kernel with Just Enough Emulation

Welcome to `faux_s` (pronounced "OS") which is a fake OS running on Linux. `faux_s` is an OS emulation environment focused on in-class exercises used in GWU's undergraduate OS class by Gabe. The code is pretty straightforward; the usefulness of this in how it emphasizes core OS concepts. I would *not* recommend using it as a replacement for actual assignments.

Simple `make` comments include:

- `make` to build everything
- `make build` ditto
- `make clean` to remove all generated files
- `make doc` to generate the .pdf from this readme.

`faux_s` Design

`faux_s` has a number of components:

- **Interrupts** - `faux_interrupts.c` includes logic to emulate an interrupt controller that contains a vector of Interrupt Service Routines (ISRs). It uses Linux `signal`s to emulate interrupts. Signals, like interrupts, stop whatever is currently executing, and use their stack for signal execution (similar to if kernel code is executing when an interrupt arrives).
- **Devices** - `faux_dev.c` emulates a few devices. The devices that current work are the *boop device* which creates Penny boops, and the *meme device* which spits out random memes. Each device has a register we can *write* into which controls how frequently the boops and memes arrive, and a register that can be *read*. The latter gives us the data in the device! This can be the characters of the boop, or the characters of the meme. The function `faux-s_dev_reg_read` is well documented, and you should read and understand its signature. `faux_s` creates a lot of overhead for each read to a device to *emulate* the large costs of going out to I/O.
- **DMA** - `faux_dma.c` includes a simple DMA interface (that essentially uses a "ring" with a single entry). The interface enables an OS to enqueue a new, empty buffer that the device will later populate. Then the OS can dequeue that populated buffer and process the data.

The different *modes* of execution for the code can be toggled in the `#define`s at the top of `main.c`.

You should read through `main.c` to get the gist of most of it.

Exercises

We use `faux_s` as a test-bed to understand

1. the trade-offs between polling and interrupts, and
2. the trade-off between byte-by-byte reading of data, and DMA.

It emulates two devices, a boop generator, and a meme generator.

Figure 1:

We'll use `faux_s` to investigate these issues.

Polling vs. Interrupts

You're going to run the code in two configurations:

1. Using polling to get device data. This makes sure that `#define POLL_IO` is not commented out. You can run the program for the boop device (`#define BOOPDEV` uncommented), and for the meme device.
2. With polling disabled, instead using interrupts. In this configuration, the `dev_isr` will receive periodic activations from your device!

You should be able to get by mostly only knowing the code in `main.c`. Please answer the following questions:

- **Q1:** What happens when you run the system using polling?
- **Q2:** What happens when you run the system using interrupts?
- **Q3:** How can you explain the difference? Please be specific.
- **Q4:** A famous google interview question: How can you tell by which way a stack grows in C on your architecture? Brainstorm this as a group and test it out. Use what you learned from that exercise to figure out which stack the interrupt handler `dev_isr` is executing on. Explain what you think is happening, and how that is possible? In other words: how are stacks used with signals in Linux?
- **Q5:** Use the meme device both modes. What is happening here? What solutions do you foresee?

DMA

Now lets hack in some DMA!

- **Q1:** Only the meme device provides DMA. Why? Why does DMA make more sense for the meme device?
- **Q2:** Not so much a question as a puzzle: Implement the `main.c` code to use the DMA features of the meme device! Note that there is a lot of documentation of the DMA functions.
- **Q3:** Why does DMA have such an impact here? The constant `WORK_AMNT` in `faux_dev.c` changes the simulated latency of the interconnect used to "talk" to the device. How does changing `WORK_AMNT` impact the effectiveness of DMA? What does this mean for non-simulated systems?
- **Q4:** Does your implementation pass `valgrind`[1]? Valgrind is a tool that helps you debug your C. It detects errors in using `malloc` and `free`; double freeing a piece of memory? not freeing memory?

A Note on `gdb`

If you want to use `gdb` with programs that use frequent signals, and wish to avoid it reporting every signal, use:

```
(gdb) handle SIGUSR1 noprint nostop
```

[1] Note, you might need to `apt-get install valgrind`