

实验 1：机器启动

本实验作为 ChCore 操作系统课程实验的第一个实验，分为两个部分：第一部分介绍实验所需的基础知识，第二部分熟悉 ChCore 内核的启动过程。

实验中的“思考题”，请在实验报告中用文字或示意图简述，“练习题”则需在 ChCore 代码中填空，并在实验报告阐述实现过程，“挑战题”为难度稍高的练习题，此后的实验也类似，不再重复说明。

第一部分：基础知识

构建系统

在 ChCore 根目录运行下面命令可以构建并运行 ChCore：

```
1 $ make build # 构建 ChCore
2 $ make qemu # 使用 QEMU 运行 ChCore（初始时运行不会有任何输出），按 Ctrl-A 再按 x 退出
```

ChCore 采用 CMake 编写的构建系统管理其构建过程，并通过 Shell 脚本 `./chbuild` 对 CMake 的配置（configure）、构建（build）和清理（clean）的操作进行管理，另外，为了同学们更方便地进行实验，在 ChCore 实验中又添加了 Makefile 进一步对 `./chbuild` 脚本进行封装。

具体地，在根目录的 `CMakeLists.txt` 中，通过 `chcore_add_subproject` 命令（实际上就是 CMake 内置的 `ExternalProject_Add`）添加了 kernel 子项目，并传入根级 `CMakeLists.txt` 在 configure 步骤后获得的配置（CMake cache 变量）；在 kernel 子项目中，通过各级子目录共同构建了 `kernel.img` 文件，并放在 ChCore 根目录的 `build` 目录下。

关于 CMake 的更多信息请参考 [IPADS 新人培训：CMake](#)。

AArch64 汇编

AArch64 是 ARMv8 ISA 的 64 位执行状态。在 ChCore 实验中需要理解 AArch64 架构的一些特性，并能看懂和填写 AArch64 汇编代码，因此请先参考 [Arm Instruction Set Reference Guide](#) 的 [Overview of the Arm Architecture](#) 和 [Overview of AArch64 state](#) 章节以对 AArch64 架构有基本的认识，[A64 Instruction Set Reference](#) 章节则是完整的指令参考手册。

除此之外，可以阅读 [A Guide to ARM64 / AArch64 Assembly on Linux with Shellcodes and Cryptography](#) 的第 1、2 部分，以快速熟悉 AArch64 汇编。

QEMU 和 GDB

在本实验中常常需要对 ChCore 内核和用户态代码进行调试，因此需要开启 QEMU 的 GDB server，并使用 GDB（在 x86-64 平台的 Ubuntu 系统上应使用 `gdb-multiarch` 命令）连接远程目标来进行调试。

ChCore 根目录提供了 `.gdbinit` 文件来对 GDB 进行初始化，以方便使用。

要使用 GDB 调试 ChCore，需打开两个终端页面，并在 ChCore 根目录分别依次运行：

```
1 # 终端 1
2 $ make qemu-gdb # 需要先确保已运行 make build
3
4 # 终端 2
5 $ make gdb
```

不出意外的话，终端 1 将会“卡住”没有任何输出，终端 2 将会进入 GDB 调试界面，并显示从 `0x80000` 内存地址开始的一系列指令。此时在 GDB 窗口输入命令可以控制 QEMU 中 ChCore 内核的运行，例如：

- `ni` 可以执行到下一条指令
- `si` 可以执行到下一条指令，且会跟随 `bl` 进入函数
- `break [func]/b [func]` 可以在函数 `[func]` 开头打断点
- `break *[addr]` 可以在内存地址 `[addr]` 处打断点
- `info reg [reg]/i r [reg]` 可以打印 `[reg]` 寄存器的值
- `continue/c` 可以继续 ChCore 的执行，直到出发断点或手动按 Ctrl-C

更多常用的 GDB 命令和用法请参考 [GDB Quick Reference](#) 和 [Debugging with GDB](#)。

第二部分：内核启动过程

树莓派启动过程

在树莓派 3B+ 真机上，通过 SD 卡启动时，上电后会运行 ROM 中的特定固件，接着加载并运行 SD 卡上的 `bootcode.bin` 和 `start.elf`，后者进而根据 `config.txt` 中的配置，加载指定的 kernel 映像文件（纯 binary 格式，通常名为 `kernel8.img`）到内存的 `0x80000` 位置并跳转到该地址开始执行。

而在 QEMU 模拟的 `raspi3b`（旧版 QEMU 为 `raspi3`）机器上，则可以通过 `-kernel` 参数直接指定 ELF 格式的 kernel 映像文件，进而直接启动到 ELF 头部中指定的入口地址，即 `_start` 函数（实际上也在 `0x80000`，因为 ChCore 通过 linker script 强制指定了该函数在 ELF 中的位置，如有兴趣请参考附录）。

启动 CPU 0 号核

`_start` 函数（位于 `kernel/arch/aarch64/boot/raspi3/init/start.s`）是 ChCore 内核启动时执行的第一块代码。由于 QEMU 在模拟机器启动时会同时开启 4 个 CPU 核心，于是 4 个核会同时开始执行 `_start` 函数。而在内核的初始化过程中，我们通常需要首先让其中一个核进入初始化流程，待进行了一些基本的初始化后，再让其他核继续执行。

思考题 1：阅读 `_start` 函数的开头，尝试说明 ChCore 是如何让其中一个核首先进入初始化流程，并让其他核暂停执行的。

提示：可以在 [Arm Architecture Reference Manual](#) 找到 `mpidr_el1` 等系统寄存器的详细信息。

切换异常级别

AArch64 架构中，特权级被称为异常级别（Exception Level, EL），四个异常级别分别为 EL0、EL1、EL2、EL3，其中 EL3 为最高异常级别，常用于安全监控器（Secure Monitor），EL2 其次，常用于虚拟机监控器（Hypervisor），EL1 是内核常用的异常级别，也就是通常所说的内核态，EL0 是最低异常级别，也就是通常所说的用户态。

QEMU `raspi3b` 机器启动时，CPU 异常级别为 EL3，我们需要在启动代码中将异常级别降为 EL1，也就是进入内核态。具体地，这件事是在 `arm64_elX_to_el1` 函数（位于 `kernel/arch/aarch64/boot/raspi3/init/tools.S`）中完成的。

为了使 `arm64_elX_to_el1` 函数具有通用性，我们没有直接写死从 EL3 降至 EL1 的逻辑，而是首先判断当前所在的异常级别，并根据当前异常级别的不同，跳转到相应的代码执行。

练习题 2：在 `arm64_elX_to_el1` 函数的 LAB 1 TODO 1 处填写一行汇编代码，获取 CPU 当前异常级别。

提示：通过 `CurrentEL` 系统寄存器可获得当前异常级别。通过 GDB 在指令级别单步调试可验证实现是否正确。

无论从哪个异常级别跳到更低异常级别，基本的逻辑都是：

- 先设置当前级别的控制寄存器（EL3 的 `scr_el3`、EL2 的 `hcr_el2`、EL1 的 `sctlr_el1`），以控制低一级别的执行状态等行为
- 然后设置 `elr_elx`（异常链接寄存器）和 `spsr_elx`（保存的程序状态寄存器），分别控制异常返回后执行的指令地址，和返回后应恢复的程序状态（包括异常返回后的异常级别）
- 最后调用 `eret` 指令，进行异常返回

练习题 3：在 `arm64_elX_to_el1` 函数的 LAB 1 TODO 2 处填写大约 4 行汇编代码，设置从 EL3 跳转到 EL1 所需的 `elr_el3` 和 `spsr_el3` 寄存器值。具体地，我们需要在跳转到 EL1 时暂时屏蔽所有中断、并使用内核栈（`sp_el1` 寄存器指定的栈指针）。

练习完成后，可使用 GDB 跟踪内核代码的执行过程，由于此时不会有任何输出，可通过是否正确从 `arm64_elX_to_el1` 函数返回到 `_start` 来判断代码的正确性。

跳转到第一行 C 代码

降低异常级别到 EL1 后，应尽快从汇编跳转到 C 代码，以便提高代码的可复用性和可读性。因此在 `_start` 函数从 `arm64_elX_to_el1` 返回后，立即设置启动所需的栈，并跳转到第一个 C 函数 `init_c`。

思考题 4：结合此前 ICS 课的知识，并参考 `kernel.img` 的反汇编（通过 `aarch64-linux-gnu-objdump -S` 可获得），说明为什么要在进入 C 函数之前设置启动栈。如果不设置，会发生什么？

进入 `init_c` 函数后，第一件事首先通过 `clear_bss` 函数清零了 `.bss` 段，该段用于存储未初始化的全局变量和静态变量（具体请参考附录）。

思考题 5：在实验 1 中，其实不调用 `clear_bss` 也不影响内核的执行，请思考不清理 `.bss` 段在之后的何种情况下会导致内核无法工作。

初始化串口输出

到目前为止我们仍然只能通过 GDB 追踪内核的执行过程，而无法看到任何输出，这无疑是对我们写操作系统的积极性的一种打击。因此在 `init_c` 中，我们应该尽快启用某个可以输出字符的东西，而这个“东西”在树莓派上叫做 UART 串口。

在 `kernel/arch/aarch64/boot/raspi3/peripherals/uart.c` 已经给出了 `early_uart_init` 和 `early_uart_send` 函数，分别用于初始化 UART 和发送单个字符（也就是输出字符）。

练习题 6: 在 `kernel/arch/aarch64/boot/raspi3/peripherals/uart.c` 中 LAB 1 TODO 3 处实现通过 UART 输出字符串的逻辑。

恭喜！我们终于在内核中输出了第一个字符串！

启用 MMU

在内核的启动阶段，还需要配置启动页表（`init_boot_pt` 函数），并启用 MMU（`el1_mmu_activate` 函数），使可以通过虚拟地址访问内存，从而为之后跳转到高地址作准备（内核通常运行在虚拟地址空间 `0xffffffff00000000` 之后的高地址）。

关于配置启动页表的内容由于包含关于页表的细节，将在下一个实验和用户进程页表等一同实现，本次实验将直接启用 MMU。

在 EL1 异常级别启用 MMU 是通过配置系统寄存器 `sctlr_el1` 实现的（Arm Architecture Reference Manual D13.2.118）。具体需要配置的字段主要包括：

- 是否启用 MMU（`M` 字段）
- 是否启用对齐检查（`A` `SA0` `SA` `nAA` 字段）
- 是否启用指令和数据缓存（`C` `I` 字段）

练习题 7: 在 `kernel/arch/aarch64/boot/raspi3/init/tools.S` 中 LAB 1 TODO 4 处填写一行汇编代码，以启用 MMU。

由于没有配置启动页表，在启用 MMU 后，内核会立即发生地址翻译错误（Translation Fault），进而尝试跳转到异常处理函数（Exception Handler），而此时我们也没有设置异常向量表（`vbar_el1` 寄存器），因此执行流会来到 `0x200` 地址，并无限重复跳转。在 QEMU 中 `continue` 执行后，待内核输出停止后，按 Ctrl-C，可以观察到在 `0x200` 处无限循环。

附录

ELF 文件格式

如第一部分所看到的，ChCore 的构建系统将会构建出 `build/kernel.img` 文件，该文件是一个 ELF 格式的“可执行目标文件”，和我们平常在 Linux 系统中见到的可执行文件如出一辙。ELF 可执行文件以 ELF 头部（ELF header）开始，后跟几个程序段（program segment），每个程序段都是一个连续的二进制块，其中又包含不同的分段（section），加载器（loader）将它们加载到指定地址的内存中并赋予指定的可读（R）、可写（W）、可执行（E）权限，并从入口地址（entry point）开始执行。

可以通过 `aarch64-linux-gnu-readelf` 命令查看 `build/kernel.img` 文件的 ELF 元信息（比如通过 `-h` 参数查看 ELF 头部、`-l` 参数查看程序头部、`-S` 参数查看分段头部等）：

```
1 $ aarch64-linux-gnu-readelf -h build/kernel.img
2 ELF Header:
3   Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
4   Class:                               ELF64
5   Data:                                   2's complement, little endian
6   Version:                               1 (current)
7   OS/ABI:                                UNIX - System V
8   ABI Version:                           0
```

9	Type:	EXEC (Executable file)
10	Machine:	AArch64
11	Version:	0x1
12	Entry point address:	0x80000
13	Start of program headers:	64 (bytes into file)
14	Start of section headers:	271736 (bytes into file)
15	Flags:	0x0
16	Size of this header:	64 (bytes)
17	Size of program headers:	56 (bytes)
18	Number of program headers:	4
19	Size of section headers:	64 (bytes)
20	Number of section headers:	15
21	Section header string table index:	14

更多关于 ELF 格式的细节请参考 [ELF - OSDev Wiki](#)。

Linker Script

在构建的最后一步，即链接产生 `build/kernel.img` 时，ChCore 构建系统中指定了使用从 `kernel/arch/aarch64/boot/linker.tpl.ld` 模板产生的 linker script 来精细控制 ELF 加载后程序各分段在内存中布局。

具体地，将 `${init_objects}`（即 `kernel/arch/aarch64/boot/raspi3` 中的代码编成的目标文件）放在了 ELF 内存的 `TEXT_OFFSET`（即 `0x80000`）位置，`.text`（代码段）、`.data`（数据段）、`.rodata`（只读数据段）和 `.bss`（BSS 段）依次紧随其后。

这里对这些分段所存放的内容做一些解释：

- `init`：内核启动阶段代码和数据，因为此时还没有开启 MMU，内核运行在低地址，所以需要特殊处理
- `.text`：内核代码，由一条条的机器指令组成
- `.data`：已初始化的全局变量和静态变量
- `.rodata`：只读数据，包括字符串字面量等
- `.bss`：未初始化的全局变量和静态变量，由于没有初始值，因此在 ELF 中不需要真的为该分段分配空间，而是只需要记录目标内存地址和大小，在加载时需要初始化为 0

除了指定各分段的顺序和对齐，linker script 中还指定了它们运行时“认为自己所在的内存地址”和加载时“实际存放在的内存地址”。例如前面已经说到 `init` 段被放在了 `TEXT_OFFSET` 即 `0x80000` 处，由于启动时内核运行在低地址，此时它“认为自己所在的地址”也应该是 `0x80000`，而后面的 `.text` 等段则被放在紧接着 `init` 段之后，但它们在运行时“认为自己在” `KERNEL_VADDR + init_end` 也就是高地址。

更多关于 linker script 的细节请参考 [Linker Scripts](#)。