

浙江大学

本科实验报告

课程名称：操作系统

姓 名：夏尤楷

学 院：计算机科学与技术学院

系：计算机科学与技术系

专 业：计算机科学与技术

学 号：3210104331

指导教师：夏莹杰

2023 年 10 月 21 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: 459510812@qq.com 手机: 15058004449

实验地点: 玉泉曹光彪西楼 503 实验日期: 2023 年 10 月 22 日

一、实验目的和要求

1. 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。
2. 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。
3. 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。
4. 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。
5. 理解 CPU 上下文切换机制, 并正确实现上下文切换功能。
6. 编写 trap 处理函数, 完成对特定 trap 的处理。
7. 调用 OpenSBI 提供的接口, 完成对时钟中断事件的设置。

二、实验过程

(一) RV64 内核引导

1. 编写 head.S

首先为即将运行的第一个 C 函数设置程序栈(栈的大小设置为 4KB), 并将该栈放置在 .bss.stack 段。接下来通过跳转指令, 跳转至 main.c 中的 start_kernel 函数。代码如下:

```
1. .extern start_kernel
```

```

2.
3.     .section .text.entry
4.     .globl _start
5. _start:
6.     la sp,boot_stack_top
7.     # store the address of the stack top into the register sp
8.     jal start_kernel
9.     .section .bss.stack
10.    .globl boot_stack
11.boot_stack:
12.    .space 4096 # <-- change to your stack size(4KB)
13.
14.    .globl boot_stack_top
15.boot_stack_top:

```

2. 完善 Makefile 脚本

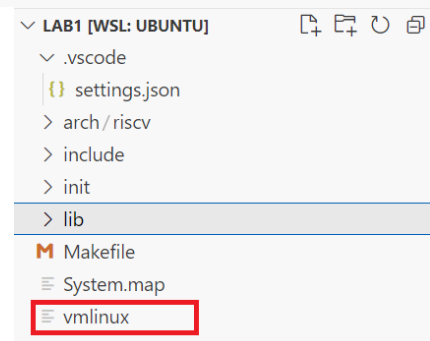
补充 lib/Makefile，使工程得以编译。补充代码如下：

```

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

```

完成此步后在工程根文件夹执行 make，可以看到工程成功编译出 vmlinux。如右图。



3. 补充 sbi.c

OpenSBI 在 M 态，为 S 态提供了多种接口，比如字符串输入输出。因此我们需要实现调用 OpenSBI 接口的功能。在 sbi.h 中，给出函数定义如下：

```

1. struct sbiret {
2.     long error;
3.     long value;
4. };
5.
6. struct sbiret sbi_ecall(int ext, int fid,
7.                         uint64 arg0, uint64 arg1, uint64 arg2,
8.                         uint64 arg3, uint64 arg4, uint64 arg5);

```

sbi_ecall 函数中，需要完成以下内容：

(1) 将 ext (Extension ID) 放入寄存器 a7 中，fid (Function ID) 放入寄存器 a6 中，将 arg0 ~ arg5 放入寄存器 a0 ~ a5 中。

(2) 使用 ecall 指令。ecall 之后系统会进入 M 模式，之后 OpenSBI 会完成相关操作。

(3) OpenSBI 的返回结果会存放在寄存器 a0, a1 中，其中 a0 为 error code, a1 为返回值，我们用 sbiret 来接受这两个返回值。

在 arch/riscv/kernel/sbi.c 中补充 sbi_ecall(), 补充后代码如下：

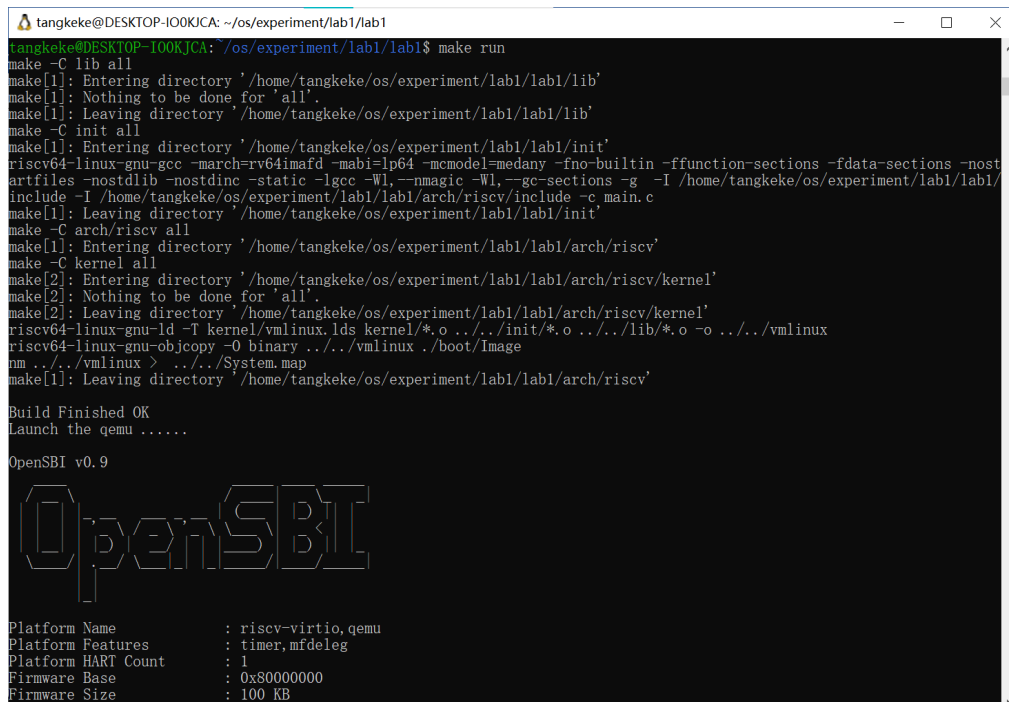
```
1. #include "types.h"
2. #include "sbi.h"
3.
4.
5. struct sbiret sbi_ecall(int ext, int fid, uint64 arg0,
6.                        uint64 arg1, uint64 arg2,
7.                        uint64 arg3, uint64 arg4,
8.                        uint64 arg5)
9. {
10.     struct sbiret retval;
11.
12.     __asm__ volatile (
13.         "mv a7,%[ext]\n"
14.         "mv a6,%[fid]\n"
15.         "mv a5,%[arg5]\n"
16.         "mv a4,%[arg4]\n"
17.         "mv a3,%[arg3]\n"
18.         "mv a2,%[arg2]\n"
19.         "mv a1,%[arg1]\n"
20.         "mv a0,%[arg0]\n"
21.         "ecall \n"
22.         "mv %[rev],a0 \n"
23.         "mv %[val],a1 \n"
24.         : [rev] "=r" (retval.error), [val] "=r" (retval.value)
25.         : [ext] "r" (ext), [fid] "r" (fid), [arg5] "r" (arg5), [arg4] "r" (arg4),
26.         [arg3] "r" (arg3), [arg2] "r" (arg2), [arg1] "r" (arg1), [arg0] "r" (arg0)
27.         : "memory"
28.     );
29.
30.     return retval;
31. }
```

4. 修改 defs.h

补充完 read_csr 这个宏定义，补充后代码如下：

```
1. #ifndef _DEFS_H
2. #define _DEFS_H
3.
4. #include "types.h"
5.
6. #define csr_read(csr) \
7. ({ \
8.     register uint64 __v; \
9.     asm volatile ("csrr " "%0, " #csr \
10.         : "=r" (__v): \
11.         : "memory"); \
12.     __v; \
13. })
14.
15. #define csr_write(csr, val) \
16. ({ \
17.     uint64 __v = (uint64)(val); \
18.     asm volatile ("csrw " #csr ", %0" \
19.         : : "r" (__v) \
20.         : "memory"); \
21. })
22.
23. #endif
```

完成后，在工程根目录通过终端输入 `make run`，即可运行由源代码编译得到的内核，如图：



```
tangkeke@DESKTOP-100KJCA: ~/os/experiment/lab1/lab1$ make run
make -C lib all
make[1]: Entering directory '/home/tangkeke/os/experiment/lab1/lab1/lib'
make[1]: Nothing to be done for 'all'.
make[1]: Leaving directory '/home/tangkeke/os/experiment/lab1/lab1/lib'
make -C init all
make[1]: Entering directory '/home/tangkeke/os/experiment/lab1/lab1/init'
riscv64-linux-gnu-gcc -march=rv64imafd -mabi=lp64 -mcmodel=medany -fno-builtin -ffunction-sections -fdata-sections -nostdlib -nostdinc -static -lgcc -Wl,-nmagic -Wl,-gc-sections -g -I /home/tangkeke/os/experiment/lab1/lab1/include -I /home/tangkeke/os/experiment/lab1/lab1/arch/riscv/include -c main.c
make[1]: Leaving directory '/home/tangkeke/os/experiment/lab1/lab1/init'
make -C arch/riscv all
make[1]: Entering directory '/home/tangkeke/os/experiment/lab1/lab1/arch/riscv'
make -C kernel all
make[2]: Entering directory '/home/tangkeke/os/experiment/lab1/lab1/arch/riscv/kernel'
make[2]: Nothing to be done for 'all'.
make[2]: Leaving directory '/home/tangkeke/os/experiment/lab1/lab1/arch/riscv/kernel'
riscv64-linux-gnu-ld -T kernel/vmlinux.lds kernel/*.o ../../init/*.o ../../lib/*.o ../../vmlinux
nm ../../vmlinux > ../../System.map
make[1]: Leaving directory '/home/tangkeke/os/experiment/lab1/lab1/arch/riscv'

Build Finished OK
Launch the qemu .....

OpenSBI v0.9

OpenSBI

Platform Name      : riscv-virtio, qemu
Platform Features  : timer, mfd deleg
Platform HART Count : 1
Firmware Base      : 0x80000000
Firmware Size      : 100 KB
```

（二）RV64 时钟中断处理

1. 修改 vmlinux.lds 和 head.S 的准备工作

修改后的 vmlinux.lds 如下：

```
1. /* 目标架构 */
2. OUTPUT_ARCH( "riscv" )
3.
4. /* 程序入口 */
5. ENTRY( _start )
6.
7. /* kernel 代码起始位置 */
8. BASE_ADDR = 0x80200000;
9.
10. SECTIONS
11. {
12.     /* . 代表当前地址 */
13.     . = BASE_ADDR;
14.
15.     /* 记录 kernel 代码的起始地址 */
16.     _skernel = .;
17.
18.     /* ALIGN(0x1000) 表示 4KB 对齐 */
19.     /* _stext, _etext 分别记录了 text 段的起始与结束地址 */
20.     .text : ALIGN(0x1000){
21.         _stext = .;
22.
23.         *(.text.init)
24.         *(.text.entry)
25.         *(.text .text.*)
26.
27.         _etext = .;
28.     }
29.
30.     .rodata : ALIGN(0x1000){
31.         _srodata = .;
32.
33.         *(.rodata .rodata.*)
34.
35.         _erodata = .;
36.     }
37.
38.     .data : ALIGN(0x1000){
39.         _sdata = .;
```

```

40.
41.     *(.data .data.*)
42.
43.     _edata = .;
44. }
45.
46. .bss : ALIGN(0x1000){
47.     _sbss = .;
48.
49.     *(.bss.stack)
50.     *(.sbss .sbss.*)
51.     *(.bss .bss.*)
52.
53.     _ebss = .;
54. }
55.
56. /* 记录 kernel 代码的结束地址 */
57. _ekernel = .;
58.}

```

修改后的 head.S 如下：

```

1. .extern start_kernel
2.
3. .section .text.init
4. .globl _start
5. _start:
6.     la sp,boot_stack_top
7.     # store the address of the stack top into the register sp
8.     jal start_kernel
9. .section .bss.stack
10. .globl boot_stack
11.boot_stack:
12. .space 4096 # <-- change to your stack size(4KB)
13.
14. .globl boot_stack_top
15.boot_stack_top:

```

2. 开启 trap 处理

除了 32 个通用寄存器之外，RISC-V 架构还有大量的控制状态寄存器（Control and Status Register, CSR），如 `stvec`、`sie`、`sstatus`、`scause`、`sepc` 等。在运行 `start_kernel` 之前要对这些 CSR 进行初始化，初始化包括以下几个步骤：

(1) 设置 `stvec`，将 `_traps` 所表示的地址写入 `stvec`，这里我们

采用 Direct 模式，而 `_traps` 则是 `trap` 处理入口函数的地址。

- (2) 开启时钟中断，将 `sie[STIE]` 置 1。查询手册，得 `sie[STIE] = sie[5]`，如下图。

SXLEN-1	10	9	8	7	6	5	4	3	2	1	0
WPRI	SEIE	UEIE	WPRI	STIE	UTIE	WPRI	SSIE	USIE			
SXLEN-10	1	1	2	1	1	2	1	1			

Figure 4.5: Supervisor interrupt-enable register (`sie`).

- (3) 设置第一次时钟中断。
- (4) 开启 S 态下的中断响应，将 `sstatus[SIE]` 置 1。查询手册，得 `sstatus[SIE] = sstatus[1]`，如下图。

SXLEN-1	SXLEN-2	34	33	32	31	20	19	18	17
SD	WPRI	UXL	WPRI	MXR	SUM	WPRI			
1	SXLEN-35	2	12	1	1	1			

16	15	14	13	12	9	8	7	6	5	4	3	2	1	0
XS[1:0]	FS[1:0]	WPRI	SPP	WPRI	SPIE	UPIE	WPRI	SIE	UIE					
2	2	4	1	2	1	1	2	1	1					

Figure 4.2: Supervisor-mode status register (`sstatus`) for RV64.

在 `arch/riscv/kernel/head.S` 中完成这些步骤，完成后代码如下：

```

1. .extern start_kernel
2.
3.     .section .text.init
4.     .globl _start
5. _start:
6.     la t0,_traps
7.     csrw stvec,t0
8.     # set stvec = _traps
9.
10.    csrr t0,sie
11.    ori t0,t0,0x20
12.    csrw sie,t0
13.    # set sie[STIE] = 1
14.
15.    andi a7,x0,0x00
16.    andi a6,x0,0
17.    andi a5,x0,0
18.    andi a4,x0,0
19.    andi a3,x0,0
20.    andi a2,x0,0
21.    andi a1,x0,0
22.    li t0,10000000

```



```

23.    rdttime a0
24.    add a0,a0,t0
25.    ecall
26.    # set first time interrupt
27.
28.    csrr t0,sstatus
29.    ori t0,t0,0x2
30.    csrwr sstatus,t0
31.    # set sstatus[SIE] = 1
32.
33.    la sp,boot_stack_top
    # store the address of the stack top into the register sp
34.    jal start_kernel
35.    .section .bss.stack
36.    .globl boot_stack
37.boot_stack:
38.    .space 4096 # <-- change to your stack size(4KB)
39.
40.    .globl boot_stack_top
41.boot_stack_top:

```

3. 实现上下文（即系统状态）切换

- (1) 在 arch/riscv/kernel/ 目录下添加 entry.S 文件。
- (2) 保存 CPU 的寄存器（上下文）到内存中（栈上）。
- (3) 将寄存器 `scause` 和 `sepc` 中的值传入 `trap` 处理函数 `trap_handler`。
- (4) 在完成对 `trap` 的处理之后，从内存中（栈上）恢复 CPU 的寄存器（上下文）。
- (5) 从 `trap` 中返回。

entry.S 的代码如下：

```

1.    .section .text.entry
2.    .align 2
3.    .globl _traps
4.    _traps:
5.        addi sp,sp,-33*8
6.        sd x0,0*8(sp)
7.        sd x1,1*8(sp)
8.        sd x2,2*8(sp)
9.        sd x3,3*8(sp)
10.       sd x4,4*8(sp)
11.       sd x5,5*8(sp)

```

```
12.    sd x6,6*8(sp)
13.    sd x7,7*8(sp)
14.    sd x8,8*8(sp)
15.    sd x9,9*8(sp)
16.    sd x10,10*8(sp)
17.    sd x11,11*8(sp)
18.    sd x12,12*8(sp)
19.    sd x13,13*8(sp)
20.    sd x14,14*8(sp)
21.    sd x15,15*8(sp)
22.    sd x16,16*8(sp)
23.    sd x17,17*8(sp)
24.    sd x18,18*8(sp)
25.    sd x19,19*8(sp)
26.    sd x20,20*8(sp)
27.    sd x21,21*8(sp)
28.    sd x22,22*8(sp)
29.    sd x23,23*8(sp)
30.    sd x24,24*8(sp)
31.    sd x25,25*8(sp)
32.    sd x26,26*8(sp)
33.    sd x27,27*8(sp)
34.    sd x28,28*8(sp)
35.    sd x29,29*8(sp)
36.    sd x30,30*8(sp)
37.    sd x31,31*8(sp)
38.    csrr t0,sepc
39.    sd t0,32*8(sp)
40.    # save 32 registers and sepc to stack
41.
42.    csrr a0,scause
43.    csrr a1,sepc
44.    jal trap_handler
45.    # call trap_handler
46.
47.    ld x0,0*8(sp)
48.    ld x1,1*8(sp)
49.    ld x2,2*8(sp)
50.    ld x3,3*8(sp)
51.    ld x4,4*8(sp)
52.    ld x5,5*8(sp)
53.    ld x6,6*8(sp)
54.    ld x7,7*8(sp)
55.    ld x8,8*8(sp)
```

```

56.    ld x9,9*8(sp)
57.    ld x10,10*8(sp)
58.    ld x11,11*8(sp)
59.    ld x12,12*8(sp)
60.    ld x13,13*8(sp)
61.    ld x14,14*8(sp)
62.    ld x15,15*8(sp)
63.    ld x16,16*8(sp)
64.    ld x17,17*8(sp)
65.    ld x18,18*8(sp)
66.    ld x19,19*8(sp)
67.    ld x20,20*8(sp)
68.    ld x21,21*8(sp)
69.    ld x22,22*8(sp)
70.    ld x23,23*8(sp)
71.    ld x24,24*8(sp)
72.    ld x25,25*8(sp)
73.    ld x26,26*8(sp)
74.    ld x27,27*8(sp)
75.    ld x28,28*8(sp)
76.    ld x29,29*8(sp)
77.    ld x30,30*8(sp)
78.    ld x31,31*8(sp)
79.    ld t0,32*8(sp)
80.    csrw sepc,t0
81.    addi x2,x2,33*8
82.    # restore sepc and 32 registers (x2(sp) should be restore last)
    from stack
83.
84.    sret
85.    # return from trap

```

4. 实现 trap 处理函数

在 arch/riscv/kernel/ 目录下添加 trap.c 文件，在其中实现 trap 处理函数 `trap_handler()`，其接收的两个参数分别是 `scause` 和 `sepc` 两个寄存器中的值。

在本实验中，我们只设置 Supervisor Timer Interrupt，查询手册得到该中断的异常码（存储于 `scause` 寄存器后 63 位）为 5，如右图。

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	9	Supervisor external interrupt
1	10-15	<i>Reserved for future standard use</i>
1	≥16	<i>Reserved for platform use</i>

trap.c 内代码如下：

```
1. #include "printk.h"
2.
3. extern void clock_set_next_event();
4.
5. void trap_handler(unsigned long scause, unsigned long sepc) {
6.     if (scause >> 63){ // 通过 `scause` 判断 trap 类型
7.         if (scause % 8 == 5) { // 如果是 interrupt 判断是否是
timer interrupt
8. // 如果是 timer interrupt 则打印输出相关信息, 并通过 clock_set_next_event()
设置下一次时钟中断
9.         printk("[S] Time interrupt!\n");
10.        clock_set_next_event();
11.    }
12. }
13. // `clock_set_next_event()` 见 4.3.4 节
14. // 其他 interrupt / exception 可以直接忽略
15. }
```

5. 实现时钟中断相关函数

在 arch/riscv/kernel/ 目录下添加 clock.c 文件, 在其中实现 get_cycles() (使用 rdttime 汇编指令获得当前 time 寄存器中的值) 和 clock_set_next_event() (调用 sbi_ecall, 设置下一个时钟中断事件)。代码如下:

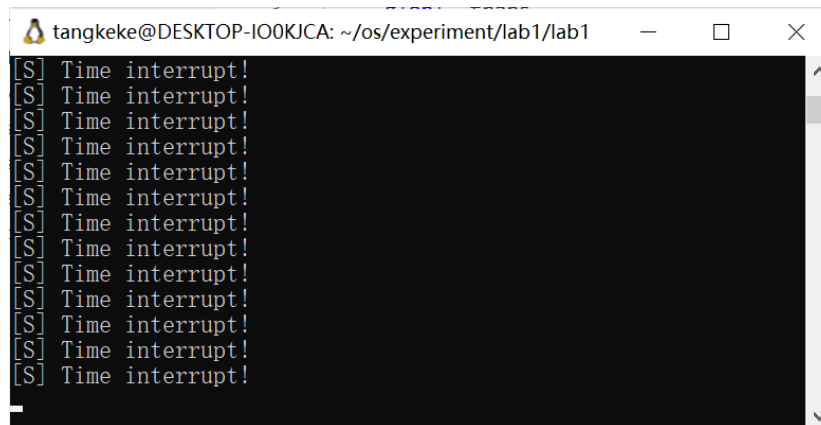
```
1. #include "sbi.h"
2.
3. // QEMU 中时钟的频率是 10MHz, 也就是 1 秒钟相当于 100000000 个时钟周期。
4. unsigned long TIMECLOCK = 100000000;
5.
6. unsigned long get_cycles() {
// 编写内联汇编, 使用 rdttime 获取 time 寄存器中 (也就是 mtime 寄存器) 的值并返回
7.     unsigned long time;
8.
9.     __asm__ volatile(
10.         "rdtime %[time]"
11.         :[time] "=r" (time)
12.         : : "memory"
13.     );
14.
15.     return time;
16. }
17.
```

```

18. void clock_set_next_event() {
19.     // 下一次 时钟中断 的时间点
20.     unsigned long next = get_cycles() + TIMECLOCK;
21.
22.     // 使用 sbi_ecall 来完成对下一次时钟中断的设置
23.     sbi_ecall(0x00,0,next,0,0,0,0,0);
24. }

```

完成以上步骤后，即可编译运行。运行时，程序每隔一秒输出一行 “[S] Time interrupt!”，如下图。



A terminal window titled 'tangkeke@DESKTOP-IOOKJCA: ~/os/experiment/lab1/lab1' showing a series of '[S] Time interrupt!' messages printed on separate lines, indicating successful timer interrupts.

三、讨论和心得

本次实验让我了解了 32 个通用寄存器之外的不同的控制状态寄存器（CSR）及其各自独特的作用，对操作系统的中断的处理机制有了更深入的理解。同时，这次实验让我初步学习了许多技能，比如 RISC-V 架构的汇编语言的编写和 Makefile 文件的编写。

四、思考题

1. calling convention:

函数调用过程通常分为 6 个阶段：

- (1) 将参数存储到函数能够访问到的位置；
- (2) 跳转到函数开始的位置（使用 RV32I 的 jal 指令）；
- (3) 获取函数需要的局部存储资源，按需保存寄存器；
- (4) 执行函数中的指令；
- (5) 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；
- (6) 返回调用函数的位置（ret 指令）。

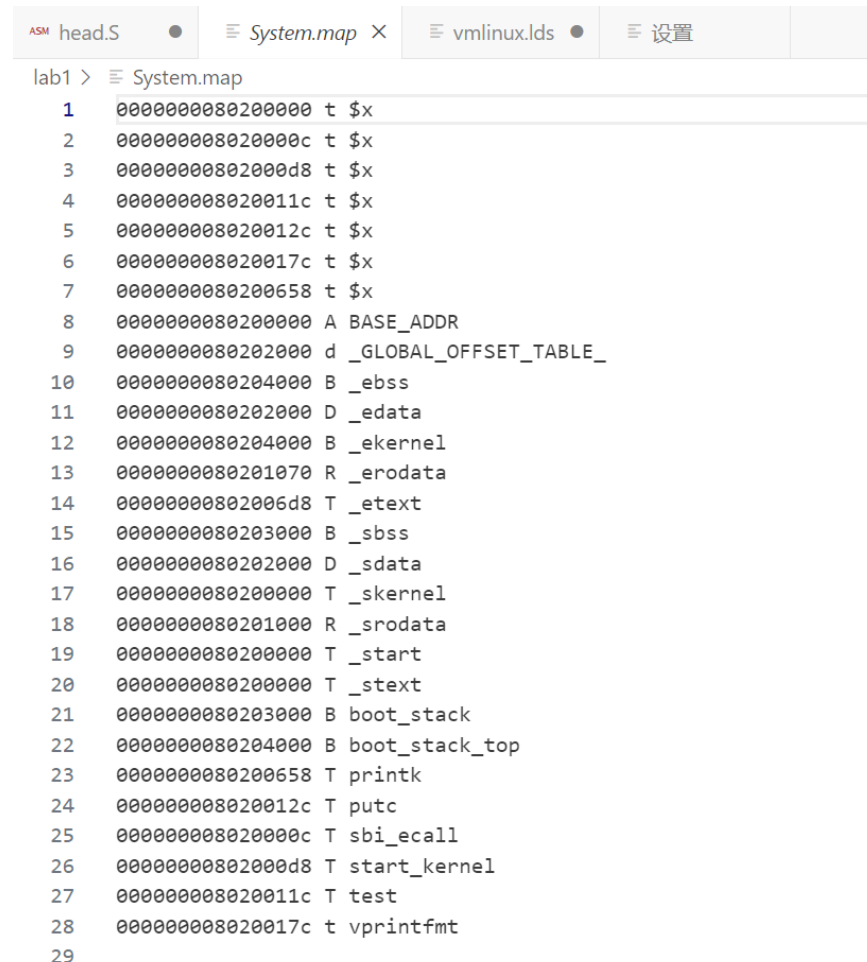
为了获得良好的性能，变量应该尽量存放在寄存器而不是内存中，但同时也要注意避免频繁地保存和恢复寄存器，因为这些操作会访问内存而降低性能。

caller saved register 指在函数调用前需要调用者主动保存的寄存器。这些寄存器在被调

用者中可以被随意更改，因而在调用前由调用者进行保存，调用后由调用者恢复；

callee saved register 指在函数调用中需要被调用者进行保存的寄存器。在调用者看来，这些寄存器内的值在调用前后应该保持不变，所以被调用者如果需要更改这种寄存器，则需要在调用开始时保存该寄存器，在调用结束时返回调用者前恢复该寄存器。

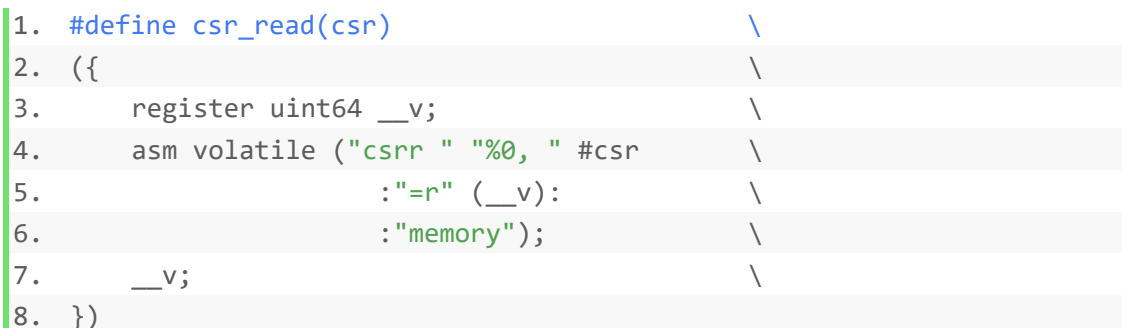
2. 如下图：



```
ASM head.S  System.map x vmlinux.lds 设置
lab1 > System.map
1 0000000080200000 t $x
2 000000008020000c t $x
3 00000000802000d8 t $x
4 000000008020011c t $x
5 000000008020012c t $x
6 000000008020017c t $x
7 0000000080200658 t $x
8 0000000080200000 A BASE_ADDR
9 0000000080202000 d _GLOBAL_OFFSET_TABLE_
10 0000000080204000 B _ebss
11 0000000080202000 D _edata
12 0000000080204000 B _kernel
13 0000000080201070 R _erodata
14 00000000802006d8 T _etext
15 0000000080203000 B _sbss
16 0000000080202000 D _sdata
17 0000000080200000 T _skernel
18 0000000080201000 R _srodata
19 0000000080200000 T _start
20 0000000080200000 T _stext
21 0000000080203000 B boot_stack
22 0000000080204000 B boot_stack_top
23 0000000080200658 T printk
24 000000008020012c T putc
25 000000008020000c T sbi_ecall
26 00000000802000d8 T start_kernel
27 000000008020011c T test
28 000000008020017c t vprintfmt
29
```

3. 如图 1，适当修改 main.c 文件，其中定义 **uint64** 型变量 **rval**，并添加语句：“**rval = csr_read(sstatus);**”

对照 RISC-V 手册中的话（如图 2），参照宏的内容：



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

可得该语句的含义是：宏 **csr_read** 通过汇编指令 **csrr** 将寄存器 **sstatus** 的值读出并传出，这个值被赋给变量 **rval**。在后续语句中，变量 **rval** 的值被移入寄存器 **a7**。之后，通过 **gdb** 可以看到，寄存器 **a7** 内的值和寄存器 **sstatus** 的值一致，说明宏成功读取了寄存器 **sstatus** 的值。

```

main.c
7 int start_kernel() {
8     uint64 rval;
9     int val = 25;
10    printk("2022");
11    printk("Hello RISC-V\n");
12    rval = csr_read(sscratch);
13    asm volatile(
14        "mv a7,%[rval]"
15        : :[rval] "r" (rval) : "memory"
16    );
17    csr_write(sscratch, val);
18
19    test(); // DO NOT DELETE !!
20
21    return 0;
22 }

remote Thread 1.1 In: start_kernel L19 PC: 0x80200130
(gdb) i r a7
a7 0x8000000000000600 -9223372036854751232
(gdb) i r sscratch
sscratch 0x19 25
(gdb)

```

(图 1)

CSrr rd, csr

x[rd] = CSRs[csr]

读控制状态寄存器 (Control and Status Register Read). 伪指令(Pseudoinstruction), RV32I and RV64I.

把控制状态寄存器 *csr* 的值写入 *x[rd]*, 等同于 **csrrs rd, csr, x0**.

(图 2)

- 如图 1, 适当修改 main.c 文件, 其中定义 int 型变量 val, 并初始化为 25。在执行语句 “csr_write(sscratch, val);”后, 可以通过 gdb 看到, 寄存器 sscratch 内的值也变成了 25, 说明通过宏的写入成功。
- 先进入源码目录, 输入命令 “make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- defconfig”, 再输入命令 “make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu-arch/arm64/kernel/sys.i”, 即可得到所需文件。下两图表明通过这两个命令, 确实生成了该文件。

```

tangkeke@DESKTOP-100KJCA: ~/os/experiment/lab0/linux-6.6-rc4/arch/arm64/kernel$ make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- defconfig
HOSTCC scripts/basic/fixdep
HOSTCC scripts/kconfig/conf.o
HOSTCC scripts/kconfig/confdata.o
HOSTCC scripts/kconfig/expr.o
HOSTCC scripts/kconfig/lexer.lex.o
HOSTCC scripts/kconfig/parser.tab.[ch]
HOSTCC scripts/kconfig/lexer.lex.o
HOSTCC scripts/kconfig/mconf.o
HOSTCC scripts/kconfig/parser.tab.o
HOSTCC scripts/kconfig/preprocess.o
HOSTCC scripts/kconfig/symbol.o
HOSTCC scripts/kconfig/util.o
HOSTLD scripts/kconfig/conf
*** Default configuration is based on 'defconfig'
#
# configuration written to: config
#
tangkeke@DESKTOP-100KJCA: ~/os/experiment/lab0/linux-6.6-rc4$ make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- arch/arm64/kernel/sys.i
SYNCC include/config/auto.conf.cmd
HOSTCC scripts/dtc/dtc.o
HOSTCC scripts/dtc/flattree.o
HOSTCC scripts/dtc/istree.o
HOSTCC scripts/dtc/data.o
HOSTCC scripts/dtc/livetree.o

tangkeke@DESKTOP-100KJCA: ~/os/experiment/lab0/linux-6.6-rc4/arch/arm64/kernel$ ls sys.i
sys.i

```

- arm32: (另需 gcc-arm-linux-gnueabi 工具链)
源码文件: arch/arm/kernel/entry-common.S、arch/arm/include/generated/calls-eabi.S、arch/arm/include/generated/calls-oabi.S (后两个需 make 生成) 等。
在 arch/arm 目录下搜索 sys_call_table, 在 kernel 目录下的 entry-common.S 文件中搜索到如下图代码段:

```

/*
 * This is the syscall table declaration for native ABI syscalls.
 * With EABI a couple syscalls are obsolete and defined as sys_ni_syscall.
 */
syscall_table_start sys_call_table
#ifdef CONFIG_AEABI
#include <calls-eabi.S>
#else
#include <calls-oabi.S>
#endif
syscall_table_end sys_call_table

```

发现其实有两套系统调用表，分别存储在名为“calls-eabi.S”和“calls-oabi.S”的源文件中。依次在源代码根目录输入以下命令，效果如下图：

1. `make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi-hf- defconfig`
2. `make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi-hf- arch/arm/kernel/entry-common.s`

```

tangkeke@DESKTOP-I00KJCA: /os/experiment/lab0/linux-6.6-rc4$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi-hf- arch/arm/kernel/entry-common.s
SYSHDR arch/arm/include/generated/uapi/asm/unistd-oabi.h
SYSHDR arch/arm/include/generated/uapi/asm/unistd-eabi.h
UPD include/generated/compile.h
SYSNR arch/arm/include/generated/asm/unistd-nr.h
GEN arch/arm/include/generated/asm/mach-types.h
SYSTBL arch/arm/include/generated/calls-oabi.S
SYSTBL arch/arm/include/generated/calls-eabi.S
CC scripts/mod/empty.o
MKELF scripts/mod/elfconfig.h
HOSTCC scripts/mod/modpost.o
CC scripts/mod/devicetable-offsets.s
UPD scripts/mod/devicetable-offsets.h
HOSTCC scripts/mod/file2alias.o
HOSTCC scripts/mod/sumversion.o
HOSTLD scripts/mod/modpost
UPD include/generated/timeconst.h
CC kernel/bounds.s
UPD include/generated/bounds.h
CC arch/arm/kernel/asm-offsets.s
UPD include/generated/asm-offsets.h
CALL scripts/checksyscalls.sh
CPP arch/arm/kernel/entry-common.s
tangkeke@DESKTOP-I00KJCA: /os/experiment/lab0/linux-6.6-rc4$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi-hf- arch/arm/kernel/entry-common.s

```

生成并打开 `arch/arm/kernel/entry-common.s` 文件，即可看到经过宏展开后的系统调用表内容，如下图：

```

ASM calls-oabi.S  C sys_oabi-compat.c  ASM entry-common.S  ASM entry-common.s X
kernel > ASM entry-common.s
1227 # 1 "./arch/arm/include/generated/calls-eabi.S" 1
1228 syscall 0, sys_restart_syscall
1229 syscall 1, sys_exit
1230 syscall 2, sys_fork
1231 syscall 3, sys_read
1232 syscall 4, sys_write
1233 syscall 5, sys_open
1234 syscall 6, sys_close
1235 syscall 7, sys_ni_syscall
1236 syscall 8, sys_creat
1237 syscall 9, sys_link
1238 syscall 10, sys_unlink
1239 syscall 11, sys_execve
1240 syscall 12, sys_chdir
1241 syscall 13, sys_ni_syscall
1242 syscall 14, sys_mknod
1243 syscall 15, sys_chmod
1244 syscall 16, sys_lchown16
1245 syscall 17, sys_ni_syscall
1246 syscall 18, sys_ni_syscall
1247 syscall 19, sys_lseek
1248 syscall 20, sys_getpid
1249 syscall 21, sys_mount
1250 syscall 22, sys_ni_syscall
1251 syscall 23, sys_setuid16
1252 syscall 24, sys_getuid16
1253 syscall 25, sys_ni_syscall
1254 syscall 26, sys_ptrace
1255 syscall 27, sys_ni_syscall
1256 syscall 28, sys_ni_syscall

```


RISC-V(32 bit):

源 码 文 件： arch/riscv/kernel/syscall_table.c 、 arch/riscv/include/asm/unistd.h 、 arch/riscv/include/uapi/asm/unistd.h 等。

进入 arch/riscv/kernel 目录，发现有文件 syscall_table.c，此即为系统调用表源文件，如下图所示：

```
tangkeke@DESKTOP-100KJCA: /os/experiment/lab0/linux-6.6-rc4/arch/riscv/kernel$ ls
Makefile          cpu_ops_spinwait.c  image-vars.h        probes              sys_riscv.c
acpi.c            cpufeature.c        irq.c                process.c           syscall_table.c
alternative.c     crash_core.c        jump_label.c        ptrace.c            time.c
asm-offsets.c    crash_dump.c        kexec_relocate.S    reset.c            traps.c
cacheinfo.c      crash_save_regs.S   kgdb.c              riscv_ksyms.c       traps_misaligned.c
cfi.c            efi-header.S        machine_kexec.c      sbi-ipi.c           vdso
compat_signal.c   efi.c               machine_kexec_file.c sbi.c               vdso.c
compat_syscall_table.c elf_kexec.c          mcount-dyn.S        setup.c             vector.c
compat_vdso       entry.S              mcount.S             signal.c             vmlinux-xip.lds.S
copy-unaligned.S fpu.S                module-sections.c    smp.c               vmlinux.lds
copy-unaligned.h ftrace.c             module.c              smptboot.c           vmlinux.lds.S
cpu-hotplug.c    head.S               patch.c              soc.c
cpu.c             head.h               perf_callchain.c     stacktrace.c
cpu_ops.c         hibernate-asm.S     perf_regs.c           suspend.c
cpu_ops_sbi.c     hibernate.c          pi                     suspend_entry.S
```

打开该文件，显示系统调用表有源自 asm/unistd.h 文件，如图：

```
void * const sys_call_table[__NR_syscalls] = {
|   [0 ... __NR_syscalls - 1] = __riscv_sys_ni_syscall,
#include <asm/unistd.h>
};
```

打开该文件，又发现该文件又有源自 uapi/asm/unistd.h 文件，如图：

```
#include <uapi/asm/unistd.h>

#define NR_syscalls ( __NR_syscalls)
```

编译所需的相关工具链在 wsl 上难以安装，故无法导出相应的系统调用表。

RISC-V(64 bit):

源 码 文 件： arch/riscv/kernel/syscall_table.c 、 arch/riscv/include/asm/unistd.h 、 arch/riscv/include/uapi/asm/unistd.h 等。

进入 arch/riscv/kernel 目录，发现有文件 syscall_table.c，此即为系统调用表源文件，如下图所示：

```
tangkeke@DESKTOP-100KJCA: /os/experiment/lab0/linux-6.6-rc4/arch/riscv/kernel$ ls
Makefile          cpu_ops_spinwait.c  image-vars.h        probes              sys_riscv.c
acpi.c            cpufeature.c        irq.c                process.c           syscall_table.c
alternative.c     crash_core.c        jump_label.c        ptrace.c            time.c
asm-offsets.c    crash_dump.c        kexec_relocate.S    reset.c            traps.c
cacheinfo.c      crash_save_regs.S   kgdb.c              riscv_ksyms.c       traps_misaligned.c
cfi.c            efi-header.S        machine_kexec.c      sbi-ipi.c           vdso
compat_signal.c   efi.c               machine_kexec_file.c sbi.c               vdso.c
compat_syscall_table.c elf_kexec.c          mcount-dyn.S        setup.c             vector.c
compat_vdso       entry.S              mcount.S             signal.c             vmlinux-xip.lds.S
copy-unaligned.S fpu.S                module-sections.c    smp.c               vmlinux.lds
copy-unaligned.h ftrace.c             module.c              smptboot.c           vmlinux.lds.S
cpu-hotplug.c    head.S               patch.c              soc.c
cpu.c             head.h               perf_callchain.c     stacktrace.c
cpu_ops.c         hibernate-asm.S     perf_regs.c           suspend.c
cpu_ops_sbi.c     hibernate.c          pi                     suspend_entry.S
```

打开该文件，显示系统调用表有源自 asm/unistd.h 文件，如图：

```
void * const sys_call_table[__NR_syscalls] = {
|   [0 ... __NR_syscalls - 1] = __riscv_sys_ni_syscall,
#include <asm/unistd.h>
};
```

打开该文件，又发现该文件又有源自 uapi/asm/unistd.h 文件，如图：

```
#include <uapi/asm/unistd.h>

#define NR_syscalls (__NR_syscalls)
```

在源代码根目录下依次执行以下命令，效果如下图：

3. `make ARCH=riscv defconfig`
4. `make ARCH=riscv CROSS_COMPILE=riscv64-linux-gnu- arch/riscv/kernel/syscall_table.i`

```
tangkeke@DESKTOP-I00KJCA:~/os/experiment/lab0/linux-6.6-rc4$ make ARCH=riscv defconfig
HOSTCC scripts/basic/fixdep
HOSTCC scripts/kconfig/conf.o
HOSTCC scripts/kconfig/confdata.o
HOSTCC scripts/kconfig/expr.o
LEX scripts/kconfig/lexer.lex.c
YACC scripts/kconfig/parser.tab.[ch]
HOSTCC scripts/kconfig/lexer.lex.o
HOSTCC scripts/kconfig/menu.o
HOSTCC scripts/kconfig/parser.tab.o
HOSTCC scripts/kconfig/preprocess.o
HOSTCC scripts/kconfig/symbol.o
HOSTCC scripts/kconfig/util.o
HOSTLD scripts/kconfig/conf
*** Default configuration is based on 'defconfig'
#
# configuration written to .config
#
tangkeke@DESKTOP-I00KJCA:~/os/experiment/lab0/linux-6.6-rc4$ make ARCH=riscv CROSS_COMPILE=riscv64-linux-
gnu- arch/riscv/kernel/syscall_table.i
SYNC include/config/auto.conf.cmd
```

生成并打开 arch/riscv/kernel/syscall_table.i 文件，即可看到经过宏展开后的系统调用表内容，如下图：

C syscall_table.c 4	C syscall_table.i X	32-bit.config	64-bit.config	C unistd.h include/asm
kernel > C syscall_table.i > ...				
58699	<code>void * const sys_call_table[453] = {</code>			
58700	<code> [0 ... 453 - 1] = __riscv_sys_ni_syscall,</code>			
58701	<code># 1 "../arch/riscv/include/asm/unistd.h" 1</code>			
58702	<code># 24 "../arch/riscv/include/asm/unistd.h"</code>			
58703	<code># 1 "../arch/riscv/include/uapi/asm/unistd.h" 1</code>			
58704	<code># 26 "../arch/riscv/include/uapi/asm/unistd.h"</code>			
58705	<code># 1 "../include/uapi/asm-generic/unistd.h" 1</code>			
58706	<code># 34 "../include/uapi/asm-generic/unistd.h"</code>			
58707	<code>[0] = __riscv_sys_io_setup,</code>			
58708				
58709	<code>[1] = __riscv_sys_io_destroy,</code>			
58710				
58711	<code>[2] = __riscv_sys_io_submit,</code>			
58712				
58713	<code>[3] = __riscv_sys_io_cancel,</code>			
58714				
58715				
58716				
58717	<code>[4] = __riscv_sys_io_getevents,</code>			
58718				
58719				
58720				
58721	<code>[5] = __riscv_sys_setxattr,</code>			
58722				
58723	<code>[6] = __riscv_sys_lsetxattr,</code>			
58724				
58725	<code>[7] = __riscv_sys_fsetxattr,</code>			
58726				
58727	<code>[8] = __riscv_sys_getxattr,</code>			
58728				

x86(32 bit): (另需 gcc-i686-linux-gnu 工具链)

源码文件: arch/x86/entry/syscall_32.c, arch/x86/include/generated/asm/syscalls_32.h (需 make 生成) 等。

进入 arch/x86/entry 目录, 发现有文件 syscall_32.c, 如图:

```
tangkeke@DESKTOP-I00KJCA: /os/experiment/lab0/linux-6.6-rc4/arch/x86/entry$ ls
Makefile  common.c  entry_32.S  entry_64.compat.S  syscall_64.c  syscalls  thunk_64.S  vsyscall
calling.h  entry.S    entry_64.S  syscall_32.c       syscall_x32.c  thunk_32.S  vds
```

查看该文件, 里面显示这是 i386 (即 x86(32 bit)) 的系统调用表源文件, 如下图:

```
Makefile ~/.../linux-6.6-rc4  M Makefile ./  C syscall_x32.c 8  C syscall_32.c 6 X
entry > C syscall_32.c > ...
1  // SPDX-License-Identifier: GPL-2.0
2  /* System call table for i386. */
```

文件里显示系统调用表有源自 asm/syscalls_32.h, 如下图:

```
__visible const sys_call_ptr_t ia32_sys_call_table[] = {
#include <asm/syscalls_32.h>
};
```

在源代码根目录下依次执行以下命令, 效果如下图:

```
tangkeke@DESKTOP-I00KJCA: /os/experiment/lab0/linux-6.6-rc4$ make ARCH=i386 CROSS_COMPILE=i686-linux-gnu- defconfig
HOSTCC scripts/basic/fixdep
HOSTCC scripts/kconfig/conf.o
HOSTCC scripts/kconfig/confdata.o
HOSTCC scripts/kconfig/expr.o
LEX scripts/kconfig/lexer.lex.c
YACC scripts/kconfig/parser.tab.[ch]
HOSTCC scripts/kconfig/lexer.lex.o
HOSTCC scripts/kconfig/menu.o
HOSTCC scripts/kconfig/parser.tab.o
HOSTCC scripts/kconfig/preprocess.o
HOSTCC scripts/kconfig/symbol.o
HOSTCC scripts/kconfig/util.o
HOSTLD scripts/kconfig/conf
*** Default configuration is based on 'i386_defconfig'
#
# No change to .config
tangkeke@DESKTOP-I00KJCA: /os/experiment/lab0/linux-6.6-rc4$ make ARCH=i386 CROSS_COMPILE=i686-linux-gnu- arch/x86/en
try/syscall_32.i
SYSHDR arch/x86/include/generated/uapi/asm/unistd_32.h
SYSHDR arch/x86/include/generated/uapi/asm/unistd_64.h
SYSHDR arch/x86/include/generated/uapi/asm/unistd_x32.h
SYSTBL arch/x86/include/generated/asm/syscalls_32.h
HOSTCC arch/x86/tools/relocs_32.o
HOSTCC arch/x86/tools/relocs_64.o
```

生成并打开 arch/x86/entry/syscall_32.i 文件, 即可看到经过宏展开后的系统调用表内容, 如下图:

```
fig  M Makefile  C syscall_32.i 9+ X  C syscall_x32.c 8  C syscall_32.c 6  C syscall_64.c
entry > C syscall_32.i > ...
60610  __attribute__((__externally_visible__)) const sys_call_ptr_t sys_call_table[] = {
60611  # 1 "/arch/x86/include/generated/asm/syscalls_32.h" 1
60612  __ia32_sys_restart_syscall,
60613  __ia32_sys_exit,
60614  __ia32_sys_fork,
60615  __ia32_sys_read,
60616  __ia32_sys_write,
60617  __ia32_sys_open,
60618  __ia32_sys_close,
60619  __ia32_sys_waitpid,
60620  __ia32_sys_creat,
60621  __ia32_sys_link,
60622  __ia32_sys_unlink,
60623  __ia32_sys_execve,
60624  __ia32_sys_chdir,
60625  __ia32_sys_time32,
60626  __ia32_sys_mknod,
60627  __ia32_sys_chmod,
60628  __ia32_sys_lchown16,
60629  __ia32_sys_ni_syscall,
60630  __ia32_sys_stat,
60631  __ia32_sys_lseek,
60632  __ia32_sys_getpid,
60633  __ia32_sys_mount,
60634  __ia32_sys_oldumount,
60635  __ia32_sys_setuid16,
60636  __ia32_sys_getuid16,
60637  __ia32_sys_stime32,
60638  __ia32_sys_ptrace,
60639  __ia32_sys_alarm,
```

x86_64: (因为我的 WSL 本身就是 x86_64 架构, 所以只需普通的 gcc 工具链)

源码文件: arch/x86/entry/syscall_64.c, arch/x86/include/generated/asm/syscalls_64.h (需 make 生成) 等。

进入 arch/x86/entry 目录, 发现有文件 syscall_64.c, 如图:

```
tangkeke@DESKTOP-100KJCA: ~/os/experiment/lab0/linux-6.6-rc4/arch/x86/entry$ ls
Makefile  common.c  entry_32.S  entry_64_compat.S  syscall_64.c  syscalls  thunk_64.S  vsyscall
calling.h  entry.S    entry_64.S  syscall_32.c       syscall_x32.c  thunk_32.S  vdsos
```

查看该文件, 里面显示这是 x86_64 的系统调用表源文件:

```
~/.../linux-6.6-rc4  M Makefile ./  C syscall_x32.c  C syscall_32.c  C syscall_64.c 6 X
entry > C syscall_64.c > ...
1  // SPDX-License-Identifier: GPL-2.0
2  /* System call table for x86-64. */
```

文件里显示系统调用表有源自 asm/syscalls_64.h, 如下图:

```
asmlinkage const sys_call_ptr_t sys_call_table[] = {
#include <asm/syscalls_64.h>
};
```

在源代码根目录下依次执行以下命令, 效果如下图:

1. make ARCH=x86_64 defconfig
2. make ARCH=x86_64 arch/x86/entry/syscall_64.i

```
tangkeke@DESKTOP-100KJCA: ~/os/experiment/lab0/linux-6.6-rc4$ make ARCH=x86_64 defconfig
HOSTCC scripts/basic/fixdep
HOSTCC scripts/kconfig/conf.o
HOSTCC scripts/kconfig/confdata.o
HOSTCC scripts/kconfig/expr.o
LEX scripts/kconfig/lexer.lex.c
YACC scripts/kconfig/parser.tab.[ch]
HOSTCC scripts/kconfig/lexer.lex.o
HOSTCC scripts/kconfig/menu.o
HOSTCC scripts/kconfig/parser.tab.o
HOSTCC scripts/kconfig/preprocess.o
HOSTCC scripts/kconfig/symbol.o
HOSTCC scripts/kconfig/util.o
HOSTLD scripts/kconfig/conf
*** Default configuration is based on 'x86_64_defconfig'
#
# configuration written to .config
#
tangkeke@DESKTOP-100KJCA: ~/os/experiment/lab0/linux-6.6-rc4$ make ARCH=x86_64 arch/x86/entry/syscall_64.i
SYNC include/config/auto.conf.cmd
GEN arch/x86/include/generated/asm/orc_hash.h
SYSHDR arch/x86/include/generated/uapi/asm/unistd_32.h
SYSHDR arch/x86/include/generated/uapi/asm/unistd_64.h
SYSHDR arch/x86/include/generated/uapi/asm/unistd_x32.h
SYSTBL arch/x86/include/generated/asm/syscalls_32.h
SYSHDR arch/x86/include/generated/asm/unistd_32_ia32.h
```

生成并打开 arch/x86/entry/syscall_64.i 文件, 即可看到经过宏展开后的系统调用表内容, 如下图:

```
entry > C syscall_64.i > ...
61316 const sys_call_ptr_t sys_call_table[] = {
61317 # 1 "arch/x86/include/generated/asm/syscalls_64.h" 1
61318 __x64_sys_read,
61319 __x64_sys_write,
61320 __x64_sys_open,
61321 __x64_sys_close,
61322 __x64_sys_newstat,
61323 __x64_sys_newfstat,
61324 __x64_sys_newlstat,
61325 __x64_sys_poll,
61326 __x64_sys_lseek,
61327 __x64_sys_mmap,
61328 __x64_sys_mprotect,
61329 __x64_sys_munmap,
61330 __x64_sys_brk,
61331 __x64_sys_rt_sigaction,
61332 __x64_sys_rt_sigprocmask,
61333 __x64_sys_rt_sigreturn,
61334 __x64_sys_ioctl,
61335 __x64_sys_pread64,
61336 __x64_sys_pwrite64,
61337 __x64_sys_readv,
61338 __x64_sys_writev,
61339 __x64_sys_access,
61340 __x64_sys_pipe,
61341 __x64_sys_select,
61342 __x64_sys_sched_yield,
61343 __x64_sys_mremap,
61344 __x64_sys_msync,
```

7. ELF (Executable and Linkable Format, 可执行与可链接格式) 文件是在 Linux 下经 gcc 编译后的目标文件, 有特定的格式, 没有特定的拓展名。主要分以下三种: 可执行文件 (以.out 为后缀)、可重定位文件 (以.o 为后缀) 和共享目标文件 (以.so 为后缀)。

使用 readelf 命令:

```
tangkeke@DESKTOP-I00KJCA:~/os/experiment/lab0/linux-6.6-rc4$ readelf -h vmlinux.o
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:       REL (Relocatable file)
  Machine:    RISC-V
  Version:    0x1
  Entry point address: 0x0
  Start of program headers: 0 (bytes into file)
  Start of section headers: 54619712 (bytes into file)
  Flags:      0x1, RVC, soft-float ABI
  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: 183
  Section header string table index: 182
```

使用 objdump 命令:

```
tangkeke@DESKTOP-I00KJCA:~/os/experiment/lab0/linux-6.6-rc4/arch/riscv/kernel$ riscv64-linux-gnu-objdump
-S soc.o

soc.o:      file format elf64-littleriscv

Disassembly of section .init.text:

0000000000000000 <soc_early_init>:
0: 7179          addi    sp, sp, -48
2: f022          sd      s0, 32(sp)
4: ec26          sd      s1, 24(sp)
6: e44e          sd      s3, 8(sp)
8: f406          sd      ra, 40(sp)
a: e84a          sd      s2, 16(sp)
c: 1800          addi    s0, sp, 48
e: 00000797      auipc   a5, 0x0
12: 0007b783      ld      a5, 0(a5) # e <soc_early_init+0xe>
16: 0007b903      ld      s2, 0(a5)
1a: 00000497      auipc   s1, 0x0
1e: 0004b483      ld      s1, 0(s1) # 1a <soc_early_init+0x1a>
22: 00000997      auipc   s3, 0x0
26: 0009b983      ld      s3, 0(s3) # 22 <soc_early_init+0x22>
```

将以下 C 语言代码用 gcc 生成 ELF 文件 “hello”:

```
1. #include <stdio.h>
2.
3. int main()
4. {
5.     printf("hello world!\n");
6.     while(1);
7. }
```

运行该文件并结束后, 输入命令 “aux |grep hello” 查看最近一次运行 “hello” 的进程号 (PID)。结果如下图:

```
tangkeke@DESKTOP-I00KJCA:~/os$ ps aux |grep hello
tangkeke 2149 1.5 0.0 2772 936 pts/0 T 16:38 0:11 ./hello
tangkeke 2176 5.6 0.0 2776 940 pts/5 T 16:39 0:40 ./hello
tangkeke 2368 16.1 0.0 2776 944 pts/5 T 16:39 1:45 ./hello
tangkeke 2734 0.8 0.0 2772 936 pts/0 T 16:41 0:05 ./hello
tangkeke 2879 19.6 0.0 2772 936 pts/0 T 16:41 1:44 ./hello
tangkeke 3275 1.0 0.0 2772 952 pts/0 T 16:44 0:03 ./hello
tangkeke 3423 76.4 0.0 2776 960 pts/5 R+ 16:45 4:01 ./hello
tangkeke 3484 11.8 0.0 2772 936 pts/0 T 16:45 0:36 ./hello
tangkeke 3581 41.6 0.0 2772 916 pts/0 T 16:46 1:43 ./hello
tangkeke 4031 11.0 0.0 2772 980 pts/6 T 16:50 0:00 ./hello
tangkeke 4051 0.0 0.0 4024 2140 pts/6 S+ 16:50 0:00 grep --color=auto hello
```

第二列即进程号。倒数第二行的进程 (进程号为 4031) 为最近一次运行 “hello” 的进程。输入命令 “cat /proc/4031/maps”, 结果显示如下图:

```
tangkeke@DESKTOP-100KJCA: /os$ cat /proc/4031/maps
55fa69e48000-55fa69e49000 r-- 00000000 08:20 749978 /home/tangkeke/os/hello
55fa69e49000-55fa69e4a000 r-xp 00001000 08:20 749978 /home/tangkeke/os/hello
55fa69e4a000-55fa69e4b000 r--p 00002000 08:20 749978 /home/tangkeke/os/hello
55fa69e4b000-55fa69e4c000 r--p 00002000 08:20 749978 /home/tangkeke/os/hello
55fa69e4c000-55fa69e4d000 rw-p 00003000 08:20 749978 /home/tangkeke/os/hello
55fa6a9a8000-55fa6a9c9000 r-p 00000000 00:00 0 [heap]
7ff93f086000-7ff93f089000 r-w-p 00000000 00:00 0
7ff93f089000-7ff93f0b1000 r-p 00000000 08:20 36928 /usr/lib/x86_64-linux-gnu/libc.so.6
7ff93f0b1000-7ff93f246000 r-xp 00028000 08:20 36928 /usr/lib/x86_64-linux-gnu/libc.so.6
7ff93f246000-7ff93f29e000 r--p 001bd000 08:20 36928 /usr/lib/x86_64-linux-gnu/libc.so.6
7ff93f29e000-7ff93f2a2000 r-p 00214000 08:20 36928 /usr/lib/x86_64-linux-gnu/libc.so.6
7ff93f2a2000-7ff93f2a4000 rw-p 00218000 08:20 36928 /usr/lib/x86_64-linux-gnu/libc.so.6
7ff93f2a4000-7ff93f2b1000 r-p 00000000 00:00 0
7ff93f2ba000-7ff93f2bc000 r-p 00000000 00:00 0
7ff93f2bc000-7ff93f2be000 r--p 00000000 08:20 36925 /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7ff93f2be000-7ff93f2e8000 r-xp 00002000 08:20 36925 /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7ff93f2e8000-7ff93f2f3000 r--p 0002c000 08:20 36925 /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7ff93f2f4000-7ff93f2f6000 r--p 00037000 08:20 36925 /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7ff93f2f6000-7ff93f2f8000 rw-p 00039000 08:20 36925 /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7ffff5eb5000-7ffff5ed6000 r-p 00000000 00:00 0 [stack]
7ffff5fb0000-7ffff5fb4000 r-p 00000000 00:00 0 [vvar]
7ffff5fb4000-7ffff5fb6000 r-xp 00000000 00:00 0 [vdso]
```

8. 默认地，所有的中断、异常是交由机器模式处理的，但是，有些中断、异常是由特权模式或者用户模式的处理，这时机器模式要将中断、异常另外交由这些模式处理。为了提高效率，RISC-V 提供了一种异常中断委托机制，`mideleg` 为中断委托寄存器，其内的某一位的值为 1，则表明该位对应的中断不必交给机器模式，而是直接交给特权级更低的某个模式处理。每一位与中断的对应关系如下图。

Machine Interrupt Delegation Register			
CSR	mideleg		
Bits	Field Name	Attr.	Description
0	Reserved	WARL	
1	MSIP	RW	Delegate Supervisor Software Interrupt
[4:2]	Reserved	WARL	
5	MTIP	RW	Delegate Supervisor Timer Interrupt
[8:6]	Reserved	WARL	
9	MEIP	RW	Delegate Supervisor External Interrupt
[63:10]	Reserved	WARL	

Table 27: mideleg Register

在本题中，mideleg 的值表示 mideleg[1]、mideleg[5]、mideleg[9]这三位被置为 1，这分别代表监管者软件中断、监管者时钟中断、监管者外部中断可以直接交给对应的低特权级模式，即监管者模式，进行处理。

五、附录

无。