



# 第十章 实验 机器启动

为了使得本科生在学习本操作系统课程的过程中能够进行动手实践、加深理解，我们设置了与课程相配套的实验。课程实验的代码部分来自于上海交通大学并行袁分布式系统用于系统内核操作系统原型。由于课程实验主要面向接触操作系统的大二或大三本科生，因此我们对ChCore代码进行了大量裁剪袁简悔，并添加了一些练习，从而形成了ChCore课程实验(ChCore Labs)。在设置课程实验的过程中，我们也受到了MIT6.828课程OS Lab的启发，同时收到了上海交通大学操作系统课程(2020年)选修学生的一些反馈，在此一并感谢。

## 19.1 简介

本实验作为ChCore课程实验的第一隔实验，分两部分：第一部分介绍ChCore实验的基本知识，包括ARM汇编、QEMU模拟器的使用；第二部分熟悉ChCore的引导加载程序(bootloader)；第三部分实现一些简单的内核态功能。

### 19.1.1 得分

实验1中，代码部分的总得分。可使用如下命令检查当前得分：

```
;?8. 9-71 3>-01
...
%>1
```

## 19.2 第一部分：实验基本知识

本部分旨在熟悉 ARM 汇编语言，以及使用 J 和 QEMU/GDB 调试。在此过程中，需要回答一些问题。

### 19.2.1 熟悉 AArch64 汇编

AArch64 是 ARMv8 ISA 的 64 位执行状态。《ARM 指令集参考指南》[1] (力直接是一隔页逐页逐页逐页的手册。在 ChCore 实验逐页，逐页逐页要在提示下可以理解一些关键汇编和编写简单的汇编代码即可。

#### 练习 1

浏览《ARM 指令集参考指南》的 A1、A3 和 D 部分，以熟悉 ARM ISA。请做好阅读笔记，如果之前学习 x86-64 的汇编，请写下与 x86-64 相比的一些差异。

### 19.2.2 构建 ChCore 内核

课呈实验逐页，使用 J (版本 = 3.1) 作为硬件模拟器来模拟 Raspberry Pi 3 (Raspi3)，QEMU 可以调用 GDB 进行调试 (如打印输出、单步调试等)。通过 QEMU 调试过的操作系统内核，可以尝试放在真实的硬件上进行测试。

在 /4/; >1 目录下，输入以下命令可以使用 Docker 进行交叉编译 (即在 x86-64 平台上编译 AArch64 代码)，构建 ChCorebootloader 和内核。

```
/4/; >1 9-71 .A580
```

如果构建呈隔页，可以找到 AArch64 的 18 593，隔页调用像 bootloader 和内核，可以作为一隔页虚拟的啥在 QEMU 上运行。调用像啥由 Ž-712581? @D 中的链接规则指定。输入以下命令运行 QEMU：

```
/4/; >1 9-71 =19A
```

ChCore 的标准输出将汇编显示在 QEMU 逐页：

```
fz~! ^4"; >1 A- >@ 5: 5@ 25: 5?410
fz~! ^00>1?? ; 2 9- 5: 5? D
```

要退出 QEMU，请输入 @8 - D (同时输入 @8 和 -，然后输入

### 19.2.3 QEMU 和 GDB

在实验过程中，由于要在 64 位系统上使用来调试 AArch64 代码，因此使用 32 位的 QEMU 代替了 64 位的。使用 GDB 调试的 QEMU 可以启动一个 32 位的目标（使用 `-target armv7` 或 `-target armv7t` 参数启动 QEMU）。在真正逐行跟踪像逐行的指令前等待目标的连接。开启 QEMU 呈现目标之镜像，可以开启 GDB 跟踪行调试，它会在某个端口上进行监听。QEMU 提供一个脚本 `30. 5: 5@` 来初始化 GDB，并且设置了其监听端口为 QEMU 的默认端口（`@/ <`）。

打开两个终端窗口，在 `>1` 目录下，输入 `71 =19A 30.` 和 `9- 71 30.` 命令可以分别打开带 GDB 调试的 QEMU 以及 GDB，在 QEMU 终端将看到下面的输出：

```
D
30.
```

#### 练习 2

启动带 GDB 的 QEMU，使用 GDB 的 `C41>1` 命令来跟踪跟踪口（第一跟踪函数）及 bootloader 的地址。

## 19.3 第二部分：内核的引导阶段 加载

Rasp3 从存储（SD 卡）中加载 32 位的 bootloader 并执行。bootloader 包括两个功能：

1. bootloader 通过函数 `>9 18* @, 18` 将处理器的异常级别从其塌缩级别切换到 L1。《ARM 指令集参考指南》的 A3.2 节简要概述了异常级别。
2. bootloader 初始引导 ART，页表和 MMU。然后 bootloader 跳转到实际的内核。在后续的实验中将会描述内存结构。

bootloader 的源代码由一个汇编文件 `@>@ %` 和一个 C 文件 `件.; ; @ 5: 5@ / /` 组成。

### 19.3.1 编译后的可执行目标文件

在编译并链接 Linux 内核的可执行目标文件时, 编译器将源文件 ( .c ) 和汇编文件 ( .s ) 编译成目标文件 ( **object file** ) ( .o )。目标文件是用二进制格式编码的机器指令编写的, 但是由于目标文件内的符号地址等信息完全缺失, 因此不能被直接运行。然而, 链接器将所有已编译的目标文件链接 ( 即在目标文件中填充其缺失的符号地址 ) 成可执行目标文件 ( **executable object file** ), 例如 A580 71>: 18 593, 这个目标文件是硬件可以运行的二进制逐条机器指令组成的。可执行目标文件的另两个格式是链接格式 ( **Executable and Linkable Format, ELF** ) 二进制文件。

ELF 可执行目标文件从 ELF 头部 ( **ELF header** ) 开始, 然后紧跟程序段 ( **program section** )。ELF 头部记录目标文件的结构, 目标文件呈哑段结构是一组力的二进制逐条块, ( 硬件或软件 ) 加载器将目标文件作为代码或数据加载到指定地址的内存中并开始执行。

以 A580 71>: 18 593 目标文件为例, 可以通过以下命令看到完整的头部信息:

```
/4/;>1 >1-0182 4 . A580 71>: 18 593
.1~ /1-01>
. 35/ 2 /
*8-??
"- @
? /; 9<8191: @ 85@@81 1: 05-:
(1>?5;:
/A>1: @
!%`fi
'žfi* %E?@19 (
`fi (1>?5;:
&E<1
,*.* .DI/A@. 81 2581
. 35/ 2 /
(1>?5;:
D
.: @E <; 5: @ -00>1??
D
%@ >@ ; 2 <>; 3>- 9 41- 01>?
. E@1? 5: @ 2581
%@ >@ ; 2 ?1/@5;: 41- 01>?
. E@1? 5: @ 2581
~8-3?
D
%5F1 ; 2 @45? 41- 01>
. E@1?
%5F1 ; 2 <>; 3>- 9 41- 01>?
. E@1?
žA9. 1> ; 2 <>; 3>- 9 41- 01>?
%5F1 ; 2 ?1/@5;: 41- 01>?
. E@1?
žA9. 1> ; 2 ?1/@5;: 41- 01>?
%l/@5;: 41- 01> ?@5: 3 @. 81 5: 01D
```

通过以下命令, 看到 A580 71>: 18 593 包含的哑段:

```
/4/;>1 >1-0182 % . A580 71>: 18 593
```



2. 逐行读入，要将 内存 的段“放到”(可通过拷贝或页表映射等方式) 拟内存地址(Virtual Memory Address, VMA), 然后开始真正逐行读入文件中的代码。

大多数情况下，一段 LMA 和 VMA 是相同的。[2]

通过; . 60A9< 也可以查看 ELF 内存 的段 一段 的 VMA :

```
/4/; >1 ; . 60A9< 4 . A580 71>: 18 593
```

#### 练习 4

查看 . A580 71>: 18 593的; . 60A9<信息。比较一段 的 LMA 是否相同，内存 什 和 LMA 不同的情况下，内核是啥 何 将一段 的 LMA 变 VMA? 提示：从一段 的加载和运行情况 进行分析

## 19.4 第 部分：内核 基础 能

### 19.4.1 内核 输 输出

为了在引导阶段和内核 行过 逐 的 试等 能， 要 呈 标准输 输出。在 Core 逐 ，内核 标准输出函数 在 71>: 18 /; 99; : <>5: @7 /中。其 能，和呈 用的逐 的 式 标准输出<>5: @2 能力 似，不同的是5: @2 是用 可以 用的系统 用，其实现是 用的是内核 的 式 输出，而现在 要实现的正是内核 的格式化输出。

#### 练习 5

以不同的 逐 打印数字的 能 ( 16 ) 尚 实现，请在 71>: 18 /; 99; : <>5: @7 /逐 填 5: @7 C>5@1 : A9以完 啥>5: @7的功能。

正确完呈 此练习 ，输 3>- 01可通过<>5: @ 41D和<>5: @ ; /@两个测试。

## 19.4.2 函数栈

表 19-1 展示了标准输出函数，程序员可以增加更多用于测试的内核功能，堆栈函数。Arch64 的函数栈使用的是 32 指令（类似于 86-64 的 32 指令），并且使用栈结构保存函数信息：函数名，函数的返回地址、传递参数等、上一个栈的指针等。因此，这些函数栈中的信息可以用来追踪函数的调用情况。

**栈指针 (Stack Pointer, SP)** 寄存器 (AArch64 使用 X30 寄存器) 指向当前正在使用的栈底（即栈上的最低地址）。栈的增量方向是内存地址从大到小的方向，弹出和压栈是栈的两项基本操作。将值压入栈时，栈指针要减少；将值写入栈时，栈指针指向的地址。从堆栈弹出一项值则是读取指向的值，然后栈指针增加。

与之相反，**帧指针 (Frame Pointer, FP)** 寄存器 (AArch64 使用 X29 寄存器) 指向当前正在使用的栈底（即栈上的最低地址）。X29 寄存器位于 SP 之间的内存空间，即当前正在执行的函数的栈空间，用于保存临时变量等。在 Arch64 中，SP 和 FP 都是 4 字节的地址，并右对齐（即保证可以被 8 整除）。

### 练习 6

内核栈初始值（即初始 SP 和 FP）的代码位于哪一行？内核栈在内存中位于哪一行？内核中有什么栈保留空间？

在调用函数时，该函数在真正执行函数内部逻辑之前会有一些初始化栈帧指针的代码：通常通过将上一函数栈帧使用过的栈来保存当前的栈，然后将当前的 SP 值复制到 FP 中。此外，这段初始化代码也记录了函数的返回地址、保存函数参数、保存寄存器的值等作用。返回地址保存在 **链接寄存器 (Link Register, LR)** (AArch64 使用 X31 寄存器) 中。根据这些调用惯例 (**calling convention**)，可以通过读取已保存的 SP 指针来追踪函数的调用顺序以及函数栈。这项特性可以用于调试，如定位代码的执行路径、查看调用函数时所用的参数等。



## 练习 7

熟悉了 AArch64 上的函数调用惯例，请在 18.9.5 节通过 GDB 找到 `?@ /7 @1?@` 函数的地址，在断点处设置断点，并检查在内核启动后的每次调用情况。观察 `?@ /7 @1?@` 函数嵌套级别将多少寄存器值压入堆栈，这些值是什么？提示：GDB 可以将寄存器打印为地址及其 64 位值或数组，力能：

```

30. D 3 D
D222222 2 71>: 18 ?@ /7 D222222 2
30. D 3 D
D222222 2 71>: 18 ?@ /7 D222222 2
D222222 0 /
D222222 2 71>: 18 ?@ /7 D
D 1
D222222 2 71>: 18 ?@ /7 D222222 2
D222222 0 /
D222222 2 71>: 18 ?@ /7 D
D 1
D222222 2 71>: 18 ?@ /7 D222222 2
D222222 0 /

```

接下来，我们将使用 `usegdb` 来读取汇编代码直观地查看 AArch64 函数调用惯例，方法是在运行第一条指令（通过 GDB 命令？）时输入 `usegdb 5 ?@ /7 @1?@` 命令以显示 `?@ /7 @1?@` 函数开始的 30 行汇编代码。

## 练习 8

在 AArch64 上，返回地址（保存在寄存器），帧指针（保存在寄存器）和参数由寄存器传递。但是，当调用者函数（**caller function**）调用被调用者函数（**callee function**）时，我们将复用这些寄存器，这些寄存器返回来的值是什么？何被存在栈中的？请使用示意图表示，将函数返回的信息（寄存器、FP、LR、参数、部分寄存器值等）在栈中具体保存的位置在哪里？

ChCore 通过调用 `?@ /7 . - /7@>- /1` 函数进行栈帧管理，该函数定义在 `71>: 18 9; 5@> /`，并且该函数忽略函数本身。该寄存器逐条的 `>1- 0 2<` 函数可以通过内力汇编的方式，直接读到当前的值。<sup>1</sup> `?@ /7 . - /7@>- /1` 的输出格式如下：

<sup>1</sup>关于 AArch64 GCC 内联汇编，可以参考《ARM GCC 内联汇编程序手册》[3]（力能接

```
%@ /7 . - /7@>- /1
1$ 222222 0 / ~" 222222 2 ~>3?
222222 2 222222 0 /
1$ 222222 0 / ~" 222222 2 ~>3? 1
222222 2 222222 0 /
1$ 222222 0 / ~" 222222 2 ~>3? 1
222222 2 222222 0 /
```

输出的第一行反映了栋用 `?@ /7 . - /7@>- /1` 的函数，第二行反映了栋用该函数的函数，依此类推，其终止条件可以通过“练习 7”得知。

输出结果的藐一行包含 FP 和 Args，并且以十六进制表示。FP 表示函数栈的帧指针（D），即隔纠啥隔函数并悔逐值。LR 表示函数返回悔悔的指令栋址，即栋用着函数下一条指令。`. 8 8 . 18` 指令跳转到 `8 . 18`，并将寄存器 D 设逐梧。最后，在 `~>3?` 之后列出的五个值是所函数的前五个参数。如果函数的参数少于 5 个，则多余的值是无效的。

力啥，第一行是 `@ /7 @1?@` 的信息。在这一行逐 LR 表示 `?@ /7 @1?@` 返悔之悔的指令栋址是隔函数初始悔栈之悔的 FP，而 Args 梧（悔藐值梧效）。

### 练习 9

使用袁示力相同的隔式 `1>: 8 9; 5@ > /逐` 实现 `?@ /7 . - /7@>- /1`。梧了忽略编译器袁悔等级的影响，逐哑要考虑 `?@ /7 @1?@` 的情况，我们已经强制了这个函数编译优化等级。

挑战：请思考，啥果考虑更多情况（力啥，多隔参数）时，应当啥何纠行悔塌操作？

正确完成该练习后，输入 `9- 71 3>- 01` 将悔显示全部分，输出啥下：

```
1 ~ @1?@
? />5<@? 3>- 01 8 .
>A: 5: 3 * 4*; >1
<>5: @ 41D !L
<>5: @ ; /@ !L
?@ /7 /; A: @ !L
?@ /7 ->3A91: @? !L
?@ /7 8> !L
% /; >1
```

参考书籍 献

[1] ARM. Arm instruction set reference guide. 4@@<?  
?@ @5/.0; /?.->9./; 9 ->9 5: ?@>A/@5; : ?1@  
>121>1: /1 3A501 1: .<02, 2018.

[2] Steve Chamberlain and Ian Lance Taylor. Using ld the gnu linker, 2010.

[3] Ethernut. Arm gcc inline assembler cookbook. 4@@<  
CCC.1@41>: A@01 1: 0; /A91: @? ->9 5: 85: 1 -?9.4@98, 2014.

实验 1：啥 码反馈



