lab1

实验步骤

1. 编写head.S

首先，要初始化sp，stack就在 `.bss.stack` 代码段，所以我们使用la指令将sp设置为 `boot_stack_top` 的地址。之后初始化s0,并且存储ra的值，再用 `jal ra start_kernel` 跳转到 `main.c` 里的 `start_kernel` 。这样， `head.S` 就编写好了。

```
# lui sp, %hi(boot_stack_top)
# addi sp, sp, %lo(boot_stack_top)

# auipc sp, %hi(boot_stack_top)
# addi sp, sp, %lo(boot_stack_top)

# auipc sp, boot_stack_top
# addi sp, sp, boot_stack_top
la sp, boot_stack_top
add s0, sp, x0

addi sp, sp, -16           # store ra
sd ra, 0(sp)
addi s0, sp, 16            # change fp / s0
jal ra, start_kernel
ld ra, 0(sp)
addi sp, sp, 16
ret

.section .bss.stack
.globl boot_stack
```

可以看到，我使用的是la伪指令设置sp的值，由下图可以看到la指令实际上是auipc与addi两条指令融合的，但是之前我直接使用 `auipc sp, %hi(boot_start_top)` 与 `addi sp, sp, %lo(boot_start_top)` ，却报错如下图。看报错，原来是 `R_RISCV_HI20` 是32位的高20位，不能直接用在这里。

Pseudoinstruction	Base Instruction(s)	Meaning	
`lla rd, symbol`	`auipc rd, symbol[31:12]` `addi rd, rd, symbol[11:0]`	Load local address	取局部地址
`la rd, symbol`	*PIC*: `auipc rd, GOT[symbol][31:12]` `l{w|d} rd, rd, GOT[symbol][11:0]` *Non-PIC*: Same as `lla rd, symbol`	Load address	取地址

```
/home/drdg8/os22fall-stu/src/lab1/arch/riscv/kernel/head.S:13:(.text.entry+0x0): relocation truncated to fit: R_RISCV_HI20 ag
ainst symbol `boot_stack_top' defined in .bss.stack section in kernel/head.o
make[1]: *** [Makefile:3: all] Error 1
```

2. 完善Makefile

这个就比较简单了。模仿 `init/Makefile` 的写法写一个相似的即可。如下图。其中wildcard表示符合 `*.c` 的所有文件，sort排序，patsubst是将文件中符合第一个条件的部分替换成第二个条件的形式。 `.PHONY` 表示抽象的目标， `$<` 表示第一个传入的文件。这些都在Makefile教程中有，不加赘述。

```
C_SRC = $(sort $(wildcard *.c))
OBJ = $(patsubst %.c, %.o, $(C_SRC))


.PHONY: all clean

all: $(OBJ)

%.o: %.c
	${GCC} ${CFLAG} -c $<

clean:
	$(shell rm *.o 2>/dev/null)
```

3. 补充sbi.c

这里有个坑，如果你直接使用 `mv a0, %[arg0]\n` 与 `[arg0] r (arg0)` 这种形式，编译运行都是正确的，但是objdump之后你会发现寄存器是乱的，如下图。圈出来的地方a6与a7赋成一样的值了。编译器似乎并没有保护寄存器。

```
    __asm__ volatile(
        "mv a7, %[ext]\n"
        "mv a6, %[fid]\n"
        "mv a5, %[arg5]\n"
        "mv a4, %[arg4]\n"
        "mv a3, %[arg3]\n"
        "mv a2, %[arg2]\n"
        "mv a1, %[arg1]\n"
        "mv a0, %[arg0]\n"
        "ecall\n"
        "mv %[error], a0\n"
        "mv %[value], a1\n"
        : [value] "=r" (ret.value), [error] "=r" (ret.error)
        : [ext] "r" (ext), [fid] "r" (fid), [arg5] "r" (arg5), [arg4] "r" (arg4), [arg3] "r" (arg3), [arg2] "r" (arg2), [arg1] "r" (arg1), [arg0] "r" (arg0)
        : "memory"
    );
```

```
      alue;
      _ volatile(
      5c:    fbc42783              lw      a5,-68(s0)
80200060:    00078813              mv      a6,a5
80200064:    fb842783              lw      a5,-72(s0)
80200068:    00078893              mv      a7,a5
8020006c:    f8843783              ld      a5,-120(s0)
80200070:    f9043703              ld      a4,-112(s0)
80200074:    f9843683              ld      a3,-104(s0)
80200078:    fa043603              ld      a2,-96(s0)
8020007c:    fa843583              ld      a1,-88(s0)
80200080:    fb043503              ld      a0,-80(s0)
80200084:    00080893              mv      a7,a6
80200088:    00088813              mv      a6,a7
8020008c:    00078793              mv      a5,a5
80200090:    00070713              mv      a4,a4
80200094:    00068693              mv      a3,a3
80200098:    00060613              mv      a2,a2
8020009c:    00058593              mv      a1,a1
802000a0:    00050513              mv      a0,a0
802000a4:    00000073              ecall
802000a8:    00050513              mv      a0,a0
802000ac:    00050793              mv      a5,a0
802000b0:    00058713              mv      a4,a1
802000b4:    fee43423              sd      a4,-24(s0)
802000b8:    fef43023              sd      a5,-32(s0)
```

所以我们需要用 `ld a0, %[arg0]\n` 与 `[arg0] m (arg0)` 直接从内存取值，从而避开寄存器的使用，如下图，可以看到现在的值都是正常的了。

```
0x8020005c <sbi_ecall+52>     ld      a7,-52(s0)
0x80200060 <sbi_ecall+56>     ld      a6,-56(s0)
0x80200064 <sbi_ecall+60>     ld      a5,-104(s0)
0x80200068 <sbi_ecall+64>     ld      a4,-96(s0)
0x8020006c <sbi_ecall+68>     ld      a3,-88(s0)
0x80200070 <sbi_ecall+72>     ld      a2,-80(s0)
0x80200074 <sbi_ecall+76>     ld      a1,-72(s0)
0x80200078 <sbi_ecall+80>     ld      a0,-64(s0)
0x8020007c <sbi_ecall+84>     ecall
0x80200080 <sbi_ecall+88>     sd      a0,-48(s0)
0x80200084 <sbi_ecall+92>     sd      a1,-40(s0)
```

```
(gdb) i r a7
a7             0x1      1
(gdb) i r a6
a6             0x0      0
(gdb) i r a5
a5             0x0      0
(gdb) i r a0
a0             0x32     50
(gdb) i r a1
a1             0x0      0
(gdb) i r a2
a2             0x0      0
(gdb)
```

```
    __asm__ volatile(
        "lw a7, %[ext]\n"
        "lw a6, %[fid]\n"
        "ld a5, %[arg5]\n"
        "ld a4, %[arg4]\n"
        "ld a3, %[arg3]\n"
        "ld a2, %[arg2]\n"
        "ld a1, %[arg1]\n"
        "ld a0, %[arg0]\n"
        "ecall\n"
        "sd a0, %[error]\n"
        "sd a1, %[value]\n"
        : [value] "=m" (ret.value), [error] "=m" (ret.error)
        : [ext] "m" (ext), [fid] "m" (fid), [arg5] "m" (arg5), [arg4] "m" (arg4), [arg3] "m" (arg3), [arg2] "m" (arg2), [arg1] "m" (arg1), [arg0] "m" (arg0)
        : "memory"
    );
```

4. 完成 `puti()` 与 `puts()`

注意，这里的puti定义最好改成 `uint64` ,不然显示不出64位的寄存器。思路都很直接，就不解释了。

```c
void puts(char *s) {
    // unimplemented
    for(int i = 0; s[i] != '\0'; i++){
        sbi_ecall(0x1, 0x0, s[i], 0, 0, 0, 0, 0);
    }
}

void puti(uint64 x) {
    // unimplemented
    char s[64];
    int len;
    // reverse
    for (len = 0; x > 0; len++){
        s[len] = x % 10;
        x /= 10;
    }
    // print
    for (int i = len - 1; i >= 0; i--){
        sbi_ecall(0x1, 0x0, s[i] + '0' , 0, 0, 0, 0, 0);
    }
}
```

5. 修改 `defs.h`

这一步也是简单的，只要找到对应的伪指令，再和 `csr_write` 对比就可写出来。

csrr rd, csr　　　　x[rd] = CSRs[csr]

读控制状态寄存器 *(Control and Status Register Read)*. 伪指令(Pesudoinstruction), 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.

```c
#define csr_read(csr)                       \
({                                          \
    register uint64 __v;                    \
    /* unimplemented */                     \
    asm volatile ("csrr %0, " #csr          \
        : "=r" (__v) : : "memory" );        \
    __v;                                    \
})

#define csr_write(csr, val)                        \
({                                                 \
    uint64 __v = (uint64)(val);                    \
    asm volatile ("csrw " #csr ", %0"              \
                    : : "r" (__v)                  \
                    : "memory");                   \
})
```

6. 交叉编译工具链的安装与中间产物的获得

只要按照步骤来即可，不多赘述，由于思考题5需要生成 `arch/arm64/kernel/sys.i` ，只要把后面的 `path/to/file` 改成这个即可。

```
drdg8@LAPTOP-U2NC5HPH:~/os22fall-stu/src/lab1$ apt-cache search aarch64
binutils-aarch64-linux-gnu - GNU binary utilities, for aarch64-linux-gnu target
binutils-aarch64-linux-gnu-dbg - GNU binary utilities, for aarch64-linux-gnu target (debug symbols)
cpp-11-aarch64-linux-gnu - GNU C preprocessor
         64-linux-gnu - GNU C preprocessor (cpp) for the arm64 architecture
g++-11-aarch64-linux-gnu - GNU C++ compiler (cross compiler for arm64 architecture)
g++-aarch64-linux-gnu - GNU C++ compiler for the arm64 architecture
gcc-11-aarch64-linux-gnu - GNU C compiler (cross compiler for arm64 architecture)
gcc-11-aarch64-linux-gnu-base - GCC, the GNU Compiler Collection (base package)
gcc-aarch64-linux-gnu - GNU C compiler for the arm64 architecture
qemu-efi-aarch64 - UEFI firmware for 64-bit ARM virtual machines
qemu-system-arm - QEMU full system emulation binaries (arm)
        arch64-linux-gnu - GNU C preprocessor
        arch64-linux-gnu - GNU C preprocessor
        rch64-linux-gnu - GNU C preprocessor
g++-10-aarch64-linux-gnu - GNU C++ compiler (cross compiler for arm64 architecture)
g++-12-aarch64-linux-gnu - GNU C++ compiler (cross compiler for arm64 architecture)
g++-9-aarch64-linux-gnu - GNU C++ compiler (cross compiler for arm64 architecture)
gcc-10-aarch64-linux-gnu - GNU C compiler (cross compiler for arm64 architecture)
gcc-10-aarch64-linux-gnu-base - GCC, the GNU Compiler Collection (base package)
gcc-10-plugin-dev-aarch64-linux-gnu - Files for GNU GCC plugin development.
gcc-11-plugin-dev-aarch64-linux-gnu - Files for GNU GCC plugin development.
gcc-12-aarch64-linux-gnu - GNU C compiler (cross compiler for arm64 architecture)
gcc-12-aarch64-linux-gnu-base - GCC, the GNU Compiler Collection (base package)
gcc-12-plugin-dev-aarch64-linux-gnu - Files for GNU GCC plugin development.
gcc-9-aarch64-linux-gnu - GNU C compiler (cross compiler for arm64 architecture)
gcc-9-aarch64-linux-gnu-base - GCC, the GNU Compiler Collection (base package)
gcc-9-plugin-dev-aarch64-linux-gnu - Files for GNU GCC plugin development.
```

```
drdg8@LAPTOP-U2NC5HPH:~/os22fall-stu/linux-6.0-rc5$ make ARCH=arm64 CROSS_COMPIL
E=aarch64-linux-gnu- defconfig
*** Default configuration is based on 'defconfig'
#
# configuration written to .config
#
drdg8@LAPTOP-U2NC5HPH:~/os22fall-stu/linux-6.0-rc5$ make ARCH=arm64 CROSS_COMPIL
E=aarch64-linux-gnu- arch/arm64/kernel/sys.i
  HOSTCC  scripts/dtc/dtc.o
  HOSTCC  scripts/dtc/flattree.o
  HOSTCC  scripts/dtc/fstree.o
  HOSTCC  scripts/dtc/data.o
  HOSTCC  scripts/dtc/livetree.o
  HOSTCC  scripts/dtc/treesource.o
  HOSTCC  scripts/dtc/srcpos.o
  HOSTCC  scripts/dtc/checks.o
```

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?

就是把要传入的参数放到对应的寄存器里，a0-a7分别对应传入的7个参数，然后跳转到被调用的函数，一般用 `jal ra, Func1` 指令。

```
    sbi_ecall(0x1, 0x0, s[i], 0, 0, 0, 0, 0);
8020017c:    fec42783             lw      a5,-20(s0)
80200180:    fd843703             ld      a4,-40(s0)
80200184:    00f707b3             add     a5,a4,a5
80200188:    0007c783             lbu     a5,0(a5)
8020018c:    00078613             mv      a2,a5
80200190:    00000893             li      a7,0
80200194:    00000813             li      a6,0
80200198:    00000793             li      a5,0
8020019c:    00000713             li      a4,0
802001a0:    00000693             li      a3,0
802001a4:    00000593             li      a1,0
802001a8:    00100513             li      a0,1
802001ac:    e85ff0ef             jal     ra,80200030 <sbi_ecall>
```

对于 Caller / Callee Saved register，可以看下图：

Each register is one of two types:

- Caller saved
  - The callee function can use them freely (if needed, the caller had to save them before invoking and will restore them afterwards)
- Callee saved
  - The callee function must save them before modifying them, and restore them before returning (avoid using them at all, and no need to save)

caller save
call callee1
caller load

caller save
call callee2
caller load

根据定义，需要被caller saved 的就是caller saved register，有ra等。对于Callee Saved Register，也有sp,s0等。

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

```
0000000080200000 t $x
0000000080200028 t $x
0000000080200108 t $x
0000000080200148 t $x
0000000080200158 t $x
00000000802001e0 t $x
0000000080200000 A BASE_ADDR
0000000080202000 d _GLOBAL_OFFSET_TABLE_
0000000080204000 B _ebss
0000000080202000 D _edata
0000000080204000 B _ekernel
000000008020100f R _erodata
00000000802002d4 T _etext
0000000080203000 B _sbss
0000000080202000 D _sdata
0000000080200000 T _skernel
0000000080201000 R _srodata
0000000080200000 T _start
0000000080200000 T _stext
0000000080203000 B boot_stack
0000000080204000 B boot_stack_top
00000000802001e0 T puti
0000000080200158 T puts
0000000080200028 T sbi_ecall
0000000080200108 T start_kernel
0000000080200148 T test
```

可以看到，被定义的值都在System.map里列出来了。

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

我将 `main.c` 加入了csr_read部分，并改变了puti的参数为uint64,如下图。之后跑出来结果是一长串数字,也就是0x8000000000006000。

```c
int start_kernel() {
    puti(2022);
    puts(" Hello RISC-V\n");

    puti(csr_read(sstatus));
    puts(" sstatus \n");
```

```
2022 Hello RISC-V
9223372036854800384 sstatus
```

查看sstatus的定义，如下图，可以看到是 `SD = 1, FS = 11` .再看定义，FS处于dirty状态。FS代表浮点寄存器是dirty的，需要先写出原先寄存器的值,再将FS置为clean状态，从而可以向内写值。SD是快速校验位，如下图。其当FS或者XS为11时置1.

XLEN-1	XLEN-2 … 20	19	18	17
SD	0	MXR	SUM	0
1	XLEN-21	1	1	1

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

图 10.9：sstatus CSR。sstatus 是 mstatus（图 10.4）的一个子集，因此它们的布局类似。SIE 和 SPIE 中分别保存了当前的和异常发生之前的中断使能，类似于 mstatus 中的 MIE 和 MPIE。RV32 的 XLEN 为 32，RV64 为 40。（来自[Waterman and Asanovic 2017]中的图 4.2；有关其他域的说明请参见该文档的第 4.1 节。）

The FS[1:0] **WARL** field and the XS[1:0] read-only field are used to reduce the cost of context save and restore by setting and tracking the current state of the floating-point unit and any other user-mode extensions respectively. The FS field encodes the status of the floating-point unit, including the CSR `fcsr` and floating-point data registers `f0`–`f31`, while the XS field encodes the status of additional user-mode extensions and associated state. These fields can be checked by a context switch routine to quickly determine whether a state save or restore is required. If a save or restore is required, additional instructions and CSRs are typically required to effect and optimize the process.

The design anticipates that most context switches will not need to save/restore state in either or both of the floating-point unit or other extensions, so provides a fast check via the SD bit.

The FS and XS fields use the same status encoding as shown in Table 3.3, with the four possible status values being Off, Initial, Clean, and Dirty.

Status	FS Meaning	XS Meaning
0	Off	All off
1	Initial	None dirty or clean, some on
2	Clean	None dirty, some clean
3	Dirty	Some dirty

The SD bit is read-only and is set when either the FS or XS bits encode a Dirty state (i.e., SD=((FS==11) OR (XS==11))). This allows privileged code to quickly determine when no additional context save is required beyond the integer register set and PC.

其他的 `WPRI UXL MXR SUM SPP SPIE UPIE SIE UIE` 符号位都是0.其中 `WPRI` 是 `Write Preserve Read Ignore`， `SPIE SIE / UPIE PIE` 表明interrupt在User/Supervisor层面是否启用以及权限，在这里都是0，也就是没有启用interrupt。 `SPP` 表明interrupt的来源，如果是user mode则置0； `SUM` 表示 `permit Supervisor User Memory access` 也就是Supervisor mode是否能够连接User mode的page，置0表示不可以。 `MXR` 表示 `Make eXecutable Readable`，置0表示load的page是只读的。 `UXL` 表示寄存器长度。这些都能在 `riscv-priviliged` 中查到。

XLEN-1	XLEN-2 … 20	19	18	17
SD	0	MXR	SUM	0
1	XLEN-21	1	1	1

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

图 10.9：sstatus CSR。sstatus 是 mstatus（图 10.4）的一个子集，因此它们的布局类似。SIE 和 SPIE 中分别保存了当前的和异常发生之前的中断使能，类似于 mstatus 中的 MIE 和 MPIE。RV32 的 XLEN 为 32，RV64 为 40。（来自[Waterman and Asanovic 2017]中的图 4.2；有关其他域的说明请参见该文档的第 4.1 节。）

The FS[1:0] **WARL** field and the XS[1:0] read-only field are used to reduce the cost of context save and restore by setting and tracking the current state of the floating-point unit and any other user-mode extensions respectively. The FS field encodes the status of the floating-point unit, including the CSR `fcsr` and floating-point data registers `f0`–`f31`, while the XS field encodes the status of additional user-mode extensions and associated state. These fields can be checked by a context switch routine to quickly determine whether a state save or restore is required. If a save or restore is required, additional instructions and CSRs are typically required to effect and optimize the process.

The design anticipates that most context switches will not need to save/restore state in either or both of the floating-point unit or other extensions, so provides a fast check via the SD bit.

The FS and XS fields use the same status encoding as shown in Table 3.3, with the four possible status values being Off, Initial, Clean, and Dirty.

Status	FS Meaning	XS Meaning
0	Off	All off
1	Initial	None dirty or clean, some on
2	Clean	None dirty, some clean
3	Dirty	Some dirty

The SD bit is read-only and is set when either the FS or XS bits encode a Dirty state (i.e., SD=((FS==11) OR (XS==11))). This allows privileged code to quickly determine when no additional context save is required beyond the integer register set and PC.

4. 用 csr_write 宏向 sscratch 寄存器写入数据，并验证是否写入成功。

我们改变main.c,放入csr_write，如下图，得到结果，其中数字是0x8020000的十进制。

```c
puti(csr_read(sscratch));
puts("before change sscratch \n");

csr_write(sscratch, 0x80200000);
puti(csr_read(sscratch));
puts("\nafter change sscratch \n");

test(); // DO NOT DELETE !!!

return 0;
```

```
before change sscratch
2149580800
after change sscratch
```

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

请看6. 实验步骤，最后结果如下所示。

```
# 0 "arch/arm64/kernel/sys.c"
# 1 "/home/drdg8/os22fall-stu/linux-6.0-rc5//"
# 0 "<built-in>"
# 0 "<command-line>"
# 1 "./././include/linux/compiler-version.h" 1
# 0 "<command-line>" 2
# 1 "./././include/linux/kconfig.h" 1




# 1 "./include/generated/autoconf.h" 1
# 6 "./././include/linux/kconfig.h" 2
# 0 "<command-line>" 2
# 1 "./././include/linux/compiler_types.h" 1
# 75 "./././include/linux/compiler_types.h"
# 1 "./include/linux/compiler_attributes.h" 1
# 76 "./././include/linux/compiler_types.h" 2
# 95 "./././include/linux/compiler_types.h"
# 1 "./include/linux/compiler-gcc.h" 1
# 96 "./././include/linux/compiler_types.h" 2
# 109 "./././include/linux/compiler_types.h"
# 1 "./arch/arm64/include/asm/compiler.h" 1
# 110 "./././include/linux/compiler_types.h" 2


struct ftrace_branch_data {
arch/arm64/kernel/sys.i
```

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.

首先，在linux-cross-reference中搜索sys_call_table，可以看到sys_call_table都在哪里定义的。

Defined in 12 files as a variable:

arch/arc/kernel/sys.c, line 15 *(as a variable)*
arch/arm64/kernel/sys.c, line 58 *(as a variable)*
arch/csky/kernel/syscall_table.c, line 11 *(as a variable)*
arch/hexagon/kernel/syscalltab.c, line 17 *(as a variable)*
arch/loongarch/kernel/syscall.c, line 32 *(as a variable)*
arch/nios2/kernel/syscall_table.c, line 15 *(as a variable)*
arch/openrisc/kernel/sys_call_table.c, line 22 *(as a variable)*
arch/riscv/kernel/syscall_table.c, line 15 *(as a variable)*
arch/x86/entry/syscall_64.c, line 16 *(as a variable)*
arch/x86/um/sys_call_table_32.c, line 34 *(as a variable)*
arch/x86/um/sys_call_table_64.c, line 29 *(as a variable)*
arch/xtensa/kernel/syscall.c, line 30 *(as a variable)*

对于arm，搜索之后发现应该下载 `gcc-arm-linux-gnueabi` ,之后 `make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- defconfig` ,

```
drdg8@LAPTOP-U2NC5HPH:~/os22fall-stu/linux-6.0-rc5$ sudo apt install gcc-arm-lin
ux-gnueabi
[sudo] password for drdg8:
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following additional packages will be installed:
```

```
drdg8@LAPTOP-U2NC5HPH:~/os22fall-stu/linux-6.0-rc5$ make ARCH=arm CROSS_COMPILE=
arm-linux-gnueabi- 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
```

直接搜没有搜到sys_call_table,我们看 `entry-common.S` 有提及。于是生成看看。

```
arch/arm/kernel/entry-common.S
├─ line 246
├─ line 366
└─ line 372
```

```
drdg8@LAPTOP-U2NC5HPH:~/os22fall-stu/linux-6.0-rc5$ make ARCH=arm CROSS_COMPILE=
arm-linux-gnueabi- 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
  HOSTCC  scripts/dtc/dtc.o
  HOSTCC  scripts/dtc/flattree.o
```

在文档中搜索，果然找到了对应的sys_call_table。

```
# 366 "arch/arm/kernel/entry-common.S"
 syscall_table_start sys_call_table

# 1 "./arch/arm/include/generated/calls-eabi.S" 1
syscall 0, sys_restart_syscall
syscall 1, sys_exit
syscall 2, sys_fork
syscall 3, sys_read
syscall 4, sys_write
syscall 5, sys_open
syscall 6, sys_close
syscall 7, sys_ni_syscall
syscall 8, sys_creat
syscall 9, sys_link
syscall 10, sys_unlink
syscall 11, sys_execve
syscall 12, sys_chdir
syscall 13, sys_ni_syscall
syscall 14, sys_mknod
syscall 15, sys_chmod
syscall 16, sys_lchown16
syscall 17, sys_ni_syscall
arch/arm/kernel/entry-common.s
```

对于riscv(32 bit)，得从网站上先下载工具链 `gcc-risc32-linux-gnu` ,由于从源码编译 `.gitmodule` 子模块下不下来，我直接使用了ubuntu2020版的工具链，似乎基本功能也可以用。

```
drdg8@LAPTOP-U2NC5HPH:~/riscv/bin$ ll
total 637652
drwxr-xr-x 2 drdg8 drdg8      4096 Sep 30 12:09 ./
drwxr-xr-x 9 drdg8 drdg8      4096 Sep 30 11:46 ../
-rwxr-xr-x 1 drdg8 drdg8   6150464 Sep 30 11:29 riscv32-unknown-linux-gnu-addr2line*
-rwxr-xr-x 2 drdg8 drdg8   6409448 Sep 30 11:29 riscv32-unknown-linux-gnu-ar*
-rwxr-xr-x 2 drdg8 drdg8   8990880 Sep 30 11:29 riscv32-unknown-linux-gnu-as*
-rwxr-xr-x 2 drdg8 drdg8   8065120 Sep 30 12:09 riscv32-unknown-linux-gnu-c++*
-rwxr-xr-x 1 drdg8 drdg8   6090336 Sep 30 11:29 riscv32-unknown-linux-gnu-c++filt*
-rwxr-xr-x 1 drdg8 drdg8   8060976 Sep 30 12:09 riscv32-unknown-linux-gnu-cpp*
-rwxr-xr-x 1 drdg8 drdg8    306952 Sep 30 11:29 riscv32-unknown-linux-gnu-elfedit*
-rwxr-xr-x 2 drdg8 drdg8   8065120 Sep 30 12:09 riscv32-unknown-linux-gnu-g++*
-rwxr-xr-x 2 drdg8 drdg8   8057232 Sep 30 12:09 riscv32-unknown-linux-gnu-gcc*
-rwxr-xr-x 2 drdg8 drdg8   8057232 Sep 30 12:09 riscv32-unknown-linux-gnu-gcc-12.2.0*
-rwxr-xr-x 1 drdg8 drdg8    192896 Sep 30 12:09 riscv32-unknown-linux-gnu-gcc-ar*
-rwxr-xr-x 1 drdg8 drdg8    192760 Sep 30 12:09 riscv32-unknown-linux-gnu-gcc-nm*
```

之后

```
make ARCH=riscv CROSS_COMPILE=/home/drdg8/riscv/bin/riscv32-unknown-linux-gnu- defconfig
```

再

```
make ARCH=riscv CROSS_COMPILE=/home/drdg8/riscv/bin/riscv32-unknown-linux-gnu- arch/riscv/kernel/syscall_table.i
```

这样，就能得到其sys_call_table。

```
drdg8@LAPTOP-U2NC5HPH:~/os22fall-stu/linux-6.0-rc5$ make ARCH=riscv CROSS_COMPILE=/home/drdg8/riscv/bin/riscv32-unknown-linux-gnu- defconfig
drdg8@LAPTOP-U2NC5HPH:~/os22fall-stu/linux-6.0-rc5$ make ARCH=riscv CROSS_COMPILE=/home/drdg8/riscv/bin/riscv32-unknown-linux-gnu- arch/riscv/kernel/syscall_table.i
```

```
void * const sys_call_table[451] = {
 [0 ... 451 - 1] = sys_ni_syscall,
# 1 "./arch/riscv/include/asm/unistd.h" 1
# 24 "./arch/riscv/include/asm/unistd.h"
# 1 "./arch/riscv/include/uapi/asm/unistd.h" 1
# 26 "./arch/riscv/include/uapi/asm/unistd.h"
# 1 "./include/uapi/asm-generic/unistd.h" 1
# 34 "./include/uapi/asm-generic/unistd.h"
[0] = (sys_io_setup),
[1] = (sys_io_destroy),
[2] = (sys_io_submit),
[3] = (sys_io_cancel),
[4] = (sys_io_getevents),
[5] = (sys_setxattr),
[6] = (sys_lsetxattr),
[7] = (sys_fsetxattr),
arch/riscv/kernel/syscall_table.i
```

对于riscv(64 bit)，实际上和riscv32一样。先

```
make ARCH=riscv CROSS_COMPILE=riscv64-linux-gnu- defconfig
```

再

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

就能得到想要的结果。

```
drdg8@LAPTOP-U2NC5HPH:~/os22fall-stu/linux-6.0-rc5$ make ARCH=riscv CROSS_COMPILE=riscv64-linux-gnu- defconfig
*** Default configuration is based on 'defconfig'
#
# configuration written to .config
#
drdg8@LAPTOP-U2NC5HPH:~/os22fall-stu/linux-6.0-rc5$ make ARCH=riscv CROSS_COMPILE=riscv64-linux-gnu- arch/riscv/kernel/syscall_table.i
  HOSTCC  scripts/dtc/dtc.o
  HOSTCC  scripts/dtc/flattree.o
```

```
void * const sys_call_table[451] = {
 [0 ... 451 - 1] = sys_ni_syscall,
# 1 "./arch/riscv/include/asm/unistd.h" 1
# 24 "./arch/riscv/include/asm/unistd.h"
# 1 "./arch/riscv/include/uapi/asm/unistd.h" 1
# 26 "./arch/riscv/include/uapi/asm/unistd.h"
# 1 "./include/uapi/asm-generic/unistd.h" 1
# 34 "./include/uapi/asm-generic/unistd.h"
[0] = (sys_io_setup),

[1] = (sys_io_destroy),

[2] = (sys_io_submit),

[3] = (sys_io_cancel),


[4] = (sys_io_getevents),




[5] = (sys_setxattr),

[6] = (sys_lsetxattr),

[7] = (sys_fsetxattr),
arch/riscv/kernel/syscall_table.i
```

对于x86(64 bit)，由于ubuntu使用的就是x86_64的架构，所以不需要交叉编译，直接编就行。先

```
make ARCH=x86_64 defconfig
```

再

```
make ARCH=x86_64 arch/x86/entry/syscall_64.i
```

```
drdg8@LAPTOP-U2NC5HPH:~/os22fall-stu/linux-6.0-rc5$ uname -a
Linux LAPTOP-U2NC5HPH 5.10.16.3-microsoft-standard-WSL2 #1 SMP Fri Apr 2 22:23:49 UTC 2021 x86_64 x86_64 x86_64 GNU/Linux
```

```
drdg8@LAPTOP-U2NC5HPH:~/os22fall-stu/linux-6.0-rc5$ 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
```

```
drdg8@LAPTOP-U2NC5HPH:~/os22fall-stu/linux-6.0-rc5$ make ARCH=x86_64 arch/x86/entry/sys
call_64.i
  CALL    scripts/checksyscalls.sh
  CALL    scripts/atomic/check-atomics.sh
  DESCEND objtool
  CPP     arch/x86/entry/syscall_64.i
```

```
 const sys_call_ptr_t sys_call_table[] = {
# 1 "./arch/x86/include/generated/asm/syscalls_64.h" 1
__x64_sys_read,
__x64_sys_write,
__x64_sys_open,
__x64_sys_close,
__x64_sys_newstat,
__x64_sys_newfstat,
__x64_sys_newlstat,
__x64_sys_poll,
__x64_sys_lseek,
__x64_sys_mmap,
__x64_sys_mprotect,
__x64_sys_munmap,
__x64_sys_brk,
__x64_sys_rt_sigaction,
__x64_sys_rt_sigprocmask,
__x64_sys_rt_sigreturn,
__x64_sys_ioctl,
__x64_sys_pread64,
__x64_sys_pwrite64,
__x64_sys_readv,
__x64_sys_writev,
```

对于x86(32 bit)，由于x86_64的编译器也能编译x86_32的文件，所以不用下载工具链。先

```
make ARCH=i386 defconfig
```

再

```
make ARCH=i386 arch/x86/entry/syscall_32.i
```

就能得到想要的结果。

```
drdg8@LAPTOP-U2NC5HPH:~/os22fall-stu/linux-6.0-rc5$ make ARCH=i386 defconfig
*** Default configuration is based on 'i386_defconfig'
#
# configuration written to .config
```

```
drdg8@LAPTOP-U2NC5HPH:~/os22fall-stu/linux-6.0-rc5$ make ARCH=i386 arch/x86/entry/syscall_32.i
  CALL    scripts/checksyscalls.sh
  CALL    scripts/atomic/check-atomics.sh
```

```
__attribute__((__externally_visible__)) const sys_call_ptr_t sys_call_table[] = {
# 1 "./arch/x86/include/generated/asm/syscalls_32.h" 1
__ia32_sys_restart_syscall,
__ia32_sys_exit,
__ia32_sys_fork,
__ia32_sys_read,
__ia32_sys_write,
__ia32_sys_open,
__ia32_sys_close,
__ia32_sys_waitpid,
__ia32_sys_creat,
__ia32_sys_link,
__ia32_sys_unlink,
__ia32_sys_execve,
__ia32_sys_chdir,
__ia32_sys_time32,
__ia32_sys_mknod,
__ia32_sys_chmod,
/x86/entry/syscall_32.i
```

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.

ELF(Executable and Linkable Format)，是一种可执行可连接的文件格式。
首先编写一个lab1.c文件，如下图。再用gcc编译，查看lab1的格式，就是ELF。

```c
#include <stdio.h>

int main(){
    printf("hello world!\n");
    while(1);
}
```

```
drdg8@LAPTOP-U2NC5HPH:~/os22fall-stu/draft$ vim lab1.c
drdg8@LAPTOP-U2NC5HPH:~/os22fall-stu/draft$ gcc -g lab1.c -o lab1
drdg8@LAPTOP-U2NC5HPH:~/os22fall-stu/draft$ /lab1
-bash: /lab1: No such file or directory
drdg8@LAPTOP-U2NC5HPH:~/os22fall-stu/draft$ ./lab1
hello world!
^C
drdg8@LAPTOP-U2NC5HPH:~/os22fall-stu/draft$ file lab1
lab1: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=a523848078eb82bc0e5f98d73bab8a768cea4407, for GNU/Linux 3.2.0, with debug_info, not stripped
```

下图是objdump与readelf的结果。

```
0000000000001149 <main>:
#include <stdio.h>

int main(){
    1149:       f3 0f 1e fa             endbr64
    114d:       55                      push   %rbp
    114e:       48 89 e5                mov    %rsp,%rbp
    printf("hello world!\n");
    1151:       48 8d 05 ac 0e 00 00    lea    0xeac(%rip),%rax        # 2004 <_IO_stdi
n_used+0x4>
    1158:       48 89 c7                mov    %rax,%rdi
    115b:       e8 f0 fe ff ff          call   1050 <puts@plt>
    while(1);
    1160:       eb fe                   jmp    1160 <main+0x17>
```

```
drdg8@LAPTOP-U2NC5HPH:~/os22fall-stu/draft$ readelf -a lab1
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:                              DYN (Position-Independent Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x1060
  Start of program headers:          64 (bytes into file)
  Start of section headers:          14640 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
  Number of section headers:         37
  Section header string table index: 36
```

之后将lab1运行并挂起，查看memory，可以看到，就像老师上课讲的一样，有 `.data` `.rodata` `.text` `.stack` 等段，由于是动态链接，中间还有一些动态库。

```
drdg8@LAPTOP-U2NC5HPH:~/os22fall-stu/draft$ ./lab1
hello world!
^Z
[1]+  Stopped                 ./lab1
drdg8@LAPTOP-U2NC5HPH:~/os22fall-stu/draft$ pgrep lab1
25286
drdg8@LAPTOP-U2NC5HPH:~/os22fall-stu/draft$ cat /proc/25286/maps
5575a2036000-5575a2037000 r--p 00000000 08:10 246486                     /home/drdg8/os22fall-stu/draft/lab1
5575a2037000-5575a2038000 r-xp 00001000 08:10 246486                     /home/drdg8/os22fall-stu/draft/lab1
5575a2038000-5575a2039000 r--p 00002000 08:10 246486                     /home/drdg8/os22fall-stu/draft/lab1
5575a2039000-5575a203a000 r--p 00002000 08:10 246486                     /home/drdg8/os22fall-stu/draft/lab1
5575a203a000-5575a203b000 rw-p 00003000 08:10 246486                     /home/drdg8/os22fall-stu/draft/lab1
5575a30a1000-5575a30c2000 rw-p 00000000 00:00 0                          [heap]
7f44dbad4000-7f44dbad7000 rw-p 00000000 00:00 0
7f44dbad7000-7f44dbaff000 r--p 00000000 08:10 17159                      /usr/lib/x86_64-linux-gnu/libc.so.6
7f44dbaff000-7f44dbc94000 r-xp 00028000 08:10 17159                      /usr/lib/x86_64-linux-gnu/libc.so.6
7f44dbc94000-7f44dbcec000 r--p 001bd000 08:10 17159                      /usr/lib/x86_64-linux-gnu/libc.so.6
7f44dbcec000-7f44dbcf0000 r--p 00214000 08:10 17159                      /usr/lib/x86_64-linux-gnu/libc.so.6
7f44dbcf0000-7f44dbcf2000 rw-p 00218000 08:10 17159                      /usr/lib/x86_64-linux-gnu/libc.so.6
7f44dbcf2000-7f44dbcff000 rw-p 00000000 00:00 0
7f44dbd0b000-7f44dbd0d000 rw-p 00000000 00:00 0
7f44dbd0d000-7f44dbd0f000 r--p 00000000 08:10 17061                      /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f44dbd0f000-7f44dbd39000 r-xp 00002000 08:10 17061                      /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f44dbd39000-7f44dbd44000 r--p 0002c000 08:10 17061                      /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f44dbd45000-7f44dbd47000 r--p 00037000 08:10 17061                      /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f44dbd47000-7f44dbd49000 rw-p 00039000 08:10 17061                      /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7ffc7a36d000-7ffc7a38e000 rw-p 00000000 00:00 0                          [stack]
7ffc7a39a000-7ffc7a39e000 r--p 00000000 00:00 0                          [vvar]
7ffc7a39e000-7ffc7a39f000 r-xp 00000000 00:00 0                          [vdso]
drdg8@LAPTOP-U2NC5HPH:~/os22fall-stu/draft$
```