第十九章　实验 1: 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与分布式系统研究所用于系统研究工作的 ChCore 微内核操作系统原型。由于课程实验主要面向刚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与简化，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 JOS 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 课程实验的第一个实验，分为三个部分：第一部分介绍 ChCore 实验的基本知识，包括 ARM 汇编语言与 QEMU 模拟器的使用；第二部分熟悉 ChCore 的**引导加载程序（bootloader）**；第三部分需要对 ChCore 实现一些简单的内核态功能。

19.1.1　准备工作

在开始本实验之前，请按如下命令行命令所示，切换到`lab1`的分支。

```
$ git checkout lab1
```

19.1.2　评分

实验 1 中，代码部分的总成绩为 80 分。可使用如下命令检查当前得分：

```
chcore$ make grade
```

```
...
Score: 100/100
```

在完成实验的过程中，可以使用`git commit`时常保存进展，以便之后调试回滚：

```
chcore$ git commit -am "some detailed information ..."
```

完成实验 1 后，请使用如下指令进行提交：

```
chcore$ git commit -am "Finish lab1"
```

评分：编码（80%）+ 文档（20%）

对于每个 ChCore 实验，需要上交一份记录实验过程的实验报告。如果对实验有任何建议，也可以在文档中记录。在实验 1 中，该文档名为`lab1.pdf`或`lab1.md`，并存储在`doc/`目录下（使用`mkdir doc`命令创建该目录）。可以通过以下方式将其添加到本地`git`仓库中：

```
chcore$ git add doc/lab1.pdf
chcore$ git commit -m "add document"
chcore$ git push origin lab1
```

在完成实验的过程中，可以使用以下方式，提交代码或者文档的更新，建议做完每个练习后提交一次：

```
chcore$ git commit -am "Exercise xx done!"
chcore$ git push origin lab1
```

在`chcore`目录下，输入`make grade`可以获取实验成绩。有两点需要注意的是：

- 除文档外，只能修改每个练习中指定的文件，并且不能创建任何其他文件或修改脚本。评分，我们将用原始文件覆盖它们。每次 git 推送时，请确保代码可以成功构建；否则，将获得 0 分。

- 除了本地测试集，我们还准备了一个远程测试集来测试您的代码。您代码的最终分数将与本地测试集和远程测试集结合在一起。

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 QEMU 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]（链接）是一个帮助入门 ARM 语法的手册。在 ChCore 实验中，只需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

课程实验中，使用 QEMU（版本 >= 3.1）作为硬件模拟器来模拟 Raspberry Pi 3（**Raspi3**），QEMU 可以结合 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`chcore`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。

```
chcore$ make build
```

如果构建成功，可以找到镜像`build/kernel.img`，该镜像包含 bootloader 和内核，可以作为一个虚拟的闪存在 QEMU 上运行。镜像入口由`CMakefiles.txt`中的链接规则指定。输入以下命令运行 QEMU：

```
chcore$ make qemu
```

ChCore 的标准输出将会显示在 QEMU 中：

```
[INFO] [ChCore] uart init finished
[INFO] Address of main() is 0x
```

要退出 QEMU，请输入`Ctrl + a x`（同时输入`Ctrl`和`a`，然后输入`x`）。

19.2.3 QEMU 与 GDB

在实验中，由于需要在 x86-64 平台上使用 GDB 来调试 AArch64 代码，因此使用`gdb-multiarch`代替了普通的`gdb`。使用 GDB 调试的原理是，QEMU 可以启动一个 GDB **远程目标(remote target)**(使用`-s`或`-S`参数启动)，QEMU 会在真正执行镜像中的指令前等待 GDB 客户端的连接。开启远程目标之后，可以开启 GDB 进行调试，它会在某个端口上进行监听。我们提供了一个 GDB 脚本`.gdbinit`来初始化 GDB，并且设置了其监听端口为 QEMU 的默认端口(`tcp::1234`)。

打开两个终端，在`chcore`目录下，输入`make qemu-gdb`和`make gdb`命令可以分别打开带有 GDB 调试的 QEMU 以及 GDB，在 QEMU 中将会看到如下的输出：

```
...
0x0000000000080000 in ?? ()
(gdb)
```

练习 2

启动带调试的 QEMU，使用 GDB 的`where`命令来跟踪入口（第一个函数）及 bootloader 的地址。

19.3 第二部分：内核的引导与加载

Raspi3 从闪存（SD 卡）中加载`.img`镜像中的 bootloader 并执行。bootloader 包括两个功能：

1. bootloader 通过函数`arm64_elX_to_el1`将处理器的异常级别从其他级别切换到 EL1。《ARM 指令集参考指南》的 A3.2 节简要描述了异常级别。

2. bootloader 初始化引导 UART，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 的源文件由一个汇编文件`boot/start.S`和一个 C 文件`boot/init_c.c`组成。

19.3.1 编译与可执行文件

在编译并链接诸如 ChCore 内核的可执行文件时，编译器会将每个 C 文件（.c）和汇编文件（.S）编译成为**目标文件（objective file）**（.o）。它是用二进制格式编码的机器指令编写的，但是由于文件内的符号地址等信息未完全生成，因此不能直接被运行。然后，链接器将所有已编译的目标文件链接（即在文件中填充其他文件中符号的地址等）成一个**可执行目标文件（executable objective file）**，例如build/kernel.img，这个文件中是硬件可以运行的二进制机器指令组成的。可执行目标文件的常见格式是**可执行和可链接格式（Executable and Linkable Format，ELF）**二进制文件。

ELF 可执行文件以 **ELF 头部（ELF header）**开始，后跟几个**程序段（program section）**。ELF 头部记录文件的结构，每个程序段都是一个连续的二进制块，（硬件或软件）加载器将它们作为代码或数据加载到指定地址的内存中并开始执行。

以build/kernel.img文件为例，可以通过以下命令看到完整的 ELF 头部信息：

```
chcore$ readelf -h build/kernel.img
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           AArch64
  Version:                           0x1
  Entry point address:               0x80000
  Start of program headers:          64 (bytes into file)
  Start of section headers:          405808 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         4
  Size of section headers:           64 (bytes)
  Number of section headers:         17
  Section header string table index: 16
```

通过以下命令，看到build/kernel.img包含的程序段：

```
chcore$ readelf -S build/kernel.img
```

现在对一些主要的程序段进行一些解释：

- `.init`：保存 bootloader 的代码和数据。这个特殊的段在`CMakefiles.txt`中定义。所有其余的程序段都是真正的 ChCore 内核。
- `.text`：保存内核程序代码，是由一条条的机器指令组成的。
- `.data`：保存初始化的全局变量或静态变量数据。定义在函数内部的局部非静态变量不在该段中存储。
- `.rodata`：保存只读数据，包括一些不可修改的常量数据，例如全局常量、`char *str = "apple"`中的字符串常量等。然而，如果使用`char str2[] = "apple"`，那么此时该字符串是动态地存在栈上的。
- `.bss`：记录未初始化的全局或静态变量，例如`int a`。由于在运行期间未初始化的全局变量被初始化为 0，因此链接器只记录地址和大小，而不是占用实际空间。

除上面列出的部分外，其他大多数段都包含调试信息，通常包含在可执行文件中，而不是加载到内存中。

练习 3

结合`readelf -S build/kernel.img`读取符号表与练习 2 中的 GDB 调试信息，请找出请找出`build/kernel.image`入口定义在哪个文件中。继续借助单步调试追踪程序的执行过程，思考一个问题：目前本实验中支持的内核是单核版本的内核，然而在 Raspi3 上电后，所有处理器会同时启动。结合`boot/start.S`中的启动代码，并说明挂起其他处理器的控制流。

19.3.2 内核的加载与执行

ELF 文件的**加载（load）**与**执行（execute）**是启动一个程序的两个重要的步骤：

1. 加载是指将程序的 ELF 文件按照链接规则从存储器（如 ROM）上按照每个段的**加载内存地址（Load Memory Address，LMA）**拷贝到内存上指定的地址。

2. 执行需要将 ELF 文件中的段“放到”（可通过拷贝或页表映射等方式）**虚拟内存地址（Virtual Memory Address，VMA）**，然后开始真正执行 ELF 文件中的代码。

大多数情况下，一个段 LMA 和 VMA 是相同的。 [2]

通过objdump也可以查看 ELF 文件中每一个段的 LMA 和 VMA：

```
chcore$ objdump -h build/kernel.img
```

练习 4

查看build/kernel.img的objdump信息。比较每一个段中的 VMA 和 LMA 是否相同，为什么？在 VMA 和 LMA 不同的情况下，内核是如何将该段的地址从 LMA 变为 VMA？**提示：**从每一个段的加载和运行情况进行分析

19.4 第三部分：内核态基础功能

19.4.1 内核态输入输出

为了支持在引导阶段和内核执行过程中的调试等功能，需要支持标准输入输出。在 ChCore 中，内核态标准输出函数printk定义在kernel/common/printk.c中。其功能，和常用的 libc 中的格式化标准输出printf()功能类似，不同的是，printf()是用户态可以调用的系统调用，其实现是需要调用的是内核态的格式化输出，而现在需要实现的正是内核态的格式化输出。

练习 5

以不同的进制打印数字的功能（例如 8、10、16）尚未实现，请在kernel/common/printk.c中 填 充printk_write_num以 完善printk的功能。

正确完成此练习后，输入make grade可通过print hex和print oct两个测试。

19.4.2 函数栈

有了格式化标准输出后，我们可以增加更多用于调测的内核态功能，例如堆栈回溯：AArch64 的函数调用使用的是`bl`指令（类似于 x86-64 中的`call`指令），并且使用栈结构保存函数调用信息：例如，函数的返回地址、所传入的参数等、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 中使用`sp`寄存器）指向当前正在使用的栈顶（即栈上的最低位置）。栈的增长方向是内存地址从大到小的方向，弹出和压入是栈的两个基本操作。将值压入堆栈需要减少 SP，然后将值写入 SP 指向的位置。从堆栈中弹出一个值则是读取 SP 指向的值，然后增加 SP。

与之相反，**帧指针（Frame Pointer，FP）**寄存器（AArch64 中使用`x29`寄存器）指向当前正在使用的栈底（即栈上的最高位置）。FP 与 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 AArch64 中，SP 和 FP 都是 64 位的地址，并且 8 对齐（即保证可以被 8 整除）。

练习 6

内核栈初始化（即初始化 SP 和 FP）的代码位于哪个函数？内核栈在内存中位于哪里？内核如何为栈保留空间？

在进入函数时，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：通常通过将上一个函数所使用的 FP 压入栈来保存旧的 FP，然后再将当前的 SP 值复制到 FP 中。此外，这段初始化代码也会记录函数的返回地址、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 中使用`x30`寄存器）中。根据这些**调用惯例（calling convention）**，可以通过遵循已保存的 FP 指针链来追溯函数的调用顺序以及函数栈。这个特点可以用于调试，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

为了熟悉 AArch64 上的函数调用惯例，请在kernel/main.c中通过 GDB 找到stack_test函数的地址，在该处设置一个断点，并检查在内核启动后的每次调用情况。每个stack_test递归嵌套级别将多少个 64 位值压入堆栈，这些值是什么含义？**提示：** GDB 可以将寄存器打印为地址及其 64 位值或数组，例如：

```
(gdb) x/g $x29
0xffffff000020f330 <kernel_stack+7984>: 0xffffff000020f350
(gdb) x/10g $x29
0xffffff000020f330 <kernel_stack+7984>: 0xffffff000020f350
      0xffffff00000d009c
0xffffff000020f340 <kernel_stack+8000>: 0x0000000000000001
      0x000000000000003e
0xffffff000020f350 <kernel_stack+8016>: 0xffffff000020f370
      0xffffff00000d009c
0xffffff000020f360 <kernel_stack+8032>: 0x0000000000000002
      0x000000000000003e
0xffffff000020f370 <kernel_stack+8048>: 0xffffff000020f390
      0xffffff00000d009c
```

下面，我们使用 GDB 来读取汇编代码直观地理解 AArch64 中函数调用惯例，方法是在运行第一条指令（通过 GDB 命令s）后输入x/30i stack_test命令以显示stack_test函数开始的 30 行汇编代码。

练习 8

在 AArch64 中，返回地址（保存在x30寄存器），帧指针（保存在x29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 fcuntion）**时，为了复用这些寄存器，这些寄存器中原来的值是如何被存在栈中的？回溯函数所需的信息（如 SP、FP、LR、参数、部分寄存器值等）在栈中具体保存的位置在哪？请画出 AArch64 函数调用的栈帧示意图。

ChCore 通过调用stack_backtrace()函数进行栈回溯，该函数定义在kernel/monitor.c，并且回溯结果忽略该函数本身。该文件中的read_fp()函数可以通过内联汇编的方式，直接读到当前 FP 的值。[1] stack_backtrace的输出格式如下：

[1] 关于 AArch64 GCC 内联汇编，可以参考《ARM GCC 内联汇编程序手册》 [3]（链接）。

```
Stack backtrace:
  LR ffffff00000d009c  FP ffffff000020f330  Args 0 0
   ffffff000020f350 ffffff00000d009c 1
  LR ffffff00000d009c  FP ffffff000020f350  Args 1 3e
   ffffff000020f370 ffffff00000d009c 2
  LR ffffff00000d009c  FP ffffff000020f370  Args 2 3e
   ffffff000020f390 ffffff00000d009c 3
  ...
```

输出的第一行反映了调用stack_backtrace的函数，第二行反映了调用该函数的函数，依此类推，其终止条件可以通过“练习 7”得知。

输出结果的每一行包含 LR，FP 和 Args，并且以十六进制表示。FP 表示函数栈的帧指针（x29），即刚进入该函数并设置 FP 后的值。LR 表示函数返回后的指令地址，即调用者函数bl指令的下一条指令。bl [label]指令跳转到label，并将寄存器x30设置为PC + 4。最后，在Args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。

例如，第一行是stack_test(0)的信息。在这一行中，LR 表示stack_test(0)返回之后的指令地址，FP 是该函数初始化栈之后的 FP，而 Args 为 0（后面 4 个值无效）。

练习 9

使用与示例相同的格式，在kernel/monitor.c中实现stack_backtrace。为了忽略编译器优化等级的影响，只需要考虑stack_test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，如果考虑更多情况（例如，多个参数）时，应当如何进行回溯操作?

正确完成该练习后，输入make grade将会显示全部 80 分，输出如下：

```
LAB1: test >>>>>>>>>>>>>>>>>
./scripts/grade-lab1
running ChCore:
  print hex: OK
  print oct: OK
  stack count: OK
  stack arguments: OK
  stack lr: OK
Score: 80/80
```

参考文献

[1] ARM. Arm instruction set reference guide. https://static.docs.arm.com/100076/0100/arm_instruction_set_reference_guide_100076_0100_00_en.pdf, 2018.

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

[3] Ethernut. Arm gcc inline assembler cookbook. http://www.ethernut.de/en/documents/arm-inline-asm.html, 2014.

实验 *1*：扫码反馈