www.jamesmolloy.co.uk

Home » JamesM's kernel development tutorials

1. Environment setup

We need a base from which to design and make our kernel. Here I will be assuming that you are using a *nix system, with the GNU toolchain. If you want to use a windows system, you must either use cygwin (which is a *nix emulation environment) or DJGPP. Either way, the makefiles and commands in this tutorial may not work.

1.1. Directory structure

My directory is laid out as follows:

```
tutorial
 |
 +-- src
 |
 +-- docs
```

All your source files will go in src, and all your documentation (you do write documentation? ;)) should go in docs.

1.2. Compiling

The examples in this tutorial should compile successfully with the GNU toolchain (gcc, ld, gas, etc). The assembly examples are written in intel syntax, which is (my personal opinion) a much more human-readable syntax than the AT&T syntax that GNU AS uses. To assemble these, you will need the Netwide Assembler.

This tutorial is *not* a bootloader tutorial. We will be using GRUB to load our kernel. To do this, we need a floppy disk image with GRUB preloaded onto it. There are tutorials to do this, but, happily, I have made a standard image, which can be found here. This goes in your 'tutorial' (or whatever you named it) directory.

1.3. Running

There is no alternative for bare hardware as a testbed system. Unfortunately, bare hardware is pretty pants at telling you where things went wrong (but of course, you're going to write completely bug-free code first time, aren't you?). Enter Bochs. Bochs is an open-source x86-64 emulator. When things go completely awry, bochs will tell you, and store the processor state in a logfile, which is extremely useful. Also it can be run and rebooted much faster than a real machine. My examples will be made to run well on bochs.

1.4. Bochs

In order to run bochs, you are going to need a bochs configuration file (bochsrc.txt). Coincidentally, a sample one is included below!

Take care with the locations of the bios files. These seem to change between installations, and if you made bochs from source it is very likely you don't even have them. Google their filenames, you can get them from the official bochs site among others.

```
megs: 32
romimage: file=/usr/share/bochs/BIOS-bochs-latest, address=0xf0000
vgaromimage: /usr/share/bochs/VGABIOS-elpin-2.40
floppya: 1_44=/dev/loop0, status=inserted
boot: a
log: bochsout.txt
mouse: enabled=0
clock: sync=realtime
cpu: ips=500000
```

This will make bochs emulate a 32MB machine with a clock speed similar to a 350MHz PentiumII. The instructions per second can be cranked up - I prefer a slower emulation speed, simply so I can see what is going on if lots of text is being scrolled.

1.5. Useful scripts

We are going to be doing several things very often - making (compiling and linking) our project, and transferring the resulting kernel binary to our floppy disk image.

1.5.1. Makefile

```
# Makefile for JamesM's kernel tutorials.
# The C and C++ rules are already setup by default.
# The only one that needs changing is the assembler
# rule, as we use nasm instead of GNU as.

SOURCES=boot.o

CFLAGS=
LDFLAGS=-Tlink.ld
ASFLAGS=-felf

all: $(SOURCES) link

clean:
	-rm *.o kernel

link:
	ld $(LDFLAGS) -o kernel $(SOURCES)

.s.o:
	nasm $(ASFLAGS) $<
```

This Makefile will compile every file in SOURCES, then link them together into one ELF binary, 'kernel'. It uses a linker script, 'link.ld' to do this:

1.5.2. Link.ld

```
/* Link.ld -- Linker script for the kernel - ensure everything goes in the */
/*            Correct place.  */
/*            Original file taken from Bran's Kernel Development */
/*            tutorials: http://www.osdever.net/bkerndev/index.php. */

ENTRY(start)
SECTIONS
{
  .text 0x100000 :
  {
    code = .; _code = .; __code = .;
    *(.text)
    . = ALIGN(4096);
  }

  .data :
  {
     data = .; _data = .; __data = .;
     *(.data)
     *(.rodata)
     . = ALIGN(4096);
```

```
  }

  .bss :
  {
    bss = .; _bss = .; __bss = .;
    *(.bss)
    . = ALIGN(4096);
  }

  end = .; _end = .; __end = .;
}
```

This script tells LD how to set up our kernel image. Firstly it tells LD that the start location of our binary should be the symbol 'start'. It then tells LD that the .text section (that's where all your code goes) should be first, and should start at 0x100000 (1MB). The .data (initialised static data) and the .bss (uninitialised static data) should be next, and each should be page-aligned (ALIGN(4096)). Linux GCC also adds in an extra data section: .rodata. This is for read-only initialised data, such as constants. For simplicity we simply bundle this in with the .data section.

1.5.3. update_image.sh

A nice little script that will poke your new kernel binary into the floppy image file (This assumes you have made a directory /mnt). Note: you will need /sbin in your $PATH to use losetup.

```
#!/bin/bash

sudo losetup /dev/loop0 floppy.img
sudo mount /dev/loop0 /mnt
sudo cp src/kernel /mnt/kernel
sudo umount /dev/loop0
sudo losetup -d /dev/loop0
```

1.5.4. run_bochs.sh

This script will setup a loopback device, run bochs on it, then disconnect it.

```
#!/bin/bash

# run_bochs.sh
# mounts the correct loopback device, runs bochs, then unmounts.

sudo /sbin/losetup /dev/loop0 floppy.img
sudo bochs -f bochsrc.txt
```

```
sudo /sbin/losetup -d /dev/loop0
```