移植任务

将mos从mips一直到riscv上。

LAB0 环境配置

我使用wsl虚拟机中的ubuntu进行相应开发，打开相应功能，依次输入：

```
wsl --update
wsl --set-default-version 2
wsl -l --online
wsl --install -d Ubuntu
wsl -d Ubuntu
```

可以通过xshell连接该虚拟机：

```
sudo apt-get install openssh-server
sudo visudo
 //在最后加上这段
<用户名> ALL=(ALL) NOPASSWD:ALL

//为了开机自动启动ssh,开机会自动执行.bashrc中的内容
vim ~/.bashrc
//在最后一行输入下面这段
sudo service ssh start
vim /etc/ssh/sshd_config

//在文本最后输入
Port 2222   #ssh端口号
PermitRootLogin yes  #可以root远程等率
PasswordAuthentication yes   # 密码验证登录

//重启服务
sudo service ssh --full-restart
```

vim以及相关插件，我使用了NERDTree和YCM。

交叉编译工具链，我选择编译安装，因为通过课程组给的

```
apt install gcc-riscv64-unknown-elf
```

直接安装，会报相应错误，比如说 64位小端并不支持32位小端等等。

同时记得在编译安装时增加多库参数以支持32位，安装完成后可以通过`--march`和`-mabi`查看支持的库。

模拟器QEMU，我选择编译安装。

LAB1 内核启动与printk实现

修改内核链接文件，内核开始地址为：`0x80200000`,结束于`0x80600000`。

修改编译内核用的Makefile文件和引用的mk文件，其中主要修改的部分为 编译 和启动需要的工具，以及相关变量。

关于printk，通过调用相应ecall，实现输入和输出以及halt。

```
#ifndef _SBI_H_
#define _SBI_H_


#define SBI_SET_TIMER 0
#define SBI_CONSOLE_PUTCHAR 1
#define SBI_CONSOLE_GETCHAR 2
#define SBI_SET_SHUTDOWN 8

#define SBI_ECALL(__num, __a0, __a1, __a2)  \
({                                          \
    register unsigned long a0 asm("a0") = (unsigned long)(__a0); \
    register unsigned long a1 asm("a1") = (unsigned long)(__a1); \
    register unsigned long a2 asm("a2") = (unsigned long)(__a2); \
    register unsigned long a7 asm("a7") = (unsigned long)(__num); \
    asm volatile("ecall"                                        \
                 : "+r"(a0)                                     \
                 : "r"(a1),"r"(a2),"r"(a7)                      \
                 : "memory");                                   \
    a0;                                                         \
})


#define SBI_ECALL_0(__num) SBI_ECALL(__num,0,0,0)
#define SBI_ECALL_1(__num,__a0) SBI_ECALL(__num,__a0,0,0)
#define SBI_PUTCHAR(__a0) SBI_ECALL_1(SBI_CONSOLE_PUTCHAR, __a0)
#define SBI_GETCHAR() SBI_ECALL_0(SBI_CONSOLE_GETCHAR)
#define SBI_TIMER(__a0) SBI_ECALL_1(SBI_SET_TIMER,__a0)
#define SBI_SHUTDOWN() SBI_ECALL_0(SBI_SET_SHUTDOWN)
#endif
```

在修改上述完成后进行测试（lab1_t1）：

```
37777777634
4294967196
4294967196
123abc
123ABC
ffedc544
FFEDC544
ABCDEFGHIJKLMNOPQRSTUVWXYZ
I love buaa scse!
I l
97
a
61
97
1100001
i love buaa
abcdefghijklmnopqrst
i love os
string
good luck
-97and97
-97anda
-97and61
-97and1100001
-97and97
0097end
0061end
97  end
61  end
97end
0097end
ffffff9fend
1100010
98
98
142
98
98
7e6
7E6
printk_1
This letter is p
start- printk_1 -end
-98 and 98
-98 and b
-98 and 62
-98 and 1100010
-98 and 98
```

LAB2 MMU设置和内存管理

riscv开启MMU之后全局都需要访问页表，而tlb则是通过硬件维护，通过`sfence.vma`来对相应tlb项进行刷新。

riscv页表项结构与mips不同，需要修改，创建有关页表和内存管理的方法集合，将地址填入`satp`寄存器中，打开开关位，正式开始页式内存管理。

在修改完成后，通过测试（lab2_t1,lab2_t2）：

```
init.c: riscv32_init() is called
Memory size: 67108864 bytes, number of pages: 16384
------------------------------------------------------------
to memory 80631000 for struct Pages.
page_init success
The number in address tmp is 1000
physical_memory_manage_check() succeeded
The number in address temp is 1000
physical_memory_manage_check_strong() succeeded
db@LAPTOP-5N9TS139:~/21371110$
```

此处为lab2_t1测试结果。

```
init.c: riscv32_init() is called
Memory size: 67108864 bytes, number of pages: 16384
------------------------------------------------------------
to memory 80631000 for struct Pages.
page_init success
123va2pa(boot_pgdir, 0x0) is 83ffe000
page2addr(pp1) is 83ffe000
start page_insert
end page_insert
page_check_strong() succeeded!
db@LAPTOP-5N9TS139:~/21371110$
```

上述为lab2_t2测试结果。

LAB3 例外处理和进程管理。

在riscv中，`sie`和`sip`控制S态响应外部中断，通过修改相应位置来开关中断。

通过设置该寄存器开启时钟中断，通过调用之前所说的SBI宏，设置中断需要的时间。

其中，异常处理需要进行上下文的切换，而切换前可以是U态也可以是S态，需要对mips的上下文切换机制进行相应的修改。利用`sscratch`寄存器保存相应栈的值（同时如果sscratch为0，则之前状态为S态，否则为U态，因为存储了S态的栈值），同时增加对SR寄存器的保护和修复（stval，sepc,scause，sstatus,sscratch(栈值)）。修改上下文对应的结构体。

根据riscv的异常种类建立异常向量表，stvec记录了异常处理程序的入口，有两种模式，第一种是不同类型异常对应不同的地址，第二种则是异常号在给定的地址上进行寻址，这里我选择第二种。

riscv中的异常种类：

amoadd.w。

Interrupt / Exception mcause[XLEN-1]	Exception Code mcause[XLEN-2:0]	Description
1	1	Supervisor software interrupt
1	3	Machine software interrupt
1	5	Supervisor timer interrupt
1	7	Machine timer interrupt
1	9	Supervisor external interrupt
1	11	Machine external interrupt
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store address misaligned
0	7	Store access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	15	Store page fault

以下为我的异常向量表：

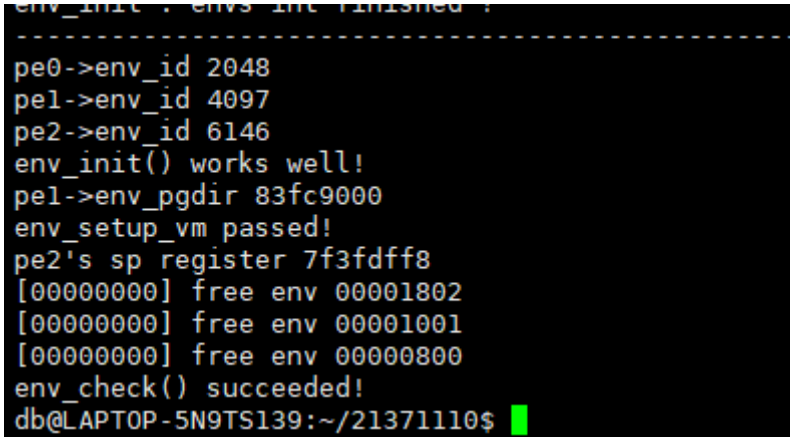
```
void (*exception_handlers[32])(void) = {
    [0 ... 31] = handle_reserved,
    [17] = handle_software,
    [21] = handle_timer,
    [3] = handle_software,
    [8] = handle_ecall_from_u,
    [12] = handle_instruction_page,
    [13] = handle_load_page,
    [15] = handle_store_page,
};
```

在`genex.S`中通过`BUILD_HANDLER`依次建立相应处理函数，实现异常处理。

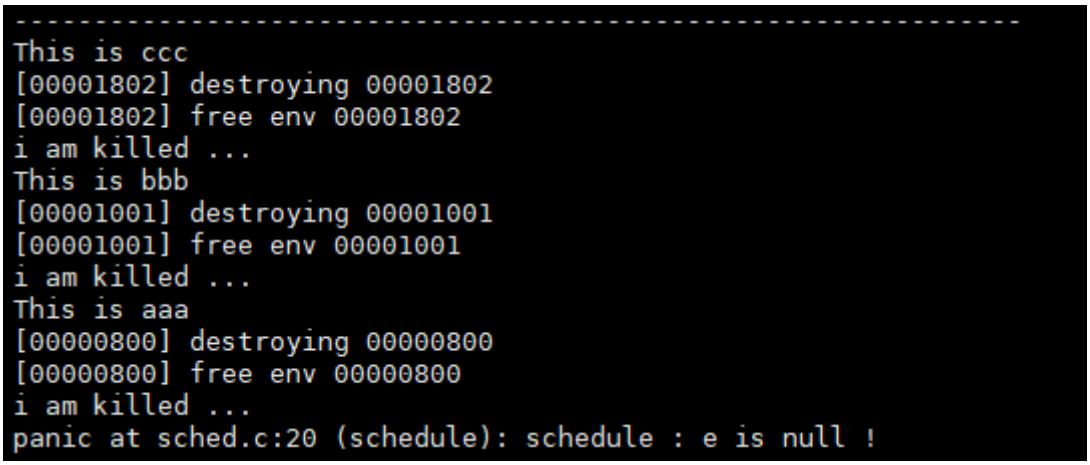
关于进程调度，由于之前所说，riscv必须全部通过页表寻址，所以需要对进程页表进行额外的处理，通过将部分内核中的代码以RX的权限映射到进程页表中，在这里上下文从S态和U态来回转换。每个进程分配10页空间大小的用户栈，在这里当时我对虚拟空间的认识并不充分，导致当时设计理解时出现了偏差，把内核态当成进程公共的，所以处理起来比较麻烦，**事实上可以跟用户栈进行同样的处理**，通过sscratch进行两者的切换。

调度时需要切换stvec中页表的地址，并通过`schedule`进行调度。

完成后通过测试(lab3_t1,lab3_t2):



以上为lab3_t1的测试结果，



以上为lab3_t2的测试结果。

LAB4 系统调用和fork

页表映射受限于当时的认知，认为一个进程需要4MB预留空间十分牺牲内存，故采取其他手段，通过系统调用将页表以只读映射到相应地址来实现用户态访问页表。

关于系统调用，通过建立系统调用表，在用户态建立系统调用用户端接口，在S态实现相关系统调用，并通过系统调用号将其联系起来。

关于fork，fork的目的是为了“复制”一份与源进程类似的进程，其中需要利用COW技术，标记不共享的可写页，在写时才复制，以此来提升效率。

利用riscv页表项中保留位，实现 `PTE_LIBRARY` 和 `PTE_COW`,通过将W可写位取消，标记上PTE_COW，在写时没有权限会触发15号写异常，在这里进行相应的处理，设置处理函数为返回地址以及相关寄存器。

完成后通过测试（lab4_1）：

```
--------------------------------------------------------------
superblock is good
read bit map is good
father!, -1
[00000800] destroying 00000800
[00000800] free env 00000800
i am killed ...
        child 1 ,1
         child 1 finished!, 1000
[00001802] destroying 00001802
[00001802] free env 00001802
i am killed ...
               child 2 ,100
               child 2 finished, -100
[00002000] destroying 00002000
[00002000] free env 00002000
i am killed ...
                       child 3 , -222222
[00002802] destroying 00002802
[00002802] free env 00002802
i am killed ...
panic at sched.c:20 (schedule): schedule : e is null !

ra:    802059bc  sp:  805ff5d8  sstatus: 00040000
scause: 00000008  sepc: 00400ca0  stval:  00000000
db@LAPTOP-5N9TS139:~/21371110$
```

LAB5 文件系统

qemu模拟的virt不能使用ide，只能使用virtio-mmio来挂载磁盘。关于外设，可以通过导出dts文件进行查询，发现共可以挂载8块磁盘。

在LAB5中，文件系统的User层和file层是通用的，只有底层的Driver层需要根据virtio进行相应的修改。

将设备寄存器的地址映射到相应地址（在env初始化时实现）。

根据文档，进行相应的初始化。

建立其相关操作集合和数据集合，比如磁盘的信息结构体和管理请求的虚拟请求结构体。

对于块设备，虚拟环形队列有3个，分别为desc，used，free三项，对于一个单独的请求，它由三部分组成，第一部分为 请求头`struct virtio_blk_req`,第二部分则是处理的数据，第三部分则是该请求的状态位。同时需要相应内存锁来进行同步，在处理就绪后，设置相应位向磁盘发送请求，根据状态位来进行相应的处理判断，通过底层驱动来实现原始扇区的读写。

完成后通过相应测试（lab5_t1,lab5_t2）：

```
superblock is good
read bit map is good
hello, file system!
buf is ;
This is /motd, the message of the day.

Welcome to the MOS kernel, now with a file system!
;
buf is ;
;
[00000800] destroying 00000800
[00000800] free env 00000800
i am killed ...
panic at sched.c:20 (schedule): schedule : e is null !
```

以上为LAB5_t1的测试结果。

```
env_init : envs init finished !
------------------------------------------------------------
superblock is good
read bit map is good
This is a different message of the day!
[00000800] destroying 00000800
[00000800] free env 00000800
i am killed ...
panic at sched.c:20 (schedule): schedule : e is null !

ra:    802059bc  sp:  805ff058  sstatus: 00040000
scause: 00000008  sepc: 00400bc4  stval:  00000000
db@LAPTOP-5N9TS139:~/21371110$
```

以上为LAB5_t2的测试结果。

LAB6 实现简易shell

根据mips中的管道来实现管道，同时模拟实现shell，不同的是spawnl需要对不定参数进行读取，而非直接作为参数进入spawn中。

以下为测试结果：

```
------------------------------------------------------------
superblock is good
read bit map is good
icode: open /motd
icode: read /motd
This is /motd, the message of the day.

Welcome to the MOS kernel, now with a file system!

icode: close /mote
icode: spawn /init
icode: exiting
[00000800] destroying 00000800
[00000800] free env 00000800
i am killed ...
init: running
init: data seems okay
init: bss seems okay
init: args: 'init' 'initarg1' 'initarg2'
init: running sh
init: starting sh

::::::::::::::::::::::::::::::::::::::::::::::::::
::                                              ::
::              MOS Shell 2023                  ::
::                                              ::
::::::::::::::::::::::::::::::::::::::::::::::::::

$
```

```
init: starting sh

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::                                                       ::
::                 MOS Shell 2023                        ::
::                                                       ::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

$ ls.b
init.b num.b sh.b motd newmotd echo.b halt.b test1.b ls.b cat.b tt
[00003004] destroying 00003004
[00003004] free env 00003004
i am killed ...
[00002803] destroying 00002803
[00002803] free env 00002803
i am killed ...

$
```

```
i am killed ...

$ cat.b /motd
This is /motd, the message of the day.

Welcome to the MOS kernel, now with a file system!
[00005004] destroying 00005004
[00005004] free env 00005004
i am killed ...
[00004803] destroying 00004803
[00004803] free env 00004803
i am killed ...

$
```

```
$ halt.b
halt at halt.c:4: halt_mos!

panic at ecall_all.c:250 (e_panic): user halt
ra:    80202028  sp:  805895c0  sstatus: 00040000
scause: 00000008  sepc: 00400034  stval:  00000000
```

以上为我移植任务的实践报告，在过程中由于不熟悉，以及知识水平的欠缺，实现的并不好，站在后来人的角度上，都有更好的实现办法。