

Lab0.5

实验目的：

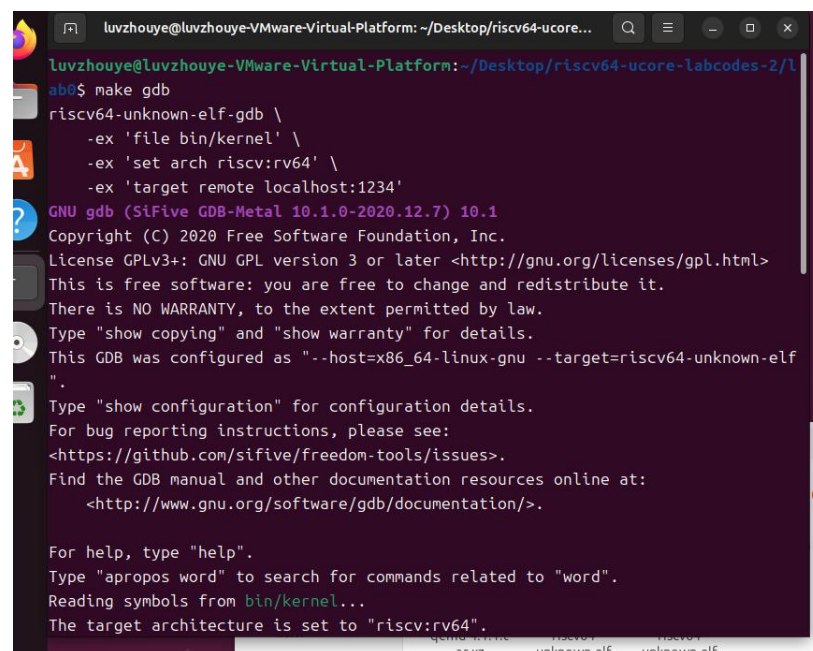
实验 0.5 主要讲解最小可执行内核和启动流程。我们的内核主要在 Qemu 模拟器上运行，它可以模拟一台 64 位 RISC-V 计算机。为了让我们的内核能够正确对接到 Qemu 模拟器上，需要了解 Qemu 模拟器的启动流程，还需要一些程序内存布局和编译流程（特别是链接）相关知识。

实验内容：

实验 0.5 主要讲解最小可执行内核和启动流程。我们的内核主要在 Qemu 模拟器上运行，它可以模拟一台 64 位 RISC-V 计算机。为了让我们的内核能够正确对接到 Qemu 模拟器上，需要了解 Qemu 模拟器的启动流程，还需要一些程序内存布局和编译流程（特别是链接）相关知识，以及通过 opensbi 固件来通过服务。

练习 1：使用 GDB 验证启动流程

为了熟悉使用 qemu 和 gdb 进行调试工作，使用 gdb 调试 QEMU 模拟的 RISC-V 计算机加电开始运行到执行应用程序的第一条指令（即跳转到 0x80200000）这个阶段的执行过程，说明 RISC-V 硬件加电后的几条指令在哪里？完成了哪些功能？要求在报告中简要写出练习过程和回答。



```
luzhouye@luzhouye-VMware-Virtual-Platform: ~/Desktop/riscv64-ucore-labcodes-2/l
ab0$ make gdb
riscv64-unknown-elf-gdb \
-ex 'file bin/kernel' \
-ex 'set arch riscv:rv64' \
-ex 'target remote localhost:1234'
GNU gdb (SiFive GDB-Metal 10.1.0-2020.12.7) 10.1
Copyright (C) 2020 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-linux-gnu --target=riscv64-unknown-elf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://github.com/sifive/freedom-tools/issues>.
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".
```

一些可能用到的 gdb 指令：

x/10i 0x80000000 : 显示 0x80000000 处的 10 条汇编指令。

```

(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:      0x8000
    0x101c:      unimp

```

x/10i \$pc : 显示即将执行的 10 条汇编指令。

这段代码是使用 RISC-V 汇编语言编写的，下面是逐行解释：

0x1000: auipc t0,0x0 将当前 PC 值的高 20 位复制到 t0。

0x1004: addi a1,t0,32 指令将寄存器 t0 的值与立即数 32 相加，结果存入寄存器 a1=0x1020。

0x1008: csrr a0,mhartid csrr 指令用于从控制和状态寄存器（CSR）读取值到寄存器 a0。mhartid 是 CSR 之一，用于获取当前硬件线程的 ID。

0x100c: ld t0,24(t0) 从内存地址 t0+24 处加载一个 64 位的值到寄存器 t0。

0x1010: jr t0 根据寄存器 t0 中的值跳转到对应的地址执行。这是一个间接跳转。

0x1014: unimp 通常用于指示未实现的功能。如果执行了这个指令，通常会触发异常。

0x1016: unimp 同上。

0x1018: unimp 同上。

0x101a: 0x8000 这是一个字节，通常在汇编代码中表示数据。

0x101c: unimp

```

S7          0x0      0
--Type <RET> for more, q to quit, c to continue without paging--q
Quit
(gdb) info r $pc
pc          0x1000      0x1000
(gdb) si
0x8000000000001004 in ?? ()
(gdb) info r $pc
pc          0x1004      0x1004
(gdb) info r to
Invalid register `to'
(gdb) info r t0
t0          0x1000      4096
(gdb) si
0x8000000000001008 in ?? ()
(gdb) info r t0
t0          0x1000      4096
(gdb) info r a0
a0          0x0         0
(gdb) info r a1
a1          0x1020      4128
(gdb) si
0x800000000000100c in ?? ()
(gdb) info r a0
a0          0x0         0
(gdb) info r $pc
pc          0x100c      0x100c
(gdb) si
0x8000000000001010 in ?? ()
(gdb) info r t0
t0          0x80000000      2147483648
(gdb) si
0x8000000000000000 in ?? ()
(gdb) info r $pc
pc          0x80000000      0x80000000

```

单步调试寄存器信息。

为了分析从加电到执行应用程序的第一条指令，在 0x80200000 加断点，调试。

```

pc          0x80000000      0x80000000
(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

```

la（Load Address）指令将标签 bootstacktop 的地址加载到寄存器 sp 中。sp 是栈指针寄存器，用于指向当前栈的顶部。内核镜像 os.bin 被加载到 0x80200000 开头的区域。

实验 1 主要讲解的是中断处理机制。通过本章的学习，我们了解了 riscv 的中断处理机制、相关寄存器与指令。我们知道在中断前后需要恢复上下文环境，用一个名为中断帧（TrapFrame）的结构体存储了要保存的各寄存器，并用了很大篇幅解释如何通过精巧的汇编代码实现上下文环境保存与恢复机制。最终，我们通过处理断点和时钟中断验证了我们正确实现了中断机制。

练习 1：理解内核启动中的程序入口操作

阅读 `kern/init/entry.S` 内容代码，结合操作系统内核启动流程，说明指令 `la sp, bootstacktop` 完成了什么操作，目的是什么？`tail kern_init` 完成了什么操作，目的是什么？

- 1) 初始化栈指针来进行函数调用等操作。
- 2) `tail` 用于实现函数的尾部调用优化。尾部调用是指在函数的末尾调用另一个函数，并且不再返回到当前函数。这种调用方式可以避免额外的栈帧分配，从而节省栈空间。

`kern_init` 指向内核初始化函数的入口点。这条指令的目的是跳转到 `kern_init` 函数执行内核初始化。由于 `tail` 指令的特性，当前函数的返回地址会被替换为 `kern_init` 的返回地址，这样在 `kern_init` 执行完毕后，可以直接返回到调用的地方，而不需要额外的栈帧。

为内核的启动做了准备。

练习 2：完善中断处理（需要编程）

请编程完善 `trap.c` 中的中断处理函数 `trap`，在对时钟中断进行处理的部分填写 `kern/trap/trap.c` 函数中处理时钟中断的部分，使操作系统每遇到 100 次时钟中断后，调用 `print_ticks` 子程序，向屏幕上打印一行文字“100 ticks”，在打印完 10 行后调用 `sbi.h` 中的 `shut_down()` 函数关机。


```

107         // directly.
108         // cprintf("Supervisor timer interrupt\n");
109         /* LAB1 EXERCISE2  YOUR CODE : */
110         /*(1)设置下次时钟中断-*/
111         clock_set_next_event();
112         /* *(2)计数器 (ticks) 加一
113         *(3)当计数器加到100的时候，我们会输出一个`100ticks`表示我们触发了100次时钟中断，同时打印次数 (num) 加一
114         * (4)判断打印次数，当打印次数为10时，调用<sbi.h>中的关机函数关机
115         */
116         ticks++;
117         if(ticks%TICK_NUM==0){
118             print_ticks();
119             num++;
120             if(num==10)
121                 sbi_shutdown();
122         }
123         break;

```

```

Platform Name      : QEMU Virt Machine
Platform HART Features : RV64ACDFIMSU
Platform Max HARTs  : 8
Current Hart       : 0
Firmware Base      : 0x80000000
Firmware Size      : 112 KB
Runtime SBI Version : 0.1

PMP0: 0x0000000000000000-0x0000000000001ffff (A)
PMP1: 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
(THU.CST) os is loading ...

Special kernel symbols:
entry 0x000000000020000a (virtual)
etext 0x00000000002009ee (virtual)
edata 0x0000000000204010 (virtual)
end   0x0000000000204028 (virtual)
Kernel executable memory footprint: 17KB
++ setup timer interrupts
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
lvzzhouye@lvzzhouye-VMware-Virtual-Platform: ~/Desktop/riscv64-ucore-labcodes-2/lab1
abi$

```

```

lvzzhouye@lvzzhouye-VMware-Virtual-Platform: ~/Desktop/riscv64-ucore-labcodes-2/lab1
abi$ make grade
gmake[1]: 进入目录"/home/lvzzhouye/Desktop/riscv64-ucore-labcodes-2/lab1" + cc kern/init/entry.S + cc kern/init/init.c + cc kern/libs/stdio.c + cc kern/debug/kdebug.c + cc kern/debug/kmonitor.c + cc kern/debug/panic.c + cc kern/driver/clock.c + cc kern/driver/console.c + cc kern/driver/intr.c + cc kern/trap/trap.c + cc kern/trap/trapentry.S + cc kern/mm/pmm.c + cc libs/printfmt.c + cc libs/readline.c + cc libs/sbi.c + cc libs/string.c + ld bin/kernel riscv64-unknown-elf-objcopy bin/kernel --strip-all -O binary bin/ucore.img gmake[1]: 离开目录"/home/lvzzhouye/Desktop/riscv64-ucore-labcodes-2/lab1"
try to run qemu
qemu pid=4002
-100 ticks: OK
Total Score: 100/100

```

扩展练习 Challenge1: 描述与理解中断流程

回答：描述 ucore 中处理中断异常的流程（从异常的产生开始），其中 `mov a0, sp` 的目的是什么？`SAVE_ALL` 中寄存器保存在栈中的位置是什么确定的？对于任何中断，`__alltraps` 中都需要保存所有寄存器吗？请说明理由。

在 uCore OS 中处理中断和异常的流程大致如下：

1) 当异常发生时，硬件会自动触发一个中断信号，硬件会响应这个信号，暂停当前程序的执行，保存当前的状态（如程序计数器、寄存器状态等）。根据中断向量查找中断描述符表（IDT），跳转到对应的中断处理函数入口。在 uCore 中，所有中断和异常的处理入口都是 `_alltraps`。在这个入口点，首先通过 `SAVE_ALL` 宏将所有寄存器的值保存到栈中。这一步是为了保证在中断处理过程中能够保存和恢复中断发生时的上下文环境。通过 `mov a0, sp` 将栈顶地址（即保存所有寄存器值的地址）传递给中断处理函数 `trap()`。这样做的目的是为了让处理函数能够访问到中断发生时的上下文信息。执行具体的中断处理逻辑。处理完成后，通过 `_trapret` 标签中的 `RESTORE_ALL` 宏来恢复之前保存的寄存器值，继续执行被中断的指令。

2) `mov a0, sp` 是为了将当前的栈指针（即包含所有寄存器状态的栈帧的顶部）的地址传递给中断处理函数，以便在处理函数中能够访问和恢复中断前的上下文环境。

`SAVE_ALL` 中寄存器保存在栈中的位置是由汇编宏 `SAVE_ALL` 和 `RESTORE_ALL` 确定的，它们定义了保存和恢复寄存器的顺序和位置。

3) 对于任何中断，`__alltraps` 中需要保存所有寄存器，因为在中断处理期间可能会修改这些寄存器的值。为了确保中断处理完成后能够正确地恢复被中断的程序的执行，必须保存和恢复这些寄存器的值。此外，如果中断处理需要调用其他函数，也需要保证这些函数的调用约定不会影响到中断返回后的状态。

扩增练习 Challenge2: 理解上下文切换机制

回答：在 `trapentry.S` 中汇编代码 `csw sscratch, sp; csrrw s0, sscratch, x0` 实现了什么操作，目的是什么？`save all` 里面保存了 `stval` `scause` 这些 `csr`，而在 `restore all` 里面却不还原它们？那这样 `store` 的意义何在呢？

1) `csw sscratch, sp`：这条指令将当前栈指针的值写入到 `sscratch` 寄存器中。`sscratch` 是一个系统寄存器，在 RISC-V 中用于保存临时数据，这里用于保存中

断发生前的栈指针，以便后续恢复。这个操作的目的是为了在中断发生时，保存当前的栈指针，因为在中断处理程序中可能会切换栈，所以需要记住原来栈的位置。

`csrrw s0, sscratch, x0`: 这条指令执行了一个读取-修改-写入操作。首先将 `sscratch` 寄存器的值读取到通用寄存器 `s0` 中，然后将 `x0` 写入 `sscratch` 寄存器。这用于在中断发生时清除 `sscratch` 寄存器，确保它不会影响后续操作。

2) `SAVE_ALL` 中保存了 `stval` 和 `scause` 这些 CSR，而在 `RESTORE_ALL` 中不恢复的原因是：

`stval`（它会记录一些中断处理所需要的辅助信息，比如指令获取(instruction fetch)、访存、缺页异常，它会把发生问题的目标地址或者出错的指令记录下来，这样我们在中断处理程序中就知道处理目标了）会从 `sepc`（它会记录触发中断的那条指令的地址）寄存器指定的地址继续执行。

`scause`（它会记录中断发生的原因，还会记录该中断是不是一个外部中断）因为处理器需要知道异常已经处理完毕，并且可以安全地恢复程序的执行。

保存这些 CSR 的意义在于能够提供给中断处理程序足够的信息来正确地处理中断，并在处理完成后恢复处理器的状态。即使某些寄存器的值在 `RESTORE_ALL` 中不恢复，它们在中断处理的调试和错误报告仍然有用。

扩展练习 Challenge3: 完善异常中断

编程完善在触发一条非法指令异常 `mret` 和，在 `kern/trap/trap.c` 的异常处理函数中捕获，并对其进行处理，简单输出异常类型和异常指令触发地址，即“`Illegal instruction caught at 0x(地址)`”，“`ebreak caught at 0x(地址)`”与“`Exception type:Illegal instruction`”，“`Exception type: breakpoint`”。

```
|  
    //初始化结束后增加两种中断，测试代码  
    __asm__ __volatile__("mret");  
    __asm__ __volatile__("ebreak");
```



```

break;
case CAUSE_ILLEGAL_INSTRUCTION:
    // 非法指令异常处理
    /* LAB1 CHALLENGE3 YOUR CODE : */
    /*(1)输出指令异常类型 ( Illegal instruction)
    ;(2)输出异常指令地址
    ;(3)更新 tf->epc寄存器
    */
    cprintf("Exception type: Illegal instruction\n");
    cprintf("Illegal instruction caught at 0x%08x\n", tf->epc);
    tf->epc += 4;
    break;
case CAUSE_BREAKPOINT:
    //断点异常处理
    /* LAB1 CHALLENGE3 YOUR CODE : */
    /*(1)输出指令异常类型 ( breakpoint)
    ;(2)输出异常指令地址
    ;(3)更新 tf->epc寄存器
    */
    cprintf("Exception type: breakpoint\n");
    cprintf("ebreak caught at 0x%08x\n", tf->epc);
    tf->epc += 2;
    break;

```

注：断点异常的时候+2 个字节是因为 breakpoint 是个短指令。

输出：

```

pork@Pork: ~/labcodes/lab1
edata 0x0000000080204010 (virtual)
end 0x0000000080204028 (virtual)
Kernel executable memory footprint: 17KB
++ setup timer interrupts
sbi_emulate_csr_read: hartid0: invalid csr_num=0x302
Exception type: Illegal instruction
Illegal instruction caught at 0x8020004e
Exception type: breakpoint
ebreak caught at 0x80200052
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
pork@Pork:~/labcodes/lab1$ make grade
make[1]: Entering directory '/home/pork/labcodes/lab1' + cc kern/init/entry.S + cc kern/init/init.c + cc kern/libs/stdio
.c + cc kern/debug/panic.c + cc kern/debug/kdebug.c + cc kern/debug/kmonitor.c + cc kern/driver/clock.c + cc kern/driver
/console.c + cc kern/driver/intr.c + cc kern/trap/trap.c + cc kern/trap/trapentry.S + cc kern/mm/pmm.c + cc libs/string.
c + cc libs/printfmt.c + cc libs/readline.c + cc libs/sbi.c + ld bin/kernel riscv64-unknown-elf-objcopy bin/kernel --str
ip-all -O binary bin/ucore.img make[1]: Leaving directory '/home/pork/labcodes/lab1'
try to run gemu
gemu pid=825
-100 ticks: OK
Total Score: 100/100
pork@Pork:~/labcodes/lab1$

```