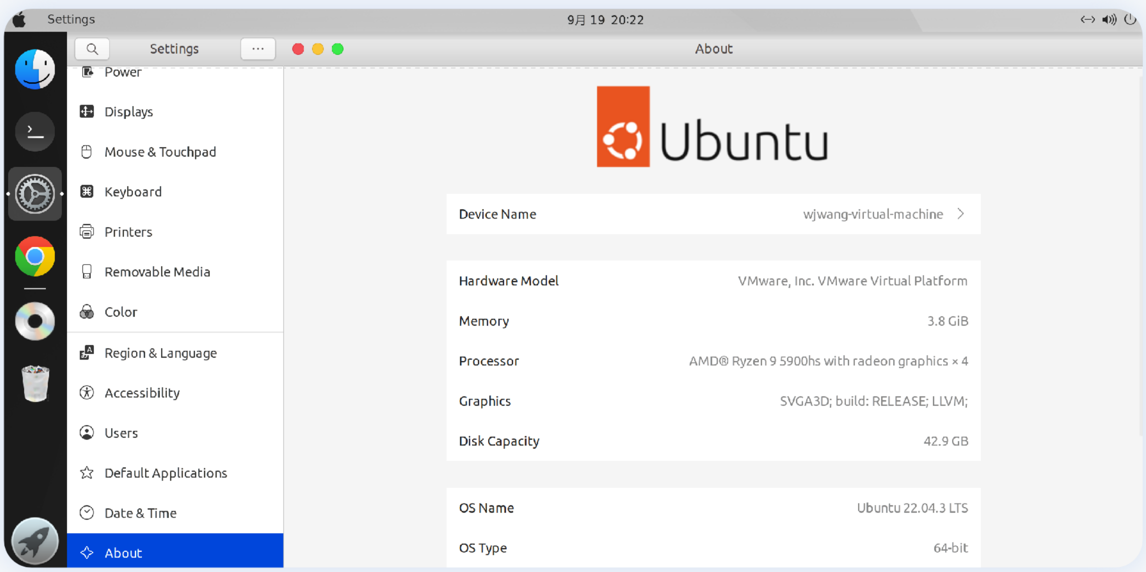
Lab7: VFS & FAT32 文件系统

学号：3210106034
姓名：王伟杰

> 实验环境：
>
> ```
> uname -a
> Linux wjwang-virtual-machine 6.2.0-33-generic #33~22.04.1-Ubuntu SMP PREEMPT_DYNAMIC Thu Sep  7 10:33:52 UTC 2 x86_64 x86_64 x86_64 GNU/Linux
> ```
>
> 
>

准备工程

同步 `os23fall-stu` 中的 `user` 文件夹，替换原有的用户态程序为 `nish`，并下载[磁盘镜像]并放置在项目目录下。

向 `include/types.h` 中补充一些类型别名，并修改 `arch/riscv/kernel/vmlinux.lds` 中的 `_sramdisk` 符号部分(将 uapp 修改为 ramdisk)

```c
typedef unsigned long uint64_t;
typedef long int64_t;
typedef unsigned int uint32_t;
typedef int int32_t;
typedef unsigned short uint16_t;
typedef short int16_t;
typedef uint64_t* pagetable_t;
typedef char int8_t;
typedef unsigned char uint8_t;
typedef uint64_t size_t;
```

这时项目能够成功编译并运行

```
make run
Boot HART PMP Granularity : 4
Boot HART PMP Address Bits: 54
Boot HART MHPM Count      : 0
Boot HART MHPM Count      : 0
Boot HART MIDELEG         : 0x0000000000000222
Boot HART MEDELEG         : 0x000000000000b109
...setup_vm done!
...buddy_init done!
...setup_vm_final done!
...proc_init done!
2022 Hello RISC-V
sstatus = 8000000000006000
sscratch = 0

switch to [PID = 1 COUNTER = 2 PRIORITY = 2]
Page Fault: sepc: 00000000000100e8, scause: 000000000000000c, stval: 00000000000
100e8
Page Fault: sepc: 0000000000011298, scause: 000000000000000c, stval: 00000000000
11298
Page Fault: sepc: 000000000001129c, scause: 000000000000000f, stval: 0000003ffff
ffff8
hello, stdout!
sys_write: fd != STDOUT
SHELL > 
```

Shell: 与内核进行交互

文件信息初始化

在task_struct中添加文件的信息指针：

```c
#include "fs.h"
struct task_struct {
    // struct thread_info* thread_info;
    uint64 state;
    uint64 counter;
    uint64 priority;
    uint64 pid;

    struct thread_struct thread;

    pagetable_t pgd;

    struct file *files;

    uint64_t vma_cnt;
    struct vm_area_struct vmas[0];
};
```

在创建进程时为进程初始化文件，当初始化进程时，先完成打开的文件的列表的初始化，直接分配一个页，并用 `files` 指向这个页。

```c
void task_init() {
    // ...
    for(int i = 1; i < INIT_TASKS; ++i){
        // ...
        task[i]->files = file_init();
        // ...
    }
    printk("...proc_init done!\n");
}
```

完成file_init函数，主要就是设置函数指针：

```c
struct file* file_init() {
    struct file *ret = (struct file*)alloc_page();

```

```c
    // stdin
    ret[0].opened = 1;
    ret[0].perms = FILE_READABLE;
    ret[0].cfo = 0;
    ret[0].lseek = NULL;
    ret[0].write = NULL;
    ret[0].read = stdin_read;
    memcpy(ret[0].path, "stdin", 6);

    // stdout
    ret[1].opened = 1;
    ret[1].perms = FILE_WRITABLE;
    ret[1].cfo = 0;
    ret[1].lseek = NULL;
    ret[1].write = stdout_write;
    ret[1].read = NULL;
    memcpy(ret[1].path, "stdout", 7);

    // stderr
    ret[2].opened = 1;
    ret[2].perms = FILE_WRITABLE;
    ret[2].cfo = 0;
    ret[2].lseek = NULL;
    ret[2].write = stderr_write;
    ret[2].read = NULL;
    memcpy(ret[2].path, "stderr", 7);

    return ret;
}
```

完成trap跳转

接下来，我们完善trap_handler，以实现ecall之后trap跳转到正确的函数：

```c
void trap_handler(unsigned long scause, unsigned long sepc, struct pt_regs *regs) {
    unsigned long temp = 1;
    if(scause & (temp << 63)) { // interrupt
        // ...
    } else {
        switch (scause & ~(temp << 63))
        {
```

```c
        case 0x8: // ECALL_FROM_U_MODE
            uint64 syscall_id = regs->a7;
            switch (syscall_id)
            {
            case SYS_READ: // sys_read
                regs->a0 = sys_read((unsigned int)regs->a0, (char*)regs->a1, (uint64_t)regs->a2);
                break;
            case SYS_WRITE: // sys_write
                regs->a0 = sys_write((unsigned int)regs->a0, (char*)regs->a1, (size_t)regs->a2);
                break;
            // ...
            }
            // 针对系统调用这一类异常， 我们需要手动将 sepc + 4
            regs->sepc += 4;
            break;
        // ...
        }
    } // exception
}
```

`write` 函数调用 `sys_write` ，间接调用我们赋值的 `stdout` 对应的函数指针：

```c
uint64 sys_write(unsigned int fd, const char* buf, uint64_t count) {
    int64_t ret;
    struct file* target_file = &(current->files[fd]);
    if (target_file->opened) {
        ret = target_file->write(target_file, buf, count);
    } else {
        printk("file not open\n");
        ret = ERROR_FILE_NOT_OPEN;
    }
    return ret;
}
```

参考 `syscall_write` 的实现，来实现 `syscall_read` ：

```c
uint64 sys_read(unsigned int fd, char* buf, uint64_t count) {
    int64_t ret;
    struct file* target_file = &(current->files[fd]);
    if (target_file->opened) {
        ret = target_file->read(target_file, buf, count);
    } else {
        printk("file not open\n");
        ret = ERROR_FILE_NOT_OPEN;
    }
    return ret;
}
```

实现串口操作

`write` 函数第一个参数是 `1` 的时候调用函数 `stdout_write` ，在这里实现输出到终端：

```c
int64_t stdout_write(struct file* file, const void* buf, uint64_t len) {
    char to_print[len + 1];
    for (int i = 0; i < len; i++) {
        to_print[i] = ((const char*)buf)[i];
    }
    to_print[len] = 0;
    return printk(buf);
}
```

同理，在 `write` 函数第一个参数是 `2` 的时候进入 `stderr_write` 函数，这里同样输出到终端：

```c
int64_t stderr_write(struct file* file, const void* buf, uint64_t len) {
    char to_print[len + 1];
    for (int i = 0; i < len; i++) {
        to_print[i] = ((const char*)buf)[i];
    }
    to_print[len] = 0;
    return printk(buf);
}
```

在处理 `stdin` 函数的时候，我们通过查阅[openSBI的手册]得知，如果控制台没有获取到可用字符，不会阻塞，而是会返回0，所以我们需要在 `stdin_read` 函数中做特殊判断，使其继续等待可用字符：

```c
int64_t stdin_read(struct file* file, void* buf, uint64_t len) {
    char* buf_c = (char*)buf;
    for (int i = 0; i < len; i++) {
        char c = uart_getchar();
        if (c == '\0') {
            i--;
            continue;
        }
        buf_c[i] = c;
    }
    return len;
}
```

内核交互测试

至此，就正确打印出 `stdout` 与 `stderr`，并且可以在 `nish` 中使用 `echo` 命令了。