操作系统内核设计报告

中国院大学
吴亦泽，吕恒磊，王东宇

一、设计思路

操作系统作为计算机硬件与软件的交互部分，有优化和便捷计算机使用的功能，操作系统的设计与计算机的性能也有着密不可分的关系。本次比赛中，我们小队设计的操作系统基于 RISC-V 指令集架构，实现了系统初始化、中断和例外、I/O、进程管理、内存管理、文件系统等一系列基本功能，本部分将描述我们在实现操作系统的各个功能时的设计思路。

首先是系统的启动和初始化。这部分需要完成的工作就是内核的运行环境，将内核加载到指定位置，然后开始运行内核。内核启动后需要做的就是各个功能的初始化，包括进程管理初始化、中断例外初始化、系统调用初始化、内存管理初始化、SD 卡驱动初始化等内容，最后通过进程调度进入第一个进程开始运行。

中断和例外功能的初始化部分在系统启动后完成，包括例外处理入口地址设置、中断号与对应中断处理接口定义、系统调用号与对应系统调用接口定义、修改特权寄存器开启中断等。触发中断时，系统从用户态陷入内核态，进入例外处理函数。用户态与内核态切换时需要保存、恢复用户态上下文，保证两种状态下的运行不会相互干扰。例外处理函数会根据中断号跳转到相应的专用中断处理函数运行，完成相应的处理逻辑。

进程管理部分要完成包括进程管理初始化、进程调度、睡眠与唤醒、克隆子进程、进程等待、阻塞与唤醒、进程退出、获取进程 ID 等多种功能。所有进程管理的功能都是建立在进程管理块（PCB）的基础上的，我们先根据各个功能的需求设计了 PCB 的结构，然后根据 PCB 完成各个功能的实现。具体的功能实现内容较多，放在实现描述部分详细介绍。

内存管理功能包括页表设计、虚地址访存的开启、页分配与回收、页错误处理等。我们使用 RISC-V 标准的 SV39 内存管理方法，页表格式与 SV39 标准一致。虚地址访存的开启在系统启动过程中完成，会在实现描述中具体描述开启方法。为了避免过于频繁地查找空闲页，以及更容易地再次使用被释放掉的页，我们设置释放页列表，释放掉的页会加入表中，下次申请页时可以直接从表中获取，而不获取新的页。页错误处理会在发生各种页错误时发生，与其他中断使用同一个统一中断处理入口，具体方法在实现描述部分。

执行文件解析即将 elf 文件导入内存。首先找到 elf 可执行文件的 elf 头，据此一一找到各个程序头，将各程序头依次分配页表并将数据存入内存。

文件系统采用 FAT32 格式，实现了文件系统识别及初始化、管道、文件打开关闭、文件读写、创建目录、切换目录、获取当前目录、打印文件信息、链接、挂载、映射等一系列功能。这一部分是我们的操作系统中最为复杂的部分，完成的功能最多。主要实现思路就是根据 FAT32 文件系统的标准，读取 SD 卡的内容，判断卡中文件系统类型。确定 SD 卡中是 FAT32 文件系统后，依次读取卡中的各个测试文件并运行。关于这些操作的详细处理方式，在实现描述部分会详细介绍。

二、实现描述

本部分将具体描述我们的操作系统实现逻辑，

1. 启动与初始化

系统的启动部分代码主要在/arch/riscv/kernel 目录下。我们将官方发布的 rustsbi 作为内核的头部，简单地将芯片启动，然后跳转到 0x80020000 地址处运行，这里也是我们内核真正的起始地址。起始地址指向的是 head.S 文件的 ENTRY(_start)位置，该部分会完成设置 global pointer，设置 SIE、SIP 等特权寄存器值，清空 bss 段，设置栈指针等工作，为后面内

核的运行配置好环境。完成这些工作后，内核会跳转到 boot.c 中的 boot_kernel 函数。这个函数里会设置虚存相关配置，并通过特权寄存器开启虚地址访存方式，最后跳转到初始化相关部分。

这里的跳转过程较为独特，我们并没有简单地使用 jump 指令跳转。在我们的设想中，内核启动后应自动运行一个用户态进程 shell，并在这个进程中执行各个测试样例。进程 shell 也应该放在硬盘中，作为一个用户应用被内核访问、运行。但由于我们无法修改云端评测机上 SD 卡的内容，这样的执行方式显然不可行。于是我们为了模拟类似的效果，将内核和用户程序 shell 分别编译为 main 和 shell 两个 elf 文件后，使用 elf2char 工具将他们以字符串的形式存放在 payload.c 中，跳转时只要将字符指针转换为函数指针，跳转至该字符串的开头地址处作为指令字段执行即可。后面需要运行 shell 程序时，就可以从这个字符串的固定位置将 shell 作为用户程序加载出来即可。

初始化部分在 start.S 的 ENTRY(_start)处开始，由于这里运行环境与之前发生了变化，为保证正确性，我们重新设置了 global pointer、bss 段、栈指针、特权寄存器等一系列运行环境，最后跳转到/int 目录下的 mian.c 运行。main 函数里所有的工作就是初始化，以及跳转到第一个进程——shell 进程中运行。初始化的内容包括进程管理初始化、例外初始化、系统调用初始化、内存管理初始化、SD 卡驱动初始化、屏幕输出初始化等，这些初始化的具体内容会放在各个部分描述。完成初始化后，设置特权寄存器开启中断，系统就会自动通过第一个时钟中断完成进程调度，进入在进程管理初始化时设置好的第一个进程 shell。至此，系统的启动和初始化就完成了。

2. 中断

中断的初始化在/kernel/irq 目录下的 irq.c 文件中，init_exception 函数，在系统初始化时被调用。我们定义了一个指针数组，用于存放各个中断处理函数地址，下标就是中断号，初始化函数的功能就是给这个数组赋值。除了专用中断处理函数，我们还需要设定统一中断处理入口，作为中断发生时的硬件自动跳转地址。我们的统一中断处理入口在/arch/riscv/kernel 目录下的 entry.S 文件内 ENTRY(exception_handler_entry)处，这里会完成用户态上下文保存、特权寄存器设置等工作，并将自己的函数返回地址设置为中断返回函数的入口，最后跳转到 irq.c 的 interrupt_helper 函数。该函数会根据作为参数传递进来的特权寄存器的值，调用前面提到的函数指针数组项，进入对应的中断处理函数。中断处理完成后，内核会返回到之前设置的中断返回函数，恢复用户态上下文，调用 sret 指令，将内核切换到用户态运行。

我们主要处理了时钟中断、系统调用、页错误几种中断。时钟中断的处理函数是 irq.c 中的 handle_int，该函数会检查阻塞队列，唤醒睡眠结束的进程，重置时钟中断计数器，最后进行进程调度。系统调用稍显复杂，因为不同的系统调用有各自不同的处理函数，我们也使用了一个指针数组，在系统初始化部分做类似中断初始化时的接口设置工作。用户会调用不同的系统调用函数，这些函数都会调用同一个函数 invoke_syscall，。该函数第一个参数是系统调用号，后面可以继续添加最多 5 个系统调用参数，并将它们存放在固定的寄存器中。设置好参数后调用 ecall 指令，发起中断，陷入内核态处理。发起中断后的处理与其他中断一样，直到统一中断处理函数调用专门中断处理函数。系统调用的处理函数是/kernel/syscall 目录下的 syscall.c，该函数能获取 invoke_syscall 设置好的系统调用参数，并将其作为参数，调用系统初始化时设置好的系统调用处理函数。处理完成后，系统调用的返回过程与其他中断一致，不再赘述。

3. 进程管理

进程管理相关内容主要在/kernel/sched 目录下的 sched.c 文件中，头文件是/include/os 目录下的 sched.h 文件。

PCB 结构体保存进程的内核栈指针 kernel_sp、用户栈指针 user_sp，这两个域被用来保证切换用户态、内核态，以及上下文切换时栈指针的位置不会发生错误；内核栈基址 kernel_stack_base 和用户栈基址 user_stack_base 保存栈的基地址位置；list 域保存进程在队列中的前驱、后继指针；wait_list 域保存进程的等待队列，在等待和退出功能中会用到；pid 域保存进程的 ID；status 域保存进程的状态，包括阻塞、正在运行、准备运行、僵尸进程、已退出等状态；priority 域保存进程的优先级，在调度时会根据准备队列中的优先级来决定调用进程的顺序、频率等；cursor_x 和 cursor_y 保存进程在屏幕上输出的光标位置，保证进程切换时光标位置正确；parent 域保存父进程信息，在克隆进程时会被用到。

进程管理的初始化在系统初始化时完成。首先我们设置 current_running 指针，指向当前正在运行的进程。为了正确性，我们添加一个没有实际意义的 0 号进程 pid0，并在初始化时将 current_running 指针指向该进程。在之后通过时钟中断调度到真正的第一个进程时，pid0 会被调度走，并且不再进入准备队列。除了添加 pid0，我们还在初始化时加入了我们操作系统自动执行的用户进程 shell，使它能在初次时钟中断时被调度到。加入 shell 进程时，我们会初始化其栈信息，设置 PCB 相关参数，并将其加入准备队列，等待调度。

我们设计的进程调度方法是优先级调度。每个进程在初始化时会设定进程优先级，刚加入准备队列时，或者运行中的进程被调度走，重新加入准备队列时设置调度优先级等于进程优先级。每次调度时从队列中选取调度优先级最高的进程运行，其他进程调度优先级提升一级。如此一来就能实现高优先级的进程能够更频繁地被调度到。优先级调度的具体算法在 do_scheduler 函数中完成，该函数会判断当前正在运行的进程状态，决定是否将其加入准备队列，然后从准备队列里用获取被调度到的进程。处理完进程队列后，函数会调用 /arch/riscv/kernel 目录下的 entry.S 文件中的汇编函数 switch_to。该函数会将内核态上下文保存在内核栈上，然后根据输入参数将栈指针移到被调度到的下一个进程的内核栈上，恢复新进程的内核上下文。这样，就完成了两个进程之间的切换。

进程克隆的处理函数是 do_clone，其基本操作就是复制父进程的信息到子进程中，但有部分信息不能完全复制。克隆进程的系统调用对父进程和子进程的返回值不同，所以该部分不能完全复制，需要对父进程和子进程进行识别后返回。另外，父进程是正在运行的进程，子进程被复制后应该处于准备运行状态进入准备队列。根据我们的进程调度方法，被调度走的进程，栈上应该保存其内核上下文，二此时父进程还处在运行状态，栈上没有内核上下文，也就没法将这些信息复制给子进程。于是我们需要在子进程的栈上根据父进程的寄存器制造一段内核上下文，以免子进程被调度到时发生错误。

进程等待功能由函数 do_wait4 完成，需要遍历全部进程列表，查找输入参数对应的子进程，将该子进程唤醒，设置为准备运行状态并加入准备队列，之后将当前进程阻塞，并置入该子进程的等待队列。当该子进程退出时，会唤醒等待队列中所有进程。

进程睡眠功能在函数 do_nanosleep 实现，可以设定睡眠时长，将当前进程阻塞，并加入全局阻塞队列。每次时钟中断时，会查看阻塞队列，并将睡眠时间足够的进程唤醒并加入准备队列。

进程阻塞与唤醒的具体操作在函数 do_block 和 do_unblock 中，作为工具函数被其他函数调用，主要操作就是修改进程状态为阻塞或准备运行，并加入阻塞队列或准备队列.

进程退出在函数 do_exit 中实现。该函数会遍历当前进程的等待队列，唤醒里面所有非退出状态的进程。最后，根据进程初始化时的退出模式选择进入僵尸状态，或者彻底退出，将进程设置为退出状态，释放进程申请的内存空间。

我们还在函数 do_exec 中实现了进程启动功能。该函数会在 PCB 列表中查找退出状态的表项，将新进程的信息初始化在这个表项中，为新进程申请内存空间，设置栈空间，并将其状态设置为准备运行，并加入准备队列。

函数 do_getpid 和 do_getppid 分别能够获取当前进程 ID 和父进程 ID 并返回，实现进程 ID 获取功能。

4. 内存管理

虚地址访存模式的开启在系统启动部分完成，具体代码在/arch/riscv/kernel 目录下的 boot.c 文件中，在 setup_vm 函数中完成。系统启动刚开始时，内核采用 32 位实地址访存方式，开启虚地址访存前需要先配置页表。除了要配置内核接下来运行时需要用到的地址段，我们也要把当前运行时正在用的 32 位实地址扩展到的 64 位虚地址也映射到实地址上，避免避免开启虚地址访存后紧接着的指令寻址失败。配置页表完成后，修改特权寄存器，开启虚地址访存即可。

内存管理的其他功能主要在/kernel/mm 目录下的 mm.c 和 ioremap.c 文件中实现。我们设置了 memCurr 指针，指向当前已分配的内存地址。另外有释放页列表，保存之前释放掉的内存页。分配页功能在 allocPage 函数中完成，先查找释放页列表，若表中有可用页，则直接使用，而不去申请新的页。若列表中没有可用页，则将 memCurr 指针向后推一个页的大小，并将这一个页的空间分配出去。释放页在 freePage 函数中完成，查询已申请页列表，将待释放地址对应的项移除，再将该项添加到已释放列表中，以便之后分配页时使用。

页错误处理在/kernel/irq 目录下的 irq.c 文件中实现。发生页错误时会触发中断，进入统一中断处理入口，最后调用函数 handle_pgfault 处理页错误。该函数会按页由大至小依次处理三级、二级、一级页表项，根据不同的错误码，查询并重新设置页表项中的控制位。若发生错误地址读、写、执行等行为，还会输出 Segmentation fault 错误信息。

5. 执行文件解析

执行文件解析即将 elf 可执行文件分配页表并读入内存，主体函数是位于 include/os/elf.h 的 load_elf 函数。首先根据 elf 文件结构找到 elf header，据此得到程序头表，依次遍历各个程序头。对于程序头数据，按照页大小为粒度进行虚页分配，当剩余数据长度不足一个页大小时，将尾部补零。若有.bss 段等不初始化内存空间或初始化为零的段（体现为当前分配页总大小小于 p_filesz 但大于 p_memsz），为其分配全零页。

6. 文件系统

文件系统部分的代码在/kernel/fat32 目录下，头文件是/include/os 目录下的 fat32.h。

我们定义了 fat_t 结构体类型，用于存储文件系统的各种基础信息；dentry_t 结构体类型，用于存储目录信息；pipe_t 结构体类型，用于存储管道信息；fd_t 结构体类型，用于存储文件描述符内容。

在系统初始化部分，完成 SD 卡驱动初始化之后，我们的文件系统就会开始进行初始化，执行 fat32.c 中的 fat32_init 函数。该函数会先读取 SD 卡的第一个扇区，并根据 FAT32 标准判断 SD 卡上的文件系统是否是 FAT32，不是的话会输出错误信息。接下来该函数会根据刚刚读到的内容获取文件系统的每扇区字节数、每簇扇区数、保留扇区数、fat 数、隐藏扇区数、总扇区数、fat 大小、根目录簇位置等信息，并一一填入 fat_t 结构体中。然后 fat32_init 函数会调用 init_inode 函数和 init_pipe 函数。这两个函数分别会获取根目录的元数据信息，以及初始化管道结构体，将所有管道读写均置为无效。最后释放读取 SD 卡第一个扇区占用的内存空间，文件系统初始化完成。

打开文件在函数 fat32_open 中实现。该函数根据传入的路径字符串，查找对应的目录。先根据路径是否以字符'/'开头，判断要从根目录开始查找还是从当前工作目录开始查找。然后按字符'/'作为分隔符查找目录名。若最后查到对应文件，则读取文件元数据，将各项信息写入对应的 dentry_t 结构体中，返回文件描述符编号；若该文件不存在，则创建新的文件，初始化其文件元数据信息，写入 dentry_t 结构体中，返回文件描述符编号；若打开文件失败，则返回-1。

关闭文件在函数 fat32_close 中实现，需要传入文件描述符作为参数。该函数会查找当

前进程的文件描述符列表，若未找要关闭的文件，则关闭失败，返回 1；若查到需要关闭的文件，则将当前进程的该文件描述符释放。若该文件使用了管道功能，同时需要修改对应管道信息。

文件读写功能分别由函数 fat32_read 和函数 fat32_write 完成。读文件时先根据文件描述符判断对应文件打开状态，无对应文件或未打开则读文件失败，返回-1。若该文件的 pipe 开启，则直接调用 pipe_read 函数。若为其他文件，则将文件总长度和传入的读取长度去最小值，确定实际需要读取的长度，然后从 SD 卡中逐个扇区进行读取，直到读取到足够长的内容。最后释放读取 SD 卡占用的页，返回实际读取的长度。写文件时也需要先判断文件是否存在，是否已打开。若写的目标是标准输出流，则可以直接向屏幕输出。若写文件 pipe 开启，则调用 pipe_write。若未开启 pipe，则向 SD 卡中写入 buffer 内容即可。

三、问题与解决方案

1.虚存开启

K210 支持的特权级为 1.9，与目前流行的 1.11 有所不同。关于虚存的区别主要在于: satp 寄存器定义的区别、sfence.vma 指令的执行、sstatus 的 SUM 位（1.9 中称为 PUM 位）的含义区别。

其中，前两个问题已经通过 RustSBI 解决。K210 相应的页表寄存器名称为 sptbr。当访问 satp 寄存器时，K210 认为这是一条非法指令，触发异常，RustSBI 在 M 态将指令进行转换，实际执行的是写 sptbr 寄存器和 sfence.vm。如果只修改 satp 寄存器而不执行 sfence.vma，就不会真的把 satp 寄存器的值（以软件形式存储）赋给实际存在的 sptbr 寄存器，也就达不到开虚存的效果。

由于了解到 K210 没有 sfence.vma 指令，只有 sfence.vm 指令，因此一开始我在设置 satp 寄存器之后执行 sfence.vm 指令来刷新 TLB，但这样并不会真正的打开虚存，也导致内核不能在虚拟地址上继续运行。

```
else if ins & 0xFE007FFF == 0x12000073 { // sfence.vma instruction
  // There is no `sfence.vma` in 1.9.1 privileged spec; however there is a `sfence.vm`.
  // For backward compability, here we emulate the first instruction using the second one.
  // sfence.vma: | 31..25 funct7=SFENCE.VMA(0001001) | 24..20 rs2/asid | 19..15 rs1/vaddr |
  //              14..12 funct3=PRIV(000) | 11..7 rd, =0 | 6..0 opcode=SYSTEM(1110011) |
  // sfence.vm(1.9):  | 31..=20 SFENCE.VM(000100000100) | 19..15 rs1/vaddr |
  //              14..12 funct3=PRIV(000) | 11..7 rd, =0 | 6..0 opcode=SYSTEM(1110011) |
  // discard rs2 // let _rs2_asid = ((ins >> 20) & 0b1_1111) as u8;
  // let rs1_vaddr = ((ins >> 15) & 0b1_1111) as u8;
  // read paging mode from satp (sptbr)
  let satp_bits = satp::read().bits();
  // bit 63..20 is not readable and writeable on K210, so we cannot
  // decide paging type from the 'satp' register.
  // that also means that the asid function is not usable on this chip.
  // we have to fix it to be Sv39.
  let ppn = satp_bits & 0xFFF_FFFF_FFFF; // 43..0 PPN WARL
  // write to sptbr
  let sptbr_bits = ppn & 0x3F_FFFF_FFFF;
  unsafe { llvm_asm!("csrw 0x180, $0"::"r"(sptbr_bits)) }; // write to sptbr
  // enable paging (in v1.9.1, mstatus: | 28..24 VM[4:0] WARL | ... )
  let mut mstatus_bits: usize;
  unsafe { llvm_asm!("csrr $0, mstatus":"=r"(mstatus_bits)) };
  mstatus_bits &= !0x1F00_0000;
  mstatus_bits |= 9 << 24;
  unsafe { llvm_asm!("csrw mstatus, $0"::"r"(mstatus_bits)) };
  // emulate with sfence.vm (declared in privileged spec v1.9)
  unsafe { llvm_asm!(".word 0x10400073") }; // sfence.vm x0
  // ::"r"(rs1_vaddr)
  mepc::write(mepc::read().wrapping_add(4)); // skip current instruction
else if mstatus::read().mpp() != MPP::Machine { // invalid instruction  can't emulate  raise tr
```

1.11 特权级中，sstatus 的 SUM 位置 1 表示允许 S 态访问 U 态的页面，但 1.9 中恰恰相反，当 PUM 位（和 SUM 是同一位，只是名称不同）置 0 时才允许。QEMU 支持特权级 1.11，故如果没注意这里的区别，则相关的程序在 QEMU 上没有问题，在 K210 上却无法运行。

```
172  void init_pcb_stack(
173      ptr_t pgdir, ptr_t kernel_stack, ptr_t user_stack, ptr_t entry_point, unsigned char* argv[],
174      pcb_t *pcb)
175  {
176      regs_context_t *pt_regs =
177          (regs_context_t *)(kernel_stack - sizeof(regs_context_t));
178
179      reg_t gp, ra;
180
181      gp = __global_pointer$;
182      ra = entry_point;
183
184      pcb->kernel_sp = (reg_t)(kernel_stack - sizeof(regs_context_t) - sizeof(switchto_context_t));
185      pcb->user_sp = (reg_t)user_stack;
186      set_stack_base(pcb, kernel_stack, user_stack);
187      pcb->pgdir = pgdir;
188
189      reg_t *regs = pt_regs->regs;
190      regs[3] = gp;  //gp
191      regs[4] = pcb; //tp
192      // regs[10] = argc; //a0=argc
193      // regs[11] = (ptr_t)argv; //a1=argv
194      pt_regs->sstatus = SR_SUM; //enable supervisor-mode userpage access
195
196  #ifdef K210
197      pt_regs->sstatus = 0; //enable supervisor-mode userpage access for K210
198  #endif
199
200      pt_regs->sepc = ra;
201      unsigned mode = SATP_MODE_SV39, asid = pcb->pid, ppn = kva2pa(pgdir) >> NORMAL_PAGE_SHIFT;
202      pt_regs->satp = (unsigned long)(((unsigned long)mode << SATP_MODE_SHIFT) | ((unsigned long)asid << SATP_ASID_SHIFT) | ppn);
203
204      switchto_context_t *switch_to_reg =
205          (switchto_context_t *)(kernel_stack - sizeof(regs_context_t) - sizeof(switchto_context_t));
206      switch_to_reg->regs[0] = &ret_from_exception;
```

2.中断号

特权级 1.11 有单独的缺页中断号，但 K210 将缺页归纳到访存异常中断中。因此，要实现 K210 上程序的写时分配，必须将处理 PageFault 的函数挂到 5 号（Load Access Fault）和

7 号（Store/AMO Access Fault）异常，而非 1.11 中的 13 和 15 号。这也是 K210 和 QEMU 模拟器的一个重大区别，且涉及关键的缺页异常处理。

```
void init_exception()
{
    /* TODO: initialize irq_table and exc_table */
    /* note: handle_int, handle_syscall, handle_other, etc.*/
    irq_table[IRQC_U_SOFT] = &handle_software;
    irq_table[IRQC_S_SOFT] = &handle_software;
    irq_table[IRQC_M_SOFT] = &handle_software;
    irq_table[IRQC_U_TIMER] = &handle_other;
    irq_table[IRQC_S_TIMER] = &handle_int;
    irq_table[IRQC_M_TIMER] = &handle_other;
    irq_table[IRQC_U_EXT] = &handle_other;
    irq_table[IRQC_S_EXT] = &handle_sext;
    irq_table[IRQC_M_EXT] = &handle_other;

    exc_table[EXCC_SYSCALL] = &handle_syscall;
    exc_table[EXCC_INST_MISALIGNED] = &handle_other;
    exc_table[EXCC_INST_ACCESS] = &handle_other;
    exc_table[EXCC_BREAKPOINT] = &handle_other;
    #ifdef K210
    exc_table[EXCC_LOAD_ACCESS] = &handle_pgfault;
    exc_table[EXCC_STORE_ACCESS] = &handle_pgfault;
    #else
    exc_table[EXCC_LOAD_ACCESS] = &handle_other;
    exc_table[EXCC_STORE_ACCESS] = &handle_other;
    #endif
    exc_table[EXCC_INST_PAGE_FAULT] = &handle_pgfault;
    exc_table[EXCC_LOAD_PAGE_FAULT] = &handle_pgfault;
    exc_table[EXCC_STORE_PAGE_FAULT] = &handle_pgfault;
}
```

3.SD 卡驱动

SD 卡驱动的框架摘取自 xv6-K210 项目，其中的代码已经实现了 SPI 协议初始化和读写 SD 卡的每一个步骤。然而，用原始的驱动代码在我自己的 SD 卡上进行实验时，经常出现 SD 卡初始化失败或读写失败的情况，令我十分苦恼。

经过 debug 分析，发现 xv6-k210 中的 SD 卡驱动有些时候只等待一次 SD 卡的 CMD 回复，如果没等到想要的回应码，就退出并认为 SD 卡出错。实际上，应该循环向 SD 卡发送多次 CMD 命令，如果仍然得不到想要的回应才认为出错，否则可能 SD 卡的回复相比程序执行要更慢，并不是真的没给出正确回应，导致误报出错。

最常见的误报出错点是 CMD9 指令，它用于读取 SD 卡的 CSD 寄存器中的信息。原始的驱动代码只发送一次 CMD9，如果没有收到正确回复就报错，这样非常容易导致误报，我添加了一个 255 次的循环，解决了这个问题。（顺手也把 CMD10 读取 CID 寄存器的也加了循环，虽然这里不那么容易误报）。

```
static uint8 sd_get_csdregister(SD_CSD *SD_csd)
{
    uint8 csd_tab[18];
    uint8 result;
    uint32 index;
    index = 255;
    while (index--){
        /*!< Send CMD9 (CSD register) or CMD10(CSD register) */
        sd_send_cmd(SD_CMD9, 0, 0);
        /*!< Wait for response in the R1 format (0x00 is no errors) */
        if (sd_get_response() != 0x00) {
            // printk("here\n"); while(1);
            sd_end_cmd();
            return 0xFF;
        }
        if ((result = sd_get_response()) != SD_START_DATA_SINGLE_BLOCK_READ) {
            // printk("there\n");
            // printk("result is %d\n", result);
            sd_end_cmd();
        }
        else
            break;
    }
    if (index == 0){
        printk("error\n"); while(1);
    }
    /*!< Store CSD register value on csd tab */
```

此外，我发现如果一次写多个 SD 卡扇区，会将 FAT 表直接破坏。目前还不知道发生这

种情况的原因，在需要写多个扇区时，只能依次一个一个的写入。读取 SD 卡并没有类似的问题。

```
1797  /* read as many sectors as we can */
1798  void sd_read(char *buf, uint32_t sec)
1799  {
1800      sd_read_sector(buf, sec, READ_BUF_CNT);
1801  }
1802
1803  /* write as many sectors as we can */
1804  void sd_write(char *buf, uint32_t sec)
1805  {
1806      for (int i = 0; i < READ_BUF_CNT; ++i)
1807      {
1808          sd_write_sector(buf, sec, 1);
1809          buf += SECTOR_SIZE;
1810          sec++;
1811      }
1812  }
```

4.长目录无法识别

FAT32 的目录项分为短目录和长目录两类。目录项的格式再次不赘述，但我在构建长目录时遇到一些困难。当我完全按照格式构建好目录项并存入 SD 卡，甚至通过 Winhex 已经能识别出长目录时，在 Windows 上打开 SD 卡却始终不能显示正确的文件名。长文件名的短目录项中存有文件名的前 6 个字符和~X，而 Windows 显示的文件名就是 ABCDEF~1，不是我在长目录项中存储的文件名。

实际上，错误原因是长目录项的校验和没有设置对。为了确保长目录项和短目录项正确匹配，长目录项有一个校验和字段，该字段是通过这个文件短目录项的前 11 个字符（文件名+扩展名）来得到的。如果校验和不匹配，则系统不认为这些长目录和相应的短目录是匹配的，也就不会去显示存储在长目录项中的正确文件名了。修改这个 bug 后，在 Windows 系统上打开 SD 卡读卡器，可以看到长文件名的正确显示。

```
long_entries[j].sequence = (j == demand - 2)? (0x40 | (j + 1)) & 0x4f : (j + 1) & 0x0f;
long_entries[j].attribute = FILE_ATTRIBUTE_LONG;
long_entries[j].reserved = 0;
// checksum
uint8 sum = 0; uchar *fcb = &new_dentry;
for (int i = 0; i < 11; ++i)
    sum = (sum & 0x1 ? 0x80 : 0) + (sum >> 1) + *fcb++;
long_entries[j].checksum = sum;
long_entries[j].reserved2 = 0lu;
```

5.pipe read

在实现管道 pipe 时，我实现的是阻塞读、无阻塞写。一开始，我发现读端口总是会少读一个字符，拿测试样例来说，本应该先输出两个空格，再输出字符串内容，但我的 pipe 只能输出一个空格。

最起初，我以为空格输出在了某些句子的末尾，因为不可见而让我以为没输出。但我把测试源码的空格改成字符后，也是不能正常输出的，排除了这个情况。

之后我认为可能是串口输出的问题，因为两个进程同时向串口输出可能引发未定义的行为（实际上，同一时间确实只有一个进程在输出，所以这个想法肯定是错的）。通过对串口输出的分析，也排除了这个可能性。

实际上，原因是地址空间的问题。读进程进行 pipe_read 时，如果发现还没收到写进来的数据，则被阻塞并调用调度器，去继续执行其它进程。等到写进程将其唤醒，它就继续进行读。虽说读进程和写进程是亲子关系，但二者的页表共享发生在 clone 的时候。负责存储 pipe_read 的缓存是静态数组，故存储在用户栈中。clone 时，用户栈被复制，子进程拥有了一个自己的用户栈，也把缓存的地址映射到此处。读进程可能是在写进程试图从 pipe_write 系统调用返回时被调度的，而我设计的内核在切换进程时并不切换页表，只是在返回用户态

之前才切换页表，因此读进程正处在写进程的地址空间中，如果直接向缓存地址写数据，会写到写进程的栈中，引发错误。

```
600  /* read from pipe */
601  /* return read count */
602  int64 pipe_read(uchar *buf, pipe_num_t pip_num, size_t count)
603  {
604      uint64_t buf_kva = get_kva_of(buf, current_running->pgdir);
605      while (pipes[pip_num].r_valid != PIPE_INVALID && pipes[pip_num].top == pipes[pip_num].bottom){
606          do_block(&current_running->list, &pipes[pip_num].wait_list);
607          do_scheduler();
608      }
609      uint32_t readsize = min(pipes[pip_num].top - pipes[pip_num].bottom, count); // neq count
610      memcpy(buf_kva, pipes[pip_num].buff + pipes[pip_num].bottom, readsize); // maybe exceed?
611      pipes[pip_num].bottom += readsize;
612      return readsize;
613  }
```

PIPE 的 r_valid 和 w_valid：初始化时设置为 PIPE_ERROR=2。当进程关闭带有管道的文件描述符时，会把相应的 valid 值减 1。如果关闭的是读端口，表明自己已经准备写了，故把 w_valid 减 1，设置成 PIPE_VALID=1；如果关闭的是写端口，表明自己已经准备读了，故把 r_valid 减 1，设置成 PIPE_VALID=1。当两个进程都关闭了读（写）端口，w_valid(r_valid)的值变为 0，就表明写（读）已经不能继续进行。

该设计是为了控制阻塞读，当读端口发现 r_valid 已经不再有效，表明写端已经关闭了它的文件描述符。此时即使缓存里没有任何数据，也不能再等待了，而是应该直接返回 0。

```
600  /* read from pipe */
601  /* return read count */
602  int64 pipe_read(uchar *buf, pipe_num_t pip_num, size_t count)
603  {
604      uint64_t buf_kva = get_kva_of(buf, current_running->pgdir);
605      while (pipes[pip_num].r_valid != PIPE_INVALID && pipes[pip_num].top == pipes[pip_num].bottom){
606          do_block(&current_running->list, &pipes[pip_num].wait_list);
607          do_scheduler();
608      }
609      uint32_t readsize = min(pipes[pip_num].top - pipes[pip_num].bottom, count); // neq count
610      memcpy(buf_kva, pipes[pip_num].buff + pipes[pip_num].bottom, readsize); // maybe exceed?
611      pipes[pip_num].bottom += readsize;
612      return readsize;
613  }
```

四、总结

尽管在操作系统内核开发过程中遇到了一些问题，我们还是努力将其克服，完成了一个具备基本功能的系统内核，同时也加深了自己对操作系统的理解。接下来我们还会继续完善这个系统内核，使其功能更加完备，性能更加优良。