浙江大學

课程名称：	操作系统原理与实践
题　　目：	Lab6: VFS & FAT32 文件系统
授课教师：	申文博
助　　教：	王鹤翔、陈淦豪、许昊瑞
姓　　名：	潘潇然
学　　号：	3220106049
地　　点：	32舍367

一、实验过程

准备工作

1. 禁用LOG，在根目录的Makefile中添加

```
LOG := 1
CFLAG    :=  $(CF) $(INCLUDE) -DTEST_SCHED=$(TEST_SCHED)  -DLOG=$(LOG)
```

同时在 `include/printk.h` 中根据LOG值进行判断，从而使得在编译时加上 `LOG=0` 就可以禁用log输出

```
#if LOG
#define Log(format, ...) \
    printk("\33[1;35m[%s,%d,%s] " format "\33[0m\n", \
        __FILE__, __LINE__, __func__, ## __VA_ARGS__)
#else
#define Log(format, ...);
#endif
```

2. 之后从仓库同步文件，这里记得要将 `user` 文件夹的全部内容都用本次lab6文件夹下的内容进行替换
3. 修改 `proc.c`，将lab5中新设置的 `nr_tasks` 变量设置为2（实际上lab5已经是2了）
4. 由于增加了 `fs` 文件夹，因此需要在 `arch/riscv` 下的Makefile中添加相关编译产物，修改第3行为

```
${LD} -T kernel/vmlinux.lds kernel/*.o ../../init/*.o ../../lib/*.o ../../fs/*.o
../../user/uapp.o -o ../../vmlinux
```

文件数据结构添加与初始化

1. 首先在 `task_struct` 中添加指向文件表的指针，以下为添加后的结构体

```
struct task_struct {
  uint64_t state;     // 线程状态
  uint64_t counter;   // 运行剩余时间
  uint64_t priority;  // 运行优先级 1 最低 10 最高
  uint64_t pid;       // 线程 id

  struct thread_struct thread;
  uint64_t *pgd;  // 用户态页表
  struct mm_struct mm;
  struct files_struct *files;
};
```

2. 完成函数 `file_init`，对 `stdout/err/in` 进行初始化。首先我们根据 `files_struct` 结构体的大小分配对应空间。接下来，我们分别处理 `stdin/out/err`，对应的fd分别为0/1/2。我们将 `opened` 设为1，`cfo` 设置为0，将 `stdin` 的权限设置为可读，`read` 设置为我们在后续实现的 `stdin_read`。其余两个权限设置为可写，同时 `write` 设置为后续实现的 `stdout_write` 与 `stderr_write`。最后，我们将其余未使用的文件的 `opened` 字段为0。

```
struct files_struct *file_init() {
  // todo: alloc pages for files_struct, and initialize stdin, stdout, stderr
  struct files_struct *ret;
  uint64_t size = sizeof(struct files_struct);
  uint64_t pages = (size + PGSIZE - 1) / PGSIZE;
  printk("file_init: size = %d, pages = %d\n", size, pages);
  ret = (struct files_struct *)alloc_pages(pages);

  // stdin
  ret->fd_array[0].opened = 1;
  ret->fd_array[0].perms = 1;
  ret->fd_array[0].cfo = 0;
  ret->fd_array[0].lseek = NULL;
  ret->fd_array[0].write = NULL;
  ret->fd_array[0].read = stdin_read;

  // stdout
  ret->fd_array[1].opened = 1;
  ret->fd_array[1].perms = 2;
  ret->fd_array[1].cfo = 0;
  ret->fd_array[1].lseek = NULL;
  ret->fd_array[1].write = stdout_write;
  ret->fd_array[1].read = NULL;

  // stderr
```

```
  ret->fd_array[2].opened = 1;
  ret->fd_array[2].perms = 2;
  ret->fd_array[2].cfo = 0;
  ret->fd_array[2].lseek = NULL;
  ret->fd_array[2].write = stderr_write;
  ret->fd_array[2].read = NULL;

  for (uint64_t i = 3; i < MAX_FILE_NUMBER; i++) ret->fd_array[i].opened = 0;

  return ret;
}
```

3. 完成后在 `proc.h` 的 `task_init` 函数中调用以完成对初始化。

```
void task_init() {
    ...
    task[0]->files = file_init();
    for (int i = 1; i < nr_tasks; ++i) {
        ...
        do_mmap(&task[i]->mm, USER_END - PGSIZE, PGSIZE, 0, PGSIZE,
            VM_READ | VM_WRITE | VM_ANON);
        task[i]->files = file_init();
}
```

处理stdout/err的写入

1. 首先修改先前实验中完成的 `sys_write` 的函数。首先获取 `fd` ， `buf` ， `count` 这三个在后续需要使用的变量。之后根据 `fd` 获取对应文件，对比文件是否open，权限是否正确，若检查通过后则调用 `write`

```
void sys_call(struct pt_regs *regs) {
  //   printk("syscall %d\n", regs->s[17]);
  switch (regs->s[17]) {
    case SYS_WRITE:
      sys_write(regs);
      break;
    case SYS_GETPID:
      sys_getpid(regs);
      break;
    case SYS_CLONE:
      // do_fork(regs);
      do_cow_fork(regs);
      break;
    case SYS_READ:
```

```
        sys_read(regs);
        break;
      default:
        Err("not support syscall id = %d", regs->s[17]);
    }
    regs->sepc += 4;
}
uint64_t sys_write(struct pt_regs *regs) {
    uint64_t fd = regs->s[10];
    char *buf = (char *)regs->s[11];
    uint64_t count = regs->s[12];
    struct file *target_file = &current->files->fd_array[fd];
    if (target_file->opened == 0) {
      printk("file not opened\n");
      return ERROR_FILE_NOT_OPEN;
    } else if (target_file->perms & FILE_WRITABLE)
      return target_file->write(target_file, buf, count);
}
```

2. 之后完成 `stderr_write`，此函数和 `stdout_write` 一样，我们只要直接通过printk进行串口输出即可

```
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);
}

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);
}
```

处理stdin的读取

1. 在 `sbi.h` 中加入以下内容，此函数用于实现读取终端输入

```
struct sbiret sbi_debug_console_read(uint64_t num_bytes, uint64_t base_addr_lo,
                                     uint64_t base_addr_hi);
```

2. 在 `sbi.c` 中完成

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

3. 最后利用 `uart_getchar` 完成 `stdin_read`，就完成啦

```
int64_t stdin_read(struct file *file, void *buf, uint64_t len) {
  // todo: use uart_getchar() to get `len` chars
  for (int i = 0; i < len; i++) {
    ((char *)buf)[i] = uart_getchar();
  }
}
```

4. 但此时运行会发现缺少 `memcmp` 和 `strlen`，因此需要在根目录下的 `./lib/string.c` 和 `./include/string.h` 补充相关内容
   - `./include/string.h`

```
#ifndef __STRING_H__
#define __STRING_H__

#include <stddef.h>

#include "stdint.h"

void *memset(void *, int, uint64_t);
void *memcpy(void *str1, const void *str2, size_t n);
int memcmp(const void *cs, const void *ct, size_t count);

static inline int strlen(const char *str) {
  int len = 0;
  while (*str++) len++;
  return len;
```

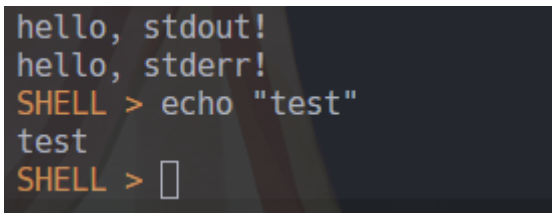
```
}

#endif
```

- `./lib/string.c`

```
#include "string.h"

#include <stddef.h>

#include "printk.h"
#include "stdint.h"

void *memset(void *dest, int c, uint64_t n) {
  char *s = (char *)dest;
  for (uint64_t i = 0; i < n; ++i) {
    s[i] = c;
  }
  return dest;
}

void *memcpy(void *str1, const void *str2, size_t n) {
  for (int i = 0; i < n; i++) {
    ((char *)str1)[i] = ((const char *)str2)[i];
  }
  return str1;
}

int memcmp(const void *cs, const void *ct, size_t count) {
  const unsigned char *su1, *su2;
  int res = 0;
  for (su1 = cs, su2 = ct; 0 < count; ++su1, ++su2, count--)
    if ((res = *su1 - *su2) != 0) break;
  return res;
}
```

二、实验结果

到此，一切的工都已准备完毕，我们可以`make run LOG=0`运行，如下图所示



我们可以正确使用 `write` 输出内容，同时 `echo` 指令可以正常使用，读取终端输出，并输出

三、实验心得

这次实验代码本身部分相对容易得多，主要遇到的问题是之前第一次下载仓库新文件的时候，内容好像大体还是去年的版本，后来第二次clone更新后的，复制进原项目的时候出现了些小问题，出现了奇奇怪怪的报错，比如 `$(OBJDUMP) -S ...` 出现报错 `-S not found`，后来才发现是makefile没有使用最新的。

到此，操作系统实验就到此结束了，感谢老师和三位助教这学期以来的付出与指导，学完这门课也收获良多，祝老师和三位助教未来一帆风顺，心想事成！