

Lab6: VFS & FAT32 文件系统

姓名：姜雨童

学号：3220103450

1 实验内容及原理

1.1 实验目的

- 为用户态的 Shell 提供 `read` 和 `write` syscall 的实现（完成该部分的所有实现方得 60 分）
- 实现 FAT32 文件系统的基本功能，并对其中的文件进行读写（完成该部分的所有实现方得 40 分）

1.2 实验环境

- Environment in previous labs

2 实验过程与代码实现

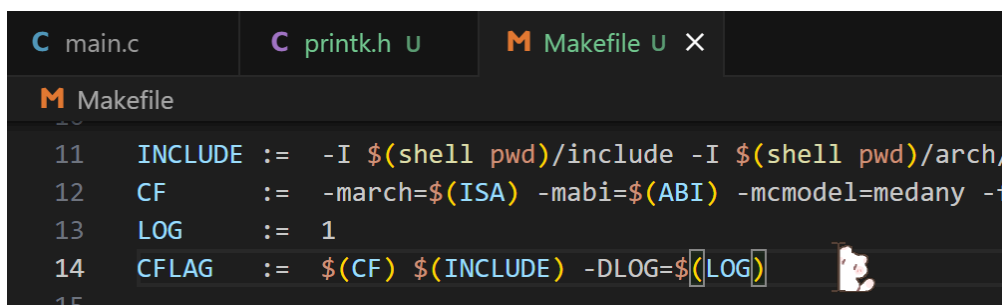
（本实验我只完成了第一部分。）

2.1 准备工作

- 从仓库同步所需文件，递归且不覆盖地将 `lab5` 中内容拷贝到 `lab6`：

```
1 | jyt@fine:~/os24fall-stu/src$ cp -Rn lab5/* lab6
```

- 在根目录下的 Makefile 中添加内容以实现运行 `make run LOG=0` 时，不会出现 Log 的输出。注意不同路径的两个 `printk.h` 都需要改，这里额外定义了一个 `Log6` 便于后续输出。



```

C main.c
C printk.h U
M Makefile U X
M Makefile
11 INCLUDE := -I $(shell pwd)/include -I $(shell pwd)/arch/
12 CF      := -march=$(ISA) -mabi=$(ABI) -mcmmodel=medany -f
13 LOG     := 1
14 CFLAG   := $(CF) $(INCLUDE) -DLOG=$(LOG)
15

```

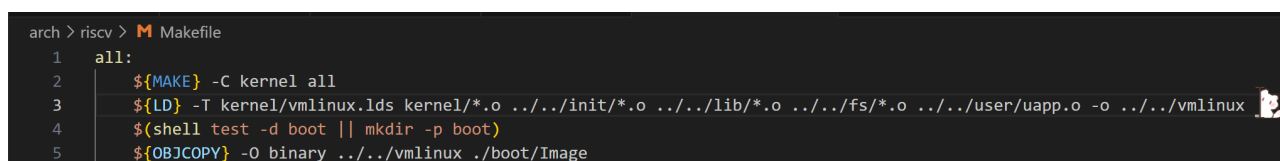


```

include > C printk.h > LogB(format, __VA_ARGS__)
1  #ifndef __PRINTK_H__
19 #if LOG
44     while(1);
45 }
46 #else
47 #define Log(format, ...)
48 #define LogDG(format, ...)
49 #define LogB(format, ...)
50 #define LogY(format, ...)
51 #define LogG(format, ...)
52 #define Err(format, ...)
53 #define Log6(format, ...) \
54     printk("\33[1;34m[%s,%d,%s] " format "\33[0m\n", \
55         __FILE__, __LINE__, __func__, ## __VA_ARGS__)
56 #endif
57

```

- 在 `arch/riscv/Makefile` 里面添加相关编译产物以编译新加入的 fs 文件夹下的内容



```

arch > riscv > M Makefile
1  all:
2  ${MAKE} -C kernel all
3  ${LD} -T kernel/vmlinux.lds kernel/*.o ../../init/*.o ../../lib/*.o ../../fs/*.o ../../user/uapp.o -o ../../vmlinux
4  $(shell test -d boot || mkdir -p boot)
5  ${OBJCOPY} -O binary ../../vmlinux ./boot/Image

```

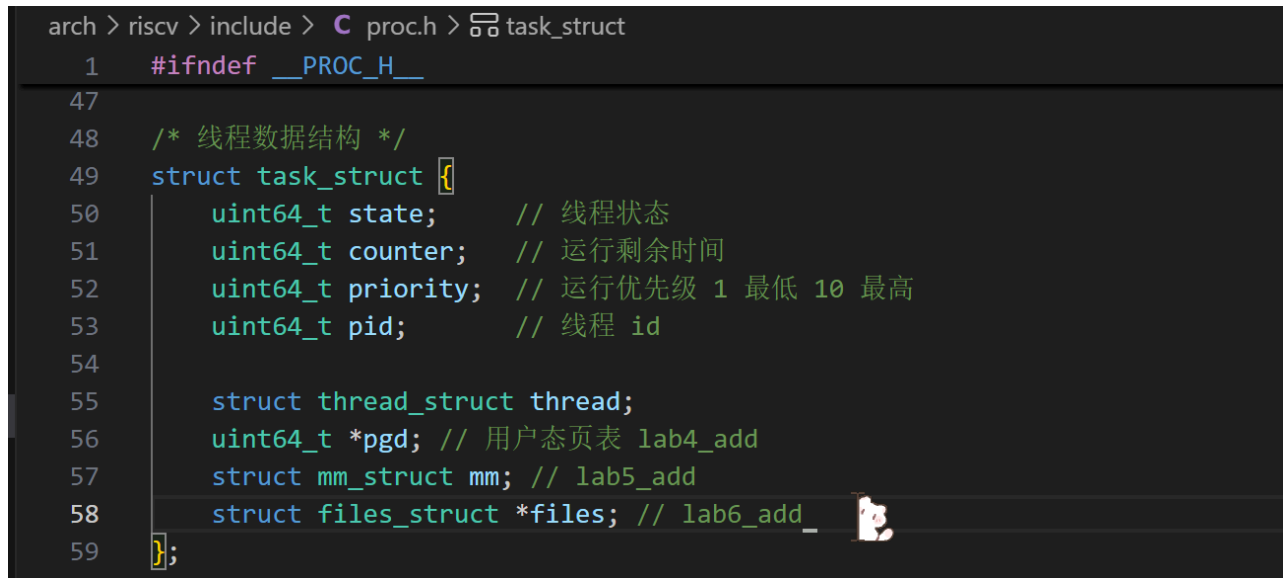
2.2 Shell: 与内核进行交互

实验提供了用户态程序 "nish" (Not Implemented SHell) 来与实验中完成的 kernel 进行交互，提供命令：

```
1 echo [string]    # 将 string 输出到 stdout
2 cat [path]       # 将路径为 path 的文件的内容输出到 stdout
3 edit [path] [offset] [string] # 将路径为 path 的文件，偏移量为 offset 的部分开始，写为 string
```

2.2.1 文件系统抽象

修改 proc.h，为进程 task_struct 结构体添加一个指向文件表的指针：



```
arch > riscv > include > C proc.h > task_struct
1  #ifndef __PROC_H__
47
48  /* 线程数据结构 */
49  struct task_struct {
50      uint64_t state;      // 线程状态
51      uint64_t counter;    // 运行剩余时间
52      uint64_t priority;   // 运行优先级 1 最低 10 最高
53      uint64_t pid;       // 线程 id
54
55      struct thread_struct thread;
56      uint64_t *pgd; // 用户态页表 lab4_add
57      struct mm_struct mm; // lab5_add
58      struct files_struct *files; // lab6_add
59  };
60
```

2.2.2 stdout/err/in 初始化

在 fs/fs.c 文件中，定义了函数 file_init（用法：在 proc.c 中的 task_init 函数中为每个进程调用，创建文件表并保存在 task struct 中），需要实现：

- 根据 files_struct 的大小分配页空间：含有 MAX_FILE_NUMBER=16 个 file 结构体，因此只需要分配一个页面空间就够了。
- 为 stdin、stdout、stderr 赋值：注意启动一个用户态程序（包括 nish）时默认打开了这三个文件，因此这里赋值时，opened 字段都要赋一。
- 保证其他未使用的文件的 opened 字段为 0

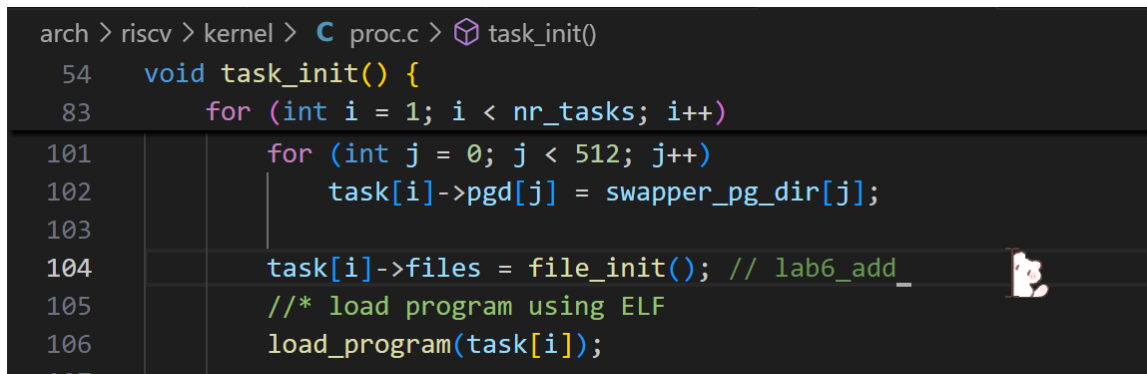
```
1 struct files_struct *file_init() {
2     struct files_struct *ret = (struct files_struct *)alloc_page();
3
4     // stdin
5     ret->fd_array[0].opened = 1;
6     ret->fd_array[0].perms = FILE_READABLE;
7     ret->fd_array[0].cfo = 0;
8     ret->fd_array[0].lseek = NULL;
9     ret->fd_array[0].write = NULL;
10    ret->fd_array[0].read = stdin_read;
11    memcpy(ret->fd_array[0].path, "stdin", 6);
12
13    // stdout
14    ret->fd_array[1].opened = 1;
```

```

15     ret->fd_array[1].perms = FILE_WRITABLE;
16     ret->fd_array[1].cfo = 0;
17     ret->fd_array[1].lseek = NULL;
18     ret->fd_array[1].write = stdout_write;
19     ret->fd_array[1].read = NULL;
20     memcpy(ret->fd_array[1].path, "stdout", 7);
21
22     // stderr
23     ret->fd_array[2].opened = 1;
24     ret->fd_array[2].perms = FILE_WRITABLE;
25     ret->fd_array[2].cfo = 0;
26     ret->fd_array[2].lseek = NULL;
27     ret->fd_array[2].write = stderr_write;
28     ret->fd_array[2].read = NULL;
29     memcpy(ret->fd_array[2].path, "stderr", 7);
30
31     return ret;
32 }

```

进程调用:



```

arch > riscv > kernel > C proc.c > task_init()
54 void task_init() {
83     for (int i = 1; i < nr_tasks; i++)
101         for (int j = 0; j < 512; j++)
102             task[i]->pgd[j] = swapper_pg_dir[j];
103
104     task[i]->files = file_init(); // lab6_add_
105     /* load program using ELF
106     load_program(task[i]);
107

```

2.2.3 处理 stdout/err 的写入

用户态程序在开始的时候会通过 `write` 函数来向内核发起 syscall 进行测试。在捕获到 `write` 的 syscall 之后，可以查找对应的 `fd`，并通过对应的 `write` 函数调用来进行输出。

`syscall.c`:

```

1 int64_t sys_write(uint64_t fd, const char *buf, uint64_t len) {
2     int64_t ret;
3     struct file *file = &(current->files->fd_array[fd]);
4     if (file->opened == 0) {
5         printk("file not opened\n");
6         return ERROR_FILE_NOT_OPEN;
7     } else {
8         ret = file->write(file, buf, len);
9     }
10    return ret;
11 }

```

- 这里更改了原来使用的函数声明，需要在对应头文件中改过来。

在 `fs/vfs.c` 中通过 `printk` 进行串口输出:

```

1 int64_t stdout_write(struct file *file, const void *buf, uint64_t len) {
2     char to_print[len + 1];

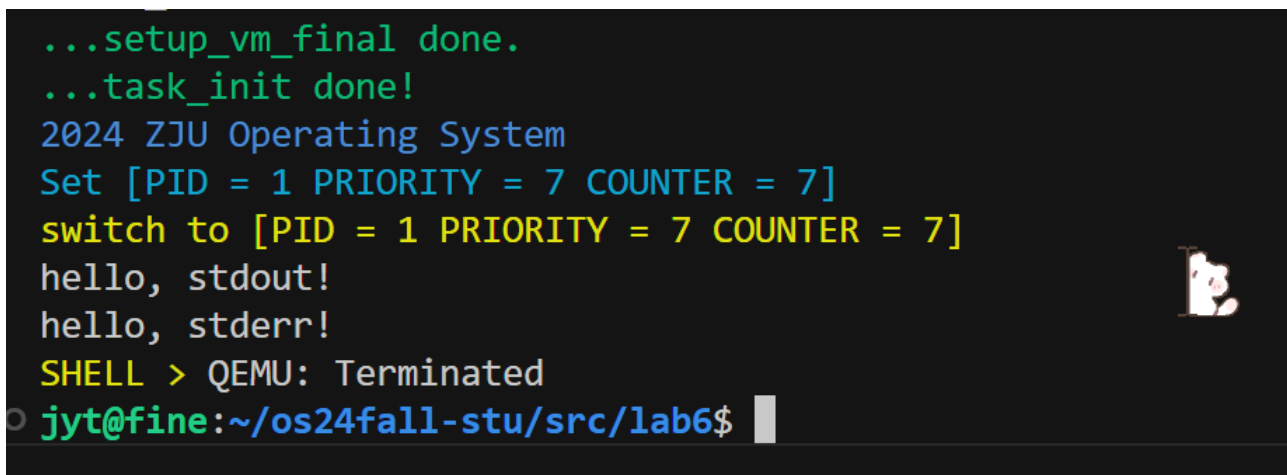
```

```

3   for (int i = 0; i < len; i++) {
4       to_print[i] = ((const char *)buf)[i];
5   }
6   to_print[len] = 0;
7   return printk(to_print);
8 }
9
10  int64_t stderr_write(struct file *file, const void *buf, uint64_t len) {
11      char to_print[len + 1];
12      for (int i = 0; i < len; i++) {
13          to_print[i] = ((const char *)buf)[i];
14      }
15      to_print[len] = 0;
16      return printk(to_print);
17 }

```

测试发现已经能够打印出输出：



```

...setup_vm_final done.
...task_init done!
2024 ZJU Operating System
Set [PID = 1 PRIORITY = 7 COUNTER = 7]
switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
hello, stdout!
hello, stderr!
SHELL > QEMU: Terminated
jyt@fine:~/os24fall-stu/src/lab6$

```

2.2.4 处理 stdin 的读取

该部分要实现 `stdin` 以获取键盘键入的内容，进而实现与用户的交互。

- 通过 `sbi` 来完成终端的输入，并把函数声明加入头文件中（详细说明见报告第4节）：

`arch/risc/kernel/sbi.c`

```

1  struct sbiret sbi_debug_console_read(uint64_t num_bytes, uint64_t base_addr_lo,
2      uint64_t base_addr_hi) {
3      return sbi_ecall(0x4442434e, 1, num_bytes, base_addr_lo, base_addr_hi, 0,0,0);
4  }

```

- 调用 `fs/vfs.c` 中的 `uart_getchar()` 函数，实现 `vfs.c/stdin_read` 对 `len` 个字符的读取：

```

1  int64_t stdin_read(struct file *file, void *buf, uint64_t len) {
2      // use uart_getchar() to get `len` chars
3      // Log6("stdin_read called: file %p, buf %p, len %d", file, buf, len);
4      char *to_read = (char *)buf;
5      for (int i = 0; i < len; i++) {
6          char c = uart_getchar();
7          if (c == '\n') {
8              i--;
9              continue;
10         }

```

```

11     to_read[i] = c;
12 }
13 return len;
14 }

```

- 在 `trap_handler` 捕获 63 号系统调用 `read`，并仿照 `write` 实现 `sys_read`，把函数声明加入头文件：

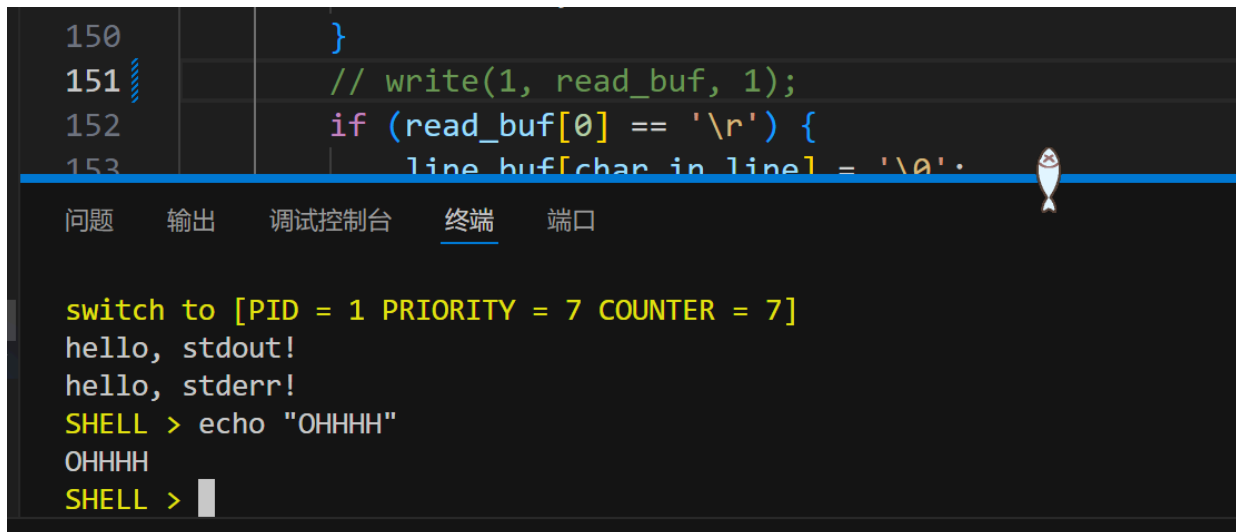
```

1 // arch/riscv/kernel/trap.c
2 ...
3 case 63: // sys_read
4     regs->a0 = sys_read(regs->a0, (const char *)regs->a1, regs->a2);
5 case 64: // sys_write
6     regs->a0 = sys_write(regs->a0, (const char *)regs->a1, regs->a2);
7 ...

1 // syscall.c
2 int64_t sys_read(uint64_t fd, char *buf, uint64_t len) {
3     int64_t ret;
4     struct file *file = &(current->files->fd_array[fd]);
5     if (file->opened == 0) {
6         printk("file not opened\n");
7         return ERROR_FILE_NOT_OPEN;
8     } else {
9         ret = file->read(file, buf, len);
10    }
11    return ret;
12 }

```

实现输入：



```

150     }
151 // write(1, read_buf, 1);
152 if (read_buf[0] == '\r') {
153     line_buf[char_in_line] = '\0';

```

问题 输出 调试控制台 终端 端口

```

switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
hello, stdout!
hello, stderr!
SHELL > echo "OHHHH"
OHHHH
SHELL >

```

3 实验中遇到的问题及解决方法

最初测试写功能时，报错如下图所示。产生原因是更改了 `sys_write` 的函数声明，但是没有在头文件中改过来。

```

syscall.c
syscall.c:22:9: error: conflicting types for 'sys_write'; have 'int64_t(uint64_t, const char *, uint64_t)' {aka 'long long int(long long unsigned int, c
char *, long long unsigned int)'}
22 | int64_t sys_write(uint64_t fd, const char *buf, uint64_t len) {
    |         ^~~~~~
In file included from syscall.c:1:
../include/syscall.h:9:10: note: previous declaration of 'sys_write' with type 'uint64_t(unsigned int, const char *, size_t)' {aka 'long long unsigned in
signed int, const char *, long unsigned int)'}
9 | uint64_t sys_write(unsigned int fd, const char* buf, size_t count);
  |         ^~~~~~
syscall.c: In function 'sys_write':
syscall.c:24:41: error: invalid use of undefined type 'struct file_struct'
24 |     struct file *file = &(current->files->fd_array[fd]);
    |

```

显示非法类型 `struct files_struct`，为 `proc.h` 加入 `#include "fs.h"` 解决：

```
syscall.c
syscall.c: In function 'sys_write':
syscall.c:24:41: error: invalid use of undefined type 'struct files_struct'
 24 |     struct file *file = &(current->files->fd_array[fd]);
    |                                         ^~
syscall.c:25:13: error: invalid use of undefined type 'struct file'
 25 |     if (file->opened == 0) {
    |             ^~
syscall.c:27:16: error: 'ERROR_FILE_NOT_OPEN' undeclared (first use in this function)
 27 |     return ERROR_FILE_NOT_OPEN;
    |            ^~~~~~
syscall.c:27:16: note: each undeclared identifier is reported only once for each function it appears in
syscall.c:29:19: error: invalid use of undefined type 'struct file'
 29 |     ret = file->write(file, buf, len);
    |             ^~
syscall.c: In function 'do_fork':
```

显示没有文件或文件夹不存在，检查后发现fs文件夹下的文件均没有被编译。在根目录的 `Makefile` 下添加内容解决报错（类似之前实验给 `user` 文件夹添加）

```
make[2]: Leaving directory '/home/jyt/os24fall-stu/src/lab6/arch/riscv/kernel'
riscv64-linux-gnu-ld -T kernel/vmlinux.lds kernel/*.o ../../init/*.o ../../lib/*.o ../../fs/*.o ../../user/uapp.o -o ../../vmlin
ux
riscv64-linux-gnu-ld: cannot find ../../fs/*.o: No such file or directory
make[1]: *** [Makefile:3: all] Error 1
make[1]: Leaving directory '/home/jyt/os24fall-stu/src/lab6/arch/riscv'
make: *** [Makefile:21: all] Error 2
jyt@fine:~/os24fall-stu/src/lab6$ make run LOG=0
```

```
1 all: clean
2     ...
3     $(MAKE) -C fs all
4
5 clean:
6     ...
7     $(MAKE) -C fs clean
```

正确编译fs文件夹下的文件后，显示找不到函数声明，发现在实验一中没写这个接口，补充（实验指导4.2.4）：

```
home/jyt/os24fall-stu/src/lab6/arch/riscv/include -DLOG=0 -c vfs.c
vfs.c: In function 'uart_getchar':
vfs.c:10:36: warning: implicit declaration of function 'sbi_debug_console_read'; did you mean 'sbi_debug_console
-Wimplicit-function-declaration]
 10 |     struct sbiret sbi_result = sbi_debug_console_read(1, ((uint64_t)&ret - PA2VA_OFFSET), 0);
    |                                ^~~~~~
    |                                sbi_debug_console_write_byte
vfs.c:10:36: error: invalid initializer
make[1]: *** [Makefile:7: vfs.o] Error 1
make[1]: Leaving directory '/home/jyt/os24fall-stu/src/lab6/fs'
make: *** [Makefile:21: all] Error 2
jyt@fine:~/os24fall-stu/src/lab6$
```

```
1 struct sbiret sbi_debug_console_read(uint64_t num_bytes, uint64_t base_addr_lo,
  uint64_t base_addr_hi) {
2     return sbi_ecall(0x4442434e, 1, num_bytes, base_addr_lo, base_addr_hi, 0, 0, 0);
3 }
```

Function Name	Description	Extension ID	Function ID
<code>sbi_set_timer</code>	设置时钟相关寄存器	0x54494d45	0
<code>sbi_debug_console_write</code>	向终端写入数据	0x4442434e	0
<code>sbi_debug_console_read</code>	从终端读取数据	0x4442434e	1

有多个头文件 `string.h`，首先尝试了修改引用方式（vscode中按住ctrl点击可以实现页面跳转，但是再次运行的时候还是显示找不到），仍然不对后直接写了一个；没有找到 `memcmp函数`，写一个。

```
riscv64-linux-gnu-ld -T kernel/vmlinux.lds kernel/*.o ../../init/*.o ../../lib/*.o ../../fs/*.o ../../user/uapp.o -o ../../vmlin
nux
riscv64-linux-gnu-ld: ../../fs/fs.o: in function `get_fs_type':
/home/jyt/os24fall-stu/src/lab6/fs/fs.c:44:(.text.get_fs_type+0x24): undefined reference to `memcmp'
riscv64-linux-gnu-ld: /home/jyt/os24fall-stu/src/lab6/fs/fs.c:46:(.text.get_fs_type+0x50): undefined reference to `memcmp'
riscv64-linux-gnu-ld: ../../fs/fs.o: in function `file_open':
/home/jyt/os24fall-stu/src/lab6/fs/fs.c:59:(.text.file_open+0x68): undefined reference to `strlen'
make[1]: *** [Makefile:3: all] Error 1
make[1]: Leaving directory '/home/jyt/os24fall-stu/src/lab6/arch/riscv'
make: *** [Makefile:22: all] Error 2
jyt@fine:~/os24fall-stu/src/lab6$
```

include > C string.h > ...	user > C string.h > ...
1 <code>#ifndef __STRING_H__</code>	1 <code>#ifndef __STRING_H__</code>
2 <code>#define __STRING_H__</code>	2 <code>#define __STRING_H__</code>
3	3
4 <code>#include "stdint.h"</code>	4 <code>int strlen(const char *str);</code>
5	5
6 <code>void *memset(void *, int, uint64_t);</code>	6 <code>#endif</code>
7	
8 <code>#endif</code>	

实现输入部分的代码后，并不能进行交互，键盘输入不显示。先通过打印提示信息，发现确实通过系统调用read，后续用同样的方法确定了可以进入 `stdin_read` 和 `uart_getchar`。

<pre>...mm_init done! ...setup_vm_final done. ...task_init done! 2024 ZJU Operating System Set [PID = 1 PRIORITY = 7 COUNTER = 7] switch to [PID = 1 PRIORITY = 7 COUNTER = 7] [trap.c,98,trap_handler] write hello, stdout! [trap.c,98,trap_handler] write hello, stderr! [trap.c,98,trap_handler] write SHELL > [trap.c,95,trap_handler] read</pre>	<pre>case 63: // sys_read Log6("read"); regs->a0 = sys_read(regs->a0, case 64: // sys_write Log6("write"); regs->a0 = sys_write(regs->a0, break;</pre>
--	--

参照实验指导找到了问题：内联汇编生成了非预期的指令序列（比如 `a5->a7->a5`，这样 `a5` 的值就被覆盖了），原因是在内联汇编的最后一个破坏描述部分没有说明 `a0-a7` 是会被破坏的（即让编译器不要使用 `a0-a7` 作为临时变量）


```

ASM vmlinux.asm > ...
2137 ffffffff00020150: 00078893 10 07, 112(30)
0 references
2138 ffffffff0002015e4: 00078893 mv a7,a5
0 references
2139 ffffffff0002015e8: 00070813 mv a6,a4
0 references
2140 ffffffff0002015ec: 00068513 mv a0,a3
0 references
2141 ffffffff0002015f0: 00060593 mv a1,a2
0 references
2142 ffffffff0002015f4: 00058613 mv a2,a1
0 references
2143 ffffffff0002015f8: 00050693 mv a3,a0
0 references
2144 ffffffff0002015fc: 00080713 mv a4,a6
0 references
2145 ffffffff000201600: 00088793 mv a5,a7
0 references
2146 ffffffff000201604: 00000073 ecall

```

修改后正常响应:

```

22 : [eid] "r" (eid), [fid] "r" (fid),
23 : [arg0] "r" (arg0), [arg1] "r" (arg1), [arg2] "r" (arg2),
24 : [arg3] "r" (arg3), [arg4] "r" (arg4), [arg5] "r" (arg5)
25 : "memory", "a0", "a1", "a2", "a3", "a4", "a5", "a6", "a7"
26 );
27 return rtn;

```

问题 输出 调试控制台 终端 端口

```

[vfs.c,21,stdin_read] stdin_read called: file 0xffffffff0002d3000, buf 0xffffffffe8, len 1
[vfs.c,8,uart_getchar] uart_getchar called

test
SHELL > [trap.c,95,trap_handler] read
[syscall.c,35,sys_read] sys_read
[vfs.c,21,stdin_read] stdin_read called: file 0xffffffff0002d3000, buf 0xffffffffe8, len 1
[vfs.c,8,uart_getchar] uart_getchar called

```

输入时，每次输入都会显示两次字母（下图1），调用输出发现read结束后就已经将字母打印到屏幕上了（下图2），逐步检查过去发现是 `trap_handler()` 中没有为case加上 `break` 语句：

```

...task_init done!
2024 ZJU Operating System
Set [PID = 1 PRIORITY = 7 COUNTER = 7]
switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
hello, stdout!
hello, stderr!
SHELL > eecchhoo tteesstt
test
SHELL >

```

```

printf(YELLOW "SHELL > " CLEAR);
while (1) {
    read(0, read_buf, 1);
    write(1, "|", 1);
    if (read_buf[0] == '\n') {
        write(1, "\n", 1);
    } else if (read_buf[0] == 0x7f) {
        if (char_in_line > 0) {
            write(1, "\b \b", 3);
            char_in_line--;
        }
    }
}

```

```

Boot HART MEDELEG           : 0x0000000000f0b509
...setup_vm done.
...buddy_init done!
...mm_init done!
...setup_vm_final done.
...task_init done!
2024 ZJU Operating System
Set [PID = 1 PRIORITY = 7 COUNTER = 7]
switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
hello, stdout!
hello, stderr!
SHELL > e|

```

4 心得体会

代码不多，报错不少...报错记录占了报告的一半，写得很痛苦。

比较幸运的一点是，写实验较迟，估计是很多人报过类似的错误，实验指导里已经加入了“检查内联汇编是否生成非预期的指令序列”的指导说明，极大的减少了debug的时间，少掉了很多头发。