

Ecallfinal1_OS

概要

文档组成

 总体设计部分

 中断及异常部分

 内存管理部分

 进程部分

 同步原语

 硬盘驱动部分

 文件系统

 ELF解析

 感想与致谢

EcallFinalOS 总体设计

异常和中断

 中断入口和出口

 中断处理核心

 Syscall_execve

 Syscall_clone

 Syscall_mkdirat

 Syscall_wait4

 Syscall_times

 Syscall_getdents64

 Syscall_splice

 测试点（50分）

物理内存

 引言

 设计思路

 PMM

 PAGE

slab分配器

设计

slabB

SlabB

SlabAllocator

遇到问题

虚拟内存

PageTable

PageEntry

VirtualMemorySpace

VirtualMemoryRegion

HeapMemoryRegion

进程管理

Process

ProcessManager

进程切换

Futex

Futex应用场景

Futex系统调用

sys_futex函数定义

sys_futex函数参数

sys_futex函数行为

sys_futex函数的异常处理

futex具体实现

futexWait()

futexWake()

futexRequeue()

同步原语

硬盘驱动部分

文件系统

VFSM

FAT32文件系统

[open](#)
[fileObject](#)

[EXT4](#)

[介绍](#)

[ext4node](#)

[EXT4](#)

[ELF解析](#)

[致谢](#)

[蔡蕾](#)

[决定参加](#)

[速成操作系统的理论](#)

[最简单代码到内存管理](#)

[虚拟内存](#)

[感谢](#)

[郭伟鑫](#)

[心得与感谢](#)

概要

本操作系统为2024年CCSC操作系统内核赛的参赛作品，由南京航天航空大学计算机科学与技术学院的学生（郭伟鑫和蔡蕾）共同开发。

内核基于大赛初赛需求，基于RISCV架构开发。使用QEMU进行模拟，支持简单的linux系统调用。

文档组成

文档分为以下几个部分

总体设计部分

介绍了EcallFinal1OS整体的设计思路，及各个部分的框架流程。

中断及异常部分

中断是进程切换的核心部分，同时也是IO交互的核心，内核中几乎所有功能都依赖中断和异常的实现。

内存管理部分

内存管理主要分为物理内存和虚拟内存部分，离散式的内存管理使内存使用更加充分。此部分给出了我们内存管理的主要思想。

进程部分

进程是操作系统的核心部件，操作系统其他部件是为了进程而服务的，进程需要综合管理内核中其他模块。

同步原语

同步原语负责实现进程之间对冲突资源访问的问题

硬盘驱动部分

硬盘驱动负责从镜像文件中读取和写入数据，由于是在QEMU上开发，于是硬盘驱动主要是基于VirtIO实现的。

文件系统

文件系统负责管理文件，用于将程序载入系统，同时有许多关键设备被映射成为文件。

ELF解析

从文件中启动用户进程需要ELF解析文件

感想与致谢

这一部分是两人在一个多月的开发当中所得到的感想，已经本操作系统开发过程中受到了很多帮助，特此致谢

EcallFinalOS 总体设计

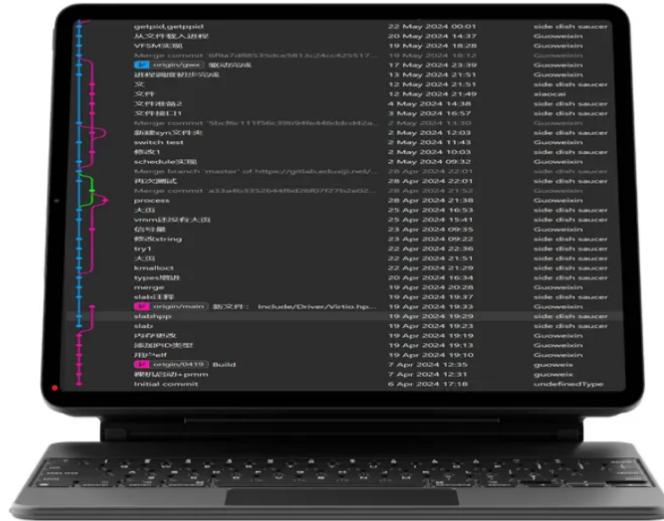
EcallFinalOS使用c++进行设计，总体采用了面向对象的编程方式，操作系统中由众多模块组成，每个模块均采用类来管理。内存，驱动，文件，进程，均使用C++中的类进行设计，同时创建管理器实体，每个管理器负责管理不同模块提供接口。

驱动，文件管理，基本库，内存，进程，同步原语，中断，系统调用，仅有部分移植其余均为自主开发或重构

构建系统内核 (从中断开始)

代码原創达70%以上

开发历时2年



图片加载失败

进程负责统筹管理其他模块，而其他模块如果是每个进程都拥有的，则由模块单体负责管理，如VMS,fileobject，信号量等。如果是仅需要使用，而不是每个进程都拥有的，则由模块管理器负责，如物理内存管理等。

- 同时我们在开发过程中引入了方便调试的工具Kout负责所有调试信息的输出，这个工具极大的方便的管理调试信息。
- 本操作系统尽可能的减少使用汇编，以减少移植到其他体系架构的成本，而使用汇编，也尽可能减少内嵌汇编的使用。

异常和中断

中断入口和出口

中断的入口和出口均由TrapEntry.S汇编负责，此汇编中有两个函数，分别负责中断的初步处理和中断的退出。

我们设定了一个TrapFrame用于保护现场,定义如下

```

1  struct RegisterContext // 通用寄存器的上下文
2  {
3      union // 使用联合体，这样既可以按名字访问寄存器，也可以按编号访问寄存器
4      {
5          RegisterData x[32];
6          struct // 按照Risc-V64规范定义的32个寄存器
7          {
8              RegisterData zero,
9                  ra, sp, gp, tp,
10             t0, t1, t2,
11             s0, s1,
12             a0, a1, a2, a3, a4, a5, a6, a7,
13             s2, s3, s4, s5, s6, s7, s8, s9, s10, s11,
14             t3, t4, t5, t6;
15         };
16     };
17
18     inline RegisterData& operator[](int index)
19     {
20         return x[index];
21     }
22 };
23
24 struct TrapFrame // 一次中断/异常所需要保存的上下文/帧
25 {
26     RegisterContext reg;
27     RegisterData status, // 原先的CPU状态
28         epc, // 原先的PC寄存器的值，表示程序执行的位置
29         tval, // 地址例外中出错的地址、发生非法指令例外的指令本身、其他异常为0
30         cause; // 中断/异常原因
31 };
32
33

```

汇编中的SAVE_ALL负责将当前的TrapFrame存到栈中，并且将栈指针传递给Trap.cpp的Trap函数而RESTORE_ALL则负责从sp中将TrapFrame恢复成为运行的进程。而这是我们调度进程的核心，传入一个TrapFrame传出另一个TrapFrame，这点将在进程处详细讲解。
而TrapEntry.S的另一个特点就是会判断当前中断是在谁处发生的，如果在用户态中断，则将TrapFrame存储到用户对应的内核的栈，防止爆了用户的栈。

中断处理核心

中断服务函数的核心则是在Trap.cpp中的Trap函数，这个函数在每次进入的时候记录一个是否要调度的变量（needSchedule），对传入的TrapFrame中的a7，判断当前应该响应什么服务，最后返回的时候查

看needSchedule变量去判断是否要进行调度。

这个函数最核心的思想是将所有中断和异常均使用相同的入口去处理，而使用switch统一管理，极大的简化了中断处理的流程，为后期处理syscall建立的基础

By:郭伟鑫

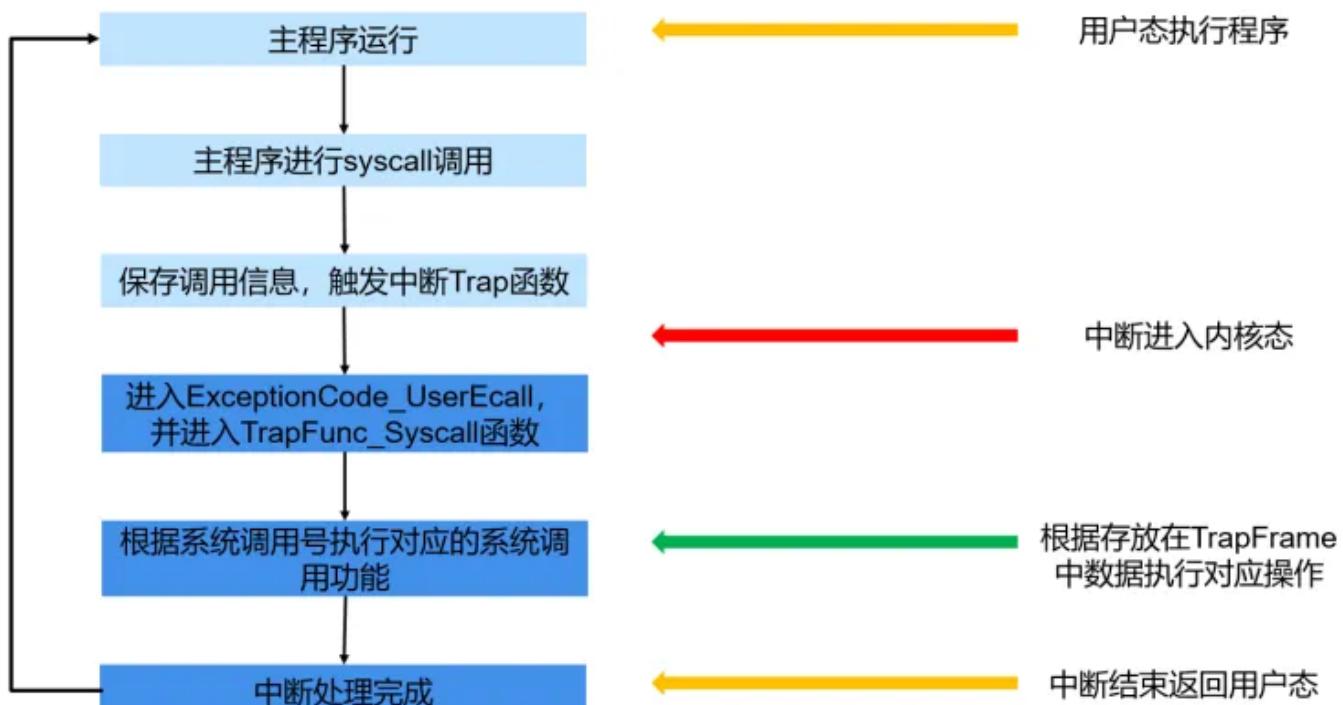
`syscall` 调用是操作系统中用户程序请求内核服务的机制。用户程序通过 `syscall` 触发特权模式的切换，使得系统能够安全地处理底层任务如文件操作、内存管理和进程控制。此过程涉及中断机制以及用户态和内核态之间的转换。

首先，用户程序通过特定的指令发起 `syscall` 调用。这条指令会生成一个软件中断或系统调用中断，使得处理器从用户态切换到内核态。用户态是指程序运行在普通的权限级别下，仅能访问受限的资源，而内核态则是操作系统运行的特权模式，具有更高的权限，能够直接访问硬件资源和系统内部数据。

当 `syscall` 指令被执行时，处理器会保存当前用户程序的状态（如程序计数器和寄存器内容），并加载内核的中断处理程序。此时，控制权转移到操作系统内核，中断服务例程会检查 `syscall` 的参数并执行相应的服务。例如，内核会根据 `syscall` 提供的系统调用号调用相应的处理函数，如 `read`、`write` 或 `open`。

在内核态，操作系统执行相应的系统调用逻辑，并可能访问硬件或系统资源。处理完请求后，内核会准备返回结果，并恢复用户程序的状态。然后，处理器切换回用户态，恢复用户程序的执行。此切换通过特定的指令完成RESTORE ALL，恢复之前保存的寄存器和程序计数器，使得用户程序能够继续执行，就像没有发生过 `syscall` 调用一样。

总之，`syscall` 调用通过中断机制在用户态和内核态之间进行切换，允许用户程序安全地请求操作系统服务。这个过程确保了系统的稳定性和安全性，同时使得操作系统能够高效地管理资源和处理各种系统请求。



Syscall_execve

`Syscall_execve` 函数模拟了系统调用 `execve` 的操作，用于加载并执行指定路径的程序。首先，函数通过 `VirtualMemorySpace::EnableAccessUser()` 启用对用户内存的访问，并获取当前进程对象。接着，利用虚拟文件系统打开待执行的 ELF 文件，并初始化一个文件对象以进行读取操作。函数计算程序参数的数量，并将 `argv` 中的参数复制到新的内存区域中。随后，`CreateProcessFromELF` 被调用以从 ELF 文件中创建新的进程，并设置其文件描述符。

在新进程创建后，函数进入一个等待循环，检查子进程的状态，直到其执行完成。当前进程在子进程结束前会被阻塞，待子进程完成后，函数会回收其资源并获取退出码。最后，当前进程调用 `cur_proc->exit(exit_value)` 退出，并释放所有临时分配的内存，如文件对象。整个流程确保了新程序能够被正确加载和执行，同时保证了资源的正确管理和进程的正确退出。

Syscall_clone

`Syscall_clone` 函数实现了创建新进程或线程的系统调用，其执行流程根据 `stack` 参数的不同而有所变化。首先，函数会禁用中断以确保原子操作，通过 `IntrSave` 保存当前中断状态。接着，获取当前进程 `cur_proc` 并分配新的进程对象 `create_proc`。如果进程分配失败，函数会记录错误并返回 -1。若 `stack` 为 `nullptr`，表明这是一个类似 `fork` 的调用，函数会创建一个新的虚拟内存空间 `vms`，并复制当前进程的虚拟内存空间到新进程中。接下来，将当前进程的上下文（`TrapFrame`）复制到新进程的栈中，并更新上下文的程序计数器 `epc` 以及返回值 `a0` 为 0，标志新进程的创建成功。

如果 `stack` 不为 `nullptr`，函数执行类似于线程创建的操作，即执行 `clone`。新进程共享当前进程的虚拟内存空间，并将 `stack` 指定为新进程的栈指针。然后，将当前进程的上下文复制到新进程的栈中，并将新进程的栈指针 `sp` 设置为 `stack`。函数还会检查 `flags` 是否包含 `SIGCHLD` 标志，如果有，则设置新进程的名称并将其标记为当前进程的子进程。最后，将新进程的状态设置为准备就绪（`S_Ready`），恢复中断状态，并返回新进程的进程 ID。

Syscall_mkdirat

首先，函数分配两个字符数组 `rela_wd` 和 `dir_path`。接着，获取当前进程对象 `cur_proc` 和当前工作目录 `cwd`。根据 `dirfd` 参数判断目录路径的基准点。如果 `dirfd` 为 `AT_FDCWD`，则使用当前工作目录作为基础路径，并将其复制到 `rela_wd`。如果 `dirfd` 不是 `AT_FDCWD`，函数通过文件描述符 `dirfd` 获取对应的文件对象 `fo`，并从中提取路径信息到 `rela_wd`。

随后，函数调用 `unified_path` 将相对路径与传入的路径合并，形成完整的目录路径 `dir_path`。接着，调用虚拟文件系统（`vfsm`）的 `create_dir` 函数在 `dir_path` 位置创建新目录。如果创建成功，函数返回 0；否则，返回 -1。在函数结束前，释放分配的内存。

Syscall_wait4

`Syscall_wait4` 函数实现了一个系统调用，用于等待子进程的状态变化，通常是等待子进程结束。这个函数接收三个参数：`pid`（待等待的进程 ID）、`status`（存储子进程退出状态的指针），以及 `options`（等待选项）。如果 `pid` 为 -1，表示等待所有子进程。如果 `status` 非空，函数会将子进程的退出状态存储到 `status` 指向的位置。

函数的执行过程如下：首先，获取当前进程 `cur_proc` 并检查是否有子进程。如果没有子进程，输出警告并返回 -1。然后，进入一个循环，遍历所有子进程查找状态为终止 (`S_Terminated`) 的进程。若找到符合条件的子进程（即其 ID 为 `pid` 或 `pid` 为 -1），函数会将该子进程的 ID 存储在 `ret` 中。如果 `status` 不为空，函数会启用用户内存访问，将子进程的退出代码左移 8 位并存储到 `status` 指向的位置。然后，销毁并回收该子进程，返回子进程的 ID。

如果在遍历过程中没有找到符合条件的子进程且 `options` 参数中包含 `WNOHANG`，则立即返回 -1，表示没有子进程退出。如果 `WNOHANG` 未设置，则当前进程会阻塞，调用其信号量等待子进程的状态变化。一旦子进程结束并发出信号量，当前进程将被唤醒并重新调度。这个等待机制保证了父进程能在子进程结束后及时回收资源。

Syscall_times

`Syscall_times` 函数用于获取当前进程的运行时间，包括用户态和核心态时间。它接受一个 `tms` 结构体指针，用于存储进程时间数据。函数首先获取当前进程 `cur_proc` 的运行时间和系统时间。`runTime` 表示总的运行时间，`sysTime` 表示系统时间（即陷入核心态的时间）。用户时间 `user_time` 是 `runTime` 减去 `sysTime`。

如果 `tms` 不为空，函数会计算用户态时间和系统态时间，按微秒为单位（`time_unit` 为 10）填充 `tms` 结构体。它还会遍历所有子进程，将它们的用户态和系统态时间累计到 `tms` 中。所有计算完成后，返回自系统启动以来的滴答数（即当前时间）。如果计算中的时间值为负，则视为出错并返回 -1。

Syscall_getdents64

`Syscall_getdents64` 函数用于从目录中读取目录项，并将其信息填充到用户提供的缓冲区 `_buf` 中。该系统调用处理的目录项结构体 `Dirent` 包含了文件的索引节点号 (`d_ino`)、到下一个目录项的偏移 (`d_off`)、当前目录项的长度 (`d_reclen`)、文件类型 (`d_type`)、以及文件名 (`d_name`)。

函数首先获取当前进程 `proc` 和文件描述符 `fd` 对应的文件对象 `dir`。通过 `fm.get_from_fd` 查找该文件对象，若文件对象无效，则返回 -1。接着，函数从虚拟文件系统 (`vfsm`) 中获取目录的根节点，并通过 `get_next_file` 方法遍历目录中的文件。`file` 指针用于存储当前目录项的信息。

接下来，函数启用用户内存访问，以便能将数据写入用户空间。初始化 `n_read` 为已读取的字节数，并设置 `i` 为当前位置。如果当前文件指针为空，则返回 0 表示没有更多目录项可读。循环中，函数将

目录项信息写入用户缓冲区 `_buf`，更新 `n_read`。如果缓冲区的剩余空间不足以容纳下一个目录项，则更新目录对象的当前位置 `pos_k` 并返回已读取的字节数。最终，函数更新目录的当前位置并返回已读取字节数。

Syscall_splice

本题目需要我们实现一个系统调用 `splice`，用于将打开的文件中指定范围的数据复制到管道中，或是将数据从管道复制到文件的指定范围。其对应用户库函数的声明为：

```
1 #include <unistd.h>
2
3 ssize_t splice(int fd_in, off_t *_Nullable off_in,
4                 int fd_out, off_t *_Nullable off_out,
5                 size_t len, unsigned int flags);
```

`splice()` 在两个文件描述符之间拷贝数据，而不涉及内核地址空间和用户地址空间之间的复制。`splice()` 从文件描述符 `fd_in` 传输至多 `len` 字节的数据到文件描述符 `fd_out`，其中一个文件描述符一定是管道，另一个一定是普通的磁盘文件。

`splice()` 对于参数的要求是：

- 如果 `fd_in` 指的是管道，那么 `off_in` 必须是 `NULL`。
- 如果 `fd_in` 不是管道，则 `off_in` 一定不是 `NULL`，且 `off_in` 指向的位置存储了将从 `fd_in` 读取的偏移量。在读取过程中，`fd_in` 文件描述符的偏移不改变，而是将 `*off_in` 增加成功复制的字节数。

同样地，对于 `fd_out` 来说：

- 如果 `fd_out` 指的是管道，那么 `off_out` 必须是 `NULL`。
- 如果 `fd_out` 不是管道，则 `off_out` 一定不是 `NULL`，且 `off_out` 指向的位置存储了将向 `fd_out` 写入的偏移量。在写入过程中，`fd_in` 文件描述符的偏移不改变，而是将 `*off_out` 增加成功复制的字节数。

本系统调用的返回值为成功复制的字节数，出现错误时返回负值。若 `*off_in` 的文件偏移超过 `fd_in` 的大小，则直接返回 0，不进行复制。若 `off_in` 或 `off_out` 为负值，则直接返回 -1。`fd_out` 的文件偏移保证不会超过文件的总字节数。

本题中，其中一个文件描述符一定是管道，另一个一定是普通的磁盘文件。`flags` 总为 0，没有实际作用。

特殊说明：

1. 尽可能向管道写入数据（当管道满时，阻塞等待）。
2. 如果文件 `fd_in` 的剩余部分小于 `len`，则将 `fd_in` 文件剩余的全部内容写入管道 `fd_out`。

3. 读取管道时，允许读取的数据量小于 `len`，但在管道有数据时不可返回0（当管道空时，阻塞等待）。

测试点 (50分)

1. 测试的 `fd_in` 为普通文件，`fd_out` 为管道，且管道在读写完成前不会关闭(10分)
2. 测试的 `fd_in` 为管道，`fd_out` 为普通文件，且管道在读写完成前不会关闭(10分)
3. 测试文件到管道的拷贝时，普通文件 `fd_in` 剩余的字节数小于 `len` (10分)
4. 测试管道到文件的拷贝时，`fd_in` 管道数据不足 `len` bytes(10分)
5. 边界情况处理(10分)

```

1  inline long syscall_splice(int fd_in, long *off_in, int fd_out, long *off
   _out, size_t len, unsigned int flags)
2  {
3      kout[Debug] << "Entering syscall_splice" << endl;
4      kout[Debug] << "fd_in: " << fd_in << ", fd_out: " << fd_out << ", le
   n: " << len << ", flags: " << flags << endl;
5
6  if (fd_in < 0 || fd_out < 0 || len <= 0) {
7      kout[Debug] << "Invalid parameters: fd_in or fd_out is negative,
   or len is non-positive" << endl;
8      return -1;
9  }
10
11 if ((off_in && *off_in < 0) || (off_out && *off_out < 0)) {
12     kout[Debug] << "Invalid offsets: off_in or off_out is negative" <
   < endl;
13     return -1;
14 }
15
16 Process *cur_proc = pm.getCurProc();
17 file_object *fo_in = fom.get_from_fd(cur_proc->fo_head, fd_in);
18 if (fo_in == nullptr) {
19     kout[Error] << "Syscall_splice can't open fd_in" << endl;
20     return -1;
21 }
22 file_object *fo_out = fom.get_from_fd(cur_proc->fo_head, fd_out);
23 if (fo_out == nullptr) {
24     kout[Error] << "Syscall_splice can't open fd_out" << endl;
25     return -1;
26 }
27
28 bool is_pipe_in = (off_in == nullptr);
29 bool is_pipe_out = (off_out == nullptr);
30 kout[Debug] << "is_pipe_in: " << is_pipe_in << ", is_pipe_out: " <<
   is_pipe_out << endl;
31 if (is_pipe_in && off_in != nullptr) {
32     kout[Error] << "Invalid offset for pipe input" << endl;
33     return -1;
34 }
35 if (is_pipe_out && off_out != nullptr) {
36     kout[Error] << "Invalid offset for pipe output" << endl;
37     return -1;
38 }
39
40 long long read_offset = 0;
41 if (!is_pipe_in) {
42     read_offset = *off_in;

```

```

43         if (read_offset > fo_in->file->size()) {
44             kout[Debug] << "read_offset exceeds file size" << endl;
45             return 0;
46         }
47         if(fo_in->file->size()-read_offset<len){
48             kout[Debug] << "Adjusting len from " << len << " to " << (fo_
49             in->file->size() - read_offset) << endl;
50             len=fo_in->file->size();
51         }
52     }
53
54     long long write_offset = 0;
55     if (!is_pipe_out) {
56         write_offset = *off_out;
57         if (write_offset > fo_out->file->size()) {
58             kout[Debug] << "write_offset exceeds file size" << endl;
59             return 0;
60         }
61         unsigned char *buffer = new unsigned char[len];
62         VirtualMemorySpace::EnableAccessUser();
63
64         // 从输入文件描述符中读取数据
65         long long bytes_read = -1;
66         if (is_pipe_in) {
67             kout[Debug] << "Reading from pipe" << endl;
68             fo_in->pos_k=0;
69             bytes_read = fom.read_fo(fo_in, buffer, len);
70             fo_in->pos_k=0;
71         } else {
72             kout[Debug] << "Reading from file, offset: " << read_offset << en
73             dl;
74             fo_in->pos_k=read_offset;
75             bytes_read = fom.read_fo(fo_in, buffer, len);
76             fo_in->pos_k=read_offset;
77         }
78
79         if (bytes_read <= 0) {
80             kout[Debug] << "Read failed or no data to read, bytes_read: " <<
81             bytes_read << endl;
82             delete[] buffer;
83             VirtualMemorySpace::DisableAccessUser();
84             return bytes_read;
85         }
86
87         long long bytes_written = -1;
88         if (is_pipe_out) {
89             kout[Debug] << "Writing to pipe" << endl;

```

```

88         bytes_written = fom.write_fo(fo_out, buffer, bytes_read);
89         fo_out->pos_k=0;
90     } else {
91         kout[Debug] << "Writing to file, offset: " << write_offset << endl;
92         fo_out->pos_k=write_offset;
93         bytes_written = fom.write_fo(fo_out, buffer, bytes_read);
94         fo_out->pos_k=write_offset;
95     }
96
97     delete[] buffer;
98     VirtualMemorySpace::DisableAccessUser();
99
100    if (!is_pipe_in) {
101        *off_in += bytes_read;
102    }
103    if (!is_pipe_out) {
104        *off_out += bytes_written;
105    }
106
107    kout[Debug] << "Bytes written: " << bytes_written << endl;
108    return bytes_written;
109}

```

主要用于在文件描述符之间传输数据。首先，它检查传入的参数的有效性。如果输入文件描述符 (`fd_in`) 或输出文件描述符 (`fd_out`) 负值，或数据长度 (`len`) 非正，则返回错误。其次，函数还验证了输入和输出文件的偏移量是否有效，确保它们在合理的范围内，尤其是当这些文件描述符是管道时。接下来，函数通过当前进程的文件对象列表获取输入和输出文件对象。如果输入文件描述符无效或输出文件描述符无效，函数返回错误。然后，函数根据输入和输出文件描述符是否为管道来确定是否使用偏移量进行读写。如果不是管道，它将检查并调整偏移量以确保它们在文件的有效范围内。同时，函数分配了一个缓冲区用于存储从输入文件读取的数据，并设置虚拟内存的访问权限。

在数据传输的核心部分，函数从输入文件描述符读取数据到缓冲区。如果输入文件描述符是管道，读取操作从管道的起始位置开始，否则从指定的偏移量开始。读取操作完成后，函数将数据写入到输出文件描述符。如果输出文件描述符是管道，则从管道的起始位置开始写入，否则从指定的偏移量开始。函数处理完所有操作后，释放缓冲区，并更新文件偏移量。如果数据读取或写入失败，函数会返回失败的字节数。

物理内存

引言

本小队先对物理内存进行开发，保证物理内存malloc和free正确的情况下，再更进一步进行虚拟内存开发。

设计思路

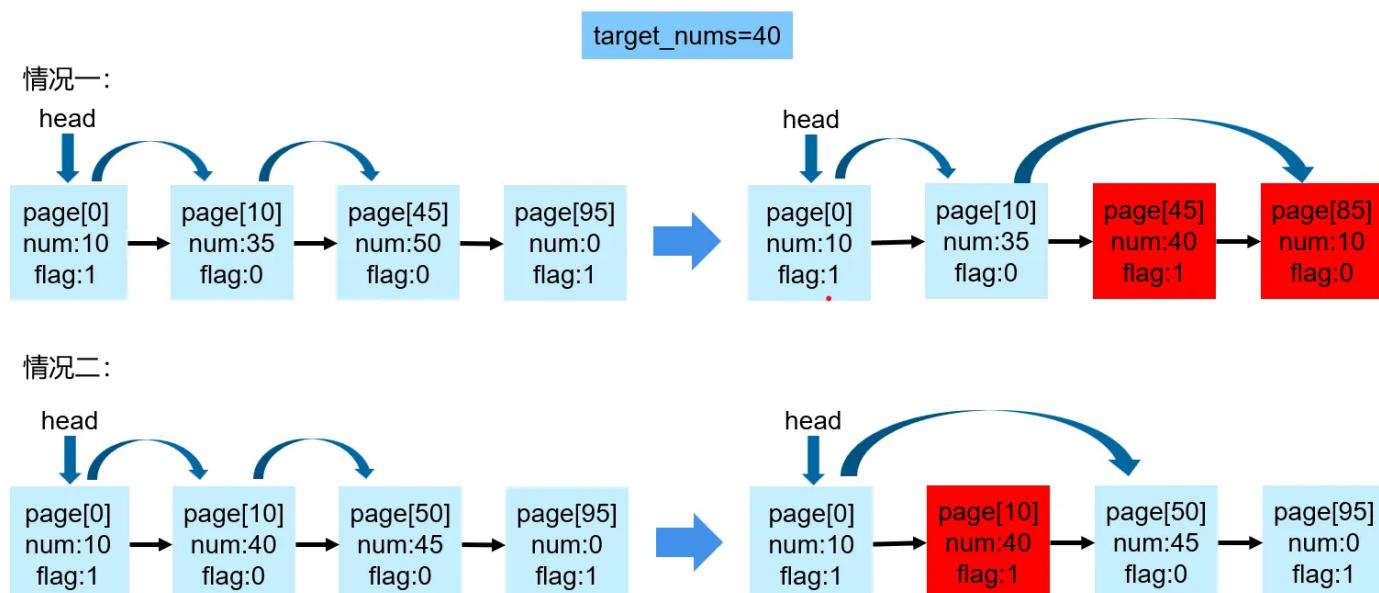
内存空间是一块连续的物理内存，我们采用链表的方法进行管理。设置PAGE用来管理每一个页的空间，再用PMM来进一步管理众多连续的PAGE。

计算管理这些页所需要多少个PAGE大小，将这些结构体按数组的内存分配布局放入最前面的几页，并将这几页的PAGE标记为非空闲，同时在PMM中记录第一个空闲页（双向链表的head指针）就是完成了PMM的初始化工作。

对于alloc_pages函数，实现了物理内存分配逻辑，采用链表管理思路：先从头结点开始搜索符合空间大小的内存空间，若没有则采用insert_page进行合并直到找到第一个能用于分配指定大小的页空间。

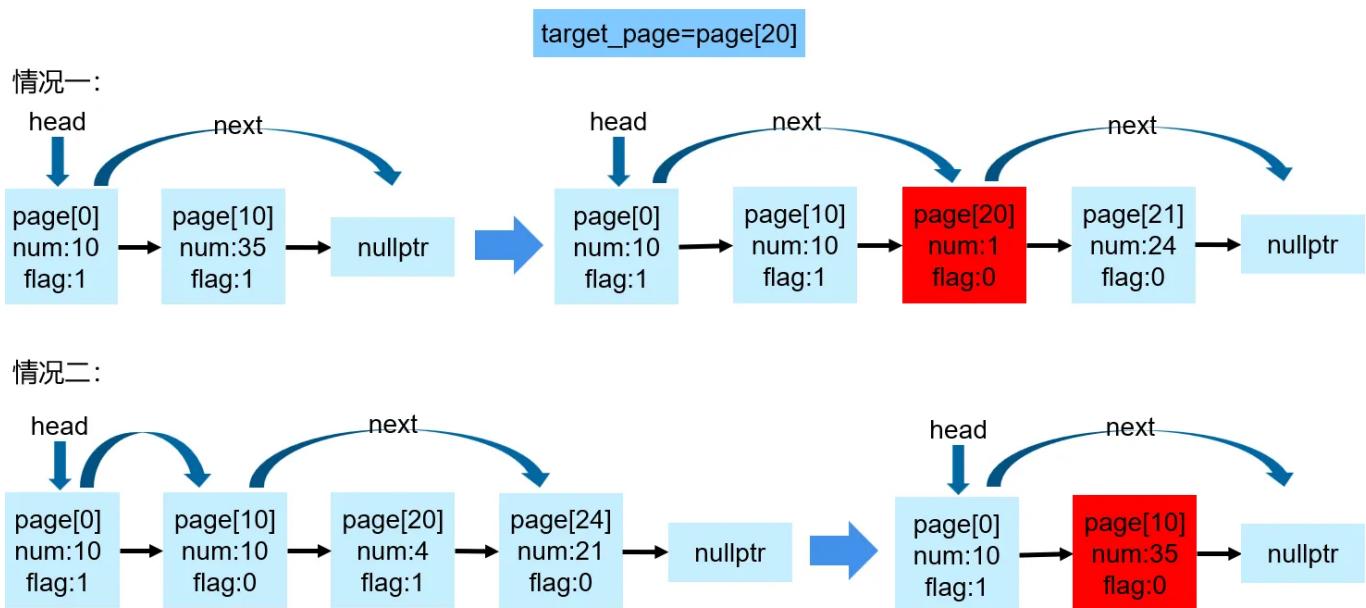


物理内存——PMM页分配：首次适应算法





物理内存——PMM页释放

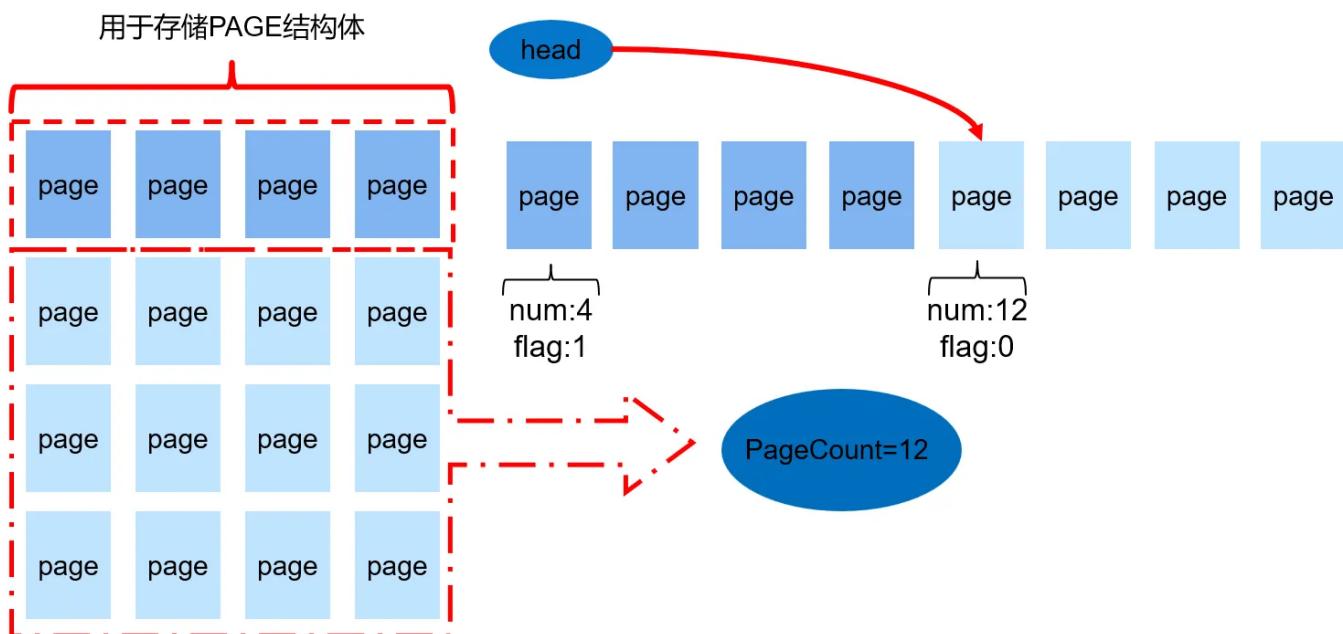


对于free_pages函数，实现了页的回收逻辑。首先先将总的空闲页增加并将该页标记为空闲，最后根据结构体PAGE的地址的大小插入会双向链表（管理PAGE*的head），同时查看是否可以合并并合并。

对于get_page_from_addr函数，用于从物理地址获取对应页的PAGE指针，从而便于free回收逻辑。

对于malloc函数，实现了PMM的内存分配，由于本小队打算完善小内存的管理，所有添加了每页的标记位的数（具体将于slab中进行讲述）。

对于free函数，实现了物理页的回收逻辑：首先找到回收地址对应的PAGE指针，然后调用free_pages函数进行页的回收。



PMM

在PMM处，我们设计了管理PAGE结构体的表头，记录最后一个结构体后一个的物理地址，便于判断页是否超内存空间大小。以及记录可用空闲页数的PageCount。最后将PMM的变量pmm作为全局变量，便于其他部分的使用。

```
1 class PMM {
2     private:
3         PAGE head; // 表头
4         PAGE* PagesEndAddr; // 结束地址
5         UInt64 PageCount; // 可用空闲页数
6 }
```

PAGE

在PAGE处，设计了标记位flags（0表示空闲，1表示不是slab，2表示slab64B，3表示slab512B，4表示slab4KB），以及当前页结构体后面有多少个空闲的连续的页，对于ref是为了后续虚拟内存设计的接口，便于找到虚拟内存多级页表是否有页表中的项指向它。最后是pre和nxt双向链表指针，便于页的分配和回收。

在Index函数，实现了找到该PAGE结构体对应于污泥内存中第几个页。

KAddr函数，实现了具体指向的页的地址而不是PAGE的地址。

PAddr函数，实现了具体的物理地址，而不是虚拟地址（多个偏移）

```
1 struct PAGE {
2     UInt64 flags, // 0表示空闲，1表示不是slab，2表示slab64B，3表示slab512B，4表示
3     slab4KB
4     num, // 当前表后面有连续的几页
5     ref; // 有多少个其他的页表中的项指向它
6     PAGE *pre,
7     *nxt;
8 }
```

slab分配器

设计

为了便于小内存的分配，本小组写了一个简易的slab内存分配器。和物理内存采用类似的结构进行管理和分配。pmm管理物理内存的最小单位是物理内存页 page，而从内核实际运行过程中来看，对于

内存的需求量往往是以字节为单位，远远小于一个页面的大小。如果我们仅仅为了这几十字节的内存需求，而专门为它分配一个整个内存页面，这无疑是对宝贵内存资源的一种巨大浪费。于是在内核中，这种专门小内存分配就应运而生。slab 首先会向pmm一次性申请一个或者多个物理内存页面，正是这些物理内存页组成了 slab。随后 slab会将这些连续的物理内存页面划分成多个大小相同的小内存块出来，同一种 slab下，划分出来的小内存块尺寸是一样的。内核会针对不同尺寸的小内存分配需求，预先创建出多个 slab出来。

用于零散小内存块分配的内存池 —— slab 分配器

slab 向pmm一次申请一个或多个物理页，构成 slab

```
void *SlabAllocator::allocateSlab(UINT64 size, UINT32 usage){  
    ...  
    return pmm.malloc(size, usage);  
}
```

针对不同尺寸的小内存分配需求，创建出3个 slab

```
void SlabAllocator::Init(){  
    ...  
    sB1.Init(SLAB_SIZE_B1, SLAB_PAGE1, 2);  
    sB2.Init(SLAB_SIZE_B2, SLAB_PAGE2, 3);  
    sB3.Init(SLAB_SIZE_B3, SLAB_PAGE3, 4);  
}
```

每个slab仿照pmm进行实现，分配器管理与pmm对接

```
class SlabB {  
private:  
    ...  
    slabB head; //表头  
    slabB* PagesEndAddr; //结束地址  
    ...  
    ...  
    ...  
public:  
    void Init(UINT32 size,UINT32 PageCounts,UINT32 Usage);  
    ...  
    ...  
    ...  
};
```

```
class SlabAllocator {  
private:  
    ...  
    ...  
    ...  
public:  
    SlabAllocator(){...}  
    void Init();  
    void free(void* freeaddress,UINT64 usage);  
    ...  
    void* allocate(UINT64 bytesize);  
    ...  
    void* allocateSlab(UINT64 size,UINT32 usage);  
};
```

slabB

与PAGE类似，用来管理每个小的内存空间。

```

1 struct slabB
2 {
3     Uint64 flags, //0表示空闲, 1表示不
4         SlabStartAddress,
5         Slab_Size,
6         num; //当前表后面有连续的几页
7     slabB* pre,
8         * nxt;
9
10    inline Uint64 Index() const
11    {return ((Uint64)this-SlabStartAddress)/sizeof(slabB);} //第几页
12
13    inline void* KAddr() const
14    {return (void*)(SlabStartAddress+Index()*Slab_Size);} //具体页的地址
15
16    inline void* PAddr() const
17    {return (void*)(SlabStartAddress+Index()*Slab_Size-0xfffffffff00000000)
18     ;} //物理地址
19 };

```

SlabB

SlabB代表管理每个slabB的结构体，具体实现采用PMM的方式进行实现。

```

1 // 定义Slab结构
2 class SlabB {
3 private:
4     slabB head; //表头
5     slabB* PagesEndAddr; //结束地址
6     Uint64 SlabStartAddress,
7         Slab_Size,
8         slabBCount; //可用空闲块数
9
10 public:
11     void Init(Uint32 size, Uint32 PageCounts, Uint32 Usage);
12     slabB* allocslabB(Uint64 num);
13     bool insert_page(slabB* src);
14     bool freeslabB(slabB* t);
15     slabB* get_from_addr(void* addr);
16     inline Uint64 getSlabBCount(){return slabBCount;}
17     void free(void* freeaddress);
18 };

```

SlabAllocator

该类主要用于最终slab的分配，内部设置三个管理对应大小的slab链表。Init在一开始会给每个大小分配指定大小的页专门用于分配大小。

```
1 class SlabAllocator {
2     private:
3         SlabB sB1, //链表
4                 sB2,
5                 sB3;
6 }
```

具体分配逻辑采用最简单的谁符合就分配对应slab大小的块

```
1 // 这边具体逻辑还需要算法修改
2 void* SlabAllocator::allocate(UInt64 bytesize)
3 {
4     if (bytesize <= SLAB_SIZE_B1) {
5         if (sB1.getSlabBCount() <= 0)
6             return nullptr;
7         else
8             return sB1.allocslabB(1)->KAddr();
9     } else if (bytesize <= SLAB_SIZE_B2) {
10        if (sB2.getSlabBCount() <= 0)
11            return nullptr;
12        else
13            return sB2.allocslabB(1)->KAddr();
14    } else if (bytesize <= 4096) {
15        int num = bytesize / SLAB_SIZE_B3;
16        if (bytesize % SLAB_SIZE_B2 != 0) {
17            num++;
18        }
19        if (sB3.getSlabBCount() <= 0)
20            return nullptr;
21        else
22            return sB3.allocslabB(num)->KAddr();
23    }
24    return nullptr;
25 }
```

在释放时，找到对应的slab链表再进行对应的free即可。

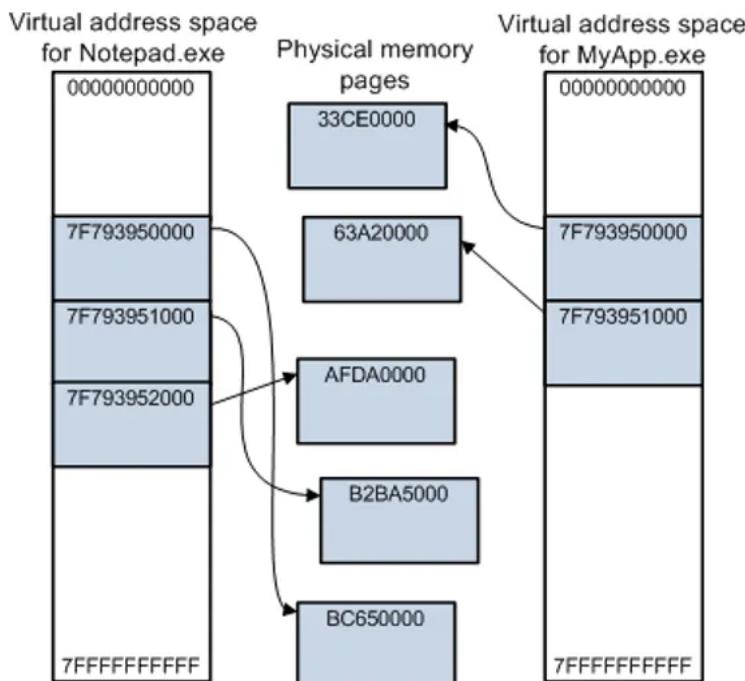
遇到问题

1.slab管理结构体本身占用太多内存，还需要进一步进行优化。

2.同时对于一页+一小块的内存仍无法拆分进行分配还是会造相应的内存浪费。

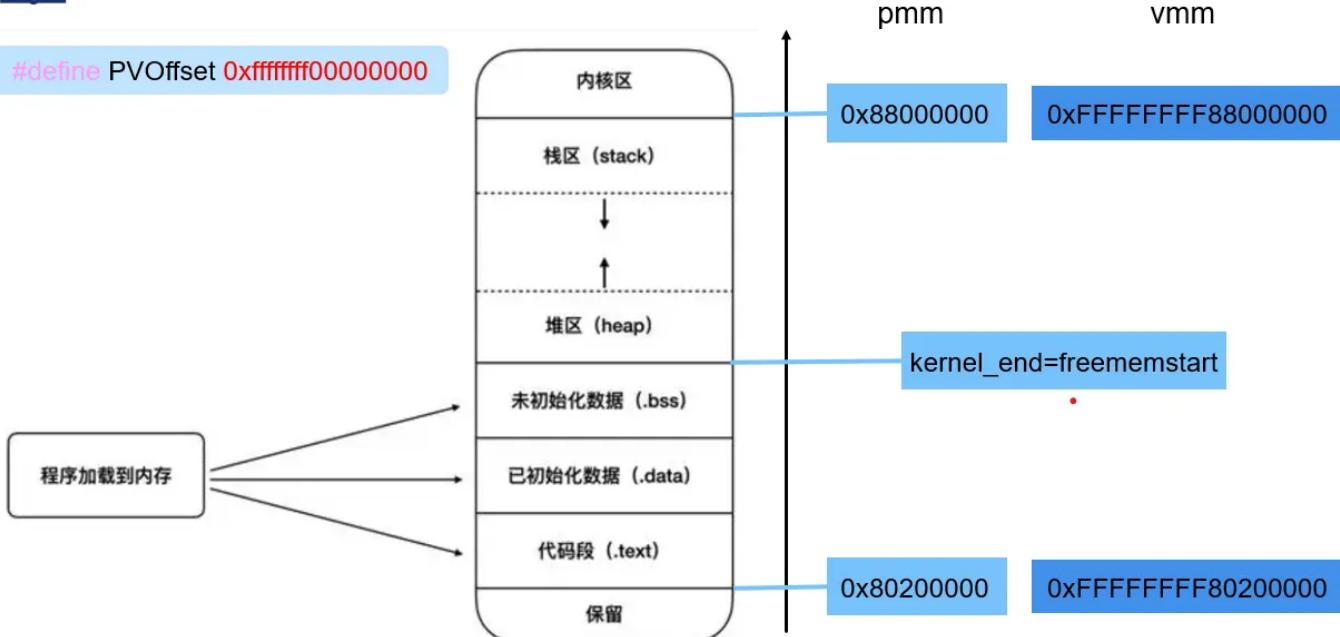
虚拟内存

虚拟内存是现代主流操作系统支持的管理方案，它需要硬件提供MMU进行分页机制。虚拟内存部分主要由三个类/结构体进行控制，分别为虚拟内存空间（Virtual Memory Space，简称VMS），虚拟内存区域（Virtual Memory Region，简称VMR），页表（PageTable），堆内存区域管理（HeapMemoryRegion）。



虚拟内存划分

```
#define PVOFFSET 0xffffffff00000000
```



PageTable

页表是我们根据SV39页表的规范构造的结构体，在目前的虚拟内存机制下它是由512个页表项组成的，大小为一个4k页的结构体，特别注意，创建该对象时不能用一般的Kmalloc或new，而需要用PMM中提供的分配物理页方法，以保证4KB对齐。

PageEntry

页表项是页表的成员，可以作为次级页表索引，也可以作为物理页索引，此外其上还标记了一些标志位，用于权限控制等。为了提高效率，读写标志位的部分均使用模板实现，最大限度地提高效率，同时保留易用性。页表项的XWR三个位用来标识当前页表项的属性，当全为0时表示次级页表，索引时需要拿到次级页表的物理页号，转换成内核虚拟页号，一级一级向下找。

VirtualMemorySpace

VMS用于管理概念上的一个逻辑地址空间，拥有一个根页表（PDT）和若干VMR串成的链表。

成员变量如下：

```
1 class VirtualMemorySpace {
2     protected:
3         static VirtualMemorySpace *CurrentVMS, // 目前层的VMS
4             *BootVMS, // 启动时使用的vms
5             *KernelVMS; // 内核层vms
6
7         POS::LinkTableT<VirtualMemoryRegion> vmrHead; // 管理vmr的链表
8         UInt32 VmrCount; // vmr的数量
9         VirtualMemoryRegion* VmrCache; // 最近使用的vmr，便于命中
10        PageTable* PDT; // 根页表指针
11        UInt32 SharedCount; // 共享计数，当共享计数达到0时可以自动销毁自身空间
12 }
```

类方法如下：

```

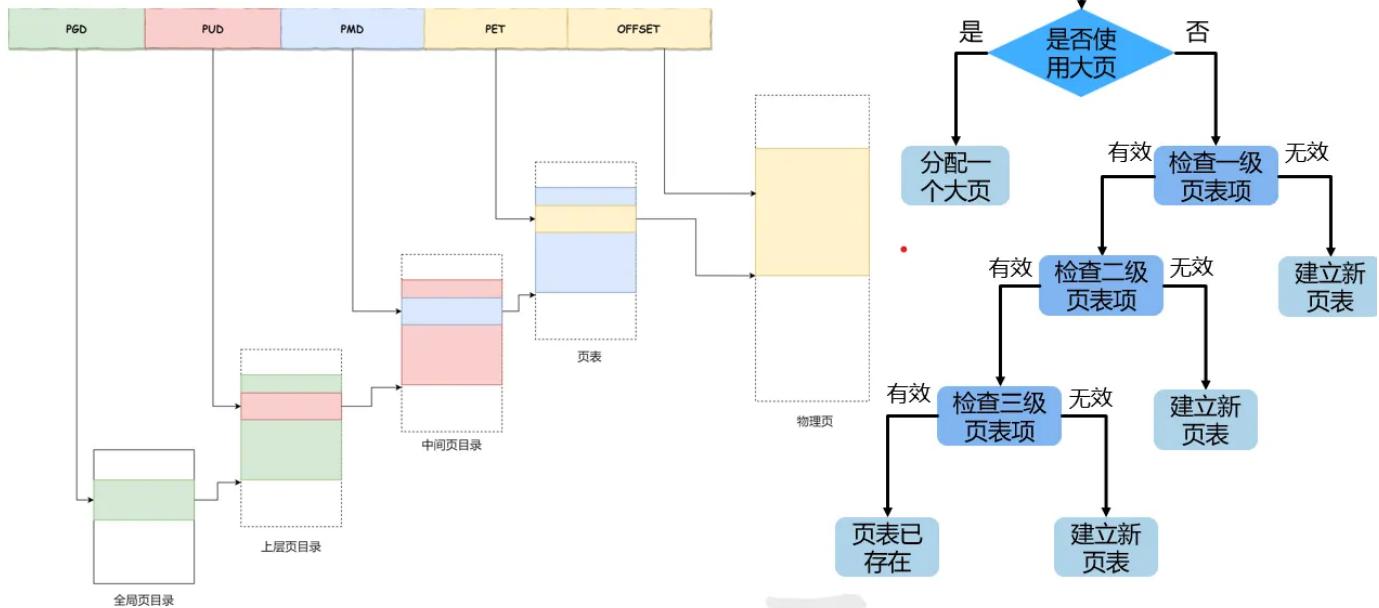
1  class VirtualMemorySpace {
2  protected:
3      ErrorType ClearVMR(){};//情况vmr链表
4      ErrorType CreatePDT() {};//创造页表项
5      static ErrorType InitForBoot(){};//Bootvms初始化
6      static ErrorType InitForKernel(){};//内核层vms初始化
7  public:
8      inline static VirtualMemorySpace* Current() {};//返回当前vms
9      inline static VirtualMemorySpace* Boot(){};//返回Bootvms
10     inline static VirtualMemorySpace* Kernel(){};//返回内核层vms
11     inline static void EnableAccessUser(){};//默认情况下，内核态是不可以访问用
12 户态地址空间的,
13 当用户态传递了某个指针必须由内核访问时，可以 暂时开启访问允许，不需要后立刻关闭，以最大程
度避免误操作
14     inline static void DisableAccessUser(){};//与上一个类似，为关闭允许访问用户
空间使用。
15     ErrorType Init(){};//vms整体变量初始化
16     static ErrorType InitStatic(){};//内核和Bootvms的初始化，同时当前态进入内核
态
17     void InsertVMR(VirtualMemoryRegion* vmr){};//插入一个构造好的VMR到当前VMS
管理的链表中,
18 表示启用地址空间中的某一区域
19     VirtualMemoryRegion* FindVMR(PtrUInt p){};//查找某个地址所属的VMR,
20 如果不存在则返回nullptr
21     void RemoveVMR(VirtualMemoryRegion* vmr, bool DeleteVm){};//从地址空间
中移除某个VMR
22     void Enter(){};//VMS的核心功能之一，即进入到当前虚拟地址空间,
23 具体操作时是对页表进行更换，并刷新TLB缓存
24     inline void Leave(){};//概念上与Enter相反的功能，由于进程必须呆在地址空间中,
25 因此Leave实际上是Enter到了KernelVMS中
26     inline void show(){};//展示vmr里的所有结点信息
27     ErrorType SolvePageFault(TrapFrame* tf){};//解决缺页异常
28     ErrorType Create(){};//创建一个指定类型的虚拟内存空间
29     ErrorType Destroy(){};
30     ErrorType showVMRCount(){};//打印VMRcount
31     ErrorType CreateFrom(VirtualMemorySpace* src) {};//从给定的虚拟内存空间创
建一个新的VMS，即拷贝VMS
32 };

```

在处理缺页异常时，增加了大页的逻辑，但只是简单的根据内存区域的标志来决定是否采用大页，后续还需要进一步完善。



虚拟内存——多级页表与缺页异常



```
1 // 检查地址对齐
2 if ((faultAddress & (2 * 1024 * 1024 - 1)) != 0) {
3     return false; // 地址没有适当对齐, 不能使用大页
4 }
5
6 // 根据内存区域的标志来决定
7 if (flags & VM_Heap) {
8     // 对于堆区域, 如果预期会频繁使用大量内存, 可能倾向于使用大页
9     return true;
10 } else if (flags & VM_Stack) {
11     // 栈通常不适合使用大页, 因为栈的增长通常是小块的
12     return false;
13 } else if (flags & VM_Exec) {
14     // 对于执行代码区域, 使用大页可以提高指令缓存的效率
15     return true;
16 }
17 // 默认情况, 不使用大页
18 return false;
```

VirtualMemoryRegion

虚拟内存区域VMR表示的是VMS中的一个合法区间，通过VMR我们可以对进程的不同段进行划分，区分内存映射文件等。

成员变量如下：

```
1 class VirtualMemoryRegion : public POS::LinkTableT<VirtualMemoryRegion> {
2     friend class VirtualMemorySpace;
3 public:
4     enum : UInt32 // 标志位
5     {
6         VM_Read = 1 << 0,
7         VM_Write = 1 << 1,
8         VM_Exec = 1 << 2,
9         VM_Stack = 1 << 3,
10        VM_Heap = 1 << 4,
11        VM_Kernel = 1 << 5,
12        VM_Shared = 1 << 6,
13        VM_Device = 1 << 7,
14        VM_File = 1 << 8,
15        VM_Dynamic = 1 << 9,
16
17        VM_RW = VM_Read | VM_Write,
18        VM_RwX = VM_RW | VM_Exec,
19        VM_KERNEL = VM_Kernel | VM_RwX,
20        VM_USERSTACK = VM_RW | VM_Stack | VM_Dynamic,
21        VM_USERHEAP = VM_RW | VM_Heap | VM_Dynamic,
22        VM_MMIO = VM_RW | VM_Kernel | VM_Device,
23    };
24
25 protected:
26     PtrUInt StartAddress, // 起始地址
27             EndAddress; // 终结地址
28     VirtualMemorySpace* VMS; // 管理的存储信息
29     UInt32 Flags; // 这块vms的权限
30 }
```

类方法如下：

```

1  class VirtualMemoryRegion : public POS::LinkTableT<VirtualMemoryRegion> {
2  public:
3      inline PageTableEntryType ToPageEntryFlags(){};
4      inline bool Intersect(PtrUInt l, PtrUInt r) const // r在中间
5      inline bool In(PtrUInt l, PtrUInt r) const // 包含l,r
6      inline bool In(PtrUInt p)
7      inline PtrUInt GetStart(){};//获取该管理空间的起始地
8      inline PtrUInt GetEnd(){};//获取终地址
9      inline PtrUInt GetLength(){};//获取内存空间大小
10     bool ShouldUseLargePage(UInt32 flags, PtrUInt faultAddress){};//是否启用大页的逻辑
11     inline UInt32 GetFlags(){};//获取标志位信息
12     ErrorType Init(PtrUInt start, PtrUInt end, UInt32 flags){};//vmr初始化
13     ErrorType CopyMemory(PageTable& pt, const PageTable& src, int level, U
14         int64 l) {};
15     //为了实现Clone的功能，必须要支持对VMS进行拷贝，而VMS的拷贝实际上就是对所属的VMR
16     //一个一个进行拷贝，  

17     //因此我们设计将VMR作为拷贝的单元。
18 };

```

HeapMemoryRegion

原先设计的VMR并不支持动态地调整大小，为了提供动态调整大小的功能，从VMR派生出HeapMemoryRegion（HMR），用来支持用户程序的brk调用，即调整堆段大小。当进行调整时，函数会判断这次调用是否能成功，保证不与其他VMR发生重叠。HMR一般在载入用户进程时创建，根据ELF文件段表的描述，我们在最后一个段出现的位置后面用GetUsableVMR来获取一个可用区域作为堆段。

```

1  class HeapMemoryRegion : public VirtualMemoryRegion {
2  protected:
3      UInt64 BreakPointLength = 0;
4  public:
5      inline PtrUInt BreakPoint() {};
6      inline ErrorType Resize(Sint64 delta) {};
7      inline ErrorType Init(PtrUInt start, UInt64 len = PAGESIZE, UInt64 flag
8          s = VM_USERHEAP) {};
9  };

```

进程管理

进程的状态由Process结构体定义，充分利用C++面向对象的理念，针对单个进程的操作都尽可能由方法实现。

Process

```

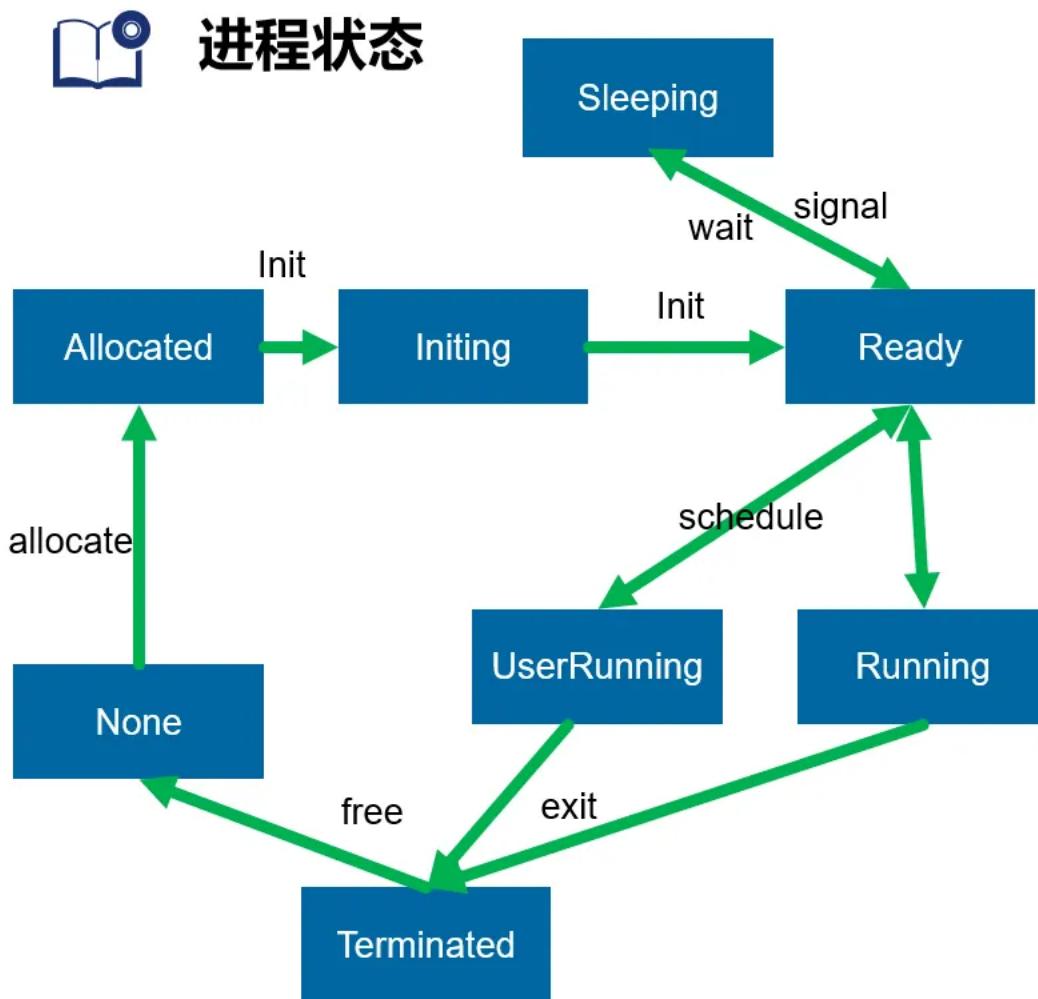
1  class Process {
2      friend ProcessManager;
3      friend Semaphore;
4      friend FileObjectManager;
5  public:
6      ClockTime timeBase; // Round Robin时间片轮转调度实现需要 计时起点
7      ClockTime runTime; // 进程运行的时间
8      ClockTime
9          trapSysTimeBase; // 为用户进程设计的 记录当前陷入系统调用时的起始时刻
10     ClockTime sysTime; // 为用户设计的时间 进行系统调用陷入核心态的时间
11     ClockTime sleepTime; // 挂起态等待时间 wait系统调用会更新
12             // 其他像时间等系统调用会使用
13     ClockTime waitTimeLimit; // 进程睡眠需要 设置睡眠时间的限制
14             // 当sleepTime达到即可自唤醒
15     ClockTime readyTime; // 就绪态等待时间(保留设计 暂不使用)
16     Uint32 SemRef; // wait的进程数
17
18     PID id;//pid 从0开始计数
19     ProcessManager* pm;//与之相关联的进程管理器
20     ProcStatus status;//进程状态
21     void* stack;//进程的内核栈
22     Uint32 stacksize;
23     VirtualMemorySpace* VMS;//虚拟内存管理
24     file_object* fo_head;//文件object拥有通过文件描述符管理文件，实际是一个打开文
件的链表
25
26  public:
27      // 关于父节点及子节点的链接
28      Process* father;
29      Process* broPre;
30      Process* broNext;
31      Process* fstChild;
32
33  private:
34
35      char* curWorkDir;//工作路径
36      Semaphore* waitSem;//进程专属信号量
37      HeapMemoryRegion* Heap;//进程的堆区
38      TrapFrame* context;//上下文
39      Uint64 flags;//表示进程的状态，如用户态还是内核态，同时可以实现自动内存回收
40      char name[PROC_NAME_LEN];//进程名称
41      Uint32 nameSpace;
42      Uint32 exitCode;//结束返回值
43      //... 剩余方法就不再赘述了
44  }
45

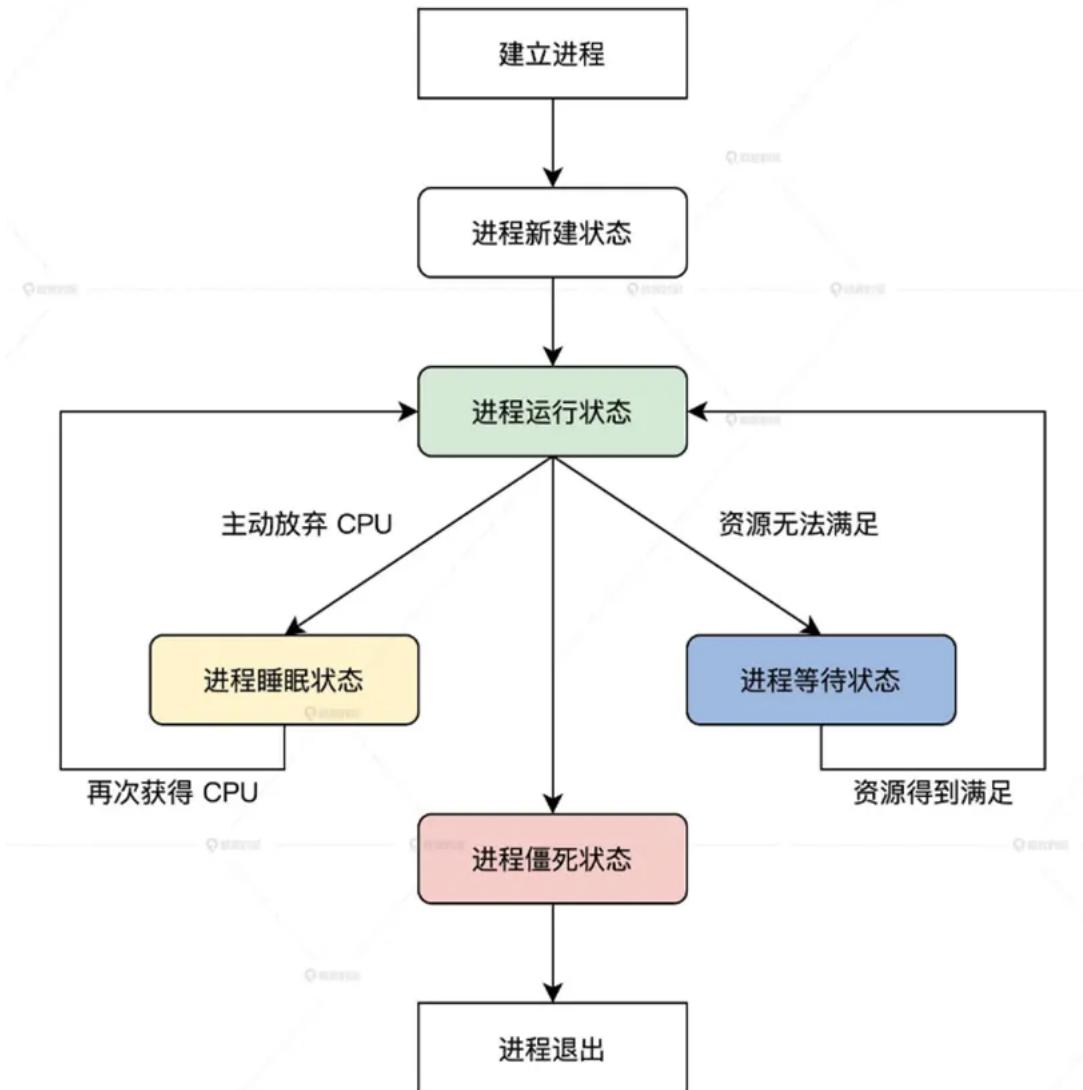
```

其中进程分为8个状态，用于描述进程的生命周期

```
1 enum ProcStatus : UInt32 {
2     S_None = 0,
3     S_Allocated,
4     S_Initing,
5     S_Ready,
6     S_Running,
7     S_UserRunning,
8     S_Sleeping,
9     S_Terminated,
10
11 };
```

进程获得空间为allocated状态，initing为正在初始化状态，防止其他进程打断该状态。start和init函数会使进程转化为Ready为可调度状态，schedule会调度ready状态的进程，被调度的进程转化为Running态，UserRunning为用户运行态用于和内核进程区分。wait操作后转为Sleep挂起态，进程exit后则为Terminal僵死态，最后再次被回收资源destroy后回归None的状态。





ProcessManager

在管理众多进程中Process暂时使用了数组进行管理，使得分配进程和释放进程都变得极为简单。在进程管理的核心则是调度算法，使用了简单的时间片流转算法。

```

1 TrapFrame* ProcessManager::Schedule(TrapFrame* preContext)
2 {
3     Process* tar;
4
5     kout[Debug] << "Schedule NOW " << curProc->getName() << endl;
6     curProc->context = preContext;//记录当前状态, 防止只有一个进程但是触发调度,
导致进程号错乱
7     // kout<<Blue<<procCount<<endl;
8     if (curProc != nullptr && procCount >= 2) {
9         int i, p;
10        ClockTime minWaitingTarget = -1;
11        RetrySchedule:
12        for (i = 1, p = curProc->id; i < MaxProcessCount; ++i) {
13            tar = &Proc[(i + p) % MaxProcessCount];
14            // if (tar->status == S_Sleeping && NotInSet(tar->SemWaitingTa
rgetTime, 0ull, (Uint64)-1)) {//Sleep的休眠时间管理, 目前还未实现
15            //     minWaitingTarget = minN(minWaitingTarget, tar->SemWaiti
ngTargetTime);
16            //     if (GetClockTime() >= tar->SemWaitingTargetTime)
17            //         tar->SwitchStat(Process::S_Ready);
18            // }
19            // kout<<p<<"P+i "<<(p+i)%MaxProcessCount<<tar->status<<endl;
20            // pm.show();
21            if (tar->status == S_Ready) {//如果是ready态则进行切换
22                tar->getVMS()->showVMRCount();
23                tar->run();
24                return tar->context;
25            } else if (tar->status == S_Terminated && (tar->flags & F_Auto
Destroy))//如果为自动销毁且为僵死态则进行销毁
26                tar->destroy();
27            }
28        }
29
30        return pm.getKernelProc()->context;//如果没有任何调度则进入内核态, 防止出错
31    }
32

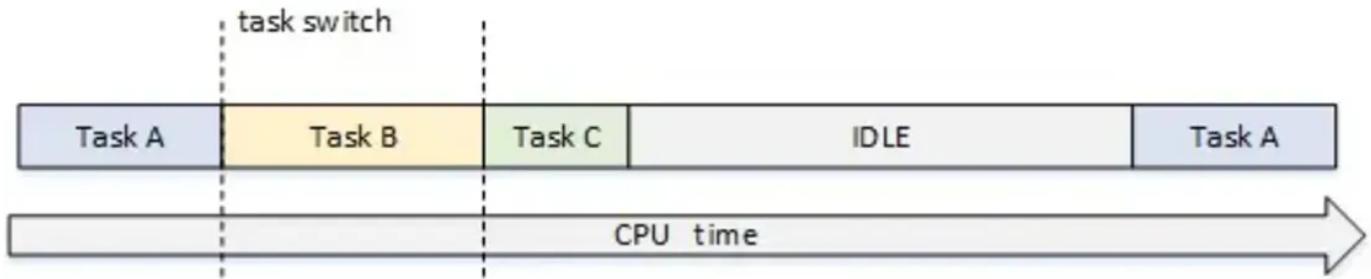
```

此进程管理器的精髓在于传入TrapFrame和返回TrapFrame，使用同样的a0去修改TrapFrame，对于中断来说，如果不更改a0则是中断，而更改a0则是调度，使用了相同的接口去实现不同的状态

进程切换

进程切换是操作系统在并发执行环境中，为了管理多个进程的执行而执行的一种操作。进程切换是操作系统在多个进程之间共享CPU时间的一种方式，通过快速保存和恢复进程它涉及保存当前运行进程的上下文（包括CPU寄存器状态、程序计数器、内存管理信息等），以便之后能够恢复该进程的执行，并加

载下一个要运行的进程的上下文到CPU中，从而允许该进程继续执行。简而言之，进程切换是操作系统在多个进程之间共享CPU时间的一种方式，通过快速保存和恢复进程的上下文来实现进程之间的无缝切换。



Futex

Futex (Fast Userspace Mutex) 是一种用户态和内核态混合的同步机制，对于我们的操作系统来说，Futex起到了关键的同步与并发控制的作用。

Futex应用场景

Futex主要用于进程间的同步。在这种情况下，需要同步的进程通过共享一段内存，futex变量就位于这段共享的内存中并且其操作是原子的。当进程试图进入或退出临界区的时候，会先去查看共享内存中的futex变量。

如果没有竞争发生（例如没有其他进程试图修改这个futex变量），那么进程只需要修改futex变量，而不必执行系统调用。这样可以避免系统调用带来的开销，从而在没有竞争的情况下提高了效率。

如果有竞争发生（例如有其他进程也试图修改这个futex变量），那么进程会执行系统调用，陷入内核态，由内核来处理竞争。简单的说，Futex通过在用户态的检查，避免了不必要的内核态切换，大大提高了低竞争时的效率。

Futex系统调用

在我们的操作系统中，futex系统调用是通过 `sys_futex()` 函数实现的。该函数根据futex操作的类型进行相应的处理，这些类型可以是FUTEX_WAIT、FUTEX_WAKE或FUTEX_REQUEUE。接下来，我们详细介绍这个函数。

sys_futex函数定义

```
Uint64 Syscall_futex(Uint64 *uaddr, int op, Uint64 val, struct timespec2 *timeout, Uint64 *uaddr2, Uint64 val3)
```

这个函数的返回类型是 `uint64`，表明它返回一个64位的整型值。它没有任何参数，所有参数通过系统调用接口来传递。

sys_futex函数参数

`sys_futex()` 函数的参数通过系统调用接口来获取，这些参数包括：

1. `uaddr`：这是一个指向futex变量的指针。
2. `futex_op`：这是一个整型数，表示futex操作的类型。
3. `val`：这个参数在不同的futex操作中有不同的含义。在FUTEX_WAIT操作中，它是期望的futex变量的值；在FUTEX_WAKE操作中，它是最多唤醒的线程数；在FUTEX_REQUEUE操作中，它是最多唤醒和重新排队的线程数。
4. `timeout`：这是一个指向 `TimeSpec2` 结构的指针，表示等待的最大时间。只有在 FUTEX_WAIT操作中才使用这个参数。
5. `uaddr2`：这是一个指向futex变量的指针，只有在FUTEX_REQUEUE操作中才使用这个参数。
6. `val3`：这个参数只有在某些特殊的futex操作中才使用，我们的 `sys_futex()` 函数没有使用这个参数。

sys_futex函数行为

`sys_futex()` 函数首先获取用户态传递来的参数。如果任何一个参数获取失败，那么它会立即返回-1。

然后，它根据futex操作的类型来进行相应的处理：

1. 如果是FUTEX_WAIT操作，那么就调用 `futexWait()` 函数，让当前线程等待futex变量的值变为期望的值。
2. 如果是FUTEX_WAKE操作，那么就调用 `futexWake()` 函数，唤醒等待futex变量的线程。
3. 如果是FUTEX_REQUEUE操作，那么就调用 `futexRequeue()` 函数，将等待一个futex变量的线程重新排队到另一个futex变量上。

如果操作类型不是以上三种，那么 `sys_futex()` 函数会产生panic，因为它不支持其他类型的操作。

在所有操作完成之后，`sys_futex()` 函数返回0，表示系统调用成功。

sys_futex函数的异常处理

`sys_futex()` 函数对各种可能出现的异常进行了处理。如果参数获取失败，那么它会立即返回-1。在 FUTEX_WAIT操作中，如果futex变量的值不等于期望的值，那么它也会返回-1。如果操作类型不是支持的类型，那么它会产生panic。这样的异常处理使得 `sys_futex()` 函数在各种异常情况下都能有正确的行为。

总的来说，`sys_futex()` 函数是我们的操作系统AVX512OS中实现futex机制的核心函数。它将复杂的futex操作封装在一个简单的接口中，使得在用户态的进程可以方便地使用futex机制进行同步。

futex具体实现

在AVX512OS中，我们针对FUTEX_WAIT、FUTEX_WAKE和FUTEX_REQUEUE三种类型的操作进行了处理。在代码实现中，我们定义了一个FutexQueue的队列结构，用于存放处于等待状态的线程信息。

- `addr` : 等待的Futex变量的地址。
- `thread` : 需要等待的线程。
- `valid` : 该队列项是否有效。

```
1 struct FutexQueue
2 {
3     uint64 addr;
4     thread* thread;
5     uint8 valid;
6 } FutexQueue;
```

FutexQueue队列的长度是FUTEX_COUNT，这是我们设定的最大等待线程数量。当线程需要等待一个Futex变量时，它会被加入到这个队列中。

以下是三个函数的具体实现：

futexWait()

```
1 void futexWait(uint64 addr, thread* th, TimeSpec2* ts);
```

在futexWait函数中，线程首先会在FutexQueue队列中找到一个未被使用的项，然后将自己的信息填入这个项中，并将项标记为已使用。然后根据是否有timeout参数来决定线程的状态。如果有timeout，那么线程的状态将被设为S_Sleeping，并计算出应当唤醒的时间。如果没有timeout，那么线程的状态将被设为S_Sleeping。最后，线程会切换到可运行状态，并调度下一个线程运行。

futexWake()

```
1 void futexWake(uint64 addr, int n);
```

在futexWake函数中，我们会遍历FutexQueue队列，找到等待给定地址的Futex变量的线程，并将其唤醒，直到唤醒的线程数量达到n个。唤醒一个线程的操作是将其状态设为S_Ready，并将返回值设为0。同时我们还需要将这个FutexQueue队列项标记为未使用。

futexRequeue()

```
1 void futexRequeue(uint64 addr, int n, uint64 newAddr);
```

futexRequeue函数首先会执行和futexWake一样的操作，唤醒n个等待给定地址的Futex变量的线程。然后，它会遍历FutexQueue队列，找到所有还在等待原来地址的Futex变量的线程，并将它们等待的地址改为newAddr。这样，这些线程在被唤醒时，将会操作newAddr指向的Futex变量。

最后，我们实现了一个futexClear函数，用于清理一个线程的所有等待状态。当线程退出时，我们需要调用这个函数，来避免其他线程在等待一个已经不存在的线程。

```
1 void futexClear(thread* thread);
```

在这个函数中，我们会遍历FutexQueue队列，找到所有等待给定线程的项，并将它们标记为未使用。总的来说，Futex在EcallFinal1中提供了一种高效的进程同步机制，能够在用户态解决大部分的竞争情况，从而避免了频繁的内核态切换。只有在竞争情况下，才会切换到内核态，由内核来完成竞争的解决。

同步原语

同步原语包括三个部分

- 自旋锁

自旋锁通过g++编译器中的`_sync_synchronize()`完成，本质上是调用riscv指令集中的栅障指令实现，主要用于保持操作的原子性，实现较为简单。

- 互斥锁

互斥锁通过自旋锁来实现，不过增加了进程号，同样的锁只允许上锁进程解锁。保证了资源的“互斥”。

- 信号量

信号量则是通过系统调度来实现进程的阻塞。

这里使用的信号量是简易的队列信号量，通过入队和出队操作实现信号量。

而在资源用尽的时候则改变进程状态，为了使操作互不干扰，在wait阻塞进程后不会触发立即调度，而是等待时间片结束后再进行立即调度。至于是否立即调度可以由调用者自己决定。将来会添加定时自唤醒功能，目前时间有限就暂时放弃了。

硬盘驱动部分

硬盘驱动主要是与Virtio进行通信

重点则是要搞清楚协议的设定，Virtio不只是一个设备，而是支持许多设备，硬盘主要是使用Virtio_blk协议。

本协议主要在于Vring的管理，宿主机和客户OS共同维护设备环。分为四个部分

- 初始化

按照文档将对应内存中的值填好即可

客户OS要注意分配Vring所在位置，同时管理avail部分

- 客户OS操作Vring环

分配Vring上三个Desc块，操作完成之后，修改avail部分的可用值，并对特定内存写入通知宿主机

- 宿主机读取环后执行操作，并通知客户OS

宿主机读取avail后读取Desc块执行相应操作，修改used部分的值，并通过提前约定的中断值通知客户OS

- 客户OS在接受到消息后再次确认

可以看出宿主OS可以使用轮询used或者接收中断的方式相应操作，由于实现驱动的时候信号量测试还不完全，于是使用了轮询的方式实现查询

```
1      //....
2      InterruptEnable();
3      *R(VIRTIO_MMIO_QUEUE_NOTIFY) = 0; //通知QEMU
4
5      while (last_used_idx==used->id) { //轮询操作,等待中断
6      }
7
8      // kout[Debug] << "!!!!!4!!!!!" << endl;
9      InterruptDisable();
10
11     free_chain(idx[0]);
12     IntrRestore(a);
13 }
14
15 void VirtioDisk::virtio_disk_intr()
16 {
17     // kout << "intr" << endl;
18     // kout[Debug] << used->id << endl;
19     // kout[Debug] << last_used_idx << endl;
20     *R(VIRTIO_MMIO_INTERRUPT_ACK) = *R(VIRTIO_MMIO_INTERRUPT_STATUS) & 0x3
21     ;//确认受到响应
22 }
23
24
```

本操作系统使用轮询+中断的方式实现，中断用于返回确认值。由于时间紧张，将来会改成信号量管理

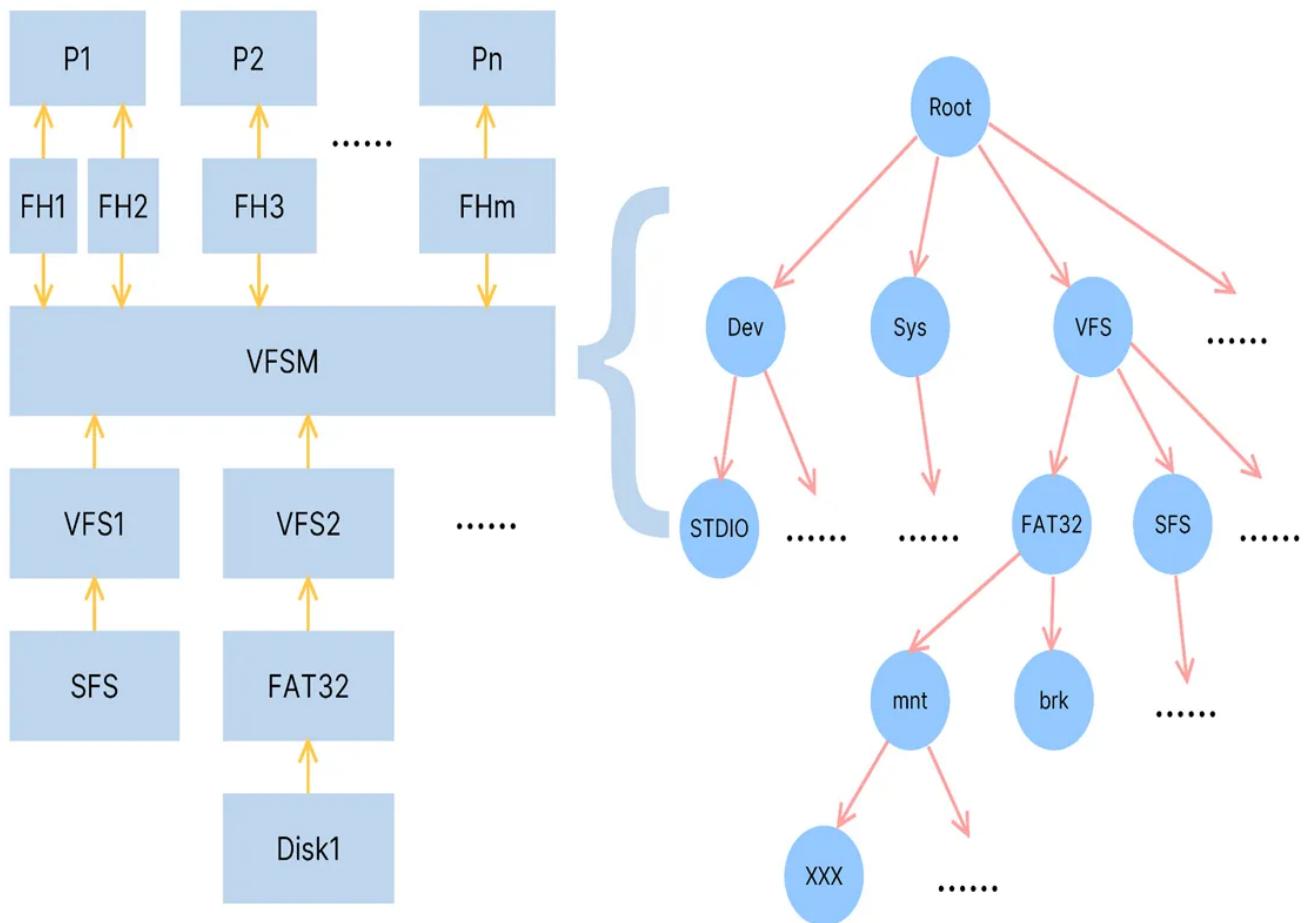
文件系统

VFSM

文件系统先通过VFSM将管理虚拟文件系统，虽然目前只支持FAT32,但是方便了将来的扩展
VFSM通过链表去管理当前打开的文件，通过路径去实现所有操作。同时提供了一些路径操作的方式。
除此之外VFSM的其他操作是通过调用其他VFS实现的，由于我们这里只实现了FAT32,所以VFSM主要是通过调用FAT32的功能实现

```
1  class VFSM
2  {
3      private:
4          FAT32FILE* OpenedFile;
5
6          FAT32FILE* find_file_by_path(char* path, bool& isOpened);
7
8      public:
9          FAT32FILE* open(const char* path, char* cwd);
10         FAT32FILE* open(FAT32FILE* file);
11         void close(FAT32FILE* t);
12
13         void showRootDirFile();
14         bool create_file(const char* path, char* cwd, char* fileName, Uint8 type = FATtable::FILE);
15         bool create_dir(const char* path, char* cwd, char* dirName);
16         bool del_file(const char* path, char* cwd);
17         bool link(const char* srcpath, const char* ref_path, char* cwd); //ref_path为被指向的文件
18         bool unlink(const char* path, char* cwd);
19         bool unlink(char* abs_path);
20         FAT32FILE* get_next_file(FAT32FILE* dir, FAT32FILE* cur = nullptr, bool (*p)(FATtable* temp) = VALID); // 获取到的dir下cur的下一个满足p条件的文件，如果没有则返回空
21
22         char* unified_path(const char* path, char* cwd);
23         void show_opened_file();
24         bool init();
25         bool destory();
26         FAT32FILE* get_root();
27     };
28 }
```

VFSM中还提供了PATH的通用工具用于处理路径。



图中包含了多个节点，分区（P₁, P₂, ..., P_n）、根目录（Root）、文件层次结构（F_{H1}, F_{H2}, ..., F_{Hm}）、设备文件（Dev）、系统文件（Sys）、虚拟文件系统（VFS）及其管理模块（VFSM）和具体实例（VFS1, VFS2）。此外，还展示了标准输入输出接口（STDIO）、文件系统类型（如FAT32）以及挂载点（mnt）和堆扩展点（brk）。

FAT32文件系统

FAT32文件系统的难点在于从网络中的各个FAT版本中构建出一个可以读取vfat的协议，根据协议编写相应的文件系统。

- 初始化

从FAT32前512个字节中读取出基本的Fat32数据，包括fat1lba fat2lba 和 data1lba的分布。

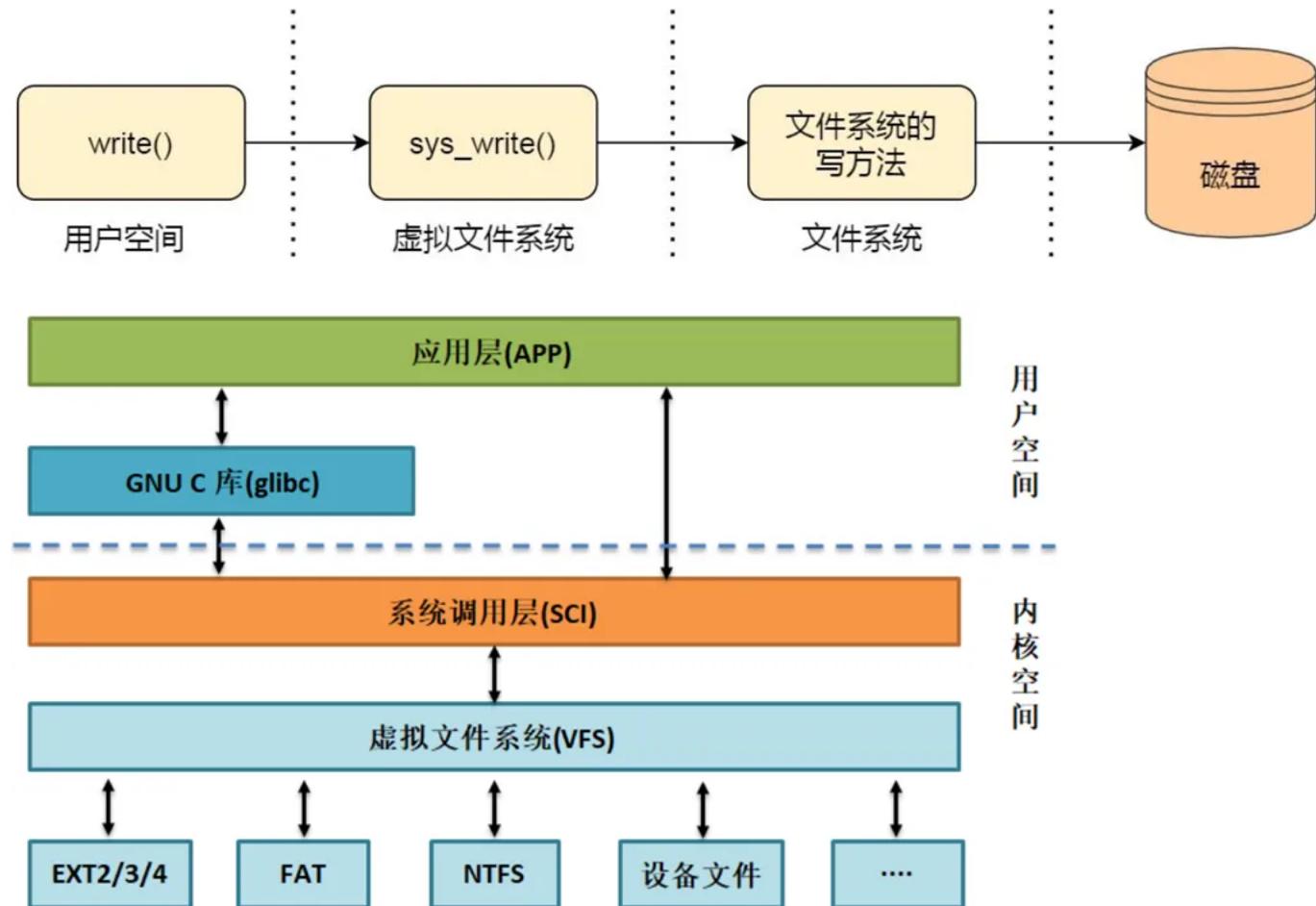
- 基础功能

- clus号到lba,lba到clus号
- 从clus中读取数据，向clus中填写数据,清除clus数据

- 设置clus的下一个clus
- 查找空的clus
- 读写fatTable
- 文件功能实现
 - 打开文件
通过读写fatTable中的clus找到装载文件的clus,递归处理, 就可以读取到目标路径
 - 读取写件
通过读取fatTable中的clus找到文件内容, 且可以通过获取下一个clus读取到全部的信息。设置同理, 对于多出的文件内容, 可以通过查找空的clus进行设置
 - 设置文件基本信息
设置fatTable即可设置文件基本信息

open

VFSM通过链表管理打开的文件, 将打开的文件存储到链表中, 加快文件读取速度。同时mount也是Fat32不支持的操作, 所以在用另一个链表去管理mount的映射关系



fileObject

通过进一步抽象，将文件抽象成为FileObject,同时使用FileObjectManage进行管理，获取相应的文件描述符，来管理文件。仍然使用链表管理。

EXT4

介绍

Ext4文件系统是以块（Block）的方式管理文件的，默认单位为4KB一块。一个1GB大小的空间，ext4 文件系统将它分隔成了0~7的8个Group。每个Group中又有superblock、Group descriptors、bitmap、Inode table、usrer data、还有一些保留空间。



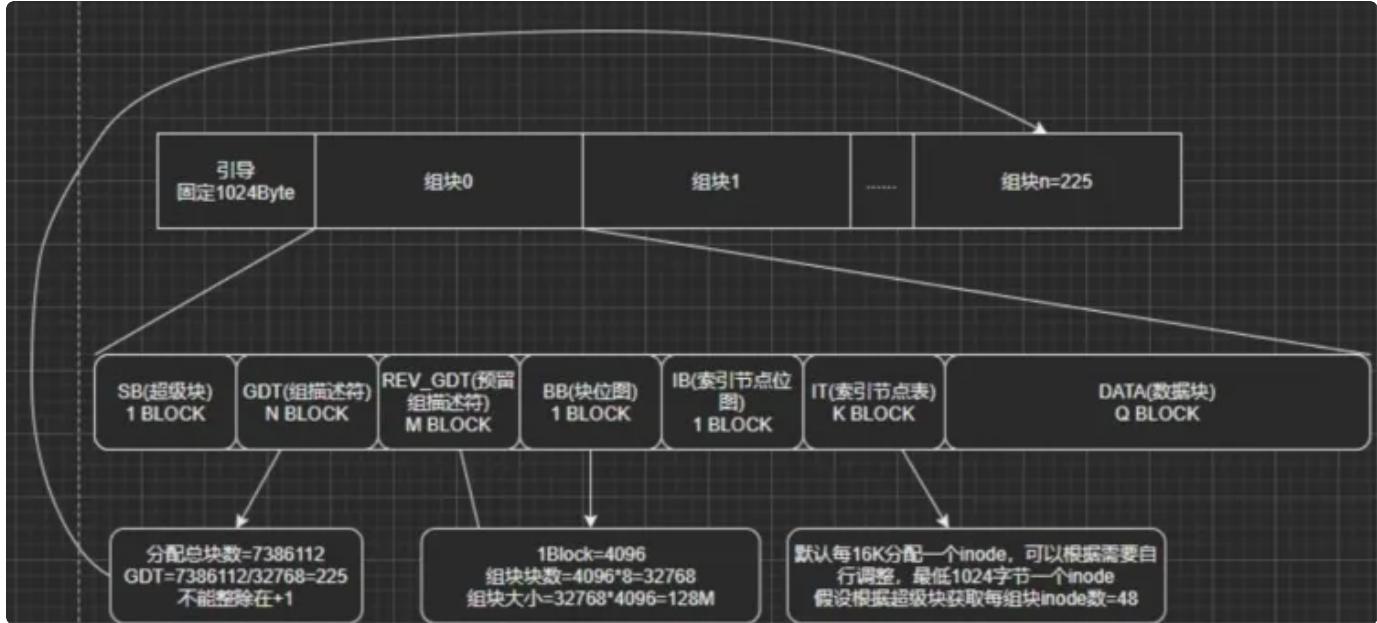
inode (索引节点)：inode是文件系统中的一个数据结构，用于存储文件或目录的元数据信息，如文件类型、权限、拥有者、大小、时间戳等。每个文件或目录都有一个对应的 inode来描述其属性和位置。

数据区块：数据区块是用于存储文件内容的实际数据块。当文件被创建或修改时，其内容被存储在数据区块中。ext4 文件系统将文件内容分散存储在多个数据块中，以提高文件系统的效率和性能。

超级块：超级块是 ext4 文件系统的关键数据结构之一，它存储了文件系统的元数据信息，如文件系统的大小、inode数量、数据块数量、挂载选项等。每个文件系统只有一个超级块，位于文件系统的开头位置。

块组 (Block Group)：块组是 ext4 文件系统的逻辑单元，用于组织和管理文件系统中的数据。每个块组包含一组连续的数据块、inode和位图等。块组有助于提高文件系统的性能和可管理性。

位图：位图是用于跟踪数据块和 inode的使用情况的数据结构。每个块组都有自己的位图，用于标记已分配和未分配的数据块和 inode。



通过移植官方给的C实现的EXT4到我们的项目。我们构造了myext4.hpp用来对接官方项目和我们的vfsm.hpp。

ext4node

```

1  class ext4node : public FileNode {
2
3     DEBUG_CLASS_HEADER(ext4node);
4
5 public:
6     ext4_file fp;
7     ext4_dir fd;
8     Sint64 offset;
9
10    void initlink(ext4_file tb, const char* Name, EXT4* fat_);
11    void initfile(ext4_file tb, const char* Name, EXT4* fat_);
12    void initdir(ext4_dir tb, const char* Name, EXT4* fat_);
13
14    bool set_name(char* _name) override; //
15    Sint64 read(void* dst, Uint64 pos, Uint64 size) override; //
16    Sint64 read(void* dst, Uint64 size) override; //
17    Sint64 write(void* src, Uint64 pos, Uint64 size) override; //
18    Sint64 write(void* src, Uint64 size) override; //
19    Sint64 close()
20 {
21     if (name) {
22         delete[] name;
23     }
24     name=nullptr;
25 }
26
27 // bool del(); //这个我看fat32也没有实现，而是实现vfs的del，所以我也先空着了
28
29 void show(); //
30 ext4node() {
31     // kout<<"ext4node create"<<endl;
32 };
33 ~ext4node() {};
34 };

```

继承FileNode, FileNode即通用的文件结点接口。查看ext4.h中的结构体信息，我们提取出

```
1  /**@brief  File descriptor. */
2  typedef struct ext4_file {
3
4      /**@brief  Mount point handle.*/
5      struct ext4_mountpoint *mp;
6
7      /**@brief  File inode id.*/
8      uint32_t inode;
9
10     /**@brief  Open flags.*/
11     uint32_t flags;
12
13     /**@brief  File size.*/
14     uint64_t fsize;
15
16     /**@brief  Actual file position.*/
17     uint64_t fpos;
18 } ext4_file;
```

以及用于存储文件夹格式的

```
1  /**@brief  Directory descriptor. */
2  typedef struct ext4_dir {
3      /**@brief  File descriptor.*/
4      ext4_file f;
5      /**@brief  Current directory entry.*/
6      ext4_dirent de;
7      /**@brief  Next entry offset.*/
8      uint64_t next_off;
9 } ext4_dir;
```

, 更有负责获取文件夹中下一个文件的结构体用于实现get_next_file:

```
1  /**@brief  Directory entry descriptor. */
2  typedef struct ext4_dirent {
3      uint32_t inode;
4      uint16_t entry_length;
5      uint8_t name_length;
6      uint8_t inode_type;
7      uint8_t name[255];
8 } ext4_dirent;
```

还有用于存储文件挂载点信息的结构体:

```
1  /**@brief  Some of the filesystem stats. */
2  struct ext4_mount_stats {
3      uint32_t inodes_count;
4      uint32_t free_inodes_count;
5      uint64_t blocks_count;
6      uint64_t free_blocks_count;
7
8      uint32_t block_size;
9      uint32_t block_group_count;
10     uint32_t blocks_per_group;
11     uint32_t inodes_per_group;
12
13     char volume_name[16];
14 };
```

熟悉需要用到的结构体信息并将这些结构体加入ext4node类的成员变量中，方便成员函数的调用。对于成员函数，运用在ext4.h中的几个重要函数来实现：

```

1  /**@brief  File open function.
2   *
3   * @param file  File handle.
4   * @param path  File path, has to start from mount point:/my_partition/f
5   * @param flags File open flags.
6   * |-----|
7   * | r or rb          O_RDONLY
8   * |-----|
9   * | w or wb          O_WRONLY|O_CREAT|O_TRUNC
10  * |-----|
11  * | a or ab          O_WRONLY|O_CREAT|O_APPEND
12  * |-----|
13  * | r+ or rb+ or r+b O_RDWR
14  * |-----|
15  * | w+ or wb+ or w+b O_RDWR|O_CREAT|O_TRUNC
16  * |-----|
17  * | a+ or ab+ or a+b O_RDWR|O_CREAT|O_APPEND
18  * |-----|
19  *
20  * @return Standard error code.*/
21 int ext4_fopen(ext4_file *file, const char *path, const char *flags);
22
23 /**@brief  File close function.
24   *
25   * @param file File handle.
26   *
27   * @return Standard error code.*/
28 int ext4_fclose(ext4_file *file);
29
30 /**@brief  Read data from file.
31   *
32   * @param file File handle.
33   * @param buf  Output buffer.
34   * @param size Bytes to read.
35   * @param rcnt Bytes read (NULL allowed).
36   *
37   * @return Standard error code.*/
38 int ext4_fread(ext4_file *file, void *buf, size_t size, size_t *rcnt);
39
40 /**@brief  Write data to file.
41   *
42   * @param file File handle.
43   * @param buf  Data to write
44   * @param size Write length..
45   * @param wcnt Bytes written (NULL allowed).
46   *

```

```

47     * @return Standard error code.*/
48 int ext4_fwrite(ext4_file *file, const void *buf, size_t size, size_t *wcn
t);
49
50 /**
51  * @brief File seek operation.
52  *
53  * @param file File handle.
54  * @param offset Offset to seek.
55  * @param origin Seek type:
56  *               @ref SEEK_SET
57  *               @ref SEEK_CUR
58  *               @ref SEEK_END
59  *
60  * @return Standard error code.*/
61 int ext4_fseek(ext4_file *file, int64_t offset, uint32_t origin);
62
63 /**
64  * @brief Rename/move directory.
65  *
66  * @param path      Source path.
67  * @param new_path Destination path.
68  *
69  * @return Standard error code.*/
70 int ext4_dir_mv(const char *path, const char *new_path);
71
72 /**
73  * @brief Create new directory.
74  *
75  * @param path Directory name.
76  *
77  * @return Standard error code.*/
78 int ext4_dir_mk(const char *path);
79
80 /**
81  * @brief Directory open.
82  *
83  * @param dir  Directory handle.
84  * @param path Directory path.
85  *
86  * @return Standard error code.*/
87 int ext4_dir_open(ext4_dir *dir, const char *path);
88
89 /**
90  * @brief Directory close.
91  *
92  * @param dir directory handle.
93  *
94  * @return Standard error code.*/
95 int ext4_dir_close(ext4_dir *dir);
96
97 /**
98  * @brief Return next directory entry.
99  *

```

```

94     * @param    dir Directory handle.
95     *
96     * @return   Directory entry id (NULL if no entry) */
97 const ext4_dirent *ext4_dir_entry_next(ext4_dir *dir);

```

即可实现从FileNode处继承来的函数



ext4node继承FileNode

```

class ext4node : public FileNode {
... DEBUG_CLASS_HEADER(ext4node);
public:
... ext4_file fp;
... ext4_dir fd;
... Sint64 offset;

... void initlink(ext4_file tb, const char* Name, EXT4* fat_);
... void initfile(ext4_file tb, const char* Name, EXT4* fat_);
... void initdir(ext4_dir tb, const char* Name, EXT4* fat_);

... bool set_name(char* _name) override; //

... Sint64 read(void* dst, Uint64 pos, Uint64 size) override; //
... Sint64 read(void* dst, Uint64 size) override; //
... Sint64 readoffset(void* buf_, Uint64 offset, Uint64 size) override;
...
... |
... Sint64 write(void* src, Uint64 pos, Uint64 size) override; //
... Sint64 write(void* src, Uint64 size) override; //

```

```

int ext4_rename(const char *path, const char
*new_path);
int ext4_dir_mv(const char *path, const char
*new_path);

int ext4_fopen(ext4_file *file, const char *path, const
char *flags);
int ext4_fread(ext4_file *file, void *buf, size_t size,
size_t *rcnt);
int ext4_fseek(ext4_file *file, int64_t offset, uint32_t
origin);

int ext4_fopen(ext4_file *file, const char *path, const
char *flags);
int ext4_fwrite(ext4_file *file, const void *buf, size_t
size, size_t *wcnt);
int ext4_fseek(ext4_file *file, int64_t offset, uint32_t
origin);

```

EXT4

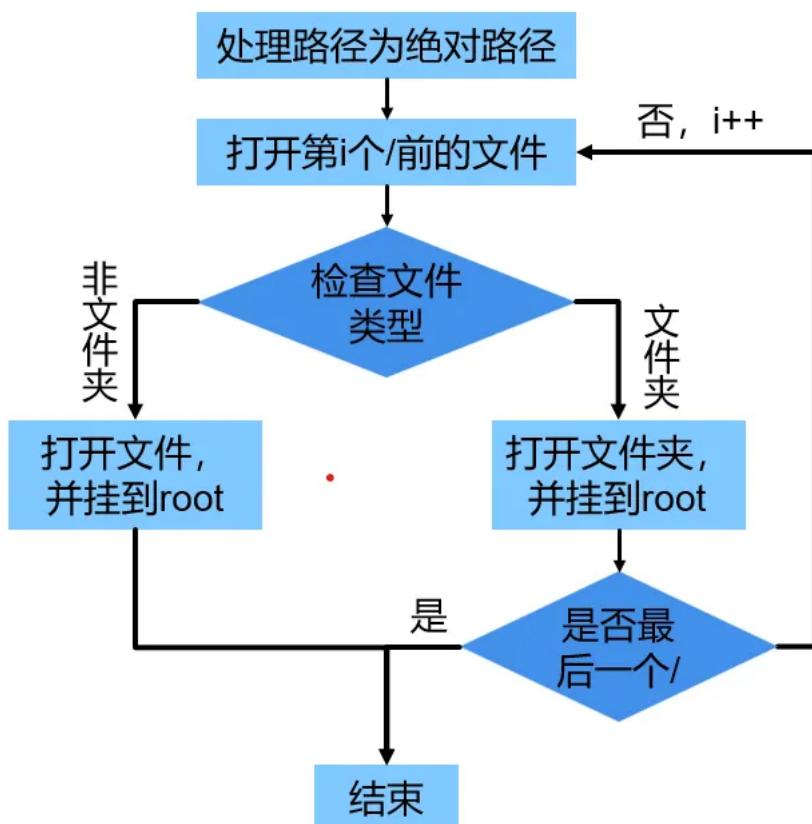
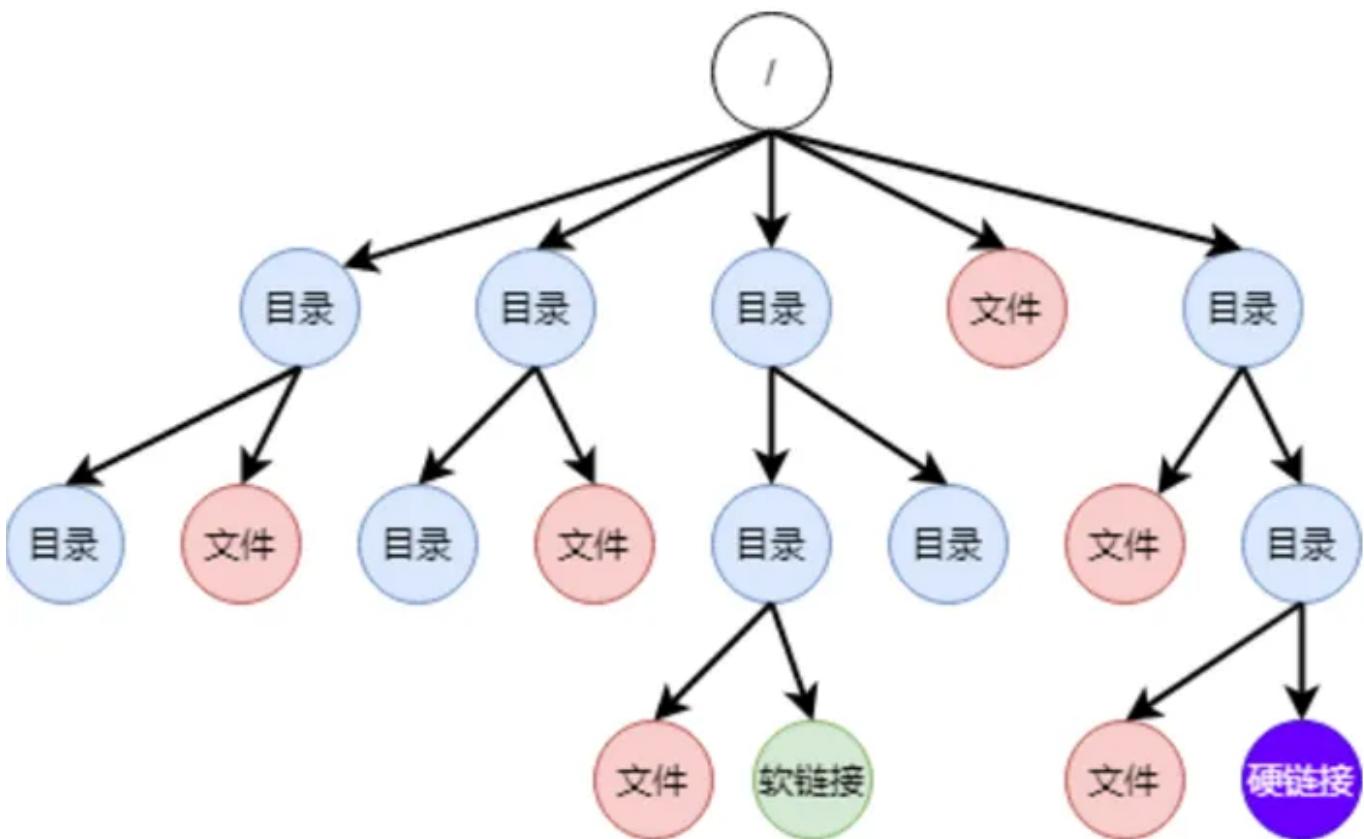
EXT4继承先前的VFS，方便与vfsm对接。

```

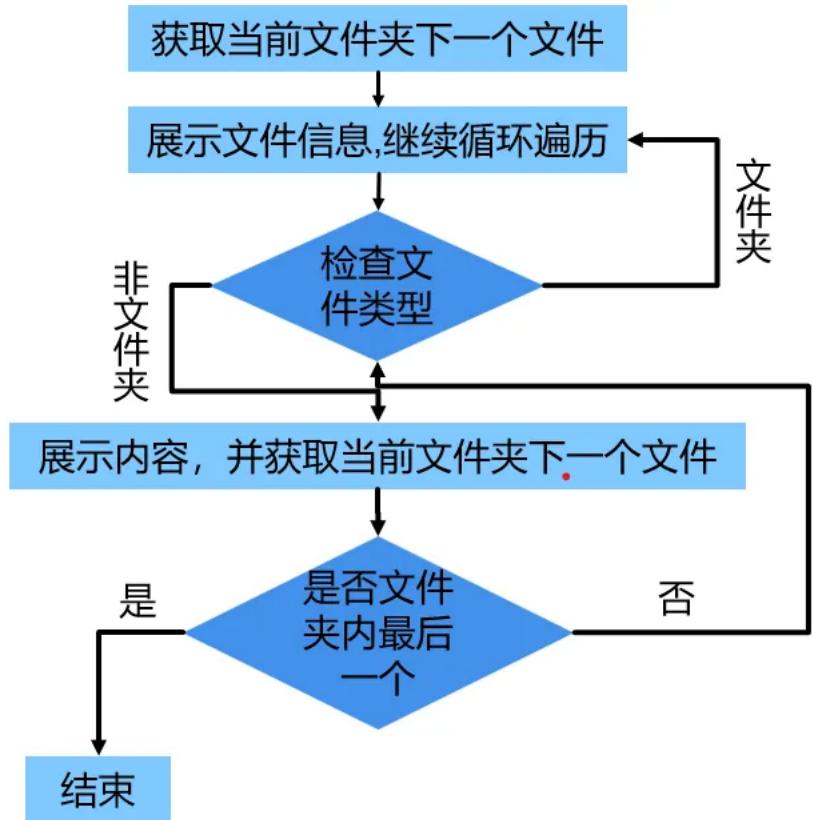
1  class EXT4 : public VFS {
2
3      DEBUG_CLASS_HEADER(EXT4);
4      friend class ext4node;
5      friend class VFSM;
6
7  public:
8      char* FileSystemName() { return nullptr; };
9      void show_all_file_in_dir(ext4node* dir, UInt32 level = 0); //
10     FileNode** get_all_file_in_dir(FileNode* dir, bool (*p)(FileType type)
11 ) override
12     {
13         return nullptr;
14     }; //
15     ext4node* open(const char* _path, FileNode* parent) override; //
16     ext4node* get_node(const char* path) override
17     {
18         return nullptr;
19     };
20     bool close(FileNode* p) override
21     {
22         return true;
23     }; //
24     bool del(FileNode* file) override;
25
26     ext4node* create_file(FileNode* dir_, const char* fileName, FileType t
27 ype = FileType::__FILE__ override; //
28     ext4node* create_dir(FileNode* dir_, const char* fileName) override;
29     //
30
31     bool init() { return 1; }; //
32
33     void get_next_file(ext4node* dir, ext4node* cur, ext4node* tag); //
34     EXT4() {};
35     ~EXT4() {};
36 };

```

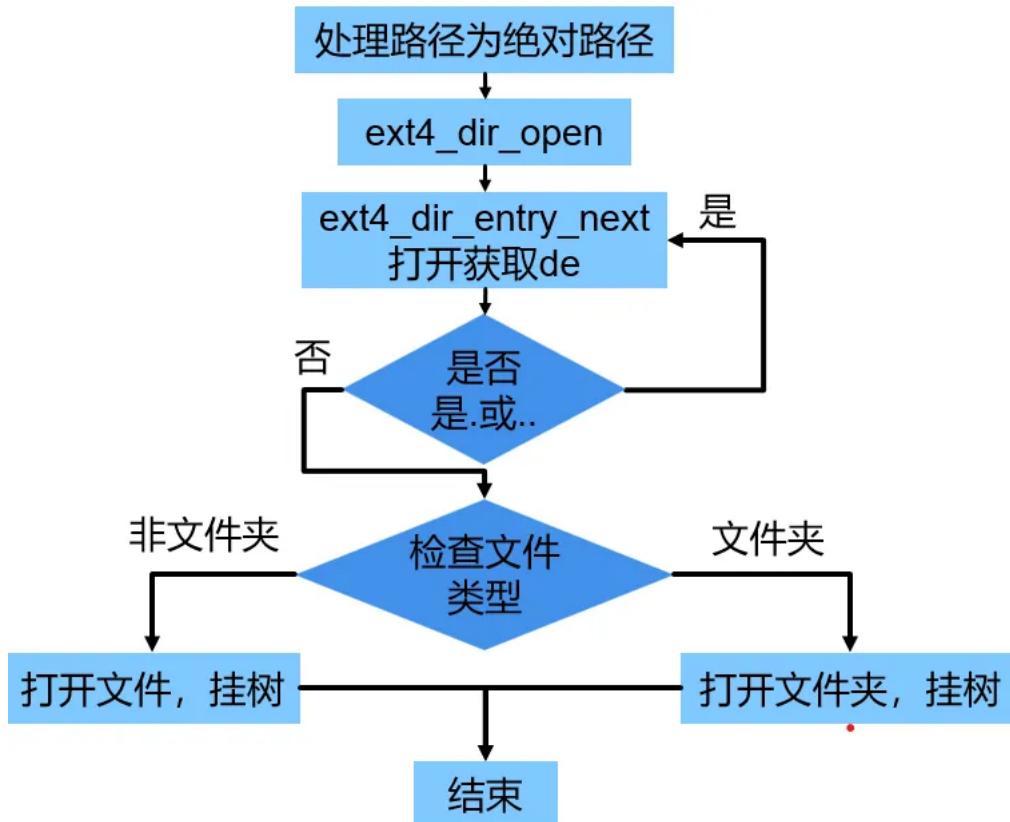
重写继承自vfs的函数，用ext4内部函数来具体实现。比较重要的是open，open沿着绝对路径打开并将打开的文件挂载到文件树上



show_all_file_in_dir则根据get_next_file来实现.



get_next_file则根据ext.h中的ext_direntry来实现。



ELF解析

ELF解析主要是从ELF头中解析出各段的作用，重点主要在于从磁盘到到虚拟内存的VMR设置，保证VMR权限设置合理。

- 解析头

```
1 struct Elf_Phdr
2 {
3     UInt32 p_type;           // 本程序头描述的段的类型
4     UInt32 p_flags;          // 本段内容的属性
5     UInt64 p_offset;         // 本段内容在文件中的位置 段内容的
开始位置相对于文件头的偏移量
6     UInt64 p_vaddr;          // 本段内容的开始位置在进程空间中的
虚拟地址
7     UInt64 p_paddr;          // 本段内容的开始位置在进程空间中的
物理地址
8     UInt64 p_filesz;         // 本段内容在文件中的大小 以字节为
单位
9     UInt64 p_memsz;          // 本段内容在内容镜像中的大小
10    UInt64 p_align;          // 对齐方式 可装载段来讲应该要向内
存页面对齐
11 } __attribute__((packed));
12
13
```

- 解析段

```
1 // 段类型的枚举表示
2 enum P_type
3 {
4     PT_NULL = 0,              // 本程序头是未使用的
5     PT_LOAD = 1,              // 本程序头指向一个可装载的段
6     PT_DYNAMIC = 2,            // 本段指明了动态链接的信息
7     PT_INTERP = 3,             // 本段指向一个字符串 是一个ELF解
析器的路径
8     PT_NOTE = 4,              // 本段指向一个字符串 包含一些附加
的信息
9     PT_SHLIB = 5,              // 本段类型是保留的 未定义语法
10    PT_PHDR = 6,              // 自身所在的程序头表在文件或内存中
的位置和大小
11    PT_LOPROC = 0x70000000,      // 为特定处理器保留使用的区间
12    PT_HIPROC = 0x7fffffff,
13 };
14
```

- 设置VMR

在VMS中设置了方便的接口，直接根据内存地址设置即可

```

1 // 权限统计完 可以在进程的虚拟空间中加入这一片VMR区域了
2 // 这边使用的memsz信息来作为vmr的信息
3 // 输出提示信息
4     UInt64 vmr_begin = pgm_hdr.p_vaddr;
5     UInt64 vmr_memsize = pgm_hdr.p_memsz;
6     UInt64 vmr_end = vmr_begin + vmr_memsize;
7     kout << "Add VMR from " << vmr_begin << " to " << vmr_end << " " <
8     < (void*)vmr_flags << endl;
9
10    // VirtualMemoryRegion* vmr_add = (VirtualMemoryRegion*)kmalloc(si
11    // zeof(VirtualMemoryRegion));
12    // vmr_add->Init(vmr_begin, vmr_end, vmr_flags);
13    VirtualMemoryRegion* vmr_add = new VirtualMemoryRegion(vmr_begin,
14    vmr_end, vmr_flags);
15    vms->InsertVMR(vmr_add);
16    vms->Enter();
17
18    memset((char*)vmr_begin, 0, vmr_memsize);
19    vms->Leave();
20
21    UInt64 tmp_end = vmr_add->GetEnd();
22    if (tmp_end > breakpoint) {
23        // 更新用户空间的数据段(heap)
24        breakpoint = tmp_end;
25    }
26
27    // 上面读取了程序头
28    // 接下来继续去读取段在文件中的内容
29    // 将对应的内容存放到进程中vaddr的区域
30    // 这边就是用filesz来读取具体的内容
31    fom.seek_fo(fo, pgm_hdr.p_offset, file_ptr::Seek_beg);
32    vms->Enter();
33    // kout<<Red<<"START"<<pgm_hdr.p_filesz<<endl;
34    rd_size = fom.read_fo(fo, (void*)vmr_begin, pgm_hdr.p_filesz);
35    vms->Leave();

```

- 写入VMR

由于riscv权限设置，专门设计了用内核态访问用户空间的函数

```

1     vms->EnableAccessUser();
2     vms->DisableAccessUser();

```

致谢

蔡蕾

决定参加

其实在这之前从来没有想过要去参加操作系统的比赛，偶然的机会了解到计算机系统能力大赛。同时，champion学长的建议参加os的内核实现赛，要知道在这之前我从来没学过什么是操作系统，但还是决定迈出第一步，去挑战不可能！

速成操作系统的理论

那两个星期基本上每天一有功夫就立刻抱着书去啃，有不懂的圈圈画画找学长们寻求解答，当时只觉得脑子很乱很乱，但后来自己的项目中也有很大的系统成分发现理解的更加透彻，很感谢当时自己的坚持（以前看书总是半途而废）。还会去看学长的培训回放以及计算机组成原理的知识，给我的感觉就是很乱很乱，甚至连什么时钟都理解不了，不过随着看代码的深入掌握算是略懂一点了，也使我计组学的更轻松了。在看代码的过程中，第一次知道c++那些extern, inline等的用途，原来以前的c++都是皮毛啊！想完全理解计算机还有很长的路要走啊！

最简单代码到内存管理

通过自己写裸机打印hello，更深刻理解了一个程序会经历什么以及内存布局的特点（linker文件）。同时了解到原来c++的cout等的基本原理是在系统调用啊，是一种中断后处理中断啊！还知道原来还是内嵌汇编的说法，也解决了很多需要通过汇编直接改变标志位甚至寄存器值的情况。作为初学者的我来说，看懂一份物理内存代码也耗时了好久，惊叹里面的算法（虽然本质只是简单的链表）想后续去更好的了解伙伴管理系统，现在的自己还差太多，连线段树是什么都不知道，哎！一个报错都找不到在哪的问题，虽说有自己的cout但还是有点不习惯这种很少的调试信息。加深了以前不怎么使用的位运算，受益匪浅！

虚拟内存

首先了解了虚拟内存是什么，但还是被一道题问住了，为什么call main会报错哩！可能现在的自己也不能给出非常好的回答，但大概测试感觉应该会是call是组合指令里面存在一些猫饼！希望随着自己知识的增长能理解的更为透彻吧！学会了在linker中添加偏移，在汇编代码中启动MMU机制……学到了太多太多学校不会教的知识，也在无形中让自己的代码能力得到了一点点锻炼！

感谢

我想特别感谢帮助过我的郭学长，champion学长以及李学长！他们本没有义务去带我这个小白，但他们仍不遗余力的教会我，离他们的目标我还差的太多太远！希望通过后续的锻炼我能更加对得起他们的培养，在明年的今天我也能成为学姐带领学弟学妹们探索操作系统的世界，将学长们托付的传承下去，回

馈学长们！

感谢比赛方给我们提供的机会，能深入了解一个操作系统的工作机理！能看到自己所学理论的用武之地！如果有幸进入决赛，我们会充满斗志的去面对新的巨大的挑战，实现我们能做到的最好的操作系统！

郭伟鑫

心得与感谢

与操作系统开发已经打了两年的交道了，从去年负责实现内存，文件，中断和驱动。今年的我也开始负责进程，信号量和syscalls了，也算走过了开发操作系统的全部流程了。真是纸上得来终觉浅，绝知此事要躬行。书本上看似简单的概念在实现的时候完全变成了另一个样子。即使做了一年的准备，但是又一次开始编写操作系统还是觉得自己做的太少了，开始的时间太晚了。

每次看到进程的调度都会感叹操作系统的精妙，多个程序的复杂并发在操作系统的指挥下就像有交通信号灯的路口井然有序。特别是将所有进程的管理跟随自己的想法进行调度，程序的每一步都在预料中的感觉真是十分奇妙。

系统开发给我的体验是分成两个部分的，第一种就是实现现有标准的功能，比如elf解析，fat32文件系统开发。这种开发往往是先查好几个小时的标准，然后理解标准后很流畅的就能将代码写出来。在开发这部分代码的时候最深刻的体会是前人制定标准的精妙，往往是读完协议不解其意。但是在开发的时候发现协议中不起眼的小设计往往在后续的开发中都发挥了极大的作用，经常写完代码拍案叫绝。

第二个部分则是自己定义自己的内核，比如进程调度，信号量的实现。这部分的开发对我这种没什么经验的开发者来说压力是巨大的。往往还不知道全貌就得开始编写代码，之后出现问题又得推到重做。参考了许多前辈优秀的代码，但是抽象程度过高的代码光是理解都花很长时间。往往基础的功能都没理解就让各种设计绕了进去。好不容易找到可以理解的代码，自己砍去那些看似不重要的设计，后续开发又发现bug，然后狼狈的将设计加回去。不过在开发了这么多模块之后，我现在认为这也是一种很好的学习方式，毕竟人的注意力是有限的，太优秀的代码对于初学者来说往往很难抓住重点。（尤其是一些设计的核心代码往往就十几行，还藏在上千页的代码当中）。不过在上手之后脑子中就会自动想出改良的设计，然后沉迷在设计自己代码中，是一种先苦后甜的开发模式

开发操作系统中最美妙的体验还是在于彻夜通宵将bug解决，看到程序执行的结果一行一行的输出在屏幕上的感觉。不过有时候的bug是真的没有头绪，不同于其他逐行执行的程序，操作系统的时间片流转有时导致的bug总是稀奇古怪。抓耳挠腮好几个小时却连程序在什么地方出错了都不清楚。这里必须感谢钱品亦学长的无私帮助，有时候遇到再困难的问题，一听到学长的声音就好像吃了一颗定心丸。也感谢他常常常用好几个小时帮助我们去调试代码。

ps:最后再夸夸钱学长，学长不只是代码能力强，更厉害是无私的热心助人。经常帮我们调试代码到凌晨，真是一个帅气的男人。