

Lab0.5

练习1：使用GDB验证启动流程

实验过程

- 首先进入到riscv64-ucore-labcodes下的lab0，同时打开两个shell后分别使用命令如下，结合gdb和qemu源码级调试ucore：

```
1 | make debug
2 | make gdb
```

从下图可以看到RISC-V计算机加点开始启动首先停在了0x0000000000001000处(即程序计数器PC首先被初始化为0x1000)，这是因为实际上RISC-V计算机上电时，第一件做的事其实是复位。复位通常是指将计算机系统的各个组件（包括处理器、内存、设备等）置于初始状态，并且会启动Bootloader。而QEMU模拟的这款riscv处理器的复位地址是0x1000，而不是0x80000000。所以首先停在了这个位置上。

```
hyx@hyx-virtual-machine:/mnt/hgfs/ShareUbuntuOS/riscv64-ucore-labcodes/lab0$ make gdb
riscv64-unknown-elf-gdb \
  -ex 'file bin/kernel' \
  -ex 'set arch riscv:rv64' \
  -ex 'target remote localhost:1234'
GNU gdb (GDB) 15.1
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "--host=x86_64-pc-linux-gnu --target=riscv64-unknown-elf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
  <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word".
Reading symbols from bin/kernel...
The target architecture is set to "riscv:rv64".
Remote debugging using localhost:1234
0x0000000000001000 in ?? ()
(gdb)
```

- 在gdb中使用命令如下查看0x1000处的10条汇编指令，

```
1 | x/10i $pc
```

结果如下图所示：

```
0x0000000000001000 in ?? ()
(gdb) x/10i $pc
=> 0x1000:    auipc   t0,0x0
    0x1004:    addi    a1,t0,32
    0x1008:    csrr    a0,mhartid
    0x100c:    ld      t0,24(t0)
    0x1010:    jr      t0
    0x1014:    unimp
    0x1016:    unimp
    0x1018:    unimp
    0x101a:    .insn   2, 0x8000
    0x101c:    unimp
(gdb)
```

图片中的指令是RISC-V硬件加电后的几条指令的一部分，由固化在Qemu内的一小段汇编程序负责，首先位于物理地址0x1000处。用于在RISC-V架构中进行程序控制和加载还有Bootloader的启动。下面是对几条不太熟悉指令的解释(后面指令解释同理)：

auipc: 当前PC(程序计数器)加上立即数，并将结果存储到目标寄存器中。

csrr: 从控制状态寄存器读取值的指令

UNIMP: 未实现的指令，通常用于在指令流中作为未定义行为或占位符

.insn: 汇编中的伪指令，用于手动插入自定义的机器指令

- 使用si命令逐行调试执行到jr t0指令处即可跳转到物理地址0x80000000对应的指令处，然后即启动Bootloader

```
1 | si
```

想要查看0x80000000处的指令可以直接使用命令x/10i 0x80000000，也可以当执行到0x80000000处后，再使用x/10i \$pc进行查看，此处我使用的后面一种方法，结果图如下：

```
(gdb) si
0x0000000080000000 in ?? ()
(gdb) x/10i $pc
=> 0x80000000: csrr    a6,mhartid
    0x80000004: bgtz   a6,0x80000108
    0x80000008: auipc   t0,0x0
    0x8000000c: addi    t0,t0,1032
    0x80000010: auipc   t1,0x0
    0x80000014: addi    t1,t1,-16
    0x80000018: sd      t1,0(t0)
    0x8000001c: auipc   t0,0x0
    0x80000020: addi    t0,t0,1020
    0x80000024: ld      t0,0(t0)
(gdb)
```

这些指令的含义与具体实现有关，特别是在特定的硬件和系统架构上即RISC-V架构上。这些也是RISC-V硬件加电后的几条指令的一部分，由RISC-V架构中的Bootloader即OpenSBI获取计算机控制权后负责实现，位于物理地址0x80000000处。

bgtz: 一个条件分支指令，表示如果a6中的值大于零，就跳转到地址0x80000108

- 为了正确地和上一阶段的OpenSBI对接，我们需要保证内核的第一条指令位于物理地址0x80200000处，因为这里的代码是地址相关的，这个地址是由处理器，即Qemu指定的。为此，我们需要将内核镜像预先加载到Qemu物理内存以地址0x80200000开头的区域上。

通过命令break *0x80200000在内核镜像位置0x80200000打上断点，并使用命令continue运行到该位置。之后再使用命令x/10i \$pc查看此处的10条指令。

```
1 | break *0x80200000
2 | continue
3 | x/10i $pc
```

运行结果如下图所示：

```
(gdb) break *0x80200000
Breakpoint 1 at 0x80200000: file kern/init/entry.S, line 7.
(gdb) continue
Continuing.

Breakpoint 1, kern_entry () at kern/init/entry.S:7
7      la sp, bootstacktop
(gdb) x/10i $pc
=> 0x80200000 <kern_entry>:    auipc   sp,0x3
  0x80200004 <kern_entry+4>:   mv      sp,sp
  0x80200008 <kern_entry+8>:   j       0x8020000a <kern_init>
  0x8020000a <kern_init>:     auipc   a0,0x3
  0x8020000e <kern_init+4>:   addi    a0,a0,-2
  0x80200012 <kern_init+8>:   auipc   a2,0x3
  0x80200016 <kern_init+12>:  addi    a2,a2,-10
  0x8020001a <kern_init+16>:  addi    sp,sp,-16
  0x8020001c <kern_init+18>:  li      a1,0
  0x8020001e <kern_init+20>:  sub     a2,a2,a0
(gdb)
```

li:将立即数加载到寄存器

实验中的知识点

BootLoader的启动

BootLoader的启动分为两个阶段：

(1) 第一个阶段通常用汇编语言来实现，包含依赖于CPU的体系架构的硬件的初始化代码。该阶段的任务：

- 1.对硬件设备进行初始化（如屏蔽所有中断，关闭处理器内部指令/数据Cache等）；
- 2.为第二阶段准备RAM空间；
- 3.复制BootLoader的第二阶段代码到RAM中；
- 4.设置堆栈为第二阶段的C语言环境做准备。

第一阶段关闭Cache的原因：通常使用Cache是为了提高系统性能，但此时Cache的使用可能改变访问主存的数量，类型或时间，BootLoader是不需要的，即BootLoader直接访问主存即可，不需要通过缓存来访问。

(2) 第一阶段执行完会跳转到第二阶段的C程序入口点，该阶段是由C语言完成，以便实现更复杂的功能，也是程序有更好的可读性和可移植性。该阶段的任务：

- 1.初始化本阶段要用到的硬件设备；
- 2.检测系统的内存映射；
- 3.将操作系统程序从Flash读到RAM；

- 4.为操作系统设置启动参数；
- 5.调用操作系统代码进行启动

固体

固件(firmware)是一种特定的计算机软件，它为设备的特定硬件提供低级控制，也可以进一步加载其他软件。固件可以为设备更复杂的软件（如操作系统）提供标准化的操作环境。对于不太复杂的设备，固件可以直接充当设备的完整操作系统，执行所有控制、监视和数据操作功能。在基于 x86 的计算机系统中，BIOS 或 UEFI 是固件；在基于 riscv 的计算机系统中，OpenSBI 是固件。OpenSBI运行在M态（M-mode），因为固件需要直接访问硬件。

地址无关代码 (PIC):

使得码可以在任何内存地址执行而无需修改且不依赖于程序在内存中的特定位置

优点：

- 灵活性：代码可以在任意内存地址执行，且无需重新编译。加载时不必进行地址重定位。
- 节省内存：多个进程可以共享相同的代码段，减少了内存占用，因为每个进程都可以将该代码映射到不同的地址空间。

缺点：

- 性能开销：PIC 在某些情况下可能会导致较小的性能损失，特别是在函数调用和全局变量访问时。

地址相关代码 (PDC):

地址相关代码依赖于它在内存中的固定位置，即在编译时，代码生成时已经假定了它将在某个固定的地址上执行。该代码如果加载到不同的地址，必须进行重定位或重新编译。

优点：

- 简单高效：由于直接使用绝对地址，代码在执行时不需要计算相对地址，因此访问内存和函数时的性能较好。

缺点：

- 缺乏灵活性：代码只能在特定的地址运行，如果要加载到不同地址，必须进行重定位。