MAKING A KERNEL

CAMILO TALERO
&
NASIR OSMAN

Making a kernel

Contents

Introduction and Setup

1 Introduction

This text is intended as both a log for the PiOS project and as a reference guide for anyone trying a similar project in the future. All text here is pertienent to the Raspberry Pi model 3 B, as this is the model we are using, past and different models may have different architectures, and as such, certain sections fo this guide may be useless for thsoe working on those machines. This is not a documentation manual, for the PiOS documentation, please refer to the PiOS documentation manual.

2 Hardware and Physical tools

To start creating a kernel for the Raspberry Pi, the following is needed:

- Raspberry Pi board (model 3 B is used in this document)
- Micro SD card
- Micro USB adapter and cable to power the PI (5V @ 2A recommended)
- An independent machine with an operating system to compile the source code (aka PC)
- A micro SD to USB adapter or any other method to store the compiled binaries into the micro SD card

The following is optional:

- Raspberry Pi case
- JTAG system for debugging
- A monitor compatible with the Pi's video core

3 Software

The following software is needed to create the kernel:

- Cross compiler to generate the binaries. Options include the GNU ARM Embedded Toolchain (used in this document)

- Raspberry Pi firmware binaries. In order for the boot process to be executed correctly it is needed to have the bootcode.bin and start.elf binaries inside of the micro SD card.
- A text editor or IDE (no help will be provided as to how to setup an IDE)

It is also recommended that the reader has enough experience with C and ARM assembly, as it is assumed that the reader understands the syntax of these languages. An understanding of linker scripts is also helpful.

4 Compilation

The following are the bare minimum commands needed to build a working kernel image.

To compile C files:

```
arm-none-eabi-gcc -<opt> -<arch> <src> -o -c <o_file>
```

Where <opt> refers to the level of optimization, <arch> refers to the target architecture, <src> is the source file(s) and <o_file> is the generated object file. The -c argument tells the compiler to generate the .o file without linking, this argument is VERY important, and things won't work without it.

Example:

```
arm-none-eabi-gcc -O0 -march=armv8-a source/MainFiles/PiTest.c
   -nostartfiles -c -o objects/MainFiles/PiTest.o
```

To compile ARM assembly files:

```
arm-none-eabi-as -<arch> <src> -c -o <o_file>
```

The flags here are the same as above, except that <src> shoulkd be an ARM assembly file instead of a C file.

Example:

```
arm-none-eabi-as -march=armv8-a source/boot/boot.s -c -o
   objects/boot/boot.o
```

To link all object files:

```
arm-none-eabi-ld <o_file(s)> -o <elf> -T <linker(s)>
```

Where <o_files> refers to all object binaries that will makeup the final kernel image, <elf> refers to the output .elf file and <linker(s)> refers to all linker scripts (normally just 1), needed to link the objects.

Example:

```
arm-none-eabi-ld ./objects/MainFiles/PiTest.o ./objects/boot/boot.o -o
   build/kernel.elf -T ./source/kernel.ld
```

To extract the raw image binary:

```
arm-none-eabi-objcopy <elf> -O binary <image>
```

Where <elf> is the .elf file created on the above step and <image> is the final kernel image binary.

Example:

```
arm-none-eabi-objcopy build/kernel.elf -O binary kernel.img
```

To dissassemble the final .img binary for debugging:

```
arm-none-eabi-objdump -D <elf>
```

Although not necessary for compilation, it is recommended to execute and store the ouput of this command in a file, as it is very helpful for debugging and veryfing the correctness of the final binary.

Example:

```
arm-none-eabi-objdump -D build/kernel.elf > logs/kernel.list
```

For more information refer to the official GNU documentation for your current version (or to the documentation of your toolchain).

5 First Program

After all prerequisites are met, it is necessary to verify that things work properly. This section simply provides a minimal example to test that all requirements were installed correctly. A more in depth explanation is provided on the Mailbox section.
To test whether everything is working fine, we just want to turn on the Pi's ACT LED on, however on the Pi 3, the led is not connected to the GPIO lines, so we have to communicate with the video core through the mailbox to turn it on. To be

extra-sure everything is fine, we want it to blink. The following C code is used to send the appropriate message to the mailbox to turn the LED on.

```
#include <stdint.h>

#define REGISTERS_BASE 0x3F000000
#define MAIL_BASE 0xB880 // Base address for the mailbox registers
// This bit is set in the status register if there is no space to write
   into the mailbox
#define MAIL_FULL 0x80000000
// This bit is set in the status register if there is nothing to read
   from the mailbox
#define MAIL_EMPTY 0x40000000

struct Message
{
   uint32_t messageSize;
   uint32_t requestCode;
   uint32_t tagID;
   uint32_t bufferSize;
   uint32_t requestSize;
   uint32_t pinNum;
   uint32_t on_off_switch;
   uint32_t end;
};

volatile struct Message m =
{
   .messageSize = sizeof(struct Message),
   .requestCode =0,
   .tagID = 0x00038041,
   .bufferSize = 8,
   .requestSize =0,
   .pinNum = 130,
   .on_off_switch = 1,
   .end = 0,
};

/** Main function - we'll never return from here */
int kernel_main(void)
{
uint32_t mailbox = MAIL_BASE + REGISTERS_BASE + 0x18;
volatile uint32_t status;

while(1)
{
   do
      status = *(volatile uint32_t *)(mailbox);
   while((status & 0x80000000));

   *(volatile uint32_t *)(MAIL_BASE + REGISTERS_BASE + 0x20) =
      ((uint32_t)(&m) & 0xfffffff0) | (uint32_t)(8);
```

```
    int i=0;

    while(i<0xF0000)
       i++;

    if(m.on_off_switch == 0)
       m.on_off_switch = 1;

    else
       m.on_off_switch = 0;

    m.requestCode = 0;
    m.requestSize = 0;
    m.pinNum = 130;
    m.end = 0;

    do
       status = *(volatile uint32_t *)(mailbox);
    while((status & 0x40000000));

    uint32_t temp;

    do
       temp = *(uint32_t *)(MAIL_BASE + REGISTERS_BASE);
       temp = temp & 0xF;
    while(temp != 8);

    }

}
```

As big as this seems for a "hello world!" example, it is the smallest C code we could come up with that nlink the LED. And we are not even done yet. This code is using memory, and if one where to disassemble the binaries, one could see that the sp register (stack pointer). Is used, as such we need to initialize it properly.
The following assembly initializes the stack pointer and branches to our main loop:

```
.section .init
.global _start

_start:
   ldr sp, =8000
   b kernel_main
```

However we still aren't done. The Raspberry Pi always begins execution at address 0x8000, so whatever instruction is at that address will be the first to run. We need to ensure this instruction is also the first instruction in the above code. For this we use the following linker file:

```
SECTIONS
{

   .init 0x8000 :
   {
   *(.init)
   }

   .text :
   {
   *(.text)
   }

   .data ALIGN(0x20) :
   {
   *(.data)
   }

   .comment :
   {
   *(.comment)
   }

   .ARM.attributes :
   {
   *(.ARM.attributes)
   }
}
```

If the reader doesn't fully understand what the C code is doing, it will be further explain in future sections. However the functioning of linker scripts, or assembly alnguage will not be explained.

6 Mailbox