Report of Lab2

1. 环境搭建

1.1 建立映射

用lab1的方法建立本地目录与docker image内实验目录的映射

```
wang@wang-virtual-machine:~$ cd Documents/OS/lab/lab2
wang@wang-virtual-machine:~/Documents/OS/lab/lab2$ docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
oslab               2020                37619cb29f3f        4 weeks ago         2.89GB
hello-world         latest              bf756fb1ae65        10 months ago       13.3kB
wang@wang-virtual-machine:~/Documents/OS/lab/lab2$ docker run -it -v `pwd`:/home/oslab/lab2 -u oslab -w /home/oslab 3761 /bin/bash
oslab@b8d1b95e2b4c:~$
```

1.2 组织文件结构

组织文件结构如下:

```
lab2
├── arch
│   └── riscv
│       ├── include
│       │   └── put.h
│       ├── kernel
│       │   ├── entry.S
│       │   ├── head.S
│       │   ├── Makefile
│       │   ├── strap.c
│       │   └── vmlinux.lds
│       └── Makefile
├── include
│   ├── put.h
│   └── test.h
├── init
│   ├── main.c
│   ├── Makefile
│   └── test.c
├── lib
│   ├── Makefile
│   └── put.c
└── Makefile
```

1.3 文件必要修改

1. 修改head.S中的`.text`命名为`.text.init`
2. 修改entry.S中的`.text`命名为`.text.entry`
3. 修改lds文件中的`.text`展开方式

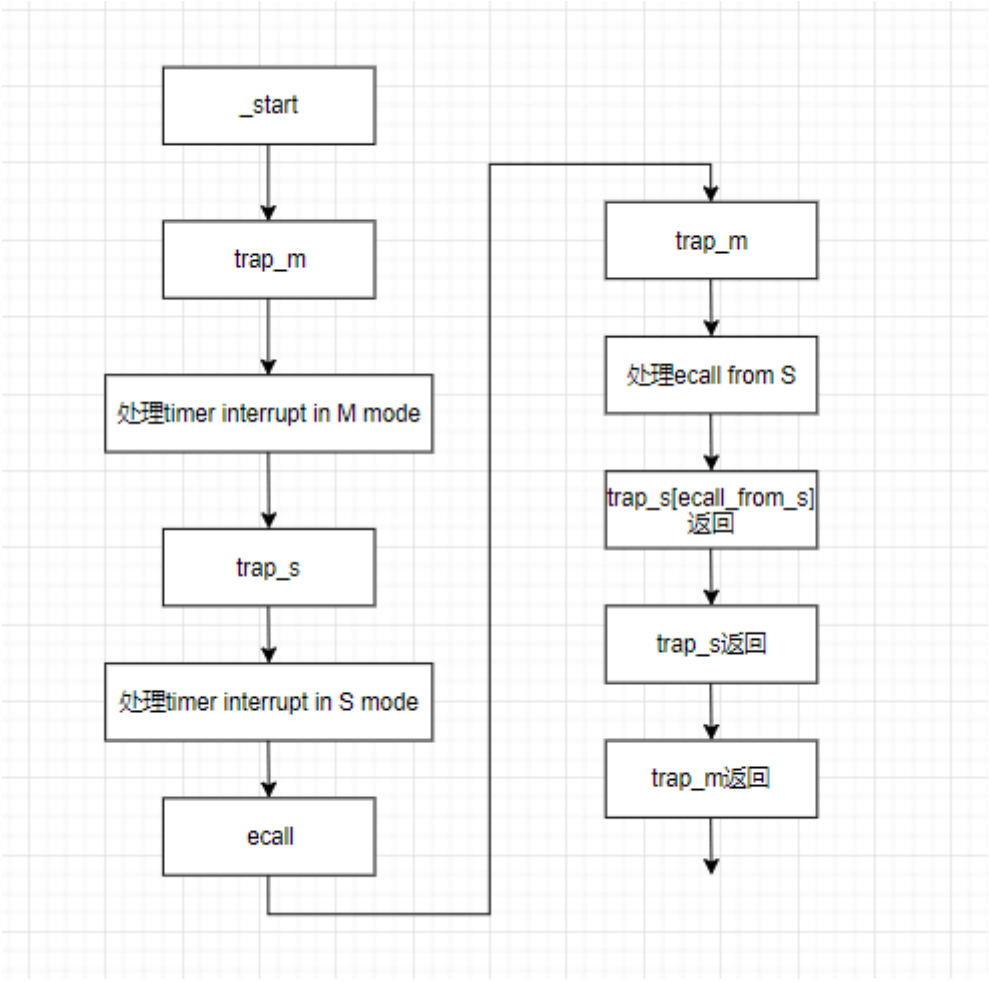
```
<<<<<<< before
.text : { *(.text) }
===============
.text : {
        *(.text.init)
        *(.text.entry)
        *(.text)
}
>>>>>>> after
```

4. 修改lds文件中的bss段，前后加上标志符`bss_start`与`bss_end`，方便后续对bss段进行初始化

```
<<<<<<< before
.bss : { *(.bss) }
===============
bss_start =.;
.bss : { *(.bss) }
bss_end =.;
>>>>>>> after
```

2. 实验流程说明

1. 整个程序从head.S中的`_start`开始。在`_start`中，首先对[m|s]status及[m|s]ie寄存器进行赋值，使得时钟中断能够在两个模式下触发
2. 当mtimecmp小于mtime时，触发M mode下的中断，进入`trap_m`，判断是否为时钟中断，若是则进入处理timer interrupt的函数
    - 上下文切换（保存所有寄存器及必要的CSRs）
    - 清除`mie[mtie]`，避免之后的S mode处理时钟中断同时继续触发时钟中断
    - 设置`mip[stip]`，为之后触发S mode下的时钟中断做准备
    - 当`trap_m`内时钟中断处理结束，此时`sstatus[tie]=1`，`sie[stie]=1`且`sip[stip]=1`，于是触发S mode的时钟中断，stvec自动保存PC，跳转到`trap_s`函数（委托）
3. 在`trap_s`内分析异常原因，判断为时钟中断，于是进行以下操作
    - 上下文切换
    - 异常处理（输出已经产生的时钟中断个数）
    - 触发ecall，跳转到M mode异常处理函数`trap_m`
4. `trap_m`分析异常原因，发现为`ecall from S-mode`，于是进入对应处理函数
    - 设置 mtimecmp += 100000，此时设置mtimecmp硬件同时会清除`mtime[mtip]`
    - 设置`mie[mtie]`，恢复M mode的时钟中断使能，保证下一次时钟中断可以触发
5. 函数逐级返回，整个委托的时钟中断处理完毕



3. head.S模式切换前添加功能

3.1 初始化bss段

利用在lds中定义的标识符`bss_start`与`bss_end`来进行遍历，将该段全部清零

```
    # initialize bss segment
    la t0, bss_start
    la t1, bss_end
1:
    sd x0, (t0)
    add t0, t0, 8
    blt t0, t1, 1b
```

3.2 初始化mtimecmp寄存器

将mtimecmp寄存器值初始化为mtime+1e6。其中mtime对应的映射地址为0x200bff8，mtimecmp对应的映射地址为0x2004000

```
    # set mtimecmp = mtime + 1000000
    li t0, 0x200bff8    # t0 = addr[mtime]
    li t1, 0x2004000    # t1 = addr[mtimecmp]
    ld t2, (t0)         # t2 = val[mtime]
    addi t2, t2, 1000000
    sd t2, (t1)         # val[mtimecmp] = val[mtime] + 1000000
```

3.3 设置时钟中断委托

- 设置`mideleg`的第5位来使能S模式的时钟中断委托，将其置位后，S mode产生的时钟中断会委托给S mode处理（仍需后续手动设置触发S mode下的时钟中断）
- 设置`mstatus[mie], sstatus[sie], mie[mtie], sie[stie]`来打开时钟中断使能

```
    # set mideleg[5]
    li t0, 32           # t0 = 0010 0000
    csrs mideleg, t0    # set mideleg[5] = 1

    # write mstatus reg
    li t0, 0x080A       # set mstatus.mie = 1 && mstatus.sie = 1 &&
mstatus.mpp = 01
    csrw mstatus, t0

    # set sstatus[sie]
    li t0, 0x02
    csrs sstatus, t0    # set sstatus[sie] = 1

    # set mie[mtie]     mie.MTIE(bit 7)
    li t0,0x80
    csrs mie, t0        # set mie[mtie] = 1

    # set sie[stie]
    li t0, 32
    csrs sie, t0        # set sie[stie] = 1
```

4. 编写machine mode的异常处理代码

4.1 上下文切换

利用sp开辟一段内存空间空间，并保存所有寄存器以及必要的CSRs的值。需要注意的是，因为寄存器x2为sp，x3为gp，x4为tp，因此这三个寄存器无需保存。在mret前也要恢复寄存器及CSRs内的值，并恢复sp。

需要注意的是，CSR寄存器不能直接存入栈中，需要用临时寄存器进行中转。

具体上下文切换代码如下：

```
# define WORD_SIZE 8
# define CONTEXT_SIZE (30 * WORD_SIZE)

    # save all the registers
    addi sp, sp, -CONTEXT_SIZE
    sd x1, 0 * WORD_SIZE(sp)
    sd x5, 1 * WORD_SIZE(sp)
    sd x6, 2 * WORD_SIZE(sp)
    sd x7, 3 * WORD_SIZE(sp)
    sd x8, 4 * WORD_SIZE(sp)
    sd x9, 5 * WORD_SIZE(sp)
    sd x10, 6 * WORD_SIZE(sp)
    sd x11, 7 * WORD_SIZE(sp)
    sd x12, 8 * WORD_SIZE(sp)
    sd x13, 9 * WORD_SIZE(sp)
    sd x14, 10 * WORD_SIZE(sp)
```

```
    sd x15, 11 * WORD_SIZE(sp)
    sd x16, 12 * WORD_SIZE(sp)
    sd x17, 13 * WORD_SIZE(sp)
    sd x18, 14 * WORD_SIZE(sp)
    sd x19, 15 * WORD_SIZE(sp)
    sd x20, 16 * WORD_SIZE(sp)
    sd x21, 17 * WORD_SIZE(sp)
    sd x22, 18 * WORD_SIZE(sp)
    sd x23, 19 * WORD_SIZE(sp)
    sd x24, 20 * WORD_SIZE(sp)
    sd x25, 21 * WORD_SIZE(sp)
    sd x26, 22 * WORD_SIZE(sp)
    sd x27, 23 * WORD_SIZE(sp)
    sd x28, 24 * WORD_SIZE(sp)
    sd x29, 25 * WORD_SIZE(sp)
    sd x30, 26 * WORD_SIZE(sp)
    sd x31, 27 * WORD_SIZE(sp)

    # save the needed CSRs
    csrr t0, mstatus
    sd t0, 28 * WORD_SIZE(sp)
    csrr t0, mtvec
    sd t0, 29 * WORD_SIZE(sp)
    csrr t0, mepc
    sd t0, 30 * WORD_SIZE(sp)
```

4.2 编写处理代码

4.2.1 时钟中断处理

- 在处理时钟中断前需要对异常类型进行判断，最终确认为时钟中断。判断过程如下：
  - 读取`mcause`寄存器，根据正负性判断异常类型为interrupt还是exception(`mcause`首位为1则是interrupt)
  - 若是interrupt，则判断exception code是否为7（Machine timer interrupt）
  - 若是则进入处理时钟中断部分

```
    csrr t0, mcause         # read trap cause
    bgez t0, m_exception    # determine interrupt(<0) or exception(>=0), branch if exception
    andi t0, t0, 0x3f       # isolate the exception code field
    li t1, 7
    bne t0, t1, m_other_int # branch if not a timer interrupt
```

- 处理时钟中断时，要完成以下功能
  - disable mie[mtie]：禁用M mode下的时钟中断，避免之后S mode处理时钟中断的同时继续触发M mode 的时钟中断
- enable sip[stip]：设置S mode时钟中断的pending位，为之后触发S mode下的时钟中断做准备

  注意： sip[stip]是只读的，因此我们只能通过修改mip的stip位来间接修改sip[stip]

```
    # disable mie[mtie] (bit 7)
    li t0, 0x080
    csrc mie t0

    # enable mip[stip] (bit 5)
    li t0, 0x20
    csrs mip, t0
```

4.2.2 ecall from S-mode处理

处理ecall from S-mode时，需要完成以下功能

- 设置 mtimecmp += 100000，此时设置mtimecmp硬件同时会清除`mtime[mtip]`
- 设置`mie[mtie]`，恢复M mode的时钟中断使能，保证下一次时钟中断可以触发

```
.globl ecall_from_s
.align 3
ecall_from_s:                   # handle ecall from s mode
    # set mtimecmp += 100000
    li t0, 0x2004000            # t0 = addr[mtimecmp]
    ld t1, (t0)
    addi t1, t1, 1000000
    sd t1, (t0)

    # enable mie[mtie] (bit 7)
    li t0, 0x080
    csrs mie t0

    # modify the mepc in the stack (mpec+=4)
    ld t0, 30 * WORD_SIZE(sp)
    addi t0, t0, 4
    sd t0, 30 * WORD_SIZE(sp)

    j trap_m_ret
```

需要注意的是，因为ecall是同步异常，因此需要在退出前将之前栈中保存的`mepc`值加4

5. 编写Supervisor mode的异常处理代码

将head.S中的trap_s函数移动到entry.S中，并首先保存所有寄存器及必要的CSRs的值

5.1 上下文切换

与trap_m内上下文切换相似，只是将保存的CSRs修改为S mode对应的CSRs，在此不再赘述。

5.2 异常处理

对异常进行处理，在时钟中断的处理函数内，用一个内存变量`COUNT`记录产生的中断个数，当`COUNT=1e5`时，将`COUNT`清零，并利用`call print_message`调用strap.c内的`print_message()`函数。在strap.c内我用一个全局变量`count`来记录调用`print_message`的次数，实现输出

trap_s内的对应代码如下:

```
    csrr t0, scause
```

```
    bgez t0, s_exception    # determine interrupt(<0) or exception(>=0)
    andi t0, t0, 0x3f       # isolate the exception code field
    li t1, 5
    bne t0, t1, s_other_int # branch if not a timer interrupt

    # handle timer interrupt

    # increase the count
    la t0, COUNT
    ld t1, (t0)             # t1 = val(COUNT1)
    addi t1, t1, 1          # COUNT1 += 1
    sd t1, (t0)

    # print the interrupt count
    li t2, 100000
    blt t1, t2, skip_print
    and t1, t1, x0
    sd t1, (t0)             # set COUNT1 = 0

    call print_message

    # clear sip[stip] (bit 5)
skip_print:
    li t0, 32
    csrc sip, t0

    # init ecall jump to M mode
    ecall
```

strap.c内的代码如下:

```
#include "put.h"

int count = 0;

void print_message(void){
    const char *msg =" [S] Supervisor Mode Timer Interrupt ";
    puts(msg);
    puti(count);
    count++;

    const char *enter = "\n";
    puts(enter);
}
```

6. 编译及测试

- 在设置了环境变量之后执行make

```
oslab@ff3d0b3e90b5:~/lab2$ export TOP=`pwd`
oslab@ff3d0b3e90b5:~/lab2$ export RISCV=/opt/riscv
oslab@ff3d0b3e90b5:~/lab2$ export PATH=$PATH:$RISCV/bin
oslab@ff3d0b3e90b5:~/lab2$ make
make -C lib
make[1]: Entering directory '/home/oslab/lab2/lib'
riscv64-unknown-elf-gcc -c -g -march=rv64imafd -mabi=lp64 -mcmodel=medany -ffunction-sections -fdata-sections -nostartfiles -nostdlib -nostdinc -static -lgcc -Wl,--nmagic -Wl,--gc-sections -I ../include
-c put.c -o put.o
make[1]: Leaving directory '/home/oslab/lab2/lib'
make -C init
make[1]: Entering directory '/home/oslab/lab2/init'
riscv64-unknown-elf-gcc -c -g -march=rv64imafd -mabi=lp64 -mcmodel=medany -ffunction-sections -fdata-sections -nostartfiles -nostdlib -nostdinc -static -lgcc -Wl,--nmagic -Wl,--gc-sections -I ../include
-c test.c -o test.o
riscv64-unknown-elf-gcc -c -g -march=rv64imafd -mabi=lp64 -mcmodel=medany -ffunction-sections -fdata-sections -nostartfiles -nostdlib -nostdinc -static -lgcc -Wl,--nmagic -Wl,--gc-sections -I ../include
-c main.c -o main.o
make[1]: Leaving directory '/home/oslab/lab2/init'
make -C arch/riscv
make[1]: Entering directory '/home/oslab/lab2/arch/riscv'
make -C kernel
make[2]: Entering directory '/home/oslab/lab2/arch/riscv/kernel'
riscv64-unknown-elf-gcc -c -g -march=rv64imafd -mabi=lp64 -mcmodel=medany -ffunction-sections -fdata-sections -nostartfiles -nostdlib -nostdinc -static -lgcc -Wl,--nmagic -Wl,--gc-sections -I ../include
head.S -o head.o
riscv64-unknown-elf-gcc -c -g -march=rv64imafd -mabi=lp64 -mcmodel=medany -ffunction-sections -fdata-sections -nostartfiles -nostdlib -nostdinc -static -lgcc -Wl,--nmagic -Wl,--gc-sections -I ../include
entry.S -o entry.o
riscv64-unknown-elf-gcc -c -g -march=rv64imafd -mabi=lp64 -mcmodel=medany -ffunction-sections -fdata-sections -nostartfiles -nostdlib -nostdinc -static -lgcc -Wl,--nmagic -Wl,--gc-sections -I ../include
strap.c -o strap.o
make[2]: Leaving directory '/home/oslab/lab2/arch/riscv/kernel'
riscv64-unknown-elf-ld ../../init/test.o ../../init/main.o ./kernel/head.o ./kernel/entry.o ./kernel/strap.o ../../lib/put.o -T ./kernel/vmlinux.lds -o ../../vmlinux
riscv64-unknown-elf-objcopy -O binary ../../vmlinux ./boot/Image --strip-all
nm ../../vmlinux > ../../System.map
make[1]: Leaving directory '/home/oslab/lab2/arch/riscv'
```

- 运行make run

```
oslab@ff3d0b3e90b5:~/lab2$ make run
qemu-system-riscv64 -nographic -machine virt -kernel vmlinux
qemu-system-riscv64: warning: No -bios option specified. Not loading a firmware.
qemu-system-riscv64: warning: This default will change in a future QEMU release. Please use the -bios option to avoid breakages when this happens.
qemu-system-riscv64: warning: See QEMU's deprecation documentation for details.
ZJU OS LAB 2             GROUP-32
 [S] Supervisor Mode Timer Interrupt 0
 [S] Supervisor Mode Timer Interrupt 1
 [S] Supervisor Mode Timer Interrupt 2
 [S] Supervisor Mode Timer Interrupt 3
 [S] Supervisor Mode Timer Interrupt 4
```

- 运行make clean

```
oslab@b8d1b95e2b4c:~/lab2$ ls
Makefile  System.map  arch  include  init  lib  vmlinux
oslab@b8d1b95e2b4c:~/lab2$ make clean
cd lib && make clean
make[1]: Entering directory '/home/oslab/lab2/lib'
rm -f *.o
make[1]: Leaving directory '/home/oslab/lab2/lib'
cd init && make clean
make[1]: Entering directory '/home/oslab/lab2/init'
rm -f *.o
make[1]: Leaving directory '/home/oslab/lab2/init'
cd arch/riscv && make clean
make[1]: Entering directory '/home/oslab/lab2/arch/riscv'
cd kernel && make clean
make[2]: Entering directory '/home/oslab/lab2/arch/riscv/kernel'
rm -f *.o
make[2]: Leaving directory '/home/oslab/lab2/arch/riscv/kernel'
rm ../../vmlinux
rm ../../System.map
rm ./boot/Image
make[1]: Leaving directory '/home/oslab/lab2/arch/riscv'
oslab@b8d1b95e2b4c:~/lab2$ ls
Makefile  arch  include  init  lib
```

7. 思考题

思考题1：通过观察vmlinux和image，可以发现image是vmlinux的子集。image中包含了.head.text到.sdata的内容，但不包含elf头部以及bss后的内容。image中没有对bss段内的内容补零，因此我们需要在head.S中对bss段进行初始化

思考题2：因为同步异常是当指令执行时由CPU控制单元产生的，只有在一条指令终止执行后CPU才会发出异常，因此同步异常处理结束后我们可以直接执行下一条指令的内容；而中断是其他硬件设备依照CPU时钟信号随机产生的，无法预知中断发生时当前指令是否执行结束，因此我们在中断结束后要重新执行当前指令的内容，因此无需`mepc+4`

8. 心得体会

本次实验感觉收获很多，不仅对RISC-V汇编语言更加熟练，也对Linux中的时钟中断处理、委托等概念的理解上升到了实践，对整个过程理解的很清楚，觉得获益匪浅。

9. 实验建议

实验报告中没有提到sip[stip]是只读的，因此在trap_m中处理时钟中断时如果通过修改sip[stip]并不会导致mip[stip]变化，我觉得如果想降低实验难度可以稍微提醒一下(也可能这是我个人的我问题)。但不提醒也可以提高debug能力，且在3.2.2 Supervisor Mode下时钟中断处理流程中提到的修改方法是正确的（修改mip[stip]）