OS Pi Lab 1 实验报告

梁远志 15231099

1. 概述

同 MIPS 仿真器上的 Lab 1 内容相似，在树莓派上的 Lab 1 也完成了系统的引导、字符输出和 printf 函数。本实验报告主要说明一下与标准实验的不同之处。

2. 工具链

此次使用的是 Raspberry Pi 3，具有一个 Aarch64 架构的 A53 四核处理器。值得注意的是这是个完全的不向下兼容的 64 位体系结构与 ARMv7 有着相当大的差异。

我们使用的是由 Linaro 提供的交叉编译工具链，而且是脱离 Lib C 的 Bare-Metal 用的 No-ELF 版本。由于是很古老的东西，经测试在 openSUSE 42.2 环境下还能正常使用，而在 Ubuntu 16.04 下就已经是无法识别的 ELF 文件了，所以我们的备选的工具链是直接从源代码编译的 GCC，其一键安装脚本来自 https://github.com/dwelch67/build_gcc。值得一提的是我们相当一部分的工作都借鉴了这位 Trailblazer 的另一个 Repo。（当然这个和学长提供的应该是一样的东西）

3. 启动

如上面所说的我们参考了那位的 Repo，所以在缺少官方资料的情况下完成了启动。以下列出几个需要注意的地方。

内核代码 text 启动的位置应该设为 0x00080000 固然我们知道这个地址设定会对日后开启 MMU 后的内核区域设定的问题带来一系列问题，不过在此暂且不予考虑。同时，这个起始地址在固件设置（config.txt）也是可以设置的。所以改写了这样一个简洁的链接脚本：

```
SECTIONS
{
    . = 0x00080000;
    .text : { *(.text*) }
    .bss  : { *(.bss*)  }
}
```

正如前面所提到的，SoC 使用的是一个四核心的处理器。首先明确一个概念，拥有多个核心的处理器每个核心拥有自己的一组寄存器，Cache 各级的共用情况不同，主存是共用的。按照推断，当内核镜像载入到主存后，四个核心都从同样的 PC 开始执行，若不做处理，在遇到对主存互斥的访问时会出现不可预知的问题。查阅了 ARMv8 的手册后，有这么一条指令可以读取系统寄存器的值 mrs x0, mpidr_el1，后者则是 CPUID 所在的系统寄存器。一般来说我们的核心是从零数起的，所以判断最后两位是不是 2'b00 即可，但是既然有给出的零号核心的 CPUID（0xC10000000），我们还是直接进行了比较。

```
_start:
    mrs x0, mpidr_el1                   将 CPUID 读入 x0 寄存器
```

```
    mov x1, #0xC1000000             在 x1 寄存器载入立即数 0xC10000000
    bic x0, x0, x1                  按位清零，即将 x1 中为 1 的位在 x0 中相应 0
    cbz x0, _start_master           比较等于 0 跳转
    b   L0                          其他核心跳转到 L0
_start_master:
    mov sp, #0x08000000             设置内核栈顶
    bl  main                        链接跳转到 C 语言的 main 函数
L0:
    b   L0                          用于其他核心的空循环
```

这样就能够完成内核载入和启动到 C 语言函数的过程。

就 Core0 的 CPUID 是多少这一说我还写了个小东西来读取，不过方法倒是相当无聊了啊。这个参见 Repo 中的 Utility 中的 myrd.c。这里利用了一个现有的 Aarch64 的 SLES Linux 作为测试工具。将代码编译为一个内核模块后则可以以内核的异常级别读取 CPUID（这个只是理论上的设想）。实验结果也十分有趣，在反复加载卸载内核模块得到的 CPUID 为 0x80000002、0x80000001。看起来这个小东西的工作机理真是莫名其妙啊。

```
.559267+08:00 pi kernel: [ 3173.101303] MyRD: Started.
.559381+08:00 pi kernel: [ 3173.101323] Current core id: 0x80000002
.119246+08:00 pi kernel: [ 3180.658988] MyRD: Finished.
.369262+08:00 pi kernel: [ 3181.907947] MyRD: Started.
.369356+08:00 pi kernel: [ 3181.907962] Current core id: 0x80000001
.799240+08:00 pi kernel: [ 3183.339287] MyRD: Finished.
.829269+08:00 pi kernel: [ 3184.366086] MyRD: Started.
.829361+08:00 pi kernel: [ 3184.366104] Current core id: 0x80000002
```

4. UART

这个地方的话参考那个 Repo 的代码就没什么问题了。这里稍微说一下的是 Pi 3 IO 的内存映射在 0x3F2…上（由结果来推导原因:/），将来在启用 MMU 后要在前面加上一堆的 F。这里需要说明的是三个汇编函数。

```
extern void set_ptr(u32, u32);
extern u32  get_ptr(u32);
extern void empty_loop(u32);
```

这个 u32 的类型定义与标准实验不同，我更愿意遵守这个规范。set_ptr 是将第一个参数指向内存位置设置为第二个参数的值。get_ptr 是取得第一个参数指向内存位置的值。empty_loop 是一个无用的消耗时钟周期的函数。这里需要提一下 ABI 规范的问题（虽然这里完全可以写成 myrd.c 中的嵌入式汇编的样子，编译后难看点而已）。按照 ARMv8 的 ABI 规范，函数调用时，参数存入寄存器中（先不管内存堆栈的问题），结果保存在 w0 寄存器中。

```
.global get_ptr
get_ptr:
    ldr w0, [x0]
    ret
.global set_ptr
set_ptr:
    str w1, [x0]
    ret
```

```
.global empty_loop
empty_loop:
    ret
```

最后解释一下 empty_loop 的用途。因为按照系统设计要求，在对 GPPUDCLK0 进行设置的时候需要前后各等待 150 个时钟周期。所以为了避免 C 编译器对空循环的优化，就有这样的一个函数来完成这种 dummy 的任务。在 AUX_MU_BAUD_REG 设定为 270 的时候，一个波特率为 115200 的 UART 就启动完成了（一般的默认 UART 就是这样的，在电脑端就不需要特别设置了）。

将 uart_send 封装为一个 printcharc 函数（标准实验代码中的函数）就可以直接移植一个 printf 了。还有就是输出换行的问题，换行从 0xd 要变为 0xa。

在提交的源码中，演示的是一个回显的功能。

5. 固件

Pi3 在发布初期使用的固件（包含 bootcode.bin, config.txt, start.elf）在启动时是要添加 kernel_old 设置，且是从 0x00000000 开始的内核 text 起始位置。然后在某个固件版本更新后，就不在需要这一设置，内核的 text 的起始位置也变成了 0x00080000，同时新增了一个文件 fixup.dat

config.txt 中的 arm_control=0x200 是指定运行在 ARMv8 也就是 Aarch64 上。

值得一提的是在旧固件中内核镜像的文件名是 kernel7.img 而在新固件中是 kernel8.img（大概是因为 ARMv7 和 v8 的意思吧）。

在链接生成 Elf 文件后需要将这个用 objcopy 导出一个二进制的镜像，也就是

```
$(CROSS)ld -o $(vmlinux_elf) -e _start -T$(link_script) $(objects)
$(CROSS)objcopy $(vmlinux_elf) -O binary $(vmlinux_img)
```

这里需要特别说明的是这里指定了入口符号_start，目前还不确定这个地方是不是会对启动造成影响，不过总之能跑起来了。

6. 感想

这个嘛，因为是一个陌生的体系结构，所以有很多东西起步起来还是很困难的。往后的困难只会越来越多，这个 Lab 在一个月前就已经完成了（也提交到 git 服务器上了），到现在都毫无更多的进展。

感觉在 UART 上学长给了较多的误导，弄得好像 UART 要用汇编来写寄存器一样:)。不过这样有点半抄袭性质完成了的 Lab 也实在是没有什么成就感。在倒腾 Bare Metal 方面已经有很多先驱者的工作成果了，我们只是以粗浅的视角再造了只轮子出来。

往后的工作能够参考的东西就十分有限了，所以更多的要去读 ARM 的手册了，再加上很多地方的代码是用汇编写的，目前觉得缺页中断这个事情要搞很久的样子。就算是一个完全就只是实现了页面管理的标准实验 Lab2 移植起来也不一定轻松。

说实话我已经有点要放弃搞这个的想法了。