

OSKernel2024-X1

操作系统设计与分析文档

RuaruaOS

目录

设计思路与实现重点-----	1
bios 启动以及相关寄存器初始化 -----	1
时钟中断 -----	2
Timerverc 中的操作 -----	3
进程管理 -----	3
上下文切换 -----	4
Exec -----	4
文件系统 -----	4
buffer(bcach e) of disk in memory -----	4
fat32 文件格式 -----	4
长文件名目录项 -----	4
短文件名目录项 -----	5

在 fat32 中寻找文件	5
创建文件	5
input 回收目录项指针	5
read / write	5
外设	6
串口	6
磁盘	8
时钟	12
trap	12
陷入机制	12
User trap	12
系统调用	14
Kernel trap	14
系统调用	15
地址管理	16
创建地址空间	16
物理内存分配	17
执行态进程内存布局	18
遇到的问题和解决方法	19
内存管理	19
并发和同步	19
设备驱动问题	19

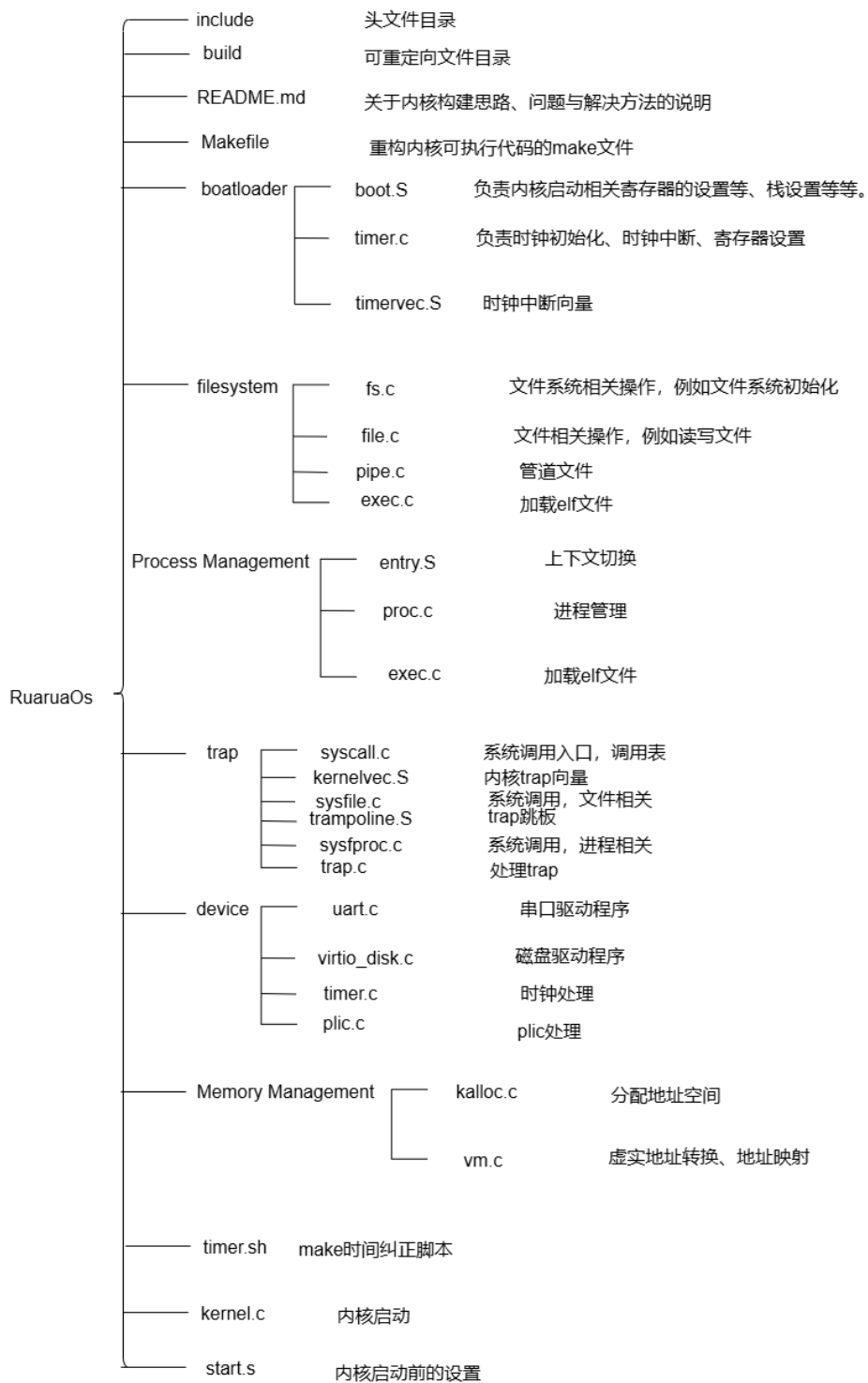


图 1 文件结构

RuaruaOS

一、设计思路与实现重点

```
la sp, stack0
li a0, 1024*4
csrr a1, mhartid
bnez a1, _sti
addi a1, a1, 1
mul a0, a0, a1
add sp, sp, a0
```

● bios启动以及相关寄存器初始化

1. 首先，从mhartid寄存器中获得当前hart的id，让id为0的hart工作，其余hart睡眠。接下来分配栈空间，否则程序不会运行。

```
csrr t6, mstatus
li t5, -6145 # 0xFFFFE7FF
and t6, t5, t6
li t5, 1024 * 2
or t6, t5, t6
csrw mstatus, t6 # set MPP = 0x00, Set MPP to 0x01 and all other bits to 1
csrr t3, mstatus
```

2. 接下来将mstatus中的MPP为置为0x01，其余位设置为1，mret之后权限模式为S态。

```
li t5, 0x80200000
csrw mepc, t5
li t5, 0
csrw satp, t5
li t5, 0xffff
csrw medeleg, t5
csrw mideleg, t5
```

3. 接下来将mepc设置为0x80200000，这是我们内核开始运行的地址，mret之后会跳转到这里，并且权限模式为S。初始化satp寄存器，并将中断（medeleg）委托给S态（mideleg）。

```
csrr t5, sie
ori t5, t5, 546
csrw sie, t5
```

4. 接下来设置sie寄存器，使能处理S态软件中断、时钟中断和M态软件中断。

```
la t5, immediate_value # 0x3fffffffffffffff
csrw pmpaddr0, t5
li t5, 0xf
csrw pmpcfg0, t5
```

5. 接下来配置pmp配置寄存器，采用TOR地址匹配模式，由于是pmpaddr0，则匹配的地址为全部的物理地址(pmpaddr0地址全为1)，它们的效果是把物理地址 $\$[0,2^{56}-1]\$$ 的范围设为所有模式(M/S/U) 都可读、可写、可执行的。

```
call timerinit
csrr t0, mhartid
mv tp, t0
mret
```

● 时钟中断

接下来就是初始化时钟中断。

0x4000	MTIMECMPLO	读/写	0xFFFFFFFF	机器模式计时器：比较值寄存器 (低 32 位)。
0x4004	MTIMECMPHI	读/写	0xFFFFFFFF	机器模式计时器：比较值寄存器 (高 32 位)。

0xBFF8	MTIMELO	读	0x00000000	机器模式计时器：当前值寄存器 (低 32 位)，该寄存器值为 pad_cpu_sys_cnt[31:0] 信号的值。
0xBFFC	MTIMEHI	读	0x00000000	机器模式计时器：当前值寄存器 (高 32 位)，该寄存器值为 pad_cpu_sys_cnt[63:32] 信号的值。

时钟的寄存器并没有专门的cpu寄存器，而是作为外设，地址如上图所示。

```
*(uint64_t*)MTIMECMP(id) = *(uint64_t*)MTIME + interval; // interval := 100000
```

设置tick，如果mtime中的值大于mtimecmp + interval，则触发中断。

```
uint64_t *scratch = &timer_scratch[id][0];
scratch[3] = MTIMECMP(id);
scratch[4] = interval;
w_mscratch((uint64_t)scratch);
```

scratch前24字节用来保存寄存器a1,a2,a3，第24~32存储了mtimecmp的地址。32~40字节存储了间隔，即上文中的interval。将scratch写入mscratch。

```
w_mtvec((uint64_t) timervec);
w_mstatus(r_mstatus() | MSTATUS_MIE);
w_mie(r_mie() | MIE_MTIE);
```

将M模式下的中断向量表机制设置为timervec的地址，由于timervec是四字节对齐，所以mtvec低2位为0，这意味着无论M-mode发生什么中断都会进入timervec中。

timervec中的操作

1. 保护寄存器
2. 将mtimecmp重新设置为mtimecmp + interval
3. 将sip设置为2，发起S-mode软件中断，处理时钟中断，即tick

● 进程管理

```
struct trapframe;----- >trapfram(最后被映射到user空间)
proc结构体中的trapfram位于内核空间
struct context;----- >进程上下文
struct proc {
    enum procstate state;
```

```

void *chan;
int killed;
int xstate;
int pid;
struct proc *parent;
uint64_t kstack;
uint64_t sz;
//uint64_t old_sz;           以后可能会需要
pagetable_t pagetable;      //user页表
pagetable_t kpagetable;     //kernel页表
struct trapframe *trapframe; //trapframe, 存放寄存器
struct context context;
struct file *ofile[NOFILE];
struct dirent *cwd;
char name[16];
};

```

第一个进程通过proc中的initcode存放的机器码来进行，initcode扫描磁盘，获取可执行文件，通过exec来执行样例。

上下文切换

通过entry.S进行上下文切换，即swtch(struct context *a0, struct context*a1)，将当前寄存器存放到a0处，然后从a1处获得新寄存器。

重头戏：exec

找到对应的文件对应磁盘的位置，读取对应的 elf 文件。将目标文件的代码段和数据段读入页表中开辟栈空间，存放参数。并且将 a0 设置为 argc，即参数的个数，a1 设置为参数地址。以上都需要为新进程开辟页并且映射到用户内存空间。对应的系统调用入口需要将每一个参数（argv）都需要开辟新的页来进行存放。

● 文件系统

buffer(bcach) of disk in memory

buf结构体是一个双向链表，并且buf的data成员为512字节，和disk一个扇区一样大。作用是将disk的内容缓存到buffer中。

fat32文件格式

在fat32的“bpb”区域，我们可以找到root目录的簇数，每个扇区字节数，每个簇的扇区数，保留扇区数。

长文件名目录项

```

typedef struct long_name_entry {
    uint8_t      order;
    uint16_t     name1[5];
    uint8_t      attr;
    uint8_t      _type;
    uint8_t      checksum;
    uint16_t     name2[6];
};

```

```

    uint16_t    _fst_clus_lo;
    uint16_t    name3[2];
} long_name_entry_t;

```

一个长文件名目录项有可以记录长度为13的文件名，如果记录不下就用多个长文件目录项来记录。其中order记录了是第几个长文件目录项还有是否是最后一个目录项。name记录了文件名。

短文件名目录项

```

typedef struct short_name_entry {
    char        name[11];
    uint8_t     attr;
    uint8_t     _nt_res;
    uint8_t     _crt_time_tenth;
    uint16_t    _crt_time;
    uint16_t    _crt_date;
    uint16_t    _lst_acce_date;
    uint16_t    fst_clus_hi;
    uint16_t    _lst_wrt_time;
    uint16_t    _lst_wrt_date;
    uint16_t    fst_clus_lo;
    uint32_t    file_size;
} short_name_entry_t;

```

短文件目录项虽然也记录名字，但是我们一般都用长文件目录项来记录名字，用短文件目录项来找到文件的创建时间等、在磁盘上的位置等信息。

在fat32中寻找文件

1. 首先要找到属于fat指定table的扇区，接着读取整个扇区
2. 在接着使用cluster数找到在此扇区中的偏移，就可以读到下一个文件的簇数。
3. 根据fat中的具体数据，判断是否为最后一个文件。

创建文件

1. 如果文件是目录，则需要创建"."和".."
2. 如果是普通文件，则直接在父目录相对偏移创建文件即可。

iput回收目录项指针

根据LRU原则，将对应指针放到root后。将ip的引用数减1，如果ip的引用数为0，则将父目录的引用数也减1.并且将ip的内容同步到disk中。

read / write

读或者写文件是需要根据文件类型来进行的。我们有三种文件类型，如下所示。

文件类型

一共有三种文件类型，分别为设备，管道，和普通文件。

管道文件

管道文件是半双工的，有读端和写端。读端和写端分别用一个file。管道结构体占一个内核页。

写管道

写管道需要将字符从用户空间copy到内核空间，然后放到管道的data区。如果data区满，则将写管道进程睡眠，读管道进程唤醒。

读管道

读时从data区copy到用户空间指定地址，读完指定字节数之后唤醒写管道进程。

普通文件

普通文件的类型为FD_INODE（2），其ip成员记录了其在disk的一些属性，例如首簇号等等。

打开文件

读、写文件之前首先需要获得目标的目录项指针，获得目录项指针需要获得从父目录下寻找对应文件名，将文件具体信息从短文件名目录项中获得。只有获得了其在磁盘中的目标位置才可以进行读写。

读文件

向文件所在位置中读取一个n个字节的内容。

写文件

向文件所在位置中写入n个字节大小的内容。如果写入的字节超过一个簇，则需要额外申请磁盘空间。

外设

● 串口

设备号

串口的设备号是1.

串口初始化

1. 关闭中断关闭**UART**（通用异步收发传输器）上的所有中断

```
WriteReg(IER, 0x00); // 禁用所有中断
```

IER寄存器控制着芯片上所有的中断的使能

这一步相当于关闭了所有UART可能发出的中断

设置波特率特殊模式 进入UART设置波特率的特殊模式。

2. 设置波特率设置**UART**的波特率为**38.4K**

```
WriteReg(LCR, LCR_BAUD_LATCH); // 设置波特率锁存位
```

当向LCR(Line Control Register)最高位(bit7)写入1时, 进入波特率设置模式

```
WriteReg(0, 0x03); // 设置波特率低位
```

```
WriteReg(1, 0x00); // 设置波特率高位
```

根据查表可知DLL、DLM两个寄存器需要设置的值

离开波特率设置模式 离开波特率设置模式, 并设置数据字长为8位, 无校验位。

3. 设置数据字长为8

```
WriteReg(LCR, LCR_EIGHT_BITS);
```

LCR的低两位控制数据字长

重置并使能FIFOs 重置UART的FIFO（先进先出）并启用它们。

4. 初始化FIFO并使能RX和TX中断

```
WriteReg(FCR, FCR_FIFO_ENABLE | FCR_FIFO_CLEAR); // 使能FIFO并清除其内容
```

FCR_FIFO_ENABLE标志用于使能输入输出两个FIFO

FCR_FIFO_CLEAR标志用于清空两个FIFO并将其计数逻辑设置为0

使能输入输出中断 使能UART的输入（RX）和输出（TX）中断。

```
WriteReg(IER, IER_TX_ENABLE | IER_RX_ENABLE); // 使能TX和RX中断
```

一旦同时使能了输入(RX)中断和FIFO, UART就会在到达trigger level时向CPU 发起一个中断

在输出THR为空时也会向CPU发起一个中断

初始化串口芯片输出缓冲区的锁 初始化os内核中UART输出缓冲区的锁。

5. 初始化UART输出缓冲区的锁

```
initlock(&uart_tx_lock, "uart");
```

内核给输出又设置了一个缓冲区, 默认大小为32

但实际上UART硬件内部已有FIFO作为缓冲, 将硬件缓冲透明化

写控制台与串口

1. 写系统调用与UART

当通过写系统调用向控制台写入数据时, 最终这些数据会通过UART（通用异步收发传输器）进行传输。UART是一种串行通信协议, 常用于微控制器和计算机之间的通信。

2. 输出缓冲区

设备会维护一个输出缓冲区uart_tx_buf, 用于存储待发送的数据。

由于使用了缓冲区, 写进程不需要等待UART实际完成发送就可以继续执行其他任务, 从而提高了系统的效率。

3. uart_putc函数

uart_putc函数负责将字符加入uart_tx_buf缓冲区。

在添加字符后, 它会调用uart_start函数来启动或继续数据的传输。

uart_putc函数会立即返回, 除非缓冲区已满, 此时它可能会选择等待或采取其他策略（如丢弃数据或阻塞调用者）。

4. uart_start函数

uart_start函数负责检查UART的传输状态, 并启动或继续数据的发送。

如果缓冲区中有数据且UART准备好发送，`uart_start`会开始传输缓冲区中的第一个字符。在传输过程中，`uart_start`不会阻塞等待字符发送完成，除非缓冲区为空或设备处于其他不能发送的状态。

5. **UART中断处理**

每当UART发送一个字节后，都会产生一个中断信号。

中断处理程序（如`uart_intr`函数）会被调用。

`uart_intr`函数会再次调用`uart_start`函数来检查是否有更多的数据需要发送。

如果缓冲区中还有数据且UART可以继续发送，`uart_start`将启动下一个字符的传输。

6. **多个字符的写入**

当进程写入多个字符时，第一个字节的传输会由`uart_putc`调用`uart_start`启动。

后续的字节将由UART发送完成后的中断处理程序（`uart_intr`）调用的`uart_start`进行传输。

读控制台与串口

1. 当用户输入一个字符，UART设备就会产生一个中断，激活os的陷阱处理程序。

程序将会调用`devintr`，读取`scause`判断是否为外部设备产生的中断。之后通过PLIC（平台级中断控制器）判断中断设备，如果是UART设备，就会调用`uartintr`函数。

2. `uartinit`从UART设备中读取所有输入字符，并将其交给`consoleintr`处理，此函数不会等待字符的输入，因为未来的输入会产生新的中断。`consoleintr`将输入保持在`buffer`中直到一整行到达，同时对一些特殊符号进行处理。当一整行到达后，就会唤醒一个正在等待的`consoleread`。

3. 当`consoleread`被唤醒时，`buffer`中就保存了完整的一行输入，此时就可以将其拷贝到用户空间并返回。

● 磁盘

数据结构

```
static struct disk {
    char pages[2*PGSIZE];
    struct virtq_desc *desc;
    struct virtq_avail *avail;
    struct virtq_used *used;

    char free[NUM];
    uint16_t used_idx;

    uint16_t nwait;
    struct {
        struct buf *b;
        char status;
        int idx1;    // first index of a chain
    } info[NUM];

    struct wait_queue queue;
    int queue_running;
    struct virtio_blk_req ops[NUM];
};
```

```
struct spinlock vdisk_lock;
} __attribute__((aligned (PGSIZE))) disk;
```

1. `struct disk`是一个代表磁盘的结构，它包含了许多与磁盘交互所需的信息和状态。
2. `pages[2*PGSIZE]`: 一个预分配的字符数组，用于存储 `virtio` 描述符等。这通常是多个连续页面的内存，用于满足某些硬件或驱动程序的特定需求。
3. `desc, avail, used`: 指向与 `virtio` 队列相关的数据结构的指针。这些结构是 `virtio` 规范的一部分，用于描述和管理磁盘的 I/O 请求。
4. `free[NUM]`: 一个数组，用于跟踪哪些描述符是空闲的。
5. `used_idx`: 一个索引，表示在 `used` 数组中查看到的位置。
6. `nwait, info[NUM]`: 用于跟踪飞行中（即已发送但尚未完成）的操作的信息。`info` 数组中的每个条目都包含与该操作相关的缓冲区、状态和索引。
7. `queue, queue_running`: 与等待队列相关的字段。`queue_running` 可能是一个标志，表示队列是否正在处理请求。
8. `ops[NUM]`: 磁盘命令头的数组。这些命令头与描述符一一对应，方便管理。
9. `vdisk_lock`: 一个自旋锁，用于保护磁盘相关的某些数据或操作。

磁盘初始化

10. 向status寄存器写入ACKNOWLEDGE(=1第0位)，表示OS已经发现有效虚拟设备
11. 向status寄存器写入DRIVER(=2第1位)，表示OS已经知道如何驱动设备
12. 向status寄存器写入FEATURES_OK(=8) 表示设备特征/具体工作方式协商完毕
13. 向status寄存器写入DRIVER_OK(=4) 表示驱动已经准备好
14. 其中，DeviceFeatures只读，DeviceFeaturesSel和DriverFeaturesSel用于分别给设备和驱动指定一套feature，os中并未使用。os中并未对原始设备feature修改，只是对驱动feature在设备feature基础上做部分禁用处理并写入DriverFeatures。
15. VIRTIO_BLK_F_RO(5-->第5位) 表示块设备只读
16. VIRTIO_BLK_F_SCSI(7) 表示虚拟块设备支持SCSI (Small Computer System Interface) 包命令
17. VIRTIO_BLK_F_CONFIG_WCE(11) 表示允许虚拟块设备在写回和写穿模式之间切换其缓存
18. VIRTIO_BLK_F_MQ(12) 表示设备支持多个虚拟队列，os中只使用了一个，默认的队列0
19. VIRTIO_F_ANY_LAYOUT(27)
出于对旧版的兼容，表示设备和驱动之间对消息帧不做任何协商
20. VIRTIO_RING_F_INDIRECT_DESC(28)
用于扩展descriptor数组到更大容量，由原本的一个队列struct virtq_desc *desc;
变为多个队列struct virtq_desc desc[len / 16]; 其中的descriptor将在后面详细解释

21. VIRTIO_RING_F_EVENT_IDX(29)

用于指示设备和驱动程序之间是否支持使用事件索引机制来提高通知性能。如果没有协商这个特性，驱动程序可以通过`available` ring中的`flags`字段来通知设备，不需要在使用缓冲区时发送通知。当然，性能相应就不高。

22. 初始化要用的队列号

```
*R(VIRTIO_MMIO_QUEUE_SEL) = 0;
```

23. 设置队列大小

```
*R(VIRTIO_MMIO_QUEUE_NUM) = NUM;
```

24. 为描述符、空闲、使用

25. 虚拟队列位置

```
*R(0x040) = ((uint64_t)disk.pages) >> PGSHIFT;
```

写入此寄存器通知设备虚拟队列在来宾物理地址空间中的位置。该值是从队列描述符表开始的页的索引号。

读写磁盘

1. 首先要申请3个空闲描述符，如果不够就等待

```
int idx[3];
while(1){
    if(alloc3_desc(idx) == 0) {
        break;
    }
    sleep(&disk.free[0], &disk.vdisk_lock);
}
```

2. 初始化请求头，使用`disk`的`ops`域存储，`ops`存储位置和第一个描述符的位置一致

```
struct virtio_blk_req *buf0 = &disk.ops[idx[0]];
struct virtq_desc *desc;
if(write)
    buf0->type = VIRTIO_BLK_T_OUT; // write the disk
else
    buf0->type = VIRTIO_BLK_T_IN; // read the disk
buf0->reserved = 0;
buf0->sector = b->sectorno;//sector;
```

3. 初始化描述符

驱动负责找到可用3个descriptor并向descriptor

table第一个位置写入请求头，第二个位置写入目标数据，第三个位置负责让磁盘向请求头的status写1，表示操作结束。

```
disk.desc[idx[0]].addr = (uint64_t) buf0;
disk.desc[idx[0]].len = sizeof(struct virtio_blk_req);
disk.desc[idx[0]].flags = VRING_DESC_F_NEXT;
disk.desc[idx[0]].next = idx[1];
disk.desc[idx[1]].addr = (uint64_t) b->data;
disk.desc[idx[1]].len = BSIZE;
```

```

if(write)
    disk.desc[idx[1]].flags = 0; // device reads b->data
else
    disk.desc[idx[1]].flags = VRING_DESC_F_WRITE; // device writes b->data
disk.desc[idx[1]].flags |= VRING_DESC_F_NEXT;
disk.desc[idx[1]].next = idx[2];
disk.info[idx[0]].status = 0xff;
disk.desc[idx[2]].addr = (uint64_t) &disk.info[idx[0]].status;
disk.desc[idx[2]].len = 1;
disk.desc[idx[2]].flags = VRING_DESC_F_WRITE; // device writes the status
disk.desc[idx[2]].next = 0;

```

4. 记录**struct buf**信息，方便中断处理程序处理

```

b->disk = 1;
disk.info[idx[0]].b = b;

```

5. 把有效**index**写入**avail ring**中

```

disk.avail->ring[disk.avail->idx % NUM] = idx[0];

```

6. 给设备发送通知消息

```

*R(VIRTIO_MMIO_QUEUE_NOTIFY) = 0;

```

7. 等待磁盘中断处理

```

while(b->disk == 1) {
    sleep(b, &disk.vdisk_lock);
}

```

8. 中断处理后清楚信息和**descriptor**链

```

disk.info[idx[0]].b = 0;
free_chain(idx[0]);

```

磁盘中断处理

1. 处理**used ring**，并且清空**disk**标志，表示处理完成

```

while((disk.used_idx % NUM) != (disk.used->id % NUM)){
    int id = disk.used->elems[disk.used_idx].id;
    if(disk.info[id].status != 0)
        panic("virtio_disk_intr status");
    disk.info[id].b->disk = 0; // disk is done with buf
    wakeup(disk.info[id].b);
    disk.used_idx = (disk.used_idx + 1) % NUM;
}

```

2. 处理中断位

```

*R(VIRTIO_MMIO_INTERRUPT_ACK) = *R(VIRTIO_MMIO_INTERRUPT_STATUS) & 0x3;

```

● 时钟

初始化

[初始化在前面提到过：timerinit。](#)

timervec操作

[timervec操作在前面提到过：timervec。](#)

● trap

1. 陷入机制

Stvec (Supervisor Trap Vector): 这是一个由内核写入的寄存器，用于存储陷阱处理程序的地址。当RISC-V处理器遇到需要处理的陷阱时，它会跳转到这个地址来执行相应的处理程序。

sepc (Supervisor Exception Program Counter): 当陷阱发生时，RISC-V会将当前的程序计数器（pc）值保存在这个寄存器中，因为陷阱处理程序执行时，原始的pc值会被stvec覆盖。sret（从陷阱返回）指令会在陷阱处理程序完成后，将sepc的值复制回pc，从而使处理器继续执行陷阱发生之前的指令。内核也可以主动写入sepc来控制sret指令的执行目标。

scause (Supervisor Cause): RISC-V在这个寄存器中放置一个数字，该数字描述了导致陷阱发生的原因。这个值对于陷阱处理程序至关重要，因为它帮助处理程序确定应该执行哪种特定的处理逻辑。

sscratch (Supervisor Scratch): 内核可以在这个寄存器中放置一个值，这个值在陷阱处理程序开始时会被使用。它为陷阱处理程序提供了一个临时的存储空间，用于传递信息或数据，而无需依赖其他内存位置或寄存器。

sstatus (Supervisor Status): 这个寄存器包含了多个状态位，其中之一是SIE（Supervisor Interrupt Enable）位。当SIE位被设置时，设备中断会被启用；如果内核清除了SIE位，RISC-V将推迟处理设备中断，直到内核再次设置SIE位。另外，sstatus中的SPP（Supervisor Previous Privilege）位指示了触发陷阱的原始模式（用户模式或管理模式），并控制sret指令返回后的执行模式。

2. user trap

- 如果用户程序发出系统调用（ecall），或者设备中断，那么在用户空间中执行时就可能会trap。来自用户空间的陷阱的高级路径是uservec，然后是usertrap；返回时，先是usertrapret，然后是userret。
- 对于user trap的向量设置，我们在进程初始化时将ra设置为forkret，而forkret会调用usertrapret，usertrapret中设置了中断向量，也就是uservec函数。

```
proc init:
p->context.ra = (uint64_t)forkret;
p->context.sp = p->kstack + PGSIZE;

void forkret(void) {
    static int first = 1;
    release(&myproc()->lock);
```

```

    if (first) {
        first = 0;
        fat32_init();
    }
    usertrapret();
}

user trap:
uint64_t trampoline_uservec = TRAMPOLINE + (uservec - trampoline);
w_stvec(trampoline_uservec);

```

- 由于RISC-V硬件在发生陷阱时不会自动切换页表，因此用户页表必须包含对uservec（即stvec寄存器所指向的陷阱向量指令）的映射。因此，设计了trampoline，将其页面映射到内核页表和每个用户页表的相同虚拟地址上。

```

kernel map trampoline:
kvmmmap(kpgtbl, TRAMPOLINE, (uint64_t)trampoline, PGSIZE, PTE_R | PTE_X);

user map trampoline:
if(mappages(pagetable, TRAMPOLINE, PGSIZE, (uint64_t)trampoline, PTE_R |
PTE_X) < 0){
    uvmfree(pagetable, 0);
    return 0;
}
if(mappages(pagetable, TRAPFRAME, PGSIZE, (uint64_t)(p->trapframe), PTE_R |
PTE_W) < 0){
    uvmunmap(pagetable, TRAMPOLINE, 1, 0);
    uvmfree(pagetable, 0);
    return 0;
}

```

- 如上所述，user页表映射了TRAPFRAME，因此，我们可以将一些重要的数值存放在这里，比如：内核页表根地址，内核栈，trap地址，发生异常的pc以及其他寄存器等等；因此，我们首先要将用户的寄存器存放到TRAPFRAME中，并且借助sscratch寄存器，我们也可以将a0存放进来(一开始a0作为TRAPFRAME地址)。
- uservec从TRAPFRAME获得当前进程内核栈的指针、当前CPU的hartid、usertrap的地址和内核页表的地址，将satp切换到内核页表，并调用usertrap。
- usertrap

函数的主要职责是识别并处理操作系统中发生的陷阱（trap），这些陷阱可能是由系统调用、设备中断或异常引起的。在处理过程中，usertrap 首先更改了陷阱向量stvec，以确保内核中的陷阱将由 kernelvec 处理。接下来，它会保存当前的用户程序计数器 sepc，这是为了防止在 usertrap 内部可能发生的进程切换覆盖这一重要信息。对于不同类型的陷阱，usertrap 采取不同的处理方式：

 - 如果是系统调用，它会调用 syscall函数来处理，并在处理完成后，将用户程序计数器 pc 加 4（因为 RISC-V在系统调用时会把程序指针留在 ecall 指令上，需要跳过该指令以继续执行后续代码）。

- 如果是设备中断，它会调用 `devintr` 函数来处理。
- 如果既不是系统调用也不是设备中断，那么它将被视为一个异常，此时内核会选择终止出错的进程。
- 返回到用户空间的第一步是调用 `usertrapret` 函数。该函数负责设置 RISC-V 的控制寄存器，以便为来自用户空间的未来陷阱做好准备。以下是 `usertrapret` 函数的主要步骤：
 - 设置陷阱向量：
`usertrapret` 首先将 RISC-V 的陷阱向量寄存器 `stvec` 设置为指向 `uservec`。`uservec` 是一个特殊的处理程序，用于处理在用户模式下发生的陷阱。
 - 准备陷阱帧字段：
 在将控制权返回给用户空间之前，`usertrapret` 需要确保 `uservec` 所依赖的陷阱帧字段已经正确设置。这些字段通常包括保存的用户程序计数器（`sepc`）、保存的陷阱帧指针（`trapframe`）等，它们对于在用户模式下正确处理陷阱至关重要。
 - 恢复用户程序计数器：
`usertrapret` 将之前保存的用户程序计数器（`sepc`）的值恢复回 RISC-V 的程序计数器寄存器。这是因为在系统调用或陷阱处理过程中，程序计数器可能会被修改。通过将 `sepc` 的值写回程序计数器，可以确保在用户空间恢复执行时从正确的指令开始。
 - 调用 `userret` 函数：
 最后，`usertrapret` 调用 `userret` 函数，该函数是 `trampoline page` 上的汇编代码段。蹦床页面是同时在用户和内核页表中映射的特殊页面，用于在用户空间和内核空间之间安全地切换。`userret` 函数的主要任务是在用户模式下恢复页表上下文，并执行必要的状态切换，以确保用户空间程序能够安全地继续执行。
- `usertrapret` 对 `userret` 的调用将指针传递到 `a0` 中的进程用户页表和 `a1` 中的 `TRAPFRAME`
- `userret` 将 `satp` 切换到进程的用户页表。
- `userret` 复制陷阱帧保存的用户 `a0` 到 `sscratch`，为以后与 `TRAPFRAME` 的交换做准备。从此刻开始，`userret` 可以使用的唯一数据是寄存器内容和 `TRAPFRAME` 的内容。下一个 `userret` 从陷阱帧中恢复保存的用户寄存器，做 `a0` 与 `sscratch` 的最后一次交换来恢复用户 `a0` 并为下一个陷阱保存 `TRAPFRAME`，并使用 `sret` 返回用户空间。

3. [systemcall](#) 系统调用

4. kernel trap

- 当在CPU上执行内核时，内核将stvec指向kernelvec的汇编代码。由于已经在内核中，kernelvec可以依赖于设置为内核页表的satp，以及指向有效内核栈的栈指针。kernelvec保存所有寄存器，以便被中断的代码最终可以不受干扰地恢复。

- 保存寄存器

kernelvec

负责将寄存器状态保存在当前被中断的内核线程的栈上，这是一个重要的步骤，因为寄存器中存储的数据与该线程的执行上下文紧密相关。当陷阱（如中断或异常）导致内核切换到另一个线程执行时，这一点尤为重要。在这种情况下，被中断线程的寄存器状态会被安全地保留在其栈上，而新线程则会使用其自己的栈空间来保存和执行其上下文。这样的设计确保了每个线程都能够独立地维护其状态，从而支持多线程的并发执行。

- kerneltrap

在保存了寄存器状态之后，kernelvec 将控制权转移至 kerneltrap。kerneltrap 函数为两种类型的陷阱（trap）做好了准备：设备中断和异常。为了处理前者，它调用了 devintr 函数（同样位于 kernel/trap.c 文件的第 177 行）。如果陷阱不是由设备中断触发的，那么它必然是一个异常。在操作系统的内核上下文中，异常通常被视为严重的错误，因为它们可能指示了内核自身的问题或者是对内核的不当访问。在这种情况下，内核不会尝试恢复或继续执行，而是会调用 panic 函数，这是一个用来报告致命错误的机制，它将停止整个系统的执行。

- 返回

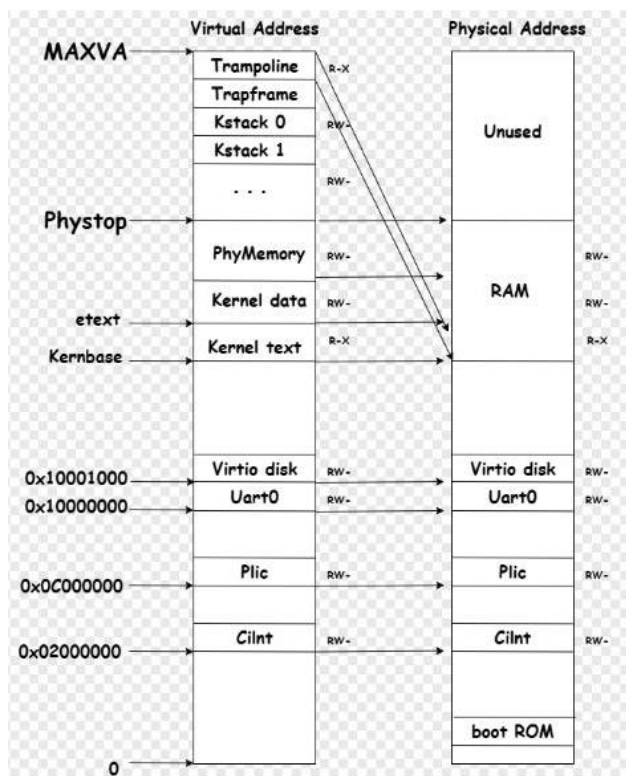
当kerneltrap的工作完成后，它需要返回到任何被陷阱中断的代码。

● 系统调用

1. 在用户空间中，当需要执行一个系统调用时，程序会将系统调用所需的参数放入寄存器 a0 和 a1 中，并将系统调用号放入寄存器 a7。在 RISC-V 架构中，ecall 指令被用来触发一个系统调用，这会导致程序的控制流从用户空间切换到内核空间。
2. 在内核中，系统调用的处理始于一个预先定义好的向量表 uservec，它指向一个处理用户trap的函数 usertrap。usertrap 函数会进一步调用 syscall 函数来处理具体的系统调用。
3. syscall 函数会从trapframe中提取出保存在 a7 中的系统调用号（通过 p->trapframe->a7）。然后，它使用这个系统调用号作为索引来查找 syscalls 数组，这是一个函数指针数组，每个条目对应一个系统调用处理函数。
4. 系统调用通常通过返回负数来表示错误，而返回零或正数则表示成功。如果 syscall 函数发现提供的系统调用号无效，它会打印一个错误消息并返回 -1。

● 地址管理

– 内核地址布局



1. 创建地址空间

○ 核心数据结构和函数

- **pagetable_t** 数据结构: 表示指向RISC-V根页表页的指针, 是用于管理地址映射的核心数据结构。
- **walk** 函数: 模拟RISC-V分页硬件, 用于为给定的虚拟地址找到对应的页表条目 (PTE)。
- **mappages** 函数: 用于向页表中装载新的地址映射, 即将一个范围的虚拟地址映射到相应的物理地址。

○ 内核页表的初始化

- **kvminit** 函数: 在启动序列的早期调用, 用于初始化内核的页表。
- **kvmmake** 函数: 分配物理内存页来保存根页表页, 并设置内核所需的地址转换。
- **kvmmap** 函数: 在kvmmake中调用, 用于将特定的物理地址范围映射到内核的虚拟地址空间。

○ 用户页表和进程管理

- **uvm**开头的函数: 用于操作和管理用户页表, 如创建、修改和删除用户进程的地址空间。

- `proc_mapstacks`函数：为每个进程分配一个内核堆栈，并通过调用`kvmmap`将其映射到虚拟地址空间。

- 数据复制与地址翻译

- `copyout` 和 `copyin`函数：用于在内核与用户空间之间复制数据。由于需要显式地翻译用户提供的虚拟地址，这些函数被实现在`vm.c`中。

- TLB管理和刷新

- `sfence.vma` 指令：RISC-V的指令，用于刷新当前CPU的转译后备缓冲（TLB），以确保使用最新的页表映射。
- TLB 刷新策略：当更改页表时，它必须确保TLB中的缓存条目被无效化，以防止使用旧的映射。在加载新的页表到`satp`寄存器后和切换到用户空间之前执行`sfence.vma`指令。

- 页表项的初始化

- 在`mappages`函数中，对于要映射的每个虚拟地址，会调用`walk`函数找到PTE地址。然后，初始化PTE以保存相关的物理页号、所需权限（`PTE_W`、`PTE_X`和`PTE_R`）以及`PTE_V`以标记PTE有效。

2. 物理内存分配

- 初始化分配器

- `kinit`函数调用：在OS的启动过程中，`main`函数调用`kinit`来初始化分配器。`Kinit`负责初始化空闲列表，将物理内存从内核结束到`PHYSTOP`之间的每一页都标记为空闲。
- 假设物理内存大小：OS假设机器有128兆字节的RAM。
- `freerange`函数调用：`kinit`调用`freerange`函数，该函数将指定范围内的物理内存页添加到空闲列表中。`freerange`通过调用`kfree`来逐个释放这些页面。
- 确保页面对齐：由于PTE只能引用在4096字节（即一个页面大小）边界上对齐的物理地址，`freerange`使用`PGROUNDUP`宏来确保只释放对齐的物理地址。
- 分配器管理空间：在初始化阶段，分配器本身没有内存来管理空闲列表。通过调用`kfree`来释放页面，这些页面随后被用作分配器的管理空间。

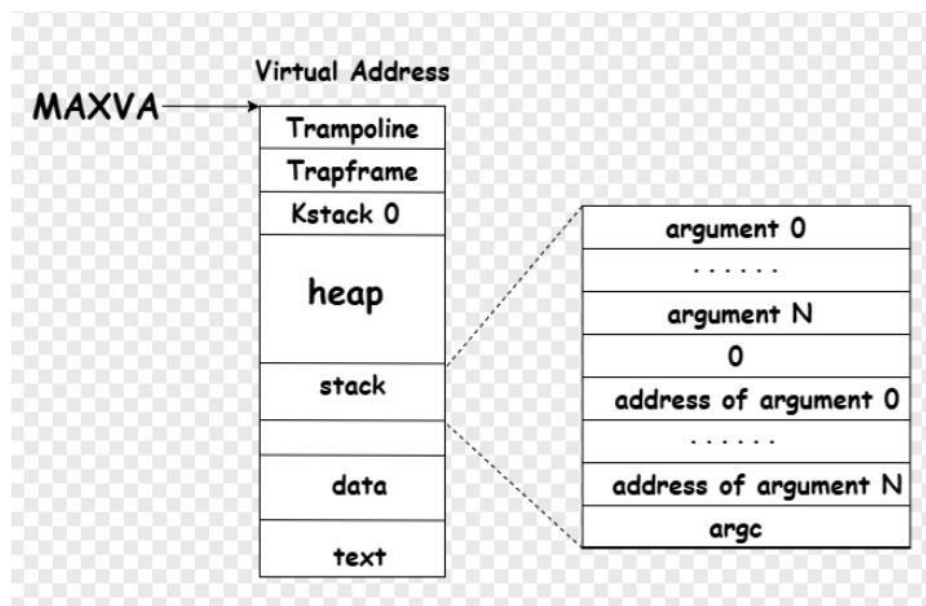
- 内存分配过程

- 空闲列表管理：空闲列表由`struct run`元素组成，每个元素代表一个空闲的物理内存页。这些`struct run`结构被存储在空闲页本身中，因为空闲页没有其他用途。

- 保护空闲列表：空闲列表受到自旋锁的保护，以确保在多处理器或多线程环境中对列表的并发访问是安全的。
 - **kfree**函数：当物理内存页不再需要时，调用**kfree**函数将其释放回空闲列表。**kfree**首先将页面上的每个字节设置为1（或其他非零值），以帮助检测对已释放内存的悬空引用。然后，它将页面添加到空闲列表的前端。
 - **kalloc**函数：当需要分配物理内存页时，调用**kalloc**函数。**kalloc**从空闲列表中删除并返回第一个元素（即第一个空闲页面）。如果列表为空（即没有空闲页面），**kalloc**将返回一个错误指示。
- 地址的双重用途
在分配器代码中，地址有时被视为整数，以便执行算术运算（如遍历所有页面）。在其他情况下，地址被用作读写内存的指针（如操纵存储在每个页面中的**struct run**结构）。这种地址的双重用途是分配器代码中类型转换频繁出现的主要原因。

3. 执行态进程内存布局

- 栈的顶部区域包含了命令行参数的字符串（**strings**）以及一个指向这些字符串的指针数组（**array of pointers**）。这些指针数组通常被称为**argv**（**argument vector**），它允许程序访问传递给它的命令行参数。这种布局使得程序能够轻松地解析命令行参数，从而根据用户的输入执行不同的操作。
- 在栈的稍低位置，是确保程序能够在**main**函数处正确启动所需的关键信息。这些信息包括**main**函数的地址、命令行参数的个数（**argc**）以及指向**argv**的指针。这些值被精心设置，以模拟一个刚刚调用了**main(argc, argv)**的C程序入口点环境。因此，当进程开始执行时，它会自动跳转到**main**函数，并带有正确的参数，使得程序能够按照预期的方式启动和运行。



二、遇到的问题解决方法

- 内存管理问题：

- ✓ 问题：内存是操作系统的核心资源，不合理的内存管理可能导致内存泄漏、碎片化和性能下降。

- ✧ 解决方法：采用先进的内存管理技术，如分页、分段、虚拟内存等，来有效地管理内存资源。同时，通过内存泄漏检测工具来及时发现和修复内存泄漏问题。

- ✓ 在测试过程中，我们经常需要将`kernel`数据`copy`到用户区，一旦操作不当就会造成错误，因此我们需要合理的`copyout`和`copyin`。

- 并发和同步问题：

- ✓ 问题：多用户、多进程、多线程的操作系统环境中，并发和同步问题是难以避免的。不当的并发和同步控制可能导致数据竞争、死锁等问题。

- ✧ 解决方法：使用锁、信号量、条件变量等同步原语来协调不同进程或线程之间的访问。同时，采用先进的并发控制技术，如无锁编程、读写锁等，来提高并发性能。

- 设备驱动程序问题：

- ✓ 问题：设备驱动程序是操作系统与硬件设备之间的接口，不稳定的驱动程序可能导致系统崩溃或硬件故障。

- ✧ 解决方法：确保设备驱动程序的稳定性和可靠性，通过严格的测试来发现和修复潜在的问题。同时，与硬件供应商保持紧密的合作，及时获取最新的驱动程序和更新。