基于树莓派操作系统课程设计指导文档

Version 2.0

序言

> 本文旨在阐述一种引导学生一步一步完成一个操作系统移植项目的过程。强调不仅能**对学生的完成情况进行考察**，还能让学生**体验到从零开始移植操作系统的乐趣**。

相信选择本任务的你，对于完成一个新内核充满了期待。

在移植过程中，可能会遇到相当相当多的困难，希望你能开动脑筋，不放弃，学会运用搜索引擎去解决它们。

当你完成之后，你会觉得这一切的付出都是值得的。

基本知识

树莓派

树莓派（Raspberry Pi）是英国树莓派基金会开发的微型单板计算机，目的是以低价硬件及自由软件促进学校的基本计算机科学教育，目前市场上最新的型号为 4B。

本实验中指定的硬件型号为 **3B**。树莓派本身做为一个流行的硬件开发板，拥有庞大的社区资源。树莓派具有的硬件已经能完成本操作系统实验的全部能容，当然其本身就能够运行 Linux 操作系统。

Aarch64

树莓派 3B 使用的是一个 ARM A53 处理器，指令集为 **Aarch64**（旧称 ARMv8）。

ARM 的 64 位处理器都使用同一个指令集，但是又因为各系列的型号实现不同，需要参照不同的文档。

这里给出两个主要参考的文档：

- ARM® Cortex®-A Series Programmer's Guide for ARMv8-A
- ARM® Cortex®-A53 MPCore Processor Technical Reference Manual

前者是 ARMv8 指令集通用的程序员指导，后者是 A53 处理器的技术参考手册（实现细节）。

另外有趣的一点是现代的手机处理器几乎都是 ARM 体系结构的 64 位处理器，Aarch64 的 Linux 内核开发已经十分成熟。

作为题外话，你可以去了解一下这几个事实：

- 树莓派官方并不单独提供一个 Aarch64 的 Raspbian（基于 Debian 的变种）Linux 操作系统，提供的反而是一个兼容旧版的 ARMv7 的版本
- ARMv8 和 ARMv7 的关系相当于 amd64（x86_64）和 i386（x86），他们的 64 位处理器均提供一种运行 32 位程序的方法
- 和 x86 不同的是，在设计上 ARMv8 和 ARMv7 设计迥异，一种很有趣的说法称 ARMv8 的设计更像是 MIPS
- 除了手机之类的低能耗设备外，Aarch64 已经逐渐渗透到高性能计算的服务器领域

Lab0: 实验环境配置

> 在本部分实验中，鼓励大家自己动手搭建自己的实验环境。

本机操作系统

因为需要在你的操作系统上运行 Aarch64 的交叉编译器，而且一般来说目标机为 64 位的编译器也需要在 64 位操作系统上运行。

这里给出两个建议的操作系统：

1. WSL (Windows Subsystem for Linux) 运行在 Windows 10 上
2. 一般的 64 位 Linux 发行版本（Ubuntu、Manjaro 等）

WSL 能够在 Windows 上运行 64 位的 ELF 可执行程序镜像，完成本实验绰绰有余。如果习惯使用 Windows 环境建议使用这种方式。

对于 macOS 用户，可能需要自行安装 GCC 自行编译一个交叉编译器。

当然最推荐的还是使用 Linux 完成本实验。

交叉编译器

由于要使用 Aarch64 指令集写内核，无法使用本机的 GCC 进行编译，因此需要下载能编译 Aarch64 的 GCC。

可以访问下面的网址去下载所需要的 GCC：https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-a/downloads

根据本机操作系统不同，你需要找到对应自己电脑的下载链接。

这里大家可以发现有两种不同的模式

- bare-metal target
- GNU/Linux target

我们这里选择前者作为我们的交叉编译器，即 aarch64-none-elf-gcc。

这里请大家思考两个问题：

- Linux Targeted 和 Bare-Metal 分别有怎样的含义呢?
- 大小端的差异是由什么决定的?

如果需要自行编译 GCC，可以前往这个 Github 上的 Repo 获得一点帮助：https://github.com/dwelch67/build_gcc

硬件模拟器

就像 MIPS 实验中使用 GXemul 来 “模拟” 一块板子进行操作一样。在树莓派移植实验中你可以使用硬件模拟器 QEMU 来完成对树莓派板子的仿真（这里也建议同学们先在硬件模拟器跑完所有的实验，验证完正确性后再烧刻在板子上去）。

对于使用 Ubuntu 20.04 进行实验的同学，可以直接使用下面的命令

```
sudo apt-get install qemu
```

会直接安装 4.2.1 版本的 QEMU。

不过这里还是建议大家手动安装更高版本的 QEMU，因为低版本的 QEMU 对于 TLB 的仿真会有一定程度的问题。

你可以去官网 (https://www.qemu.org/download/) 找到下载教程。

这里以 Linux 为例，介绍 5.0.0 版本（**推荐版本**）的安装。

```
wget https://download.qemu.org/qemu-5.0.0.tar.xz
tar xvJf qemu-5.0.0.tar.xz
cd qemu-5.0.0
./configure --target-list=aarch64-softmmu
sudo make -jX
sudo make install
# X is the number of compile thread
```

这里在安装过程中可能会报软件不存在的错误，基本都是相关依赖工具链不完全，你可以自行在搜索引擎中找到解决方案。

树莓派硬件

为了完成我们的实验，你需要下面的一些硬件准备：

1. 树莓派3B一只
2. FT232/CP2102串口通信板一只
3. MicroSD一张

树莓派使用的是Micro-USB的供电线，你可能需要一只USB接口的电源适配器和一根Micro-USB的线缆（一般来说在购买树莓派硬件时都会提供树莓派供电线），你也可以使用电脑的USB接口供电（在树莓派不连接其他外设时，一般也不需要2.5A的电流）。

串口通信版起到一个“中间商”的工作，可以将GPIO串口输出的信息通过USB端口输出到电脑上，也可以在电脑上将信息输入进树莓派中。

你需要将USB端口与电脑端相连，并且在串口通信板的另一端与GPIO相连。

你可以在下面的链接中找到管脚的对应序号Raspberry Pi 2 & 3 Pin Mappings - Windows IoT | Microsoft Docs，需要注意是，你只需要连接地线`GND`、输入`RXD0`和输出`TXD0`。特别你**需要注意**的是：串口通信版的输出端，应该对应树莓派的输入端；串口通信版的输入端，应该对应树莓派的输出端。

接下来你需要软件去接收串口的信息（有的串口你可能还需要安装驱动才能操作），对于不同的开发OS使用的软件也不太一样。

在Windows上你可以使用Putty去连接串口设备，你可以在设备管理器中找到对应的串口，注意选择Serial和设置波特率。

Linux下可以使用Minicom软件来连接串口设备（通过lsusb确认USB设备是否就位），一个典型的启动命令：

```
minicom -b 115200 -D /dev/ttyUSB0
```

其中`115200`为设置的波特率，`/dev/ttyUSB0`为一个字符设备（也就是串口通信板）。

实验要求

这一部分需要你在报告中记录 **你的实验环境。**

Lab1: 内核启动和 Printf 的实现

> 在本部分实验中，将实现内核启动和低地址的 Printf

内核编译运行

你需要自行编写 Makefile 去编译自己的内核，之后使用 QEMU 来编译内核 kernel.img.

```
qemu-system-aarch64 -M raspi3 -serial stdio -kernel kernel.img
```

这里对参数进行一下解析：

- -M raspi3: 模拟的板子是 Raspberry3
- -serial stdio: 使用 UART0 串口仿真进行交互
- -kernel kernel.img: 让仿真器加载对应的内核镜像
- -drive file=sdcard.img,format=raw: 模拟仿真器中的 SD 卡

内核启动

树莓派的启动地址在 0x00080000，以 EL2 异常级状态启动。

内核是运行在 EL1 异常级状态的，因此你需要在启动的时候从 EL2 异常级降低到 EL1 异常级。

同时树莓派 3B 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。**你需要找到解决这个问题的方法。**

这里请你思考一个问题：

- 我们的内核为什么从 EL3/EL2 异常级启动而不是 EL1 异常级启动?

任务清单

- 确定要移植的文件清单
- 修改 Makefile、配置交叉编译工具链
- 修改链接脚本 Link Script
- 实现 UART 驱动，替换字符输出相关的代码
- 撰写启动用的汇编代码

UART 驱动

虽然这是一个号称 Bare-Metal 的实验，但是其重点并不在于实现硬件驱动，这里提供简单的初始化 UART 以及一个输出字符所需要的代码（这一驱动实现的源码来源：https://github.com/bztsrc/raspi3-tutorial/tree/master/05_uart0）

```
#define UART0_IBRD      ((volatile u_int*)(0x3F201024))
#define UART0_FBRD      ((volatile u_int*)(0x3F201028))
#define UART0_LCRH      ((volatile u_int*)(0x3F20102C))
#define UART0_CR        ((volatile u_int*)(0x3F201030))
#define UART0_ICR       ((volatile u_int*)(0x3F201044))
#define GPFSEL1         ((volatile u_int*)(0x3F200004))
#define GPPUD           ((volatile u_int*)(0x3F200094))
#define GPPUDCLK0       ((volatile u_int*)(0x3F200098))
#define UART0_DR        ((volatile u_int*)(0x3F201000))
#define UART0_FR        ((volatile u_int*)(0x3F201018))

void uart_init () {
    register unsigned int r;
    *UART0_CR = 0;
```

```
    r = *GPFSEL1;
    r &= ~((7 << 12) | (7 << 15)); //gpio14, gpio15
    r |= (4 << 12) | (4 << 15);    //alt0
    *GPFSEL1 = r;
    *GPPUD = 0;            //enable pins 14 and 15
    r = 150;
    while (r--) { asm volatile ("nop"); }
    *GPPUDCLK0 = (1 << 14) | (1 << 15);
    r = 150;
    while (r--) { asm volatile ("nop"); }
    *GPPUDCLK0 = 0;        //flush GPIO setup
    *UART0_ICR = 0x7FF;    //clear interrupts
    *UART0_IBRD = 2;       // 115200 baud
    *UART0_FBRD = 0xB;
    *UART0_LCRH = 0b11 << 5; // 8n1
    *UART0_CR = 0x301;     //enable Tx, Rx, FIFO
}

void uart_send (unsigned int c) {
    do { asm volatile ("nop"); } while (*UART0_FR & 0x20);
    *UART0_DR = c;
}
```

实验要求

在这个实验中你只需要让 printf 在低地址运行即可。

这一部分需要在报告中记录**启动内核和调用驱动的方法**。

Lab2: MMU 设置和内存管理

> 在本部分实验中，你将完成 MMU 设置和内核设置为高地址

MMU 设置

在启动树莓派处理器的 MMU 时，你可以感受到现代处理器的硬件设计的复杂与强大!

在 Aarch64 体系结构中，地址翻译需要通过 MMU 进行控制，且这一部分完全由硬件控制。

先请你思考下面的问题：

- 在 “标准” MIPS 实验中，是如何进行地址翻译的呢?
- 提示阅读代码有关 **do_refill** 部分

在真正开启 MMU 进行地址翻译之前，我们需要先设置相关的寄存器。

Aarch64 地址空间布局

在 Aarch64 体系结构中，一般约定高地址（掩码全为 1）是内核态地址，低地址（掩码全为 0）是用户态地址。

我们还注意到内核的启动地址为 0x00080000，是本来属于用户态的地址，在这种情况下，一种比较好的方式是把用户态低地址的一部分设置给内核，在这里做页表的设置和 MMU 相关寄存器的设置，然后跳转到高地址。

当然另外一种方法是互换原本属于内核和用户态地址空间的位置。

一种参考设置

理解这些设置的内容，并完成自己 MMU 的设置。

内核态页表

在设置寄存器之前，我们需要先构建自己的页表。

我们需要保证在开启 MMU 之前和开启 MMU 之后都能正常访问内核代码、所有物理页面和 MMIO（设备占用的地址空间）。

在 MMU 开启前虚拟地址通过链接脚本直接映射到对应的物理地址，因此在开启 MMU 之前你需要设置各个页面的权限和类型。

因为不可避免的（照顾到 32 位系统），所以有大约 16MB 的内存牺牲给了设备占用的地址空间（`0x3F00 0000 ~ 0x3FFF FFFF`）；这时将 `0x00000000 ~ 0x3EFF FFFF` 的物理内存映射为普通内存（`PTE_NORMAL`），将设备地址空间映射为设备（`PTE_DEVICE`）。

下面给出一些页表权限位的设置：

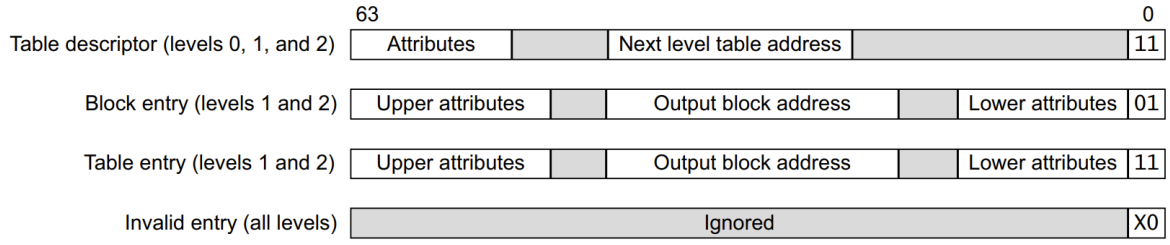

Figure 12-10 A64 Table descriptor type

(ARM® Cortex®-A Series Programmer's Guide for ARMv8-A 12-15)

```
#define PTE_VALID 1
#define PTE_TABLE (1 << 1)
```

对于页目录，需要设置 `PTE_TABLE` 和 `PTE_VALID`.

对于页表，需要设置 `PTE_VALID`.

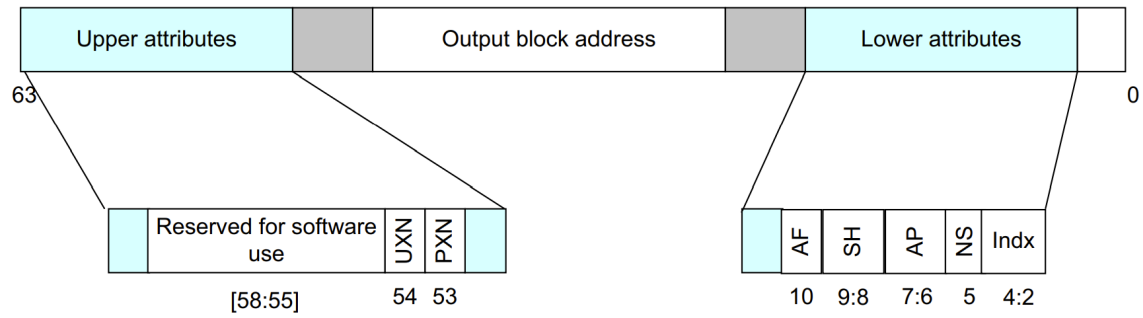

Figure 13-3 Stage 1 block memory attributes

(ARM® Cortex®-A Series Programmer's Guide for ARMv8-A 13-11)

Indx MAIR 中的 AttrIndx [4:2]

这块将在 MAIR 寄存器介绍。

AP (Access Permission) 标志位 [7:6]

```
#define PTE_KERN (0 << 6) // 仅 EL1 + 访问
#define PTE_USER (1 << 6) // EL0 + 可访问
#define PTE_RW (0 << 7) // 读写
#define PTE_RO (1 << 7) // 只读
```

SH 共享标记 [9:8]

```
#define PTE_OUTER_SHARE (2 << 8) // 外部共享（核心 Cluster 间）需要给设备内存标记
#define PTE_INNER_SHARE (3 << 8) // 内部共享（Cluster 内）需要给普通内存标记
```

AF (Access Flag) 标志位 [10]

```
#define PTE_AF (1 << 10) // 用户控制的可访问标识（AccessFlag）
```

用户控制的可访问标识相当于一个不强制的标识位与标准实验中的 PTE_V 不同，控制位会导致缺页中断，但是并不会真的影响是否能存取，仅会影响 TLB 行为和异常行为。

PXN/UXN 标志位 [54:53]

```
#define PTE_PXN (1UL << 53) // 不可执行标记 (PrivilegedExecute-Never)
#define PTE_UXN (1UL << 54) // EL0 不可执行标记 (Unprivileged Execute-Never)
```

在 “标准” MIPS 实验中，我们可以直接通过 Kseg0 区域的地址访问对应响应的物理地址。但是在 Aarch64 体系结构下并没有这样的地址映射关系，但是内核依然需要能直接访问物理地址的内容（在页表遍历时需要这个的功能），你需要建立在页表中建立**线性映射区**的结构去实现这个功能。

Reserved for software use [58:55]

保留给操作系统的权限，可以用这块标识 COW 权限。

间接内存属性寄存器

Memory Attribute Indirection Register, EL1

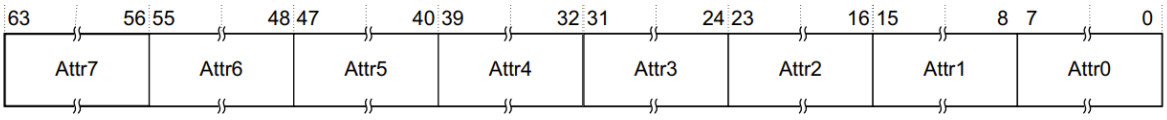

Figure 4-64 MAIR_EL1 bit assignments

(ARM® Cortex®-A53 MPCore Processor Technical Reference Manual 4-117)

用于设置一些页面属性，在翻译时会去查找该寄存器确定页面的真正属性。

总共 8 个位置，可以用 3 位 2 进制标识，对应于页表中的 MAIR [4:2].

每个位置有 8 位可以用来设置，下面两张图表现前四位和后四位的含义：

Table 4-105 Attr<n>[7:4] bit assignments

Bits	Meaning
0b0000	Device memory. See Table 4-106 for the type of Device memory.
0b00RW, RW not 00	Normal Memory, Outer Write-through transient.[a]
0b0100	Normal Memory, Outer Non-Cacheable.
0b01RW, RW not 00	Normal Memory, Outer Write-back transient.[a]
0b10RW	Normal Memory, Outer Write-through non-transient.
0b11RW	Normal Memory, Outer Write-back non-transient.

a. The transient hint is ignored.

Table 4-106 Attr<n>[3:0] bit assignments

Bits	Meaning when Attr<n>[7:4] is 0000	Meaning when Attr<n>[7:4] is not 0000
0b0000	Device-nGnRnE memory	UNPREDICTABLE
0b00RW, RW not 00	UNPREDICTABLE	Normal Memory, Inner Write-through transient
0b0100	Device-nGnRE memory	Normal memory, Inner Non-Cacheable
0b01RW, RW not 00	UNPREDICTABLE	Normal Memory, Inner Write-back transient
0b1000	Device-nGRE memory	Normal Memory, Inner Write-through non-transient (RW=00)
0b10RW, RW not 00	UNPREDICTABLE	Normal Memory, Inner Write-through non-transient
0b1100	Device-GRE memory	Normal Memory, Inner Write-back non-transient (RW=00)
0b11RW, RW not 00	UNPREDICTABLE	Normal Memory, Inner Write-back non-transient

(ARM® Cortex®-A53 MPCore Processor Technical Reference Manual 4-117)

其中 RW 两位指的是 cache 的读时分配（read-allocate）和写时分配（write-allocate）。

一些概念的说明：

1. inner 和 outer 指的是多核心下的问题，此处不需要考虑，内外相同即可
2. transient 和 non-transient 为 cache 的存留时间问题，此处不需要考虑，都设置为 non-transient（cache 内容更倾向于长时间驻留）
3. Write-through 和 Write-back 是 cache 的写策略，分别是写通达和写回，都设置为 Write-through
4. GRE 指的是设备内存区域访问的三种特性，A53 中 “仅有” nGnRE 和 GRE，一般对于外设的 IO 设置为 nGnRE

此处我们设置 3 种类型：

#	AttrIndx	Attr[7:4]	Attr[3:0]	用途
1	0	0b1000	0b1000	普通内存
2	1	0b0000	0b0100	设备，nGnRE
3	2	0b0100	0b0100	non-cache 的内存

> 你可以计算得到应该向 mair_el1 寄存器的值

使用 msr 指令对 mair_el1 进行修改。

页表项的标志位中则可以定义这三种内存类型的宏：

```
#define PTE_NORMAL (0 << 2)
#define PTE_DEVICE (1 << 2)
#define PTE_NON_CACHE (2 << 2)
```

翻译控制寄存器

Translation Control Register, EL1

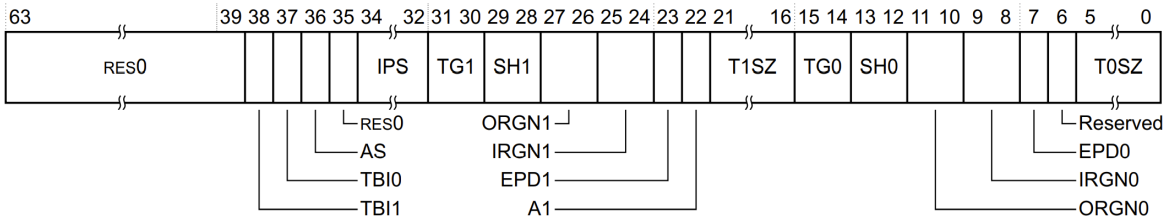

Figure 4-44 TCR_EL1 bit assignments

(ARM® Cortex®-A53 MPCore Processor Technical Reference Manual 4-87)

该寄存器用于控制地址翻译的过程。

这里给出一个参考设置：

位	值	说明
[38:38]	1	TTBR1_EL1 忽略高 8 位
[37:37]	1	TTBR0_EL1 忽略高 8 位
[36:36]	0	8 位 ASID
[34:32]	000	32 位物理内存
[31:30]	00	TTBR1_EL1 4KB 页面大小
[29:28]	11	内部共享 TTBR1_EL1
[27:26]	10	TTBR1_EL1 外写通达可缓存
[25:24]	10	TTBR1_EL1 内写通达可缓存
[23:23]	0	TTBR1_EL1 使用
[22:22]	0	TTBR1_EL1 使用 ASID
[21:16]	0d25	TTBR1_EL1 使用 25 位掩码
[15:14]	00	TTBR0_EL1 4KB 页面大小
[13:12]	11	内部共享 TTBR0_EL1
[11:10]	10	TTBR0_EL1 外写通达可缓存
[9:8]	10	TTBR0_EL1 内写通达可缓存
[5:0]	0d25	TTBR0_EL1 使用 25 位掩码

约定使用 TTBR1_EL1 用于内核的地址空间（高位），TTBR0_EL1 用于用户进程的地址空间（低位）。

25 位掩码可以构成下面的三级页表：

一个三级页表设计（64-25=39 位 VA）

63:40	39:31	30:21	20:12	11:0
高位掩码 25b	三级页表 9b	二级页表 9b	一级页表 9b	页内偏移 12b

ASID 是 aarch64 从 v6 开始引入的机制，加快 TLB 的性能。

用 ASID 来标记进程号，可以使 TLB 中包含多个进程的页表。

> 你可以计算出应该向 tcr_el1 寄存器的值

使用 msr 修改 tcr_el1 寄存器的值，并在下面使用 isb 指令。

- ISB 指令用于清空处理器流水线中的指令，在进程切换，更新 TLB 设置等操作时需要保证修改效果能够生效。

页表基地址寄存器

Translation Table Base Register 1, EL1

设置用户态页目录寄存器 **ttbr0_el1** 和内核态页表寄存器 **ttbr1_el1**。

前面我们已经将页表项已经填好，在这里我们只需要把 ** 物理 ** 地址写入寄存器即可。

```
adrp    x8, boot_ttbr0_l1
msr     ttbr0_el1, x8
adrp    x8, boot_ttbr1_l1
msr     ttbr1_el1, x8
isb
```

boot_ttbr0_l1 和 boot_ttbr1_l1 是页表的物理地址（注意在启动前低地址虚拟地址**等于**物理地址）。

注意在进程切换时修改用户页表 ttbr0_el1 即可。

系统控制寄存器

System Control Register, EL1

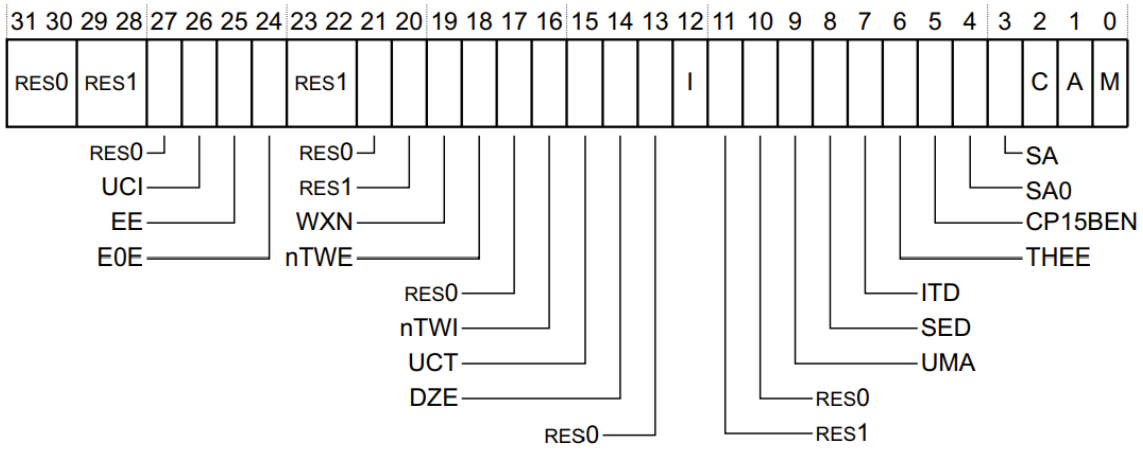

Figure 4-28 SCTLR_EL1 bit assignments

(ARM® Cortex®-A53 MPCore Processor Technical Reference Manual 4-51)

主要是对系统进行整体控制的寄存器。

这里给出一个参考设置：

位	值	说明
[29:28]	11	保留
[26:26]	0	不允许 EL0 使用 cache 指令
[25:25]	0	EL1 及异常时小端存储
[24:24]	0	EL0 时小端存储
[23:20]	1101	保留
[19:19]	0	不强制写区域不可执行
[18:18]	0	WTE 指令正常
[16:16]	0	WTI 指令正常
[15:15]	0	TTBR1_EL1 使用
[14:14]	0	EL0 级部分 cache 指令禁止
[12:12]	1	指令 cache 启动
[11:10]	10	保留
[9:9]	0	EL0 禁止访问中断屏蔽寄存器
[8:8]	0	SETEND 指令开启
[7:7]	0	IT 指令关闭
[5:5]	1	CP15 障碍（指令）开启
[4:4]	0	EL0 级栈对齐检查关闭
[3:3]	0	栈对齐检查关闭
[2:2]	1	数据 cache 启动
[1:1]	0	禁止对齐检查
[0:0]	1	**MMU 开启**

> 你可以计算出应该向 sctlr_el1 寄存器的值

不要忘记使用 isb 指令使修改生效。

在经过这些设置之后，MMU 就正式开启了。

下面所有的地址翻译都将通过硬件进行页表遍历进而翻译。

内存管理部分

这里和 MOS 区别主要是页表级数可能有些差异，页表项权限不完全一样。

不要忘记 TLB 的清空！

任务清单

- 链接脚本修改
- 页表设置
- 汇编设置 MMU
- 其他内存管理部分
- TLB 的刷新

实验要求

在这个实验要求你实现内存管理和 MMU 开启。

你需要对你的内存管理函数进行**测试**，请在你的报告中体现出来。

同时在报告中记录的还有你对于 **MMU 的设置**和**内存管理**部分的修改。

Lab3: 异常处理和进程管理

> 在本部分实验中，完成 Aarch64 体系下异常的处理和实现用户态程序运行

Aarch64 异常模型

Aarch64 共设计了 4 个异常级别（忽略 Secure 特性），分别是：

- EL0 用户的应用程序（用户态）
- EL1 操作系统内核（内核态）
- EL2 Hypervisor 虚拟化监视器（支持处理器核心的虚拟化）
- EL3 安全监视器（支持安全状态）

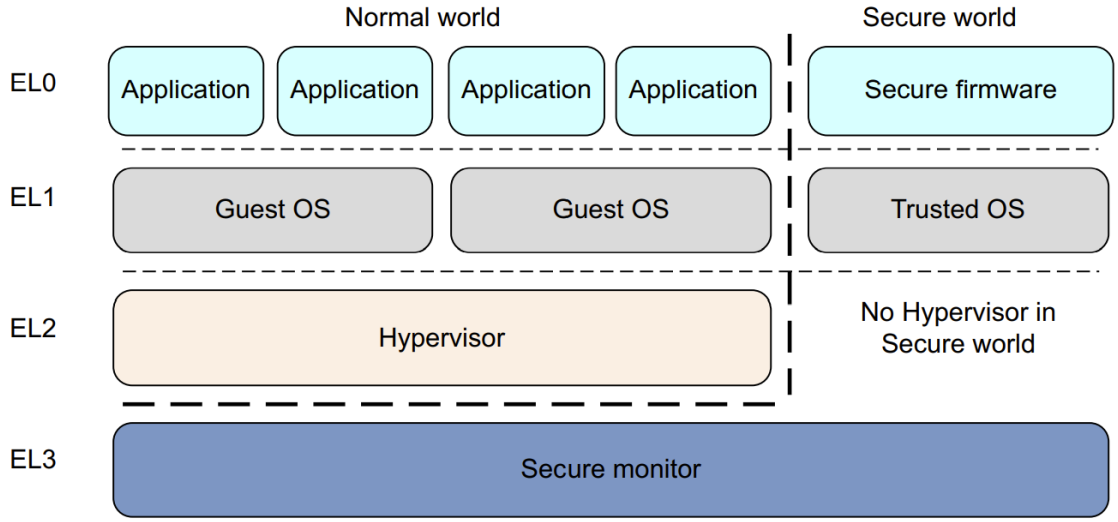

Figure 3-2 ARMv8 Exception levels in the Normal and Secure worlds

(ARM® Cortex®-A Series Programmer's Guide for ARMv8-A 3-2)

在本操作系统实验中你仅需要使用到前两个 EL。

一般而言处理器从用户态陷入内核态是需要一些条件的，比如内部因素的处理器异常（异常指令，对齐问题等）、特权指令（系统调用）、外部中断。

与 MIPS 不完全相同的是，Aarch64 结构下根据异常级和陷入原因的不同进入内核态时将跳转到不同的位置，即构成了中断向量表，如下表所示：

Address	Exception type	Description
VBAR_ELn + 0x000	Synchronous	Current EL with SP0
+ 0x080	IRQ/vIRQ	Current EL with SP0
+ 0x100	FIQ/vFIQ	Current EL with SP0
+ 0x180	SError/vSError	Current EL with SP0
+ 0x200	Synchronous	Current EL with SPx
+ 0x280	IRQ/vIRQ	Current EL with SPx
+ 0x300	FIQ/vFIQ	Current EL with SPx
+ 0x380	SError/vSError	Current EL with SPx
+ 0x400	Synchronous	Lower EL using AArch64
+ 0x480	IRQ/vIRQ	Lower EL using AArch64
+ 0x500	FIQ/vFIQ	Lower EL using AArch64
+ 0x580	SError/vSError	Lower EL using AArch64
+ 0x600	Synchronous	Lower EL using AArch32
+ 0x680	IRQ/vIRQ	Lower EL using AArch32
+ 0x700	FIQ/vFIQ	Lower EL using AArch32
+ 0x780	SError/vSError	Lower EL using AArch32

(Vector table offsets from vector table base address)

如此列举的一共有 4 组的异常向量，简单的说用途分别是 “不应该发生”、EL1 异常陷入 EL1 处理、EL0 异常陷入 EL1 处理、32 位相关的陷入。在本实验中你可能需要填充中间的两组异常向量。

请首先思考下面这个问题：

- 什么时候会发生 EL1 异常并陷入 EL1 来处理 EL1 的异常的情况?

中断向量表有一个要求是 `.align 11`，含义是这个向量的基地址的后 11 位为 0，也就是 2KB 对齐。处理器设计也要求每个向量的代码 `.text` 也满足一定的对齐要求。

> 请你查阅相关资料或代码，自行撰写一个满足对齐要求的简单的**中断向量**。

完成中断向量后，你需要将这个向量载入到 `vbar_el1` 寄存器中。

在 Sync（同步错误）中包括了本实验中的两种常见错误情况：

- 缺页错误（Page Fault）
- 系统调用（System Call）

为了区分具体错误类型，Aarch64 在设计时使用了 `esr_el1` 寄存器描述了异常的 “症状”。同时在缺页错误时，在寄存器 `far_el1` 中会去保存发生错误的虚拟地址。

> 请查阅手册，完成对缺页错误和系统调用错误的分发。
>
> 并根据 Aarch64 的访存机制，完成缺页错误的处理。（这块比 MIPS 简单很多）

IRQ 中断请求中包括了时钟中断，后文将介绍一个使用内置时钟的方法。

FIQ 在实验中可能是用不到的；Error 会在一些指令违例等情况出现。

TrapFrame

在异常发生的时候，陷入内核态的第一步就需要保存现场。

> 请查阅相关资料，确定需要保存的寄存器，设计一个合适的 `Trapframe`.
>
> 并书写相关汇编代码完成现场的保存和恢复。

注意在 Aarch64 中有能同时对两个寄存器操作的 ldp 和 stp 指令，可以加快保存现场的速度。

时钟中断

因为仿真器不具有使用博通 SoC 的时钟的功能，所以只能使用处理器内部的核心时钟，一般称为 Generic Timer。

这种内部时钟可以使用系统寄存器存取的方式（msr/mrs）来控制（当然也可以使用内存空间总线的方式）。这里涉及的系统寄存器有：

- CNTP_CTL_EL0 控制寄存器
- CNTP_TVAL_EL0 计数寄存器
- CNTP_CVAL_EL0 比较寄存器
- CNTFRQ_EL0 时钟频率

一个参考的控制位设定（来源 ARM 内核驱动），其字面意思可以帮助理解含义：

```
#define ARCH_TIMER_CTRL_ENABLE      (1 << 0)
#define ARCH_TIMER_CTRL_IT_MASK     (1 << 1)
#define ARCH_TIMER_CTRL_IT_STAT     (1 << 2)
```

一般的操作是向 TVAL 中填入一个值，则会触发一个 Countdown 的计数器模式，在启用控制寄存器后则会倒计时。如果期望的是一个具体的物理时间，则需要取出时钟频率寄存器的值，手动计算期望的寄存器的值。

> 请完成时钟相关的函数，并验证可用性。

你可以在启用计时器后轮询控制寄存器，观察其 STAT 位的值变化。当 STAT 为高位时，计时器处于异常状态。这个时候计时器的中断请求总线已经向中断控制器发出请求了，但是这个时候并没能观测到期望的时钟中断的软件过程。而很明显时钟内置的中断屏蔽位是 0，问题可能出在树莓派的中断控制器的设置，或者处理器的状态位（DAIF）设置。

树莓派的中断控制器是一组内存总线上的寄存器，可以参照文档（https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2836/QA7_rev3.4.pdf）进行设置。

虽然这是树莓派 2B 的文档，但是大部分的内存 IO 总线上的寄存器和设备都是和树莓派 3B 相同的。

> 请寻找到合适的中断控制器设定，并启用时钟的中断。

在启动时从 EL2 回落到 EL1 时，需要设置 `spsr_el12` 寄存器，寄存器的内容在异常返回（ERET）后会复制到 PSTATE 寄存器状态寄存器，请保证中断不被屏蔽。

与 “标准” MIPS 实验不同的是，这里的时钟是一次性的，并不是能周期性产生的。你需要在每次时钟结束之后重启时钟的倒计时。

由于 Arm 不能像 Mips 那样，在内核态自动屏蔽中断，因此你可能需要支持简单的**异常重入**。

> 请根据相关资料，完成时钟中断的设置。

用户程序加载

类似于 “标准” MIPS 实验中的用户程序加载，你需要将用户程序通过读 ELF 文件的方式动态加载到虚拟地址空间中。

> 请查找相关资料，完成 Aarch64ELF 文件的加载。

为了帮助测试进程调度，建议你在这里完成用户进程的链接实现。

你需要首先实现用户态的入口汇编 entry.S，设置用户栈的地址。之后跳转到 libmain 中获取进程控制块相关内容，最后跳到用户程序主函数 umain. 下面是两个参考用户程序。

```
int umain () {
    while (1) {
        printf ("a");
    }
    return 0;
}
```

```
int umain () {
    while (1) {
        printf ("b");
    }
    return 0;
}
```

这里 printf 的实现可以使用系统调用 SVC 实现。

在控制台中观察 a 和 b 的输出数量，一般来说符合认知的数量应该等于两次的时钟周期比。

> 请思考上面给出的两种时钟设置可以达到这个要求?

任务清单

- 异常向量
- 异常分发和处理函数
- 系统调用（部分）
- ELF 加载部分
- 进程管理和调度部分
- 用户态相关程序
- 用户态链接脚本
- ……

实验要求

在这个实验要求你实现时钟中断和异常处理。

你需要能验证你的内核确实产生了时钟中断，并且你确实对中断进行了处理。

你需要实现用户态进程的加载，并展示效果。

在报告中记录你对于**时钟的设置**、**异常处理**和**用户进程加载**部分的处理。

系统调用 / IPC 和 Fork

> 在本部分实验中你将完成全部系统调用和**用户态**Fork

系统调用

在你对 Aarch64 的异常模型理解的非常透彻的情况下，这部分内容比较简单，你只需类似前面的部分进行补充就可以了。

与 MIPS 结构不同，Aarch64 前 8 个参数均由寄存器传参。

> 为了减少出错，可以使用 C 代码完成系统调用的分发。

进程间通信

这部分和 MOS 中没有太大的区别，注意 32 位和 64 位区别即可。

Fork

一个完整的 Fork，主要要点有：

1. 不应该实现一个内核的系统调用的 fork（不增加系统调用）
2. 实现一种机制来遍历进程内存空间（类似于 `vpd/vpt` 的实现）
3. 实现一个由用户态处理自身进程的缺页中断的机制（Copy on Write 机制的用户态实现）
4. 实现在用户态下恢复 `Trapframe` 并回到正常状态的机制

为了实现用户态 Fork，我们还需要让用户态可以完成访问自身页表的操作，在 MIPS 实验中是如下的操作：

```
e->env_pgdir [PDX (UVPT)]  = e->env_cr3 | PTE_V | PTE_R;
```

在 Aarch64 中，你同样可以使用类似的操作完成自映射实现用户态页表的访问：

```
pgdir [GET_L1_INDEX (UVPT)] = p->pa | PTE_RO | PTE_USER;
```

代码中的 p->pa 是用户页目录的物理地址，PTE_RO 和 PTE_USER 都是权限位。

这里给出 UVPT、UVPM、UVPD 的定义：

```
#define UVPT 0x4000000000
#define UVPM (UVPT + (UVPT >> 9))
#define UVPD (UVPM + (UVPT >> 18))
```

在用户态页表遍历时，采用下面的方法即可：

```
for (i = 0; i < 512; i++) {
    if ((((*vpd)[i]) & VALID) == 0)
        continue;
    for (j = 0; j < 512; j++) {
        if (((*vpm)[(i << 9) + j] & VALID) == 0)
            continue;
        for (k = 0; k < 512; k++) {
            u64 pte = (*vpt)[(i << 18) + (j << 9) + k];
            if ((pte & VALID) == 0 || (pte & PTE_USER) == 0)
                continue;
            //do something
        }
    }
}
```

请你思考下面的问题，并实现你的内核中的页表遍历。

> 为什么那一句代码能实现自映射?

你需要仔细思考这个问题，也是笔者认为我们内核中**最美妙**的设计。

COW 的用户态处理

当发生缺页中断时，系统陷入内核态，内核通过鉴别一个同步（`sync`）错误的种类，来捕获特定的缺页中断，进入一个内核的处理函数。

> 请查阅处理器文档，找到缺页中断对应的错误编号（Exception Class），并将缺页中断导向一个内核 handle。

这里内核可以通过系统寄存器 `far_el1` 来获得发生缺页中断的地址，在进程为自身设置了 `set_pgfault_handler` 的前提下（也就是在 `env` 的 `env_pgfault_handler` 域非空），系统在判别这是不是一个尝试写入 “只读” 的 COW 页面的操作，如果是则需要直接通过设置 `elr` 和 `sp` 寄存器，分别为处理函数地址和异常处理栈指针（`XSTACKTOP`），使得此处能够返回到用户定义的缺页中断的处理函数。此处还需要传递发生缺页错误的地址信息。

用户进程设置了一个 `__asm_pgfault_handler` 的用户态处理缺页中断的过程。进入这一过程后，首先会调用 `pgfault` 函数，这个函数需要一个参数，也就是发生缺页中断的地址。从这一函数返回后则需要恢复进程原本的运行状态。

用户态恢复 Trapframe

首先在内核捕获缺页时的 Trapframe 就应该是进程原本的运行状态，而异常返回后则会有信息被破坏。所以需要先行将 Trapframe 复制到一个 “用户可见的” 的位置，比如异常处理栈内：

```
struct Trapframe *tf = (struct Trapframe *)(K_TIMESTACK_TOP - sizeof (struct
Trapframe));
bcopy (tf, (void *) (U_XSTACK_TOP - sizeof (struct Trapframe)), sizeof (struct
Trapframe));
```

> 这个时候需要将返回的栈指针设置到一个合适的位置。在返回用户态，并通过 `pgfault` 函数处理完 COW 复制后，请撰写一段汇编代码尝试恢复现场。

在 MIPS 实验代码中，对于 sp 寄存器的恢复是通过延时槽实现的。但是在 Aarch64 中并没有延时槽这种设计，同时因为受限于指令格式，不能直接通过 `LDR` 设置 `pc/sp` 寄存器。因此你必须使用一个数字编号的寄存器来作为 “中间变量”，而这种做法也带来了新的问题。

> 请查阅文档，利用一个 “保留用途” 的数字编号寄存器来完成这些操作。

任务清单

- 系统调用（全部）
- 进程间通信部分
- Fork 内核处理部分
- Fork 用户处理部分
- ……

实验要求

你需要实现**用户态 Fork**，请在报告中请体现出来。

当然对于 Fork，你依然需要进行**测试**，这点也请在报告体现。

Lab5：文件系统

> 在本部分实验你将完成文件系统

磁盘驱动

在 MIPS 实验中，IDE 磁盘驱动的实现是依赖于内核地址存取的，但是这并不是一种在树莓派上能够实现的方法。

在树莓派上最接近文件系统的硬件是 SD 卡，而通常而言 SD 卡的总线控制和 EMMC 总线相同，你需要实现一个读取 SD 卡上的扇区的驱动，进而实现一个 “磁盘驱动”。

具体的 EMMC 驱动的实现，请自行（在借助他人的代码的帮助下）完成即可。

磁盘镜像创建

使用下面的语句创建 sdcard 镜像：

```
dd if=fs/fs.img of=sdcard.img seek=10000
```

这里需要警示的是 `dd` 可是号称 Disk Destroyer 的内置命令，请在使用该命令时保持极度的谨慎！

在使用仿真器加载时，添加参数 `-drive file=sdcard.img,format=raw`。此处的扇区偏移不是必须的，因为内核读取并不依赖于 FAT 分区，你可以直接将 `fs.img` 放在 `sdcard.img` 的头部，或者直接使用 `fs.img` 作为磁盘镜像。

> 请注意文件系统镜像的字节序（Endian）的问题，这里的设定也和处理器 MMU 设置相关，你可能需要修改代码生成期望的文件系统镜像。

Windows 系统下的 WSL 并没有 USB 块设备的文件描述符实现，所以如果你使用的是 Windows 操作系统而又希望写入物理 SD 卡，你可以尝试使用 dd for windows （http://www.chrysocome.net/dd）。这里没有任何保证，请谨慎操作。

用户态 IO

在用户态下访问设备 IO 的地址空间这里给出两种典型的方案：

1. 将设备的地址总线的页面映射到文件系统服务进程的地址空间上，并在用户程序实现一个用户态的硬件驱动。
2. 实现一个系统调用，在内核中实现硬件驱动，并读取扇区，写入到用户进程的地址空间上。

> 请讨论两种实现方案的优缺点，从安全性和微内核设计原则两个角度。

块缓存

树莓派 3 使用的是 Aarch64 v8.0 架构的指令集，而 Aarch64 的页表脏位权限是从 v8.1 之后才支持的，所以在本实验中请舍弃全部的脏回写（或者全部写回）。

任务清单

- 磁盘驱动
- 磁盘镜像创建
- 文件系统部分
- ……

实验要求

请完成自己的文件系统，并对这个文件系统进行测试。

请把主要任务记录在**你的报告**中。

Lab6：Shell

> 这部分实验你将完成一个命令行的外壳接口

这部分实验与体系结构差异基本不大，你可以尽情去发挥了，当然并不是说难度不大，需要你耐心下去进行移植。

Spawn

如果你实现 Spawn 你会发现 MIPS 参数压栈和 Aarch64 参数压栈方式是不完全一样的。

你需要找到能解决 Aarch64 压栈方式的办法。

实验要求

请在实验报告中记录在 Shell 移植过程中的区别。

并在申优答辩中对你的 Shell 进行展示。

后记

树莓派实验大概并没有那么像上面说的那么容易，但是不管怎样，相信做到这一步的你是亲历亲为的完成了整个实验，我由衷的为你的坚持感到敬佩。

这版指导书是在梁远志学长版本的基础上重新修订的，感谢他在我们移植过程中的帮助。

同时感谢李思然学长在树莓派板子上移植时的帮助。

新版本指导书中包含了我们对于 Aarch64 的理解和移植过程中遇到的各种困难的思考，或多或少有不到位的地方。如果你有对实验设计、教学体验、指导书有着建议或者意见，请务必告诉我们。

陈纪源 郭衍培

2022/2/23

附录

附录 A

> 本部分记录使用 QEMU 的调试技巧

QEMU 调试技巧

退出 QEMU

`ctrl+A x`：注意是先按 ctrl+A 后再按 x

增加 Interrupt 模式

```
qemu-system-aarch64 -M raspi3 -kernel kernel.img -drive
file=qemu/fs.img,if=sd,format=raw -serial stdio -d int
```

可以输出中断异常信息，包括显示产生中断时各种寄存器的值。

增加 Asm 模式

```
qemu-system-aarch64 -M raspi3 -kernel kernel.img -drive
file=qemu/fs.img,if=sd,format=raw -serial stdio -d in_asm
```

以汇编指令的形式输出指令。

增加反汇编模式

```
aarch64-elf-objdump -D kernel.elf >kernel.txt
```

查看内核的反汇编代码，帮助了解各种虚拟地址位置。

GDB 调试

准备工作

原理主要是利用 qemu 与 gdb 进行通信，在 gdb 中进行单步调试。

需要安装 gdb-multiarch，注意不是 gdb，因为需要在 aarch64 体系结构上调试。

启动

在实验仓库下执行如下的内容：

```
qemu-system-aarch64 -M raspi3 -kernel kernel.img -nographic -drive
file=fs.img,if=sd,format=raw -s -S
```

注意：在执行该前需要编译好内核 kernel.img.

在另一个Shell中启动 gdb 指令

```
gdb-multiarch kernel.img
```

与 qemu 进行通信

```
target remote:1234
```

之后就可以使用 gdb 命令调试了。

GDB 命令

查看源码

```
list // 显示当前行往上下 5 行的源代码，注意要导入符号表后才能使用
list 5,10 // 显示第 5 行到第 10 行的源代码
```

设置断点

```
b 303 // 第 303 行设置断点
info b // 查看断点信息
b 7 if n == 6 // 条件断点
enable breakpoints // 启用断点
disable breakpoints // 禁用断点
```

运行程序

```
c //continue
```

执行下一条语句

```
n //next，函数直接执行不进入
s //step，函数会进入执行
ni //next instruction 单步汇编，函数直接执行不进入
si //step instruction 单步汇编，函数会进入执行
```

打印命令

```
p n //print，输出变量 n
p 0x80001000 //print，输出地址
display n // 追踪变量
undisplay n // 取消追踪变量
```

查看寄存器

```
info registers // 后面可以设置寄存器的名字
```

退出 gdb

```
quit
```

显示汇编模式

```
layout asm
```

附录 B

> 本部分记录向树莓派板子时需要的注意事项

说明

QEMU做为硬件模拟器，可以模拟板子上大部分硬件。但它本质上来说，还是“模拟器”，无法保证100%准确。这里还是建议同学们在QEMU上测试成功后一定要上板子进行测试。

树莓派的启动

现代微系统的启动过程一般情况下都十分复杂，以一个典型的启动过程为例：

1. 系统上电，复位
2. PC指向BIOS程序初始位置
3. BIOS完成硬件初始化，读取磁盘将系统移交给磁盘的引导记录上的程序段（一般称为Bootloader）
4. Bootloader读取磁盘，加载需要启动的操作系统的内核到内存中
5. Bootloader将系统移交给操作系统内核

近几年兴起的UEFI固件意图通过取代BIOS，能够极大的简化操作系统内核加载的过程。作为题外话，你可以研究一下这种新型的固件能够对操作系统启动部分的开发带来什么样的改变？

非常幸运的是树莓派没有这种如此繁复的过程，因为大部分的固件都不集成在树莓派的开发板上，而是以文件的方式写入在SD卡上，给出一个最简化的文件列表：

1. bootcode.bin 用于辅助CPU启动的GPU程序二进制镜像
2. config.txt 纯文本的内核启动配置
3. fixup.dat 固件的修补
4. start.elf GPU的内核ELF镜像
5. kernel8.img 需要加载的内核

前四个部分可以在Raspberry的官方Github仓库中找到：raspberrypi/firmware (github.com)，第五个就是你完成的内核。

config.txt的内容如下，其目的是将树莓派启动到Aarch64的EL2（异常级别-2）状态，同时识别并加载文件名为kernel8.img的内核镜像。

```
arm_control=0x200
```

> Pi3 在发布初期使用的固件（包含 bootcode.bin, config.txt, start.elf）在启动时需要添加 kernel_old 设置，且是从 0x00000000 开始的内核 text 起始位置。然后在固件版本更新后，就不再需要这一设置，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

你需要将上述的5个文件拷贝到以FAT文件系统格式化的SD卡的根目录中，连接好串口通信的线缆，上电。

UART0

树莓派中默认使用UART1做为串口的输入输出方式。为了使用UART0做为串口通信方式，你需要通过mailbox通知GPU修改相关配置。

> 树莓派启动时先启动GPU，在进行相关硬件设置后再将CPU启动

mailbox传输的信息你可以在上面有关UART0驱动找到。

这里提醒一下，QEMU并没有对GPU等其他硬件进行模拟。

页表的设置

在MIPS中，页表的翻译是通过软件写tlb，而在Aarch64中是由硬件写MMU。

当手动填写页表后，如果我们立马访问对应的虚拟地址，可能会由于时间过短导致页表并没有被填上。

最后的办法是在所有对于页表的修改都增加同步指令：

```
asm volatile ("dsb sy");
asm volatile ("isb");
```