

浙江大学

本科实验报告

课程名称：操作系统

姓 名：徐文皓

学 院：计算机科学与技术学院

系：计算机科学与技术系

专 业：软件工程

学 号：3210102377

指导教师：夏莹杰

2023 年 10 月 15 日

浙江大学操作系统实验报告

实验名称：_____RV64 内核引导与时钟中断处理_____

电子邮件地址：_____手机：_____

实验地点：_____线上_____实验日期：2023 年 10 月 15 日

一、实验目的和要求

本实验的目的有：学习 RISC-V 汇编，编写 head.S 实现跳转到内核运行的第一个 C 函数；学习 OpenSBI，理解 OpenSBI 在实验中所起到的作用，并调用 OpenSBI 提供的接口完成字符的输出；学习 Makefile 相关知识，补充项目中的 Makefile 文件，来完成对整个工程的管理；学习 RISC-V 的 trap 处理相关寄存器与指令，完成对 trap 处理的初始化；理解 CPU 上下文切换机制，并正确实现上下文切换功能；编写 trap 处理函数，完成对特定 trap 的处理；调用 OpenSBI 提供的接口，完成对时钟中断事件的设置。

本实验的要求是：独立完成作业；在 lab0 的基础上，搭建实验代码框架，编写 head.S、完善 Makefile 脚本、补充 sbi.c、实现 puts()和 puti()、并补充修改 defs.h 文件完成内核引导；进一步修改 vmlinux.lds、head.S、test.c，开启 trap 处理：修改 head.S，并补全_start 中的逻辑；实现上下文切换：添加 arch/riscv/kernel/entry.S，并补全_traps 中的逻辑。实现 trap 处理函数：添加 trap.c 文件，实现 trap 处理函数 trap_handler()，实现时钟中断相关函数：添加 clock.c 文件，实现 get_cycles()和 clock_set_next_event()。

二、实验过程

首先考虑实现内核引导相关功能。

head.S 中的_start 是程序入口。在其中，我们通过.space 4096 为即将运行的第一个 C 函数设置大小为 4KB 的程序栈，并将该栈放置在.bss.stack 段。将栈顶地址记录在 sp 中，随后跳转至 main.c 中的 start_kernel 函数。

```

1 ...
2 _start:
3     la sp, boot_stack_top
4     call start_kernel
5 ...
6     .space 4096
7 ...

```

补充 Makefile，使工程得以编译。

```

1 C_SRC      = $(sort $(wildcard *.c))
2 OBJ        = $(patsubst %.c,%.o,$(C_SRC))
3
4 file = print.o
5 all:$(OBJ)
6
7 %.o:%.c
8     ${GCC} ${CFLAG} -c $<
9 clean:
10     $(shell rm *.o 2>/dev/null

```

完成 `sbi_ecall()`。

```

1 struct sbiret sbi_ecall(int ext, int fid, uint64 arg0, uint64 arg1, uint64 arg2,
2                        uint64 arg3, uint64 arg4, uint64 arg5){
3     struct sbiret res;
4     register uint64 a0 asm("a0") = arg0;
5     ...
6     register uint64 a5 asm("a5") = arg5;
7     register uint64 a6 asm("a6") = (uint64)fid;
8     register uint64 a7 asm("a7") = (uint64)ext;
9
10    __asm__ volatile(
11        "ecall\n"
12        : "=r"(a0), "=r"(a1)
13        : "r"(a0), "r"(a1), "r"(a2), "r"(a3),
14          "r"(a4), "r"(a5), "r"(a6), "r"(a7)
15        : "memory");
16    res.error = a0;
17    res.value = a1;
18    return res;
19 }

```

在完成了 `sbi_ecall()` 的基础上，参考 `ExtensionID` 和 `FunctionID` 的情况，对其进一步封装，得到 `sbi_console_putchar()` 和 `sbi_set_timer()`。

```

1 void sbi_console_putchar(char c){
2     sbi_ecall(0x1, 0x0, c, 0, 0, 0, 0, 0);
3 }
4
5 void sbi_set_timer(unsigned long time){
6     sbi_ecall(0x0, 0x0, time, 0, 0, 0, 0, 0);
7 }

```

完成 `sbi_console_putchar()` 后，即可完成 `print.c` 中的 `puts()` 和 `puti()`。

```

1 void puts(char *s){
2     for (int i = 0; s[i] != 0; i++)
3         sbi_console_putchar(s[i]);
4 }
5
6 void puti(int x){
7     if (x < 0){
8         x = -x;
9         sbi_console_putchar('-');
10    }
11    int p = 1;
12    while (p * 10 <= x)
13        p = p * 10;
14    do{
15        sbi_console_putchar(x / p + '0');
16        x = x % p;
17        p = p / 10;
18    } while (p > 0);
19 }

```

在 `defs.h` 中补充 `csr_read` 这一宏定义。

```

1 #define csr_read(csr) \
2 ({ \
3     register uint64 __v; \
4     asm volatile ("csrr %0, " #csr \
5                  : "=r" (__v) : \
6                  : "memory"); \
7     __v; \
8 })

```

至此，通过 `make run` 运行内核，可以看到启动后输出 2022 Hello RISC-V。

```

Boot HART PMP Granularity : 4
Boot HART PMP Address Bits: 54
Boot HART MHPM Count      : 0
Boot HART MHPM Count      : 0
Boot HART MIDELEG         : 0x00000000000000222
Boot HART MEDELEG         : 0x0000000000000b109
2022 Hello RISC-V

```

下面我们考虑实现时钟中断。

首先修改按照指导书要求，修改 `vmlinux.lds` 以及 `head.S`，将之后的 `_traps` 放在 `entry` 程序段。修改 `_start` 的内容，在其中完成了设置中断处理入口函数基地址、开启时钟中断、设置第一次时钟中断并开启了 S 态中断响应。

```
1 _start:
2     la sp, boot_stack_top
3
4     la t0, _traps
5     csrw stvec, t0 # set stvec = _traps
6
7     li t0, 0x00000020
8     csrs sie, t0 # set sie[SIE] = 1
9
10    # set first time interrupt
11    rdtm t0
12    li t1, 100000000
13    li a0, 0
14    li a1, 0
15    add a2, t0, t1
16    li a3, 0
17    li a4, 0
18    li a5, 0
19    li a6, 0
20    li a7, 0
21    call sbi_ecall
22    csrs sstatus, 2 # set sstatus[SIE] = 1
23
24    call start_kernel
```

在 `_traps` 中实现上下文切换机制。保存现场，将 `scause` 和 `sepc` 分别放入 `a0` 和 `a1` 中，并调用 `trap_handler()` 进行处理。这里我们发现，`a0, a1` 与 `void trap_handler(unsigned long scause, unsigned long sepc)` 的参数表一一对应。

```
1 _traps:
2     # 1. save 32 registers and sepc to stack
3     addi sp, sp, -8*33
4     sd x0, 0(sp)
5     sd x1, 8(sp)
6     ...
7     sd x31, 248(sp)
8     csrr a0, sepc
9     sd a0, 256(sp)
10
11    # 2. call trap_handler
12    csrr a0, scause
13    csrr a1, sepc
14    call trap_handler
15
16    # 3. restore sepc and 32 registers (x2(sp) should be restore last) from stack
17    ld a0, 256(sp)
18    csrwr sepc, a0
19    ld x31, 248(sp)
20    ...
21    ld x1, 8(sp)
22    ld x0, 0(sp)
23    ld x2, 16(sp)
24    addi sp, sp, 8*33
25
26    # 4. return from trap
27    sret
```

考虑 `trap_handler()`。观察 `scause` 的结构，使用适当掩码将最高位和其余位分离，分别用 `interrupt` 和 `cause` 两个变量接收。我们注意到当 `trap` 类型为 S 态时钟中断时，最高位为 1 且其余位值为 5。当满足这一条件时，我们输出 “[S] Super

visor Mode Timer Interrupt”。同时，为便于调试，在不满足这一条件时我们也设置了相应输出。

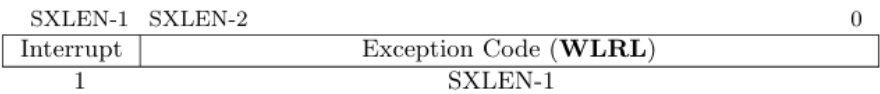


Figure 4.9: Supervisor Cause register `scause`.

Interrupt	Exception Code	Description
1	0	User software interrupt
1	1	Supervisor software interrupt
1	2–3	<i>Reserved for future standard use</i>
1	4	User timer interrupt
1	5	Supervisor timer interrupt
1	6–7	<i>Reserved for future standard use</i>
1	8	User external interrupt

```
1 void trap_handler(unsigned long scause, unsigned long sepc){
2     unsigned long interrupt = scause & 0x8000000000000000;
3     unsigned long cause = scause & 0x7fffffffffffffff;
4     if(interrupt&&cause==5){
5         puts("[S] Supervisor Mode Timer Interrupt\n");
6         clock_set_next_event();
7     }
8     else
9         puts("[NOT] Timer Interrupt\n");
10    return;
11 }
```

最后由此实现时钟中断，`clock_set_next_event()`用于设置下一个中断事件。

```
1 unsigned long get_cycles() {
2     register uint64 t0 asm("t0");
3     __asm__ volatile(
4         "rdtime t0"
5         : "=r"(t0)
6         :
7         : "memory");
8     return (unsigned long)t0;
9 }
10
11 void clock_set_next_event() {
12     unsigned long next = get_cycles() + TIMECLOCK;
13     sbi_set_timer(next);
14     return;
15 }
```

至此，启动内核，可以观察到每秒输出一行相应内容。

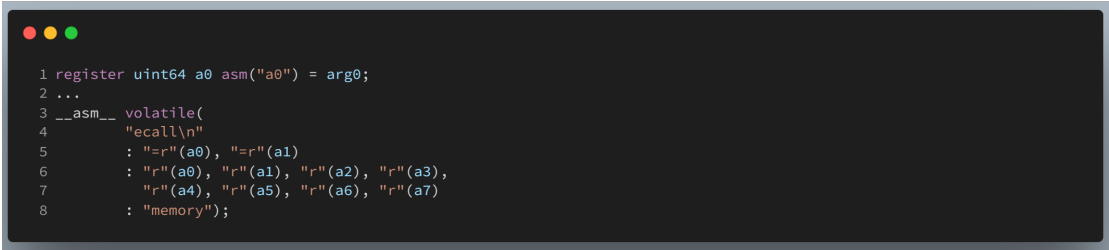
```
Boot HART MEDELEG : 0x0000000000000b109
2022 Hello RISC-V
[S] Supervisor Mode Timer Interrupt
[S] Supervisor Mode Timer Interrupt
[S] Supervisor Mode Timer Interrupt
```

三、讨论和心得

本次实验总体来说并不是很顺利，花费了大量的精力，尤其是对于之前学习的是 MIPS 的软件工程专业的我来说会更加困难一些。不过在这次实验的过程中，我更加熟练地了解了 GDB 调试，也能够看懂 Makefile 文件了，确实感觉收获很大。以下列出几个在实验过程中值得记录的具体的想法：

其一，需要注意 `la sp, book_stack_top` 这一设置栈顶位置的指令应当尽可能放在其他所有指令之前。在刚刚开始时钟中断部分时，我修改 `_start` 的内容后使得内核无法正常运行，表现在 GDB 进行汇编单步执行都无法正常完成。经检查，在 `_start` 中，在跳转至函数之前，虽然自己写的指令中没有对 `sp` 进行操作，但是一些伪指令在被汇编成具体指令时可能会用到栈。因此，在 `sp` 被设置之前尽量避免不必要的指令。

其二，在使用内联汇编时尽量手动分配寄存器。在这之前我使用的是类似于实验指导书示例一的结构，但是可能由于涉及到的寄存器过多，在众多形如“`mv a0, %[arg0]\n`”之类的汇编指令，导致在被编译器自动分配寄存器、汇编后产生了“`mv a0, a1\n mv a1, a0`”这样的无意义指令。



```
1 register uint64 a0 asm("a0") = arg0;
2 ...
3 __asm__ volatile(
4     "ecall\n"
5     : "=r"(a0), "=r"(a1)
6     : "r"(a0), "r"(a1), "r"(a2), "r"(a3),
7       "r"(a4), "r"(a5), "r"(a6), "r"(a7)
8     : "memory");
```

其三，Linux 在调用 C 函数时，参数表和 `a0,a1,...` 是一一对应的。

最后，强烈建议像交叉编译工具链的名称这样对技术要求不高但是找起来很麻烦的东西能在指导书中列出。

四、思考题

1. 请总结一下 RISC-V 的 calling convention，并解释 Caller/Callee Saved Register 有什么区别？

函数调用过程通常分为 6 个阶段：将参数存储到函数能够访问到的位置；跳

转到函数开始位置；获取函数需要的局部存储资源，按需保存寄存器；执行函数中的指令；将返回值存储到调用者能够访问到的位置，恢复寄存器，释放局部存储资源；返回调用函数的位置。

为了获得良好的性能，变量应该尽量存放在寄存器而不是内存中，但同时也要注意避免频繁地保存和恢复寄存器，因为它们同样会访问内存。为此，RISC-V的解决策略是将相关寄存器区分为 Caller Saved Register 和 Callee Saved Register。前者如 \$ra,\$t0~\$t6 等在发生函数调用时由主调函数保存和恢复，被调函数在执行指令的过程中无需考虑保存它们，可以直接当作临时寄存器使用，这样在一定程度上减轻了对内存的读写需要。后者如 \$s0~\$s7 等由被调函数进行保存和恢复，也就是说，如果被调函数想要使用这些寄存器，它必须保证函数退出时与进入时的寄存器值一致。

2. 编译之后，通过 System.map 查看 vmlinux.lds 中自定义符号的值

如图所示，逐行显示出了每个符号的地址、类型和名称。

```
root@LAPTOP-K817AQP6:~/os23fall-stu/src/lab1# cat System.map
0000000080200000 A BASE_ADDR
0000000080202000 D TIMECLOCK
0000000080204000 B _ebss
0000000080202000 D _edata
0000000080204000 B _kernel
000000008020104f R _erodata
0000000080200554 T _etext
0000000080203000 B _sbss
0000000080202000 D _sdata
0000000080200000 T _skernel
0000000080201000 R _srodata
0000000080200000 T _start
0000000080200000 T _stext
0000000080200054 T _traps
0000000080203000 B boot_stack_top
0000000080204000 B boot_stack_top
00000000802001a0 T clock_set_next_event
000000008020017c T get_cycles
000000008020046c T puti
0000000080200400 T puts
000000008020028c T sbi_console_putchar
00000000802001e8 T sbi_ecall
00000000802002dc T sbi_set_timer
00000000802003b0 T start_kernel
00000000802003f0 T test
0000000080200328 T trap_handler
```

3. 用 csr_read 宏读取 sstatus 寄存器的值，对照 RISC-V 手册解释其含义

我们在 main.c 中以如下形式读取 sstatus 的值。

```
1 register long t0 asm("t0") = csr_read(sstatus);
```

可以观察到，csr_read(sstatus)可以正确读取到 sstatus 的值。

5. Detail your steps about how to get arch/arm64/kernel/sys.i

首先在 linux 目录下通过 `sudo apt install gcc-aarch64-linux-gnu` 安装 Aarch64 交叉编译工具链。

```
● root@LAPTOP-K817AQP:~/linux# sudo apt install gcc-aarch64-linux-gnu
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following additional packages will be installed:
  binutils-aarch64-linux-gnu cpp-11-aarch64-linux-gnu cpp-aarch64-linux-gnu
  gcc-11-aarch64-linux-gnu gcc-11-aarch64-linux-gnu-base gcc-12-cross-base
  libasan6-arm64-cross libatomic1-arm64-cross libc6-arm64-cross libc6-dev-arm64-cross
  libgcc-11-dev-arm64-cross libgcc-s1-arm64-cross libgomp1-arm64-cross libhwasan0-arm64-cross
  libitm1-arm64-cross liblsan0-arm64-cross libstdc++6-arm64-cross libtsan0-arm64-cross
  libubsan1-arm64-cross linux-libc-dev-arm64-cross
```

使用默认配置。

```
● root@LAPTOP-K817AQP:~/linux# make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- defconfig
*** Default configuration is based on 'defconfig'
#
# configuration written to .config
#
```

通过 `make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- arch/arm64/kernel/sys.i`, 指定生成预处理产物 `arch/arm64/kernel/sys.i`。

```
● root@LAPTOP-K817AQP:~/linux# make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- arch/arm64/kernel/sys.i
SYNC      include/config/auto.conf.cmd
WRAP      arch/arm64/include/generated/uapi/asm/kvm_para.h
WRAP      arch/arm64/include/generated/uapi/asm/errno.h
WRAP      arch/arm64/include/generated/uapi/asm/ioctl.h
```

之后, 我们发现该目录下出现了 `sys.i` 文件。

```
● root@LAPTOP-K817AQP:~/linux# cd arch/arm64/kernel/
● root@LAPTOP-K817AQP:~/linux/arch/arm64/kernel# ls
Makefile          entry-fpsimd.S    module.c          sleep.S
acpi.c            entry-ftrace.S    mte.c             smccc-call.S
acpi_numa.c       entry.S           paravirt.c        smp.c
acpi_parking_protocol.c  fpsimd.c         patch-scs.c       smp_spin_table.c
alternative.c     ftrace.c          patching.c        stacktrace.c
armv8_deprecated.c  head.S           pci.c             suspend.c
asm-offsets.c     hibernate-asm.S  perf_callchain.c  sys.c
asm-offsets.s     hibernate.c       perf_regs.c       sys.i
cacheinfo.c       hw_breakpoint.c  pi                sys32.c
compat_alignment.c  hyp-stub.S       probes            sys_compat.c
cpu-reset.S        idle.c            process.c          syscall.c
cpu_errata.c       idreg-override.c proton-pack.c      time.c
cpu_ops.c          image-vars.h      psci.c            topology.c
cpufeature.c       image.h           ptrace.c           trace-events-emulation.h
cpuidle.c          io.c              reloc_test_core.c traps.c
cpuinfo.c          jump_label.c      reloc_test_syms.S vdso
crash_core.c       kaslr.c           relocate_kernel.S vdso-wrap.S
crash_dump.c       kexec_image.c     return_address.c  vdso.c
debug-monitors.c   kgdb.c            sdei.c            vdso32-wrap.S
efi-header.S       kuser32.S         setup.c           vmlinux.lds.S
efi.c              machine_kexec.c   signal.c           watchdog_hld.c
elfcore.c          machine_kexec_file.c  signal32.c
entry-common.c     module-plts.c     sigreturn32.S
```

6. Find system call table of Linux v6.0 for ARM32, RISC-V(32 bit), RISC-V(64 bit), x86(32 bit), x86_64 List source code file, the whole system call table with macro expanded, screenshot every step.

通过 `sudo apt install gcc-arm-linux-gnueabi`, 安装 ARM 交叉编译工具链。

```
root@LAPTOP-K817AQP6:~/linux# sudo apt install gcc-arm-linux-gnueabi
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following additional packages will be installed:
  binutils-arm-linux-gnueabi cpp-11-arm-linux-gnueabi gcc-11-arm-linux-gnueabi
  gcc-11-arm-linux-gnueabi-base libasan6-armel-cross libatomic1-armel-cross libc6-armel-cross
  libc6-dev-armel-cross libgcc-11-dev-armel-cross libgcc-s1-armel-cross libgomp1-armel-cross
  libstdc++6-armel-cross libubsan1-armel-cross linux-libc-dev-armel-cross
```

通过 `make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- defconfig`, 选择默认配置。

```
root@LAPTOP-K817AQP6:~/linux# make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- defconfig
*** Default configuration is based on 'multi_v7_defconfig'
#
# configuration written to .config
#
```

通过 `make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- /arch/arm/kernel/sys_arm.i`, 获得其预处理产物。

```
root@LAPTOP-K817AQP6:~/linux# make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- /arch/arm/kernel/sys_arm.i
SYNC      include/config/auto.conf.cmd
SYSHDR     arch/arm/include/generated/uapi/asm/unistd-oabi.h
SYSHDR     arch/arm/include/generated/uapi/asm/unistd-eabi.h
WRAP       arch/arm/include/generated/uapi/asm/kvm_para.h
```

通过 `cat -n arch/arm/kernel/sys_arm.i | less`, 可以查看其内容。

```
root@LAPTOP-K817AQP6:~/linux# cat -n arch/arm/kernel/sys_arm.i | less
```

```
1 # 0 "arch/arm/kernel/sys_arm.c"
2 # 1 "/root/linux/"
3 # 0 "<built-in>"
4 # 0 "<command-line>"
5 # 1 "./include/linux/compiler-version.h" 1
6 # 0 "<command-line>" 2
7 # 1 "./include/linux/kconfig.h" 1
8
9
10
11
12 # 1 "./include/generated/autoconf.h" 1
13 # 6 "./include/linux/kconfig.h" 2
14 # 0 "<command-line>" 2
15 # 1 "./include/linux/compiler_types.h" 1
16 # 80 "./include/linux/compiler_types.h"
17 # 1 "./include/linux/compiler_attributes.h" 1
18 # 81 "./include/linux/compiler_types.h" 2
19 # 153 "./include/linux/compiler_types.h"
20 # 1 "./include/linux/compiler-gcc.h" 1
21 # 154 "./include/linux/compiler_types.h" 2
22 # 170 "./include/linux/compiler_types.h"
23 struct ftrace_branch_data {
24     const char *func;
25     const char *file;
26     unsigned line;
27     union {
28         struct {
29             unsigned long correct;
30             unsigned long incorrect;
31         };
32         struct {
33             unsigned long miss;
34             unsigned long hit;
35         };
36     };
37 };
38
```

通过 `make ARCH=riscv CROSS_COMPILE=riscv64-linux-gnu- defconfig`, 选择默认配置。

```

root@LAPTOP-K817AQPG:~/linux# make ARCH=riscv CROSS_COMPILE=riscv64-linux-gnu- defconfig
*** Default configuration is based on 'defconfig'
#
# configuration written to .config
#

```

通过 `make ARCH=riscv CROSS_COMPILE=riscv64-linux-gnu- arch/riscv/kernel/syscall_table.i`，获得其预处理产物。

```

root@LAPTOP-K817AQPG:~/linux# make ARCH=riscv CROSS_COMPILE=riscv64-linux-gnu- arch/riscv/kernel/syscall_table.i
HOSTCC scripts/selinux/genheaders/genheaders
HOSTCC scripts/selinux/mdp/mdp
UPD include/generated/compile.h
CC scripts/mod/empty.o
MKELF scripts/mod/elfconfig.h

```

通过 `cat -n arch/riscv/kernel/syscall_table.i | less`，可以查看其内容。

```

root@LAPTOP-K817AQPG:~/linux# cat -n arch/riscv/kernel/syscall_table.i | less

```

```

1 # 0 "arch/riscv/kernel/syscall_table.c"
2 # 1 "/root/linux/"
3 # 0 "<built-in>"
4 # 0 "<command-line>"
5 # 1 "./include/linux/compiler-version.h" 1
6 # 0 "<command-line>" 2
7 # 1 "./include/linux/kconfig.h" 1
8
9
10
11
12 # 1 "./include/generated/autoconf.h" 1
13 # 6 "./include/linux/kconfig.h" 2
14 # 0 "<command-line>" 2
15 # 1 "./include/linux/compiler_types.h" 1
16 # 80 "./include/linux/compiler_types.h"
17 # 1 "./include/linux/compiler_attributes.h" 1
18 # 81 "./include/linux/compiler_types.h" 2
19 # 153 "./include/linux/compiler_types.h"
20 # 1 "./include/linux/compiler-gcc.h" 1
21 # 154 "./include/linux/compiler_types.h" 2
22 # 170 "./include/linux/compiler_types.h"
23 struct ftrace_branch_data {
24     const char *func;
25     const char *file;
26     unsigned line;
27     union {
28         struct {
29             unsigned long correct;
30             unsigned long incorrect;
31         };
32         struct {
33             unsigned long miss;
34             unsigned long hit;
35         };
36     };
37 };

```

考虑到 RISC-V32 和 RISC-V64 共用 riscv 目录，它们的 syscall_table 是同一文件，因此在进行 RISC-V32 的内容前，我们通过 `make clean`，清除编译产物。

```

root@LAPTOP-K817AQPG:~# git clone https://github.com/riscv/riscv-gnu-toolchain
Cloning into 'riscv-gnu-toolchain'...
remote: Enumerating objects: 8544, done.
remote: Counting objects: 100% (51/51), done.
remote: Compressing objects: 100% (39/39), done.
remote: Total 8544 (delta 19), reused 36 (delta 12), pack-reused 8493
Receiving objects: 100% (8544/8544), 5.19 MiB | 2.28 MiB/s, done.
Resolving deltas: 100% (4262/4262), done.

```

检查下列依赖。

```

root@LAPTOP-K817AQPG:~# sudo apt-get install autoconf automake autotools-dev curl python3 libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool patchutils bc zlib1g-dev libexpat-dev ninja-build

```

跳转到这一目录，通过 `./configure --prefix=/opt/riscv --with-arch=rv32gc --with-abi=ilp32d make` 选择安装 RISC-V32 交叉编译工具链。

在此时，下载交叉编译工具链遇到了一些困难。在和曾学姐交流后，我们不作宏展开地给出之后的文件。

对于 RISC-V32 的 `/root/linux/arch/riscv/kernel/syscall_table.c`，通过 `cat -n`

/root/linux/arch/riscv/kernel/syscall_table.c | less，查看其内容。

```
1 // SPDX-License-Identifier: GPL-2.0-only
2 /*
3  * Copyright (C) 2009 Arnd Bergmann <arnd@arndb.de>
4  * Copyright (C) 2012 Regents of the University of California
5  */
6
7 #include <linux/linkage.h>
8 #include <linux/syscalls.h>
9 #include <asm-generic/syscalls.h>
10 #include <asm/syscall.h>
11
12 #undef __SYSCALL
13 #define __SYSCALL(nr, call)    asmlinkage long __riscv_##call(const struct pt_regs *);
14 #include <asm/unistd.h>
15
16 #undef __SYSCALL
17 #define __SYSCALL(nr, call)    [nr] = __riscv_##call,
18
19 void * const sys_call_table[_NR_syscalls] = {
20     [0 ... _NR_syscalls - 1] = __riscv_sys_ni_syscall,
21 #include <asm/unistd.h>
22 };
```

对于 x86(32 bit)的/root/linux/arch/x86/entry/syscall_32.c，通过 cat -n /root/linux/arch/x86/entry/syscall_32.c | less，查看其内容。

```
1 // SPDX-License-Identifier: GPL-2.0
2 /* System call table for i386. */
3
4 #include <linux/linkage.h>
5 #include <linux/sys.h>
6 #include <linux/cache.h>
7 #include <linux/syscalls.h>
8 #include <asm/syscall.h>
9
10 #ifdef CONFIG_IA32_EMULATION
11 #define __SYSCALL_WITH_COMPAT(nr, native, compat)    __SYSCALL(nr, compat)
12 #else
13 #define __SYSCALL_WITH_COMPAT(nr, native, compat)    __SYSCALL(nr, native)
14 #endif
15
16 #define __SYSCALL(nr, sym) extern long __ia32_##sym(const struct pt_regs *);
17
18 #include <asm/syscalls_32.h>
19 #undef __SYSCALL
20
21 #define __SYSCALL(nr, sym) __ia32_##sym,
22
23 __visible const sys_call_ptr_t ia32_sys_call_table[] = {
24 #include <asm/syscalls_32.h>
25 };
```

我们同时给出/root/linux/arch/x86/entry/syscalls/syscall_32.tbl 的内容。

```
1 #
2 # 32-bit system call numbers and entry vectors
3 #
4 # The format is:
5 # <number> <abi> <name> <entry point> <compat entry point>
6 #
7 # The __ia32_sys and __ia32_compat_sys stubs are created on-the-fly for
8 # sys_*( ) system calls and compat_sys_*( ) compat system calls if
9 # IA32_EMULATION is defined, and expect struct pt_regs *regs as their only
10 # parameter.
11 #
12 # The abi is always "i386" for this file.
13 #
14 0      i386      restart_syscall      sys_restart_syscall
15 1      i386      exit                  sys_exit
16 2      i386      fork                  sys_fork
17 3      i386      read                   sys_read
18 4      i386      write                  sys_write
19 5      i386      open                   sys_open                compat_sys_open
20 6      i386      close                  sys_close
21 7      i386      waitpid                 sys_waitpid
22 8      i386      creat                  sys_creat
23 9      i386      link                   sys_link
24 10     i386      unlink                  sys_unlink
25 11     i386      execve                  sys_execve            compat_sys_execve
26 12     i386      chdir                   sys_chdir
27 13     i386      time                     sys_time32
28 14     i386      mknod                    sys_mknod
29 15     i386      chmod                     sys_chmod
30 16     i386      lchown                   sys_lchown16
31 17     i386      break
32 18     i386      oldstat                  sys_stat
33 19     i386      lseek                    sys_lseek            compat_sys_lseek
34 20     i386      getpid                   sys_getpid
```

对于 x86(64 bit)的/root/linux/arch/x86/entry/syscall_64.c, 通过 `cat -n /root/linux/arch/x86/entry/syscall_64.c | less`, 查看其内容。

```
1 // SPDX-License-Identifier: GPL-2.0
2 /* System call table for x86-64. */
3
4 #include <linux/linkage.h>
5 #include <linux/sys.h>
6 #include <linux/cache.h>
7 #include <linux/syscalls.h>
8 #include <asm/syscall.h>
9
10 #define __SYSCALL(nr, sym) extern long __x64_##sym(const struct pt_regs *);
11 #include <asm/syscalls_64.h>
12 #undef __SYSCALL
13
14 #define __SYSCALL(nr, sym) __x64_##sym,
15
16 asmlinkage const sys_call_ptr_t sys_call_table[] = {
17 #include <asm/syscalls_64.h>
18 };
```

我们同时给出/root/linux/arch/x86/entry/syscalls/syscall_64.tbl 的内容。

```
1 #
2 # 64-bit system call numbers and entry vectors
3 #
4 # The format is:
5 # <number> <abi> <name> <entry point>
6 #
7 # The __x64_sys_*( ) stubs are created on-the-fly for sys_*( ) system calls
8 #
9 # The abi is "common", "64" or "x32" for this file.
10 #
11 0      common  read      sys_read
12 1      common  write     sys_write
13 2      common  open      sys_open
14 3      common  close     sys_close
15 4      common  stat      sys_newstat
16 5      common  fstat     sys_newfstat
17 6      common  lstat     sys_newlstat
18 7      common  poll      sys_poll
19 8      common  lseek     sys_lseek
20 9      common  mmap      sys_mmap
21 10     common  mprotect   sys_mprotect
22 11     common  munmap     sys_munmap
23 12     common  brk        sys_brk
24 13     64      rt_sigaction sys_rt_sigaction
25 14     common  rt_sigprocmask sys_rt_sigprocmask
26 15     64      rt_sigreturn sys_rt_sigreturn
27 16     64      ioctl      sys_ioctl
28 17     common  pread64     sys_pread64
29 18     common  pwrite64    sys_pwrite64
30 19     64      readv      sys_readv
31 20     64      writev     sys_writev
32 21     common  access     sys_access
33 22     common  pipe       sys_pipe
34 23     common  select     sys_select
```

7. Explain what is ELF file? Try `readelf` and `objdump` command on an ELF file, give screenshot of the output. Run an ELF file and `cat /proc/PID/maps` to give its memory layout.

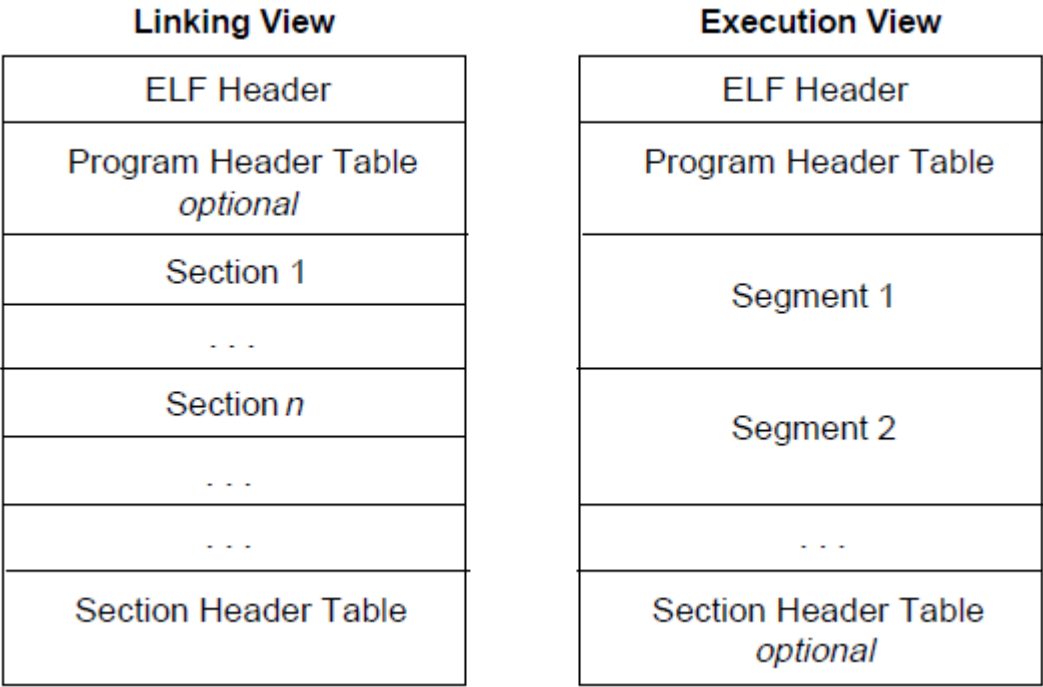
可执行与可链接格式（英语：Executable and Linkable Format，缩写 ELF，此前的写法是 Extensible Linking Format），常被称为 ELF 格式，在计算中，是一种用于可执行文件、目标代码、共享库和核心转储（core dump）的标准文件格式。在 Linux 中，主要有以下三种类型：

可重定位文件（Relocatable File），包含由编译器生成的代码以及数据。链接器会将它与其它目标文件链接起来从而创建可执行文件或者共享目标文件。在

Linux 系统中，这种文件的后缀一般为 .o 。

可执行文件（Executable File），就是我们通常在 Linux 中执行的程序。

共享目标文件（Shared Object File），包含代码和数据，这种文件是我们所称的库文件，一般以 .so 结尾。一般情况下，它有以下两种使用情景：链接器（Linker, ld）可能会处理它和其它可重定位文件以及共享目标文件，生成另外一个目标文件。动态链接器（Dynamic Linker）将它与可执行文件以及其它共享目标组合在一起生成进程镜像。



通过 readelf -a main.o，可以查看 main.o 文件的信息。

```
root@LAPTOP-K817AQPG:~/os23fall-stu/src/lab1/init# readelf -a main.o
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 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:                              REL (Relocatable file)
  Machine:                          RISC-V
  Version:                          0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:         4240 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           64 (bytes)
  Number of section headers:         24
  Section header string table index: 23

Section Headers:
 [Nr] Name              Type              Address            Offset
     Size              EntSize          Flags              Link      Info      Align
 [ 0] 0000000000000000 NULL              0000000000000000 00000000
     0000000000000000 0000000000000000 0 0 0
 [ 1] .text               PROGBITS         0000000000000000 00000040
     0000000000000000 0000000000000000 AX 0 0 4
```

通过 objdump -f main.o，可以显示 main.o 的整体头部摘要信息。


```

root@LAPTOP-K817AQP6:~/os23fall-stu/src/lab1/init# objdump -f main.o

main.o:      file format elf64-little
architecture: UNKNOWN!, flags 0x00000011:
HAS_RELOC, HAS_SYMS
start address 0x0000000000000000

```

为便于演示，我们使用如下 HelloWorld 程序。为使得程序处于运行状态，我们在其中设置了一个死循环。

```

1 #include <stdio.h>
2
3 int main(void){
4     printf("Hello World!\n");
5     while(1);
6     return 0;
7 }

```

运行此程序，通过 ps au，可以观察回到 HelloWorld.exe 的进程号为 30683。

```

root@LAPTOP-K817AQP6:~/os23fall-stu/src/lab0# ps au
USER      PID   %CPU  %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root      229   0.0   0.0    3236 1088 hvc0    Ss+   09:26   0:00 /sbin/agetty -o -p -- \u --no
root      231   0.0   0.0    3192 1080 tty1    Ss+   09:26   0:00 /sbin/agetty -o -p -- \u --no
root      363   0.0   0.0    7472 4900 pts/1    Ss    09:26   0:00 /bin/login -f
root      395   0.0   0.0    6120 4932 pts/1    S+    09:26   0:00 -bash
root      412   0.0   0.0    2888 956 pts/0    Ss+   09:26   0:00 sh -c "$VSCODE_WSL_EXT_LOCATI
root      413   0.0   0.0    2888 948 pts/0    S+    09:26   0:00 sh /mnt/c/Users/无尘 | 溯荒.v
root      418   0.0   0.0    2888 964 pts/0    S+    09:26   0:00 sh /root/.vscode-server/bin/f
root      422   0.1   0.6 962536 101904 pts/0    Rl+   09:26   0:15 /root/.vscode-server/bin/fib0
root      435   0.0   0.3 606956 60644 pts/2    Ssl+  09:26   0:03 /root/.vscode-server/bin/fib0
root      442   0.1   0.4 754792 76580 pts/0    Rl+   09:26   0:13 /root/.vscode-server/bin/fib0
root      518   0.0   0.3 601860 56520 pts/3    Ssl+  09:26   0:03 /root/.vscode-server/bin/fib0
root      528   1.8   1.2 1060324 202192 pts/0    Sl+   09:26   3:32 /root/.vscode-server/bin/fib0
root      672   9.7  12.0 2028220 1967508 pts/0    Sl+   09:26   18:38 /root/.vscode-server/extension
root      720   0.0   0.3 603372 55884 pts/0    Sl+   09:26   0:00 /root/.vscode-server/bin/fib0
root      825   0.1   0.6 852844 104404 pts/0    Sl+   09:26   0:17 /root/.vscode-server/bin/fib0
root     14071 0.0   0.0    5180 4200 pts/10    Ss+  10:28   0:00 /bin/bash --init-file /root/.
root     25571 0.0   0.0    5284 4204 pts/11    Ss    11:52   0:00 /bin/bash --init-file /root/.
root     30412 0.0   0.0 4268812 11796 pts/0    Sl+  12:35   0:00 /root/.vscode-server/extension
root     30683 99.6  0.0    2776 952 pts/11    R+   12:36   0:18 ./HelloWorld.exe
root     30699 0.0   0.0    5180 4300 pts/12    Ss    12:36   0:00 /bin/bash --init-file /root/.
root     30794 0.0   0.0    7484 3188 pts/12    R+   12:36   0:00 ps au

```

通过 cat /proc/30683/maps，即可观察到该程序的内存配置。

```

root@LAPTOP-K817AQP6:~/os23fall-stu/src/lab0# cat /proc/30683/maps
56277eac2000-56277eac3000 r--p 00000000 08:20 253704 /root/os23fall-stu/src/lab0/HelloWorld.exe
56277eac3000-56277eac4000 r-xp 00001000 08:20 253704 /root/os23fall-stu/src/lab0/HelloWorld.exe
56277eac4000-56277eac5000 r--p 00002000 08:20 253704 /root/os23fall-stu/src/lab0/HelloWorld.exe
56277eac5000-56277eac6000 r--p 00002000 08:20 253704 /root/os23fall-stu/src/lab0/HelloWorld.exe
56277eac6000-56277eac7000 rw-p 00003000 08:20 253704 /root/os23fall-stu/src/lab0/HelloWorld.exe
56277eef000-56277ef1e000 rw-p 00000000 00:00 0 [heap]
7f880477b000-7f880477e000 rw-p 00000000 00:00 0
7f880477e000-7f88047a6000 r--p 00000000 08:20 38524 /usr/lib/x86_64-linux-gnu/libc.so.6
7f88047a6000-7f880493b000 r-xp 00028000 08:20 38524 /usr/lib/x86_64-linux-gnu/libc.so.6
7f880493b000-7f8804993000 r--p 001bd000 08:20 38524 /usr/lib/x86_64-linux-gnu/libc.so.6
7f8804993000-7f8804997000 r--p 00214000 08:20 38524 /usr/lib/x86_64-linux-gnu/libc.so.6
7f8804997000-7f8804999000 rw-p 00218000 08:20 38524 /usr/lib/x86_64-linux-gnu/libc.so.6
7f8804999000-7f88049a6000 rw-p 00000000 00:00 0
7f88049a6000-7f88049b0000 rw-p 00000000 00:00 0
7f88049b0000-7f88049b2000 r--p 00000000 08:20 38521 /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f88049b2000-7f88049dc000 r-xp 00002000 08:20 38521 /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f88049dc000-7f88049e7000 r--p 0002c000 08:20 38521 /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f88049e7000-7f88049ea000 r--p 00037000 08:20 38521 /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f88049ea000-7f88049ec000 rw-p 00039000 08:20 38521 /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7ffe44e40000-7ffe44e62000 rw-p 00000000 00:00 0 [stack]
7ffe44e6b000-7ffe44e6f000 r--p 00000000 00:00 0 [vvar]
7ffe44e6f000-7ffe44e71000 r-xp 00000000 00:00 0 [vdso]

```

8. 通过查看 RISC-V Privileged Spec 中的 medeleg 和 mideleg，解释内核启动界面中 MIDELEG 值的含义

可以观察到启动界面 MIDELEG 值如下，即第 1、5、9 位为 1，其余位为 0。


```
Boot HART MIDELEG      : 0x0000000000000222
Boot HART MEDELEG      : 0x000000000000b109
```

MIDELEG 使得一些中断无需进入 M 态，它将处理工作授权(委托)给其他特权模式，在相对低的特权模式(即 S 态)下即可完成处理。MIDELEG 与 mip 的位是一一对应的，因此我们观察 mip 的结构。可以发现，对应第 1、5、9 位分别是 SSIP(software interrupts)、STIP(timer interrupts)、SEIP(external interrupts)，说明 M 态将发生在 S 态及 U 态的软件中断、时钟中断和外部中断的中断处理授权给 S 态完成。

MXLEN-1	12	11	10	9	8	7	6	5	4	3	2	1	0
WPRI	MEIP	WPRI	SEIP	UEIP	MTIP	WPRI	STIP	UTIP	MSIP	WPRI	SSIP	USIP	
MXLEN-12	1	1	1	1	1	1	1	1	1	1	1	1	

Figure 3.11: Machine interrupt-pending register (mip).