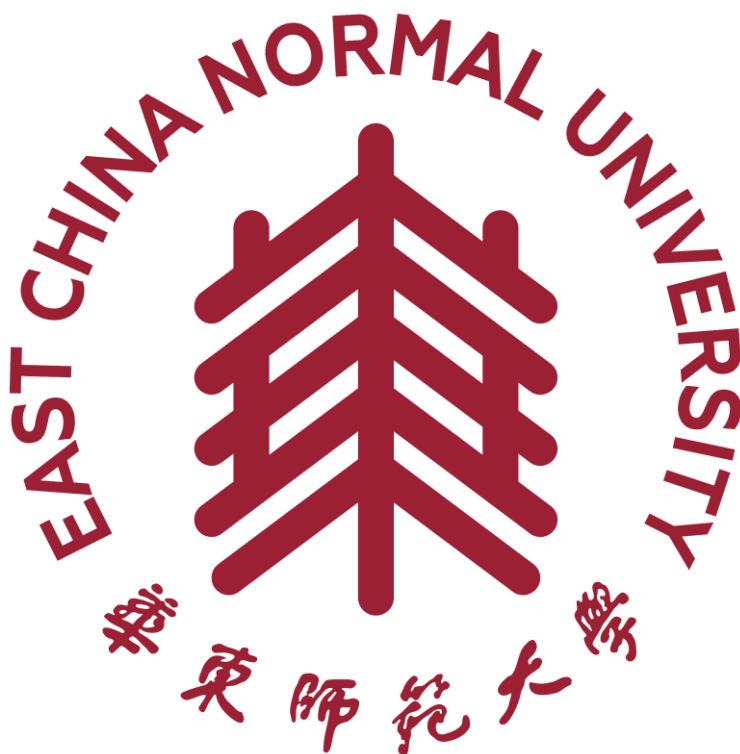


2024



## UNO-OS 内核设计手册

姓 名: 季子墨, 李彤

学 号: 10235101439, 10235101500

学 院: 数据科学与工程学院, 软件工程学院

专 业: 计算机科学拔尖班, 软件工程

指导教师: 石亮

方 向: 小型内核实现

2024 年 12 月

# 目录

摘要 .....	I
1、 UNO-OS 概览 .....	1
1.1 系统结构 .....	1
1.2 创新点 .....	1
1.3 运行方法 .....	3
2、 boot 模块.....	4
2.1 功能介绍 .....	4
2.2 实现分析 .....	4
2.3 测试 .....	5
3、 中断处理模块 .....	6
3.1 功能介绍 .....	6
3.2 中断处理的过程 .....	6
3.3 时钟中断 .....	8
3.4 系统调用 .....	10
3.5 外设中断 .....	15
3.6 测试 .....	17
4、 内存管理模块 .....	18
4.1 功能介绍 .....	18
4.2 物理内存的管理 .....	18
4.3 虚拟内存的管理 .....	27
4.4 测试 .....	28
5、 进程调度模块 .....	34
5.1 功能介绍 .....	34
5.2 PCB 块 .....	34
5.3 初始进程 .....	35
5.4 调度算法 .....	36
5.5 进程间通信 .....	41
5.6 测试 .....	41
6、 文件系统模块 .....	45
6.1 功能介绍 .....	45
6.2 中断层 (PLIC) .....	45
6.3 驱动层 .....	45
6.4 缓冲层 .....	45
6.5 文件系统层 .....	48
6.6 测试 .....	56

<b>7、 系统调用参考 .....</b>	<b>62</b>
7.1 内存分配 .....	62
7.2 进程管理 .....	64
7.3 文件读写 .....	69
<b>参考文献 .....</b>	<b>80</b>

# UNO-OS 内核设计手册

## 摘要：

该文档将对我们的 UNO-OS 作一详细的介绍。UNO-OS 是一款 RISC-V 平台的，宏内核操作系统，该内核部分参考了 Linux 和 xv6 的设计逻辑，同时也加入了一系列新的设计理念。

本系统主要参考的仓库：

[ecnu-oslab](#)

[xv6](#)

我们在这篇文档中，通过讲解大量的源代码，对我们的设计思路进行了详细的阐述。同时，针对各个系统的实现，我们给出了大量的测试用例来证明其正确性。

该系统总共经历了 100 余次修改（以 commit 数计），总的代码量约为 6000 行（去除注释）

截至 12 月 24 日，我们的系统完成情况如下所示：

版本号	完成内容	分支链接
v-0.1alpha	完成 boot 模块并实现必要的库	<a href="#">gitlab</a>
v-0.2alpha	完成虚拟内存模块	<a href="#">gitlab</a>
v-0.3alpha	完成中断处理模块	<a href="#">gitlab</a>
v-0.4alpha	完成进程结构的设计	<a href="#">gitlab</a>
v-0.5alpha	完成进程调度模块	<a href="#">gitlab</a>
v-0.6alpha	完成磁盘底层操作接口	<a href="#">gitlab</a>
v-0.7alpha	完成文件系统模块	<a href="#">gitlab</a>
v-0.8alpha	完成基础的系统调用	<a href="#">gitlab</a>
v1.0(master)	实现了 COW 和高响应比优先调度算法	<a href="#">gitlab</a>

表 0-1 版本完成内容及分支链接

## 1、UNO-OS 概览

### 1.1 系统结构

系统结构如图1-1

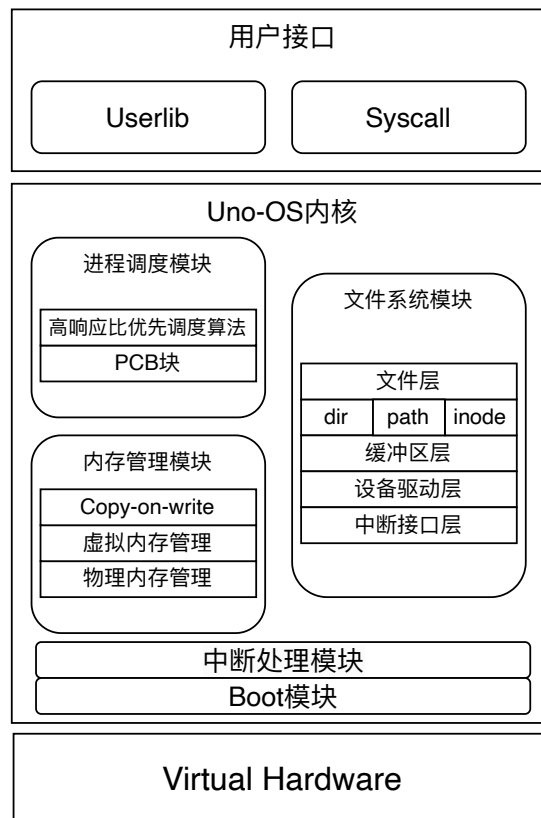


图 1-1 系统结构示意图

### 1.2 创新点

Uno-OS 通过目录的分级来完成模块化的设计，如图1-2。大体上，Uno-OS 分为 boot 模块，中断处理模块，内存管理模块，进程调度模块，文件系统模块。具体实现的系统调用，参见7

在完成了一个 OS 所必备的基本功能的基础上，Uno-OS 还进行了一系列创新，包括但不限于：

- 实现了 COW 策略，详见 4.2.3 节
- 实现了类似 UNIX 下的多级索引文件结构，详见6.5.3 节
- 实现了高响应比优先（HRRN）算法，详见5.4 节

```

./kernel/
├── boot      # 启动模块
│   ├── entry.S          # 用于加载start函数
│   ├── main.c           # 内核入口函数，完成必要的初始化工作
│   ├── start.c          # start函数，完成从M模式向S模式的跳转
│   └── Makefile
├── dev       # 设备和驱动模块
│   ├── console.c        # 控制台设备
│   ├── plic.c           # 处理PLIC外设中断
│   ├── timer.c          # 系统时钟模块
│   ├── uart.c           # 处理UART外设中断
│   ├── virtio.c         # qemu虚拟IO接口
│   └── Makefile
├── fs        # 文件和IO模块
│   ├── bitmap.c         # 文件系统(bitmap)模块
│   ├── buf.c            # IO模块的缓冲层
│   ├── dir.c            # 文件目录模块
│   ├── file.c           # 文件结构抽象模块
│   ├── fs.c             # 文件系统的管理
│   ├── inode.c          # 索引节点的管理
│   └── Makefile
├── lib       # 相关的实用工具库
│   ├── print.c          # 打印和调试
│   ├── sleeplock.c      # 睡眠锁
│   ├── spinlock.c       # 自旋锁
│   ├── str.c            # 字符串操作
│   └── Makefile
├── mem       # 内存管理模块
│   ├── kvm.c            # 内核内存管理
│   ├── mmap.c           # 内存直接映射管理
│   ├── pmem.c           # 物理内存分配管理
│   ├── uvm.c            # 用户页表管理
│   └── Makefile
├── proc      # 进程管理模块
│   ├── cpu.c            # cpu管理模块
│   ├── exec.c           # elf程序执行模块
│   ├── proc.c           # 进程调度模块
│   ├── swtch.S          # 上下文切换
│   └── Makefile
├── syscall   # 系统调用管理模块
│   ├── syscall.c        # 系统调用处理
│   ├── sysfile.c        # 文件相关系统调用
│   ├── sysproc.c        # 进程相关系统调用
│   └── Makefile
├── trap      # 中断处理模块
│   ├── trampoline.S     # 内核态和进程态切换的处理
│   ├── trap_kernel.c    # M模式中断处理
│   ├── trap.S           # 进行上下文切换相关工作
│   ├── trap_user.c      # S模式中断处理
│   └── Makefile
├── Makefile
└── kernel.ld

```

图 1-2 工程目录结构

- 优化磁盘缓冲区算法，保证即使在最坏情况下也只需遍历一次链表，详见6.4节

此外，我们预期还将实现如下创新点和优化方案：

- 实现 UNIX 下的管道机制
- 实现懒分配策略
- 进行更加完备的测试
- 实现更多系统调用

### 1.3 运行方法

Uno-OS 系统的构建和运行相关参数写在 Makefile 中，要运行该系统，在命令行下输入：

```
$ make qemu
```

即可启动系统

## 2、boot 模块

### 2.1 功能介绍

boot 模块的内容位于 **/kernel/boot** 文件夹中。用于完成系统的加载启动以及初始化工作。

### 2.2 实现分析

在 Uno-OS 的磁盘映像中，**entry.S** 的内容会被写入到磁盘的引导扇区中（地址 0x80000000）这个程序会做两件事：首先，将内核栈初始化，然后，跳转到 `start` 函数中执行必要的初始化例程。

接下来，**start.c** 中的 `start` 函数。这里主要做的事情是进行由 RISC-V 的 M 模式跳转到 S 模式的准备工作，这主要是通过设置 `mstatus` 中的字段值实现的<sup>[1]</sup>。此外，还需要为异常设置 `delegation` 以方便统一处理。最后，`mret` 函数使我们进入 S 模式的同时跳转到下面的 `main` 函数。

由于我们的 `qemu` 配置选项假定该系统运行在一个双核的机器上，我们约定在 CPU0 上执行由所有 CPU 共享的资源（如，内存管理，磁盘中断等），各 CPU 初始化自己私有的资源（一般这类函数以 `inithart` 结尾），以下为 **main.c** 中的代码：

```
1 int main()
2 {
3     int cpuid = r_tp();
4
5     if(cpuid == 0) {
6         print_init();
7         pmem_init();
8         kvm_init();
9         kvm_inithart();
10        trap_kernel_init();
11        trap_kernel_inithart();
12        plic_init();
13        plic_inithart();
14        mmap_init();
15        virtio_disk_init();
```



```
16     proc_init();
17     proc_make_first();
18
19     __sync_synchronize();
20     started = 1;
21     printf("cpu %d is booting!\n", cpuid);
22 } else {
23
24     while(started == 0);
25     __sync_synchronize();
26
27     kvm_inithart();
28     trap_kernel_inithart();
29     printf("cpu %d is booting!\n", cpuid);
30 }
31 proc_scheduler();
32
33 while (1);
34 }
```

最后，main 函数会调用 `proc_scheduler` 函数，系统进入调度器，调度器的工作将会在后面的进程调度模块中详细描述<sup>5</sup>

### 2.3 测试

在完成了所有的初始化并进入 `proc_scheduler` 之前，每个 `cpu` 都会打印一条消息表明成功启动。

```
qemu-system-riscv64 -machine virt -bios none -kernel kernel-qemu
cpu 0 is booting!
cpu 1 is booting!
```

图 2-1 boot 模块测试结果

### 3、中断处理模块

#### 3.1 功能介绍

中断是系统实现用户接口和硬件访问的重要工具，当中断或异常发生时，处理器会暂停当前正在执行的任务，保存其状态，并跳转到专门的中断处理程序来处理这些事件。这种机制确保了系统的稳定性和安全性。在 Uno-OS 中，中断模块位于 `/kernel/trap` 中。

#### 3.2 中断处理的过程

中断处理中，我们分为 `kernel` 态和 `machine` 态分别处理，`machine` 态主要处理的是时钟中断，这两类中断的核心处理函数分别为 `trap_user_handler` 和 `trap_kernel_handler` 中。此外，我们还有 `kernel_vector` 这一辅助函数来保存 `cpu` 状态以及跳转到相应的处理函数。

##### 3.2.1 M 模式下的中断处理

根据 RISC-V 规范，在 M 模式下发生中断时，CPU 会跳转到 `mtvec` 寄存器指定的地址执行中断处理函数。在 Uno-OS 中，我们需要 M 模式下的中断处理函数完成时钟中断的处理

##### 3.2.2 S 模式下的中断处理

同样的，S 模式下的中断会跳转到 `stvec` 寄存器。

S 模式下的中断处理函数需要完成如下操作的响应

- 由于时钟中断而引发的 SSI
- 外设引起的中断
- 系统调用

当在用户态触发陷阱之后，就会调用 `trap_user_handler` 处理。

```
1  if (IS_INTR(scause)) {
2  switch(trap_id)
3  {
4      case SMODE_SOFTWARE_INTERRUPT:
5          timer_interrupt_handler();
6          break;
```



在由内核态返回用户态时，需要先关闭中断，确保在设置陷阱向量和陷阱帧时不会被打断，通过 `(uint64)user_vector-(uint64)trampoline` 计算出 `user_vector` 的偏移量，以便将其地址写入 `stvec` 寄存器。

然后设置 `uservec` 需要的陷阱帧值，以正确恢复内核页表、内核栈指针、陷阱处理程序地址和 CPU ID。

`x &= SSTATUS_SPP` 将 `x` 中的 `SSTATUS_SPP` 位清零，确保 `sret` 将返回的不是超级用户模式，又通过 `x |= SSTATUS_SPIE` 启用返回模式下超级用户模式的中断，然后再将更新都写回处理器。

同理，在设置完 `sepc` 寄存器后，`uint64 satp = MAKE_SATP(p->pgtbl);` 创建并设置新的 SATP 值，以便切换到用户模式时使用新的页表。最后跳转到 **trampoline.S**，完成从内核模式到用户模式的切换，包括设置页表、恢复用户寄存器并通过 `sret` 指令进入用户模式。

### 3.3 时钟中断

#### 3.3.1 原理

##### 3.2.1. Machine Timer Registers (`mtime` and `mtimecmp`)

Platforms provide a real-time counter, exposed as a memory-mapped machine-mode read-write register, `mtime`. `mtime` must increment at constant frequency, and the platform must provide a mechanism for determining the period of an `mtime` tick. The `mtime` register will wrap around if the count overflows.

The `mtime` register has a 64-bit precision on all RV32 and RV64 systems. Platforms provide a 64-bit memory-mapped machine-mode timer compare register (`mtimecmp`). A machine timer interrupt becomes pending whenever `mtime` contains a value greater than or equal to `mtimecmp`, treating the values as unsigned integers. The interrupt remains posted until `mtimecmp` becomes greater than `mtime` (typically as a result of writing `mtimecmp`). The interrupt will only be taken if interrupts are enabled and the MTIE bit is set in the `mie` register.



Figure 27. Machine time register (memory-mapped control register).

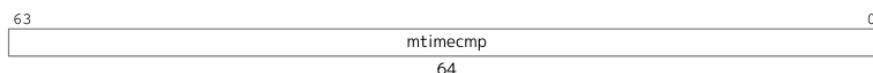


Figure 28. Machine time compare register (memory-mapped control register).

图 3-1 RISC-V 官方文档中对时钟中断相关寄存器 (MTIME) 的介绍

在 RISC-V 架构中，CLINT 定义了一个 MTIME 区域(详见 `/include/mem/memlayout.h`)。该区域储存了系统的时钟周期数，每次时钟周期数递增 1。此外，还有一个 MTIMECMP

区域，该区域用于和 MTIME 计数器进行比较，如果发现 MTIME 计数器的值大于等于 MTIMECMP，则会触发一次时钟中断。<sup>[1]</sup>

### 3.3.2 初始化的过程

有了上述背景资料的支撑，我们再来看看 Uno-OS 中对时钟中断的初始化流程，初始化的工作主要在 timer\_init 中完成。

```

1 // 时钟初始化
2 // called in start.c
3 void timer_init()
4 {
5     int current_cpuid=r_mhartid();
6
7     // 时钟中断的频率由TIMER_INTERVAL的值指定
8     *(uint64*)CLINT_MTIMECMP(current_cpuid)=*(uint64*)
        CLINT_MTIME+TIMER_INTERVAL;
9
10
11     mscratch[current_cpuid][3]=CLINT_MTIMECMP(current_cpuid);
12     mscratch[current_cpuid][4]=TIMER_INTERVAL;
13     w_mscratch((uint64)mscratch[current_cpuid]);
14     w_mtvec((uint64)timer_vector);
15
16     // 设置允许M模式下的时钟中断
17     w_mstatus(r_mstatus() | MSTATUS_MIE);
18     w_mie(r_mie() | MIE_MTIE);
19 }

```

mscratch 寄存器是一个指针，该指针指向一段内存区域，这段内存区域用于在发生中断时暂存一部分临时变量的值。mscratch[current\_cpuid][3] 存储着 MTIMECMP 的地址，mscratch[current\_cpuid][4] 存储着 TIMER\_INTERVAL，即发生时钟中断的间隔。

### 3.3.3 时钟中断处理函数

时钟中断是 M 模式下的中断，因此他会跳转到 `mtvec` 指向的函数，由上面的初始化步骤我们可以发现，`mtvec` 指向 `timer_vector` 函数，该函数位于 `/kernel/trap/trap.S` 文件中。代码不在此处列出，大体上，这个函数干了以下三件事

- 进行暂存数据的恢复与存储
- 使 `MTIMECMP` 自增 `TIMER_INTERVAL`，从而为下一次时钟中断的处理做好准备
- 触发一个 SSI（S 模式下的软件中断）

最后一个步骤相当于将对时钟中断的后续在发生 SSI 时，中断处理函数会识别出该中断的类型。然后调用 `timer_interrupt_handler` 来完成时钟中断的响应。这个函数会更新系统时钟，该操作由 `cpu0` 负责，此外，他还会强制当前进程放弃 `cpu`，这是我们实现时间片轮转调度算法的必要步骤<sup>5</sup>

## 3.4 系统调用

用户程序通过执行特定的指令（如 ‘`ecall`’）发起系统调用请求。

### 3.4.1 内核处理请求

当用户程序发生 `trap` 时，CPU 会切换到内核模式，并进入 `trap_user_handler()` 函数，此时，操作系统开始处理异常或中断。

系统首先保存了陷阱发生时的上下文信息，如 `sepc`（用户程序的 PC），`sstatus`（当前状态），以及 `scause`（触发陷阱的原因）。接着，检查是否来自用户模式，并将中断向量地址设置为 `kernel_vector`，以便后续内核能够处理异常。

函数根据 `scause` 判断此次陷阱是中断还是异常。如果是中断，则会根据具体的中断类型调用相应的中断处理函数。否则，就调用 `syscall()` 函数处理系统调用（这里只对系统调用进行处理，其他异常不做过多操作）。

```
1 void trap_user_handler()
2 {
3     // 这些寄存器保存了陷阱发生时的上下文信息。
4     uint64 sepc = r_sepc();           // 记录了发生异常时的pc值
5     uint64 sstatus = r_sstatus();    // 与特权模式和中断相关的状态信息
```

```
6      uint64 scause = r_scause();    // 引发trap的原因
7      // uint64 stval = r_stval();    // 发生trap时保存的附加
      // 信息（不同trap的附加信息不同）
8
9      // 获取当前进程的信息，trap_id是scause寄存器最低4位
10     proc_t *p = myproc();
11     int trap_id = scause & 0xf;
12
13     // 确认trap来自用户模式
14     assert((sstatus & SSTATUS_SPP) == 0, "trap_user_handler:
      not from u-mode");
15
16     // 设置S模式的中断向量地址，使得内核可以处理异常
17     w_stvec((uint64)kernel_vector);
18
19     // 保存trap前的PC值
20     p->tf->epc = sepc;
21
22     // 如果trap是中断，则调用相应的中断处理程序，略
23     // ...
24
25     // 如果是异常
26     else {
27         switch(trap_id)
28         {
29             case UMODE_SYSCALL_INTERRUPT:
30                 p->tf->epc += 4;    // 跳过ecall指令，指向返回
      // 后要执行的下一指令
31                 intr_on();
32                 syscall();
```

```
33         break;
34     default:    // 未知系统调用
35         panic("trap_user_handler: Unknown trap id %x
36               , \n\tdescription:%s", trap_id,
37               exception_info[trap_id]);
38         break;
39     }
40 }
41
42 // 处理完毕后，将控制权返回给用户程序
43 trap_user_return();
44 }
```

在处理系统调用时，内核通过维护一个系统调用表 `syscalls[]` 来映射系统调用号与对应的处理函数。每个系统调用都有一个唯一编号，存储在 `a7` 寄存器中。当用户程序发起系统调用时，内核通过该编号从 `syscalls[]` 表中查找并执行相应的处理函数。

```
1 // 系统调用跳转
2 static uint64 (*syscalls[])(void) = {
3     [SYS_print]      sys_print,
4     [SYS_brk]        sys_brk,
5     [SYS_mmap]        sys_mmap,
6     [SYS_munmap]      sys_munmap,
7     [SYS_fork]        sys_fork,
8     [SYS_wait]        sys_wait,
9     [SYS_exit]        sys_exit,
10    [SYS_sleep]        sys_sleep,
11
12    [SYS_open]         sys_open,
13    [SYS_close]        sys_close,
14 };
```

`syscall()` 函数处理系统调用，主要做两件事：



- 获取系统调用号：当用户程序发出系统调用请求时，会触发一个陷阱（trap），并将系统调用号放入 a7 寄存器中。在 syscall() 函数中，通过 p->tf->a7 获取该系统调用号，这个值用于确定要调用哪个系统调用处理函数。
- syscall() 会检查系统调用号是否有效。如果有效，函数会从 syscalls[] 表中找到对应的处理函数并执行，结果通过 a0 寄存器返回给用户程序；如果系统调用号无效，函数会触发错误，调用 panic() 报错。

```

1 // 系统调用
2 void syscall()
3 {
4     struct proc *p=myproc();
5     int syscall_num=p->tf->a7; // a7存储系统调用号
6
7     if(syscall_num >= 0 && syscall_num < sizeof(syscalls)/
        sizeof(syscalls[0])
8         && syscalls[syscall_num])
9         p->tf->a0 = syscalls[syscall_num]();
10    else{
11        panic("syscall: Unknown index in user system
            interrupt vector: %d", syscall_num);
12    }
13 }

```

### 3.4.2 返回用户状态

在内核完成系统调用处理后，trap\_user\_return() 函数负责将控制权交还给用户程序。

它通过关中断防止在返回用户态时发生中断，并使用 w\_stvec() 设置中断向量地址，确保内核重新指向用户态的陷阱处理程序。为了正确恢复内核态时的状态，函数还保存了内核页表、栈地址、陷阱处理函数以及 CPU ID。在更新 SSTATUS 寄存器后，函数恢复用户程序的 PC 并设置用户页表。最后通过跳转 trampoline.S 切换用户页表并恢复用户寄存器，sret 指令切换到用户模式，确保用户程序继续执行。

```
1 // 内核态返回用户态
2 void trap_user_return()
3 {
4     proc_t* p = myproc();
5     intr_off();
6
7     // 使得trap重新指向user_vector
8     w_stvec(TRAMPOLINE + ((uint64)user_vector - (uint64)
9         trampoline));
10
11     // 设置trap帧的内核相关信息，用于恢复内核模式时的状态
12     p->tf->kernel_satp = r_satp(); // 内核页表
13     p->tf->kernel_sp = p->kstack + PGSIZE; // 进程的内核栈
14     p->tf->kernel_trap = (uint64)trap_user_handler; // 设置
15     // 内核中断处理函数为 trap_user_handler
16     p->tf->kernel_hartid = r_tp(); // hartid 用于 CPU ID
17
18     // 更新SSTATUS寄存器，设置为用户模式的状态
19     unsigned long x = r_sstatus();
20     x &= ~SSTATUS_SPP; // 清除SSTATUS_SPP，表示当前处于用户
21     // 模式
22     x |= SSTATUS_SPIE; // 设置SSTATUS_SPIE，表示中断允许
23     w_sstatus(x);
24
25     w_sepc(p->tf->epc);
26     // 告诉trampoline.S，切换到的用户页表
27     uint64 satp = MAKE_SATP(p->pgtbl);
28
29     // 跳转到trampoline.S，切换到用户页表，恢复用户寄存器，并
30     // 通过sret切换到用户模式
```

```

27     uint64 fn = TRAMPOLINE + (user_return - trampoline);
28     ((void (*)(uint64, uint64))fn)(TRAPFRAME, satp);
29 }
    
```

下面的流程图可以更加清晰地展示整系统调用这一过程：

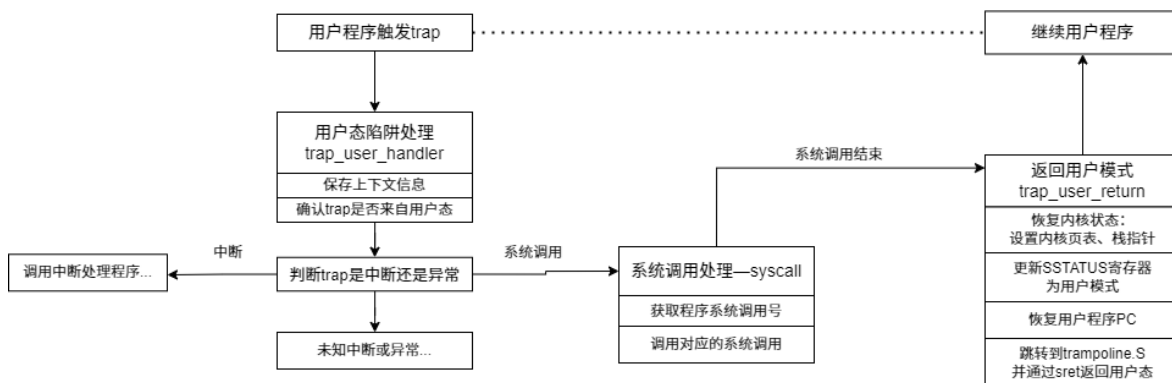


图 3-2 系统调用流程

### 3.5 外设中断

Uno-OS 中的外设中断主要基于 PLIC，可分为两类，一类是读取键盘输入以及输出字符到显示屏的 UART。另一类则是用于读写磁盘的 virtio。

在内核状态下处理外设中断时，我们根据其中断号调用对应的中断处理程序。而用户状态下对于外设中断的处理，则是通过调用内核状态的处理程序完成的。

```

1 void external_interrupt_handler()
2 {
3     int irq = plic_claim(); // 获取中断号
4
5     if (!irq) return;
6     else if (irq == UART_IRQ)
7         uart_intr();
8     else if (irq == VIRTIO_IRQ)
9         virtio_disk_intr();
10    else
11        printf("Unexpected interrupt irq = %d\n", irq);
12
    
```

```
13     plic_complete(irq);  
14 }
```

## 3.6 测试

### 3.6.1 时钟中断的实现

我们在时钟中断处理函数中，插入一条打印语句，这样每当系统成功进入时钟中断的处理函数时都会打印一条信息

```
1 void timer_interrupt_handler()  
2 {  
3     // ...  
4     printf("CPUID:%d di da\n",mycpuid());  
5 }
```

结果如下：

```
qemu-system-riscv64 -machine virt -bios none -kernel kernel-qemu -m 128M -smp 2 -nographic  
CPUID:0 di da  
hart 0 starting  
CPUID:1 di da  
hart 1 starting  
CPUID:0 di da  
CPUID:1 di da  
CPUID:0 di da  
CPUID:1 di da  
CPUID:0 di da  
CPUID:1 di da
```

图 3-3 时钟中断功能正常

## 4、内存管理模块

### 4.1 功能介绍

内存管理系统是操作系统中负责分配、回收和管理计算机内存的关键组件。它通过为进程分配合适的内存空间，确保每个进程能够高效运行，同时避免内存冲突和浪费。Uno-OS 的虚拟地址管理基于 RISC-V 的 Sv39 模式，即虚拟地址空间大小为  $2^{39} - 1$  个字节。整体设计上，Uno-OS 的虚拟内存空间设计参考了 xv6。内存管理模块位于 `/kernel/mem` 目录下。

### 4.2 物理内存的管理

#### 4.2.1 物理内存布局

因此我们可以将物理内存大致分为下面三部分。

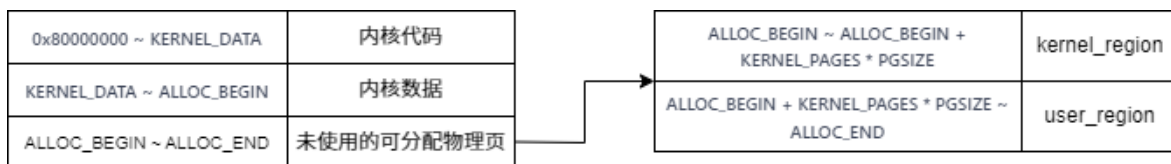


图 4-1 物理内存分布

显然，前两部分的物理页会一直被内核占用，不会动态分配和回收，我们只需要管理好第三部分的物理页。而为了避免恶意用户程序通过不断申请物理页而使内核空间耗尽，我们还是将内核物理页和用户物理页分开管理（如图 `kernel_region` 和 `user_region`）。

我们在 `common.h` 中定义物理页的大小为 `PGSIZE` (4096 Byte) 这样问题就从管理连续的地址空间 `ALLOC_BEGIN ~ ALLOC_END` 转化成了管理有限个独立的物理页。为了便于物理页的插入和移出，我们选择使用链表作为核心的数据结构。

```

1 typedef struct page_node
2 {
3     struct page_node *next; // 指向下一个物理页节点
4     uint32 ref_cnt;         // 记录物理页的引用次数
5 } page_node_t;
6
7 typedef struct alloc_region
8 {
9     uint64 begin;           // 该区域的起始物理地址
10    uint64 end;             // 该区域的终止物理地址
11    spinlock_t lk;          // 自旋锁，用于保护以下两个变量
12    uint32 allocable;        // 当前可分配的页面数量
13    page_node_t list_head;   // 可分配链表的头节点
14 } alloc_region_t;
15
16 static alloc_region_t kern_region, user_region;
17 #define KERN_PAGES 1024 // 定义内核可分配空间的页数

```

`page_node_t` 表示了一个物理页的管理单元，它包含一个指向下一个物理页的指针，以及一个记录物理页被引用次数的计数器。`alloc_region_t` 则用于管理一块连续的

可分配物理内存区域，它包含了以下信息：

- **begin** 和 **end** 字段用来记录该物理内存区域的起始和结束地址。通过这两个字段，能够明确地划分出内核区域和用户区域，确保内核空间 and 用户空间的物理页不会发生交叉。
- **list\_head** 维护了该区域内剩余可分配物理页的链表。这个链表是按照物理页的顺序组织的，每个节点指向一个物理页，便于物理页的分配和回收。
- **allocable** 用于记录当前区域中可分配的物理页数目，配合链表，可以快速确定可用的物理页数量。

#### 4.2.2 物理内存的初始化

```
1 // 物理内存初始化函数
2 void pmem_init(void)
3 {
4     kern_region.begin = (uint64)ALLOC_BEGIN;
5     kern_region.end = (uint64)ALLOC_BEGIN + (uint64)
6         KERN_PAGES * PGSIZE;
7     kern_region.allocable = (uint64)KERN_PAGES;
8     spinlock_init(&kern_region.lk, "kern_region_lock");
9     kern_region.list_head.next = NULL;
10    free_range(kern_region.begin, kern_region.end, true);
11
12    // same in user_region
13    // ...
14 }
```

初始化也是按照内核区域和用户区域分开进行。通过这种方式，内核和用户的物理内存被清晰地分隔，并且每个区域都有独立的管理结构，确保在后续的内存分配和回收过程中不会发生冲突。

#### 4.2.3 物理内存的分配与回收

物理内存的分配与回收需要靠 `pmem_alloc()` 函数和 `pmem_free()` 函数实现。因为这些函数需要同时作用与内核区域和用户区域，为了区分，我们加入一个布尔值



`in_kernel` 作为参数，当值为 `true` 时，代表函数的作用范围在内核区域，反之则在用户区域。

- `pmem_alloc()` 函数从内核或用户区域返回一个未使用的干净的物理页，如果没有可分配物理页，就触发 `panic` 并锁死程序；如果成功分配物理页，那么将分配的物理页全部清空（置为 0），并把引用数设为 1。
- `pmem_free()` 函数在将物理页引用数减 1 后，还要判断该物理页的引用数是否为 0，只有为 0 时，才能将该物理页释放。

为了操作的简单，我们在分配物理页时每次都取链表中第一个可分配的物理页——也就是 `list_head` 后面紧跟着的页面，而在归还物理页时，同样也使用头插法将物理页插入链表。

代码实现如下：

```

1 void *pmem_alloc(bool in_kernel)
2 {
3     alloc_region_t *region = in_kernel ? &kern_region : &
        user_region;
4     spinlock_acquire(&region->lk);
5
6     // 检查当前区域的可分配页面数量是否为0，若为0则触发panic
7     if (region->allocable == 0)
8     {
9         spinlock_release(&region->lk);
10        panic("There is no empty page.");
11    }
12
13    // 获取链表头部的下一个节点，即第一个可用的物理页
14    page_node_t *page = region->list_head.next;
15    if (page != NULL) {
16        region->list_head.next = page->next;
17        --region->allocable;

```

```

18     spinlock_release(&region->lk);
19     memset(page, 0, PGSIZE);
20     page->ref_cnt = 1;
21 } else {
22     spinlock_release(&region->lk);
23     panic("Failed to allocate physical page!");
24 }
25 return (void *)page;
26 }

```

cow 机制可能会导致多个进程共享同一份物理内存，因此我们不能随意释放一个物理页，只有当物理页的引用数为 0 时，才能将其释放。

```

1 void pmem_free(uint64 page, bool in_kernel)
2 {
3     alloc_region_t *region = in_kernel ? &kern_region : &
        user_region;
4     assert((page % PGSIZE == 0) && (page >= region->begin &&
        page < region->end), "pmem_free: Invalid page address"
5     );
6     spinlock_acquire(&region->lk);
7     page_node_t *free_page = (page_node_t *)page;
8
9     // 页面引用数不为0，不能直接删除
10    if(--free_page->ref_cnt > 0){
11        spinlock_release(&region->lk);
12        return;
13    }
14
15    // 通过使用memset将内存区域设置为垃圾值，可以确保在释放内存后，
16    // 任何尝试访问该内存的操作都会失败，从而避免悬空引用的问

```

```

    题。
16     memset((void *)page, 1, PGSIZE);
17     free_page->next = region->list_head.next;
18     region->list_head.next = free_page;
19     ++region->allocable;
20     spinlock_release(&region->lk);
21 }
22
23 void free_range(uint64 begin, uint64 end, bool in_kernel)
24 {
25     char *p;
26     p = (char *)PGROUNDUP((uint64)begin);
27     for (; p + PGSIZE <= (char *)end; p += PGSIZE) {
28         pmem_free((uint64)p, in_kernel);
29     }
30 }

```

其管理流程用图来表示的话大致如下：

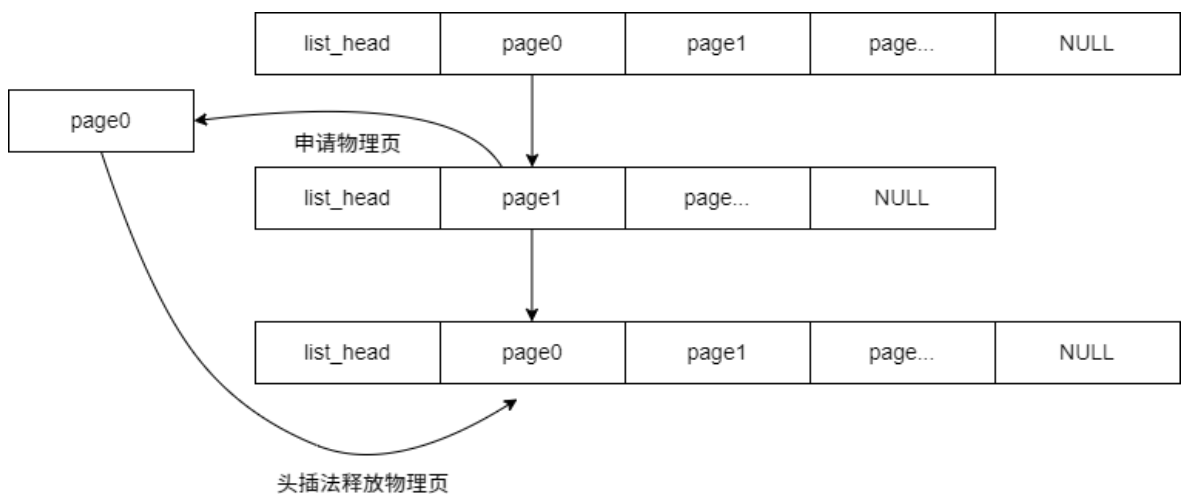


图 4-2 物理内存分配与回收示意图

#### 4.2.4 COW

在上面的代码中我们已经简单实现了物理内存的管理。但是要实现 COW，我们还需要一些工作。

#### 4.2.4.1 cow 实现

辅助函数（代码略）：

- `void pmem_inc_ref(uint64 page)` ——增加页面引用计数
- `void pmem_dec_ref(uint64 page)` ——减少页面引用计数
- `pmem_get_ref(uint64 page)` ——获取页面引用计数

在实现 COW 时，我们不再通过 `copy_range` 完全克隆父进程的物理内存，而是让父子进程共享同一物理内存。具体的做法就是：遍历父进程的内存空间，将物理地址封装为 PTE，并加入子进程的页表，将父子进程的 PTE 都设置为只读，当任一进程写入时，触发页面故障，执行实际的内存分配。

```

1 static void copy_range(pgtbl_t old, pgtbl_t new, uint64 begin
    , uint64 end)
2 {
3     uint64 va, pa, page;
4     int flags;
5     pte_t *pte;
6
7     for (va = begin; va < end; va += PGSIZE){
8         pte = vm_getpte(old, va, false);
9         assert(pte != NULL, "uvm_copy_pgtbl: pte == NULL");
10        assert((*pte) & PTE_V, "uvm_copy_pgtbl: pte not valid
            ");
11        // assert(PTE_CHECK(*pte), "uvm_copy_pgtbl: pte check
            fail");
12
13        pa = (uint64)PTE_TO_PA(*pte);
14        flags = (int)PTE_FLAGS(*pte);
15        page = (uint64)pmem_alloc(false); // 传入的不可能是
            kernel_pgtbl 吧 lol
16        memcpy((char *)page, (const char *)pa, PGSIZE);

```

```

17
18     if(flags & PTE_W){
19         flags |= PTE_COW;
20         flags &= ~PTE_W;
21         *pte = PA_TO_PTE(pa) | flags;
22     }
23
24     vm_mappages(new, va, page, PGSIZE, flags);
25     pmem_inc_ref(pa);
26 }
27 }

```

除了将 PTE 设置为只读外，我们还通过预留位来标记该页面为 COW 页面，与普通只读页面区分开：

```

1 // 页面权限控制
2 // ...
3 #define PTE_D (1 << 7) // dirty
4 #define PTE_COW (1 << 8) // cow
5 // ...

```

#### 4.2.4.2 page fault 处理

至此，我们基本实现了 COW，接下来要做的就是处理写时复制时发生的 page fault。这里只需要处理 `scause==15` 的情况，因为 13 是页面读错误，而 COW 是不会引起读错误的。

定义两个辅助函数：

- `is_cowpage` 判断一个页面是否为 COW 页面；
- `handle_cow` 处理 COW 页面的页面故障。当进程写入 COW 页面时，会根据情况复制页面或修改页表。

```

1 int is_cowpage(pgtbl_t pg, uint64 va) {
2     pte_t *pte = vm_getpte(pg, va, false);

```

```
3     if (pte == NULL)
4         return 0;
5     return (*pte & PTE_COW) != 0;
6 }
7
8 void* handle_cow(pgtbl_t pg, uint64 va) {
9     pte_t *pte = vm_getpte(pg, va, false);
10    assert(pte != NULL, "handle_cow: pte is NULL");
11
12    uint64 pa = PTE_TO_PA(*pte);
13    int flags = PTE_FLAGS(*pte);
14
15    uint64 new_page = (uint64)pmem_alloc(false);
16    assert(new_page != 0, "handle_cow: failed to allocate
17           new page");
18    // 如果该页只有当前进程引用，则直接修改页表条目并返回物理
19    // 页
20    if ((flags & PTE_W) == 0)
21        *pte = PA2PTE(pa) | flags;
22    else {
23        memcpy((char *)new_page, (char *)pa, PGSIZE);
24        *pte = PA2PTE(new_page) | (flags & ~PTE_W) | PTE_COW;
25        // 设置为新页面，去除写权限，增加COW标志
26    }
27
28    pmem_inc_ref(pa);
29    return (void *)new_page;
30 }
```

有了这两个函数，我们就能在 `trap_user_handler` 函数中增加对 `page fault` 的处理

了。

```

1 // ...
2
3 case UMODE_INSTRUCTION_PAGE_FAULT:
4     if (is_cowpage(p->pgtbl, stval))
5         assert(handle_cow(p->pgtbl != 0, stval), "
            trap_user_handler: fail to handle cowpage");
6     else
7         panic("trap_user_handler: Accessing non-COW page");
8
9 // ...
    
```

## 4.3 虚拟内存的管理

### 4.3.1 内核页表布局

Uno-OS 的内核页表设计参考 xv6，内核页表的绝大多数表项是直接映射到物理内存。同时，kstack 和 trapframe 等用户页表和内核页表所共有的部分不是直接映射，而且，他们位于地址空间的最高位。

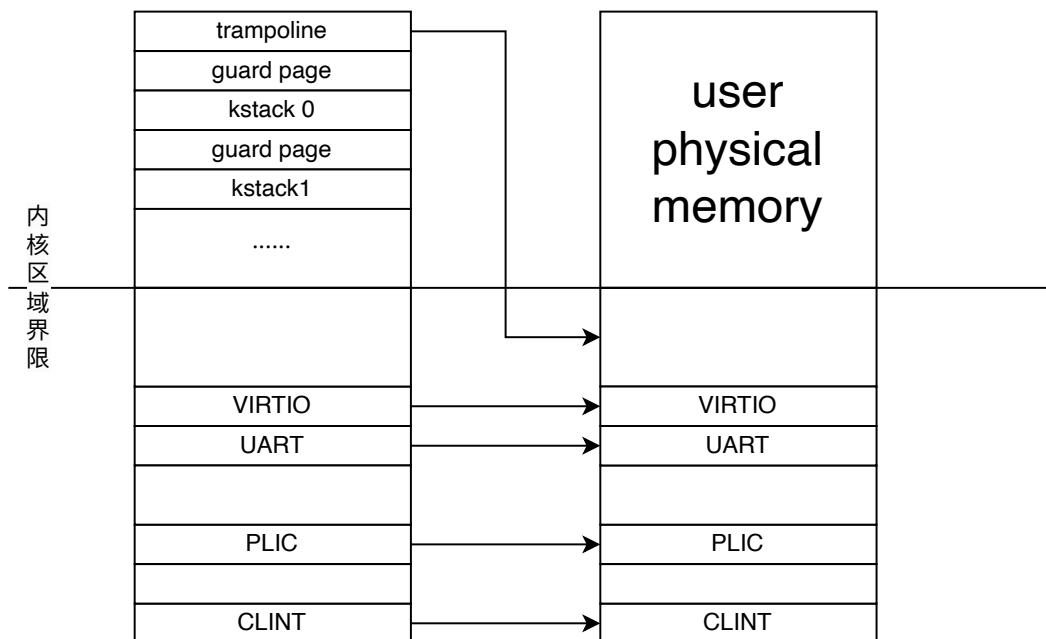


图 4-3 内核页表示意图

### 4.3.2 用户页表布局

用户页表的设计如下

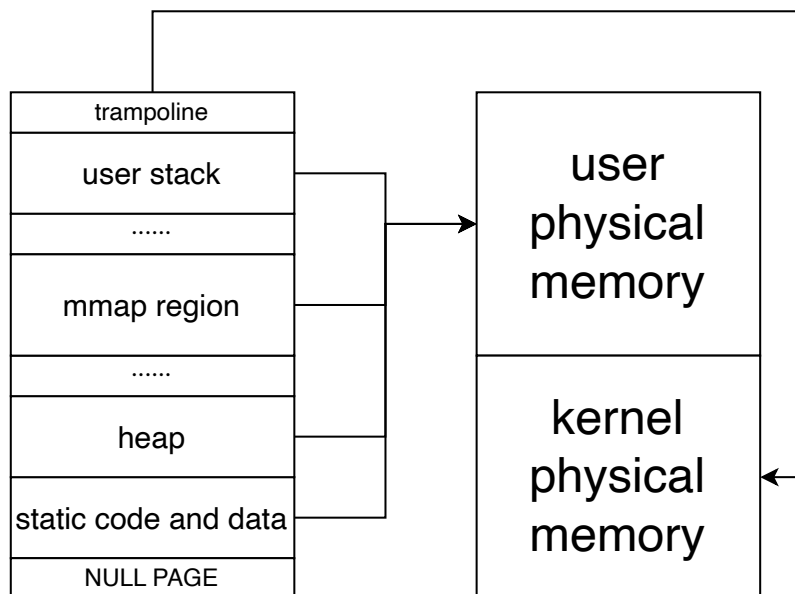


图 4-4 用户页表示意图

## 4.4 测试

### 4.4.1 物理内存的分配

```

1 volatile static int started = 0;
2 volatile static int over_1 = 0, over_2 = 0;
3 static int* mem[1024];
4
5 int main()
6 {
7     int cpuid = r_tp();
8
9     if(cpuid == 0) {
10
11         print_init();
12         pmem_init();

```



```
13
14     printf("cpu %d is booting!\n", cpuid);
15     __sync_synchronize();
16     started = 1;
17
18     for(int i = 0; i < 512; i++) {
19         mem[i] = pmem_alloc(true);
20         memset(mem[i], 1, PGSIZE);
21         printf("mem = %p, data = %d\n", mem[i], mem[i
22             ][0]);
23     }
24     printf("cpu %d alloc over\n", cpuid);
25     over_1 = 1;
26
27     while(over_1 == 0 || over_2 == 0);
28
29     for(int i = 0; i < 512; i++)
30         pmem_free((uint64)mem[i], true);
31     printf("cpu %d free over\n", cpuid);
32 } else {
33
34     while(started == 0);
35     __sync_synchronize();
36     printf("cpu %d is booting!\n", cpuid);
37
38     for(int i = 512; i < 1024; i++) {
39         mem[i] = pmem_alloc(true);
40         memset(mem[i], 1, PGSIZE);
```

```

41         printf("mem = %p, data = %d\n", mem[i], mem[i
           ] [0]);
42     }
43     printf("cpu %d alloc over\n", cpuid);
44     over_2 = 1;
45
46     while(over_1 == 0 || over_2 == 0);
47
48     for(int i = 512; i < 1024; i++)
49         pmem_free((uint64)mem[i], true);
50     printf("cpu %d free over\n", cpuid);
51
52 }
53 while (1);

```

在该测试中, **cpu-0** 和 **cpu-1** 并行申请内核中的全部物理内存, 赋值并输出信息, 待申请全部结束, 并行释放所有申请的物理内存。

测试输出为:

#### 4.4.2 页表的实现

```

1 int main()
2 {
3     int cpuid = r_tp();
4
5     if(cpuid == 0) {
6
7         print_init();
8         pmem_init();
9         kvm_init();
10        kvm_inithart();
11        printf("cpu %d is booting!\n", cpuid);

```

```

mem = 0x00000000803f5000, data = 16843009
mem = 0x00000000803f6000, data = 16843009
mem = 0x00000000803f7000, data = 16843009
mem = 0x00000000803f8000, data = 16843009
mem = 0x00000000803f9000, data = 16843009
mem = 0x00000000803fa000, data = 16843009
mem = 0x00000000803fb000, data = 16843009
mem = 0x00000000803fc000, data = 16843009
mem = 0x00000000803fd000, data = 16843009
mem = 0x00000000803fe000, data = 16843009
mem = 0x00000000803ff000, data = 16843009
mem = 0x0000000080400000, data = 16843009
mem = 0x0000000080401000, data = 16843009
mem = 0x0000000080402000, data = 16843009
mem = 0x0000000080403000, data = 16843009
cpu 0 alloc over
mem = 0x0000000080404000, data = 16843009
mem = 0x0000000080405000, data = 16843009
cpu 1 alloc over
cpu 1 free over
cpu 0 free over

```

图 4-5 物理内存的管理

```

12
13     __sync_synchronize();
14     started = 1;
15
16     pgtbl_t test_pgtbl = pmem_alloc(true);
17     uint64 mem[5];
18     for(int i = 0; i < 5; i++)
19         mem[i] = (uint64)pmem_alloc(false);
20
21     printf("\ntest-1\n\n");
22     vm_mappages(test_pgtbl, 0, mem[0], PGSIZE, PTE_R);

```

```

23     vm_mappages(test_pgtbl, PGSIZE * 10, mem[1], PGSIZE /
        2, PTE_R | PTE_W);
24     vm_mappages(test_pgtbl, PGSIZE * 512, mem[2], PGSIZE
        - 1, PTE_R | PTE_X);
25     vm_mappages(test_pgtbl, PGSIZE * 512 * 512, mem[2],
        PGSIZE, PTE_R | PTE_X);
26     vm_mappages(test_pgtbl, VA_MAX - PGSIZE, mem[4],
        PGSIZE, PTE_W);
27     vm_print(test_pgtbl);
28
29     printf("\ntest-2\n\n");
30     vm_mappages(test_pgtbl, 0, mem[0], PGSIZE, PTE_W);
31     vm_unmappages(test_pgtbl, PGSIZE * 10, PGSIZE, true);
32     vm_unmappages(test_pgtbl, PGSIZE * 512, PGSIZE, true)
        ;
33     vm_print(test_pgtbl);
34
35     } else {
36
37         while(started == 0);
38         __sync_synchronize();
39         printf("cpu %d is booting!\n", cpuid);
40
41     }
42     while (1);
43 }

```

该测试分为两个部分，第一部分测试分配页面，测试了相同页表项，不同一级页表项，不同二级页表项和虚拟内存空间顶端的虚拟地址空间。第二部分测试页面的释放，它将部分已分配区域释放。最后，我们还用一个工具函数打印当前页表：

```

qemu-system-riscv64 -machine virt -bios none -kernel kernel-qemu -m 128M -smp 2 -nographic
cpu 0 is booting!
cpu 1 is booting!

test-1

level-2 pgtbl: pa = 0x0000000080403000
.. level-1 pgtbl 0: pa = 0x0000000087fb4000
.. .. level-0 pgtbl 0: pa = 0x0000000080403000
.. .. .. physical page 0: pa = 0x0000000087fb9000 flags = 3
.. .. .. physical page 10: pa = 0x0000000087fb8000 flags = 7
.. .. level-0 pgtbl 1: pa = 0x0000000080403000
.. .. .. physical page 0: pa = 0x0000000087fb7000 flags = 11
.. level-1 pgtbl 1: pa = 0x0000000087fb1000
.. .. level-0 pgtbl 0: pa = 0x0000000080403000
.. .. .. physical page 0: pa = 0x0000000087fb7000 flags = 11
.. level-1 pgtbl 255: pa = 0x0000000087faf000
.. .. level-0 pgtbl 511: pa = 0x0000000080403000
.. .. .. physical page 511: pa = 0x0000000087fb5000 flags = 5

test-2

level-2 pgtbl: pa = 0x0000000080403000
.. level-1 pgtbl 0: pa = 0x0000000087fb4000
.. .. level-0 pgtbl 0: pa = 0x0000000080403000
.. .. .. physical page 0: pa = 0x0000000087fb9000 flags = 5
.. .. level-0 pgtbl 1: pa = 0x0000000080403000
.. level-1 pgtbl 1: pa = 0x0000000087fb1000
.. .. level-0 pgtbl 0: pa = 0x0000000080403000
.. .. .. physical page 0: pa = 0x0000000087fb7000 flags = 11
.. level-1 pgtbl 255: pa = 0x0000000087faf000
.. .. level-0 pgtbl 511: pa = 0x0000000080403000
.. .. .. physical page 511: pa = 0x0000000087fb5000 flags = 5

```

图 4-6 页表的实现

## 5、进程调度模块

### 5.1 功能介绍

进程调度模块是操作系统中负责管理和分配 CPU 使用权的核心组件，其主要功能是根据一定的调度算法选择就绪队列中的进程并分配 CPU，以实现多任务的高效运行。Uno-OS 初步实现了一个高响应比优先（HRRN）算法。进程调度模块位于 `/kernel/proc` 目录下。

### 5.2 PCB 块

进程控制块（PCB）是系统对进程进行管理的关键部分，由 `/include/proc/proc.h` 中的 `proc` 结构体定义，其结构如下：

```
1 typedef struct proc {
2     spinlock_t lk;           // 自旋锁
3
4     // 锁区域开始
5
6     int pid;                 // 标识符
7     enum proc_state state;   // 进程状态
8     struct proc* parent;     // 父进程
9     int exit_state;          // 进程退出时的状态(父进程可能关
10                             // 心)
11
12     void* sleep_space;       // 睡眠是为在等待什么
13
14     // HRRN相关
15     uint64 begin_runnable_time;
16     uint64 begin_running_time;
17     uint64 total_wait_time;
18     uint64 total_exec_time;
19
20     // 锁区域结束
21
22     pgtbl_t pgtbl;           // 用户态页表
```

```

21     uint64 heap_top;           // 用户堆顶(以字节为单位)
22     uint64 ystack_pages;      // 用户栈占用的页面数量
23     trapframe_t* tf;          // 用户态内核态切换时的运行环境
    暂存空间
24     mmap_region_t* mmap;      // 用户可映射区域的起始节点
25
26     uint64 kstack;            // 内核栈的虚拟地址
27     context_t ctx;            // 内核态进程上下文
28
29     inode_t* cwd;              // 当前目录
30     file_t* filelist[FILE_PER_PROC]; // 打开文件列表
31 } proc_t;

```

其中，锁区域之中的字段需要被保护以防止死锁或重调度的发生

所有的 PCB 定义于内核态，最大数量为 NPROCS，

### 5.3 初始进程

```

[jizimo@linuxdemo:~$ pstree
systemd--ModemManager--3*[{ModemManager}]
      --VGAAuthService
      --containerd--10*[{containerd}]
      --14*[cpptools-srv--7*[{cpptools-srv}]]
      --cron
      --dbus-daemon
      --dockerd--13*[{dockerd}]
      --dockerd--containerd--12*[{containerd}]
            --13*[{dockerd}]
      --firewalld--{firewalld}
      --fwupd--5*[{fwupd}]
      --5*[gdb-multiarch--8*[{gdb-multiarch}]]
      --login--bash
      --multipathd--6*[{multipathd}]
      --mysqld_safe--mysqld--35*[{mysqld}]
      --polkitd--3*[{polkitd}]
      --rsyslogd--3*[{rsyslogd}]
      --sh--node--5*[bash]
            --15*[{node}]
            --node--cpptools--16*[{cpptools}]
                  --node--6*[{node}]
                  --11*[{node}]
            --3*[node--12*[{node}]]

```

图 5-1 进程树示例

参考 xv6 以及 POSIX 规范，在系统启动时，会首先创建一个 proczero 进程，这个进程即所谓的“根进程”。其他的进程要想被创建，必须通过 fork 函数复制这个进

程（或其他的进程），即，进程不能凭空出现。而这样复制出来的进程又自然地 and 先前的进程形成了父子关系，如此一来，所有的进程便构成了一种树的关系（即，进程树）。上图是使用 linux 的 `ps tree` 命令显示的进程树的一部分。

我们的 `proczero` 进程便位于这棵树的根部。

## 5.4 调度算法

我们设计了一套不同于 xv6 中的时间片轮转调度算法的，高响应比优先算法 (HRRN)，该算法同时兼顾了短作业和长作业，是在实际应用中较为实用的一种调度算法。

我们定义响应比为： $ResponseRatio = [(1 + \frac{WaitTime}{ExecuteTime}) \times Granularity]$

式子后面乘上的常系数，并使用高斯函数取整是为了避免在内核中进行浮点数运算，同时提高计算响应比的精度的一种方法。实际算法如下：

```

1 // 检查响应比
2 // 当前锁被持有
3 static uint64 get_weight(proc_t *p)
4 {
5     assert(spinlock_holding(&(p->lk)), "get_weight: Not
        holding lock");
6     if(p->total_exec_time==0)
7     {
8         return -1; // -1 等价于 UINT64_MAX
9     }
10    return ((p->total_exec_time+p->total_wait_time)<<
        RATE_GRADIENT)/p->total_exec_time;
11 }

```

我们设计的调度器如下所示：

```

1 // 调度器
2 void proc_scheduler()
3 {
4     mycpu()->proc = NULL; // 当前没有正在执行的进程
5     int found = 0;

```

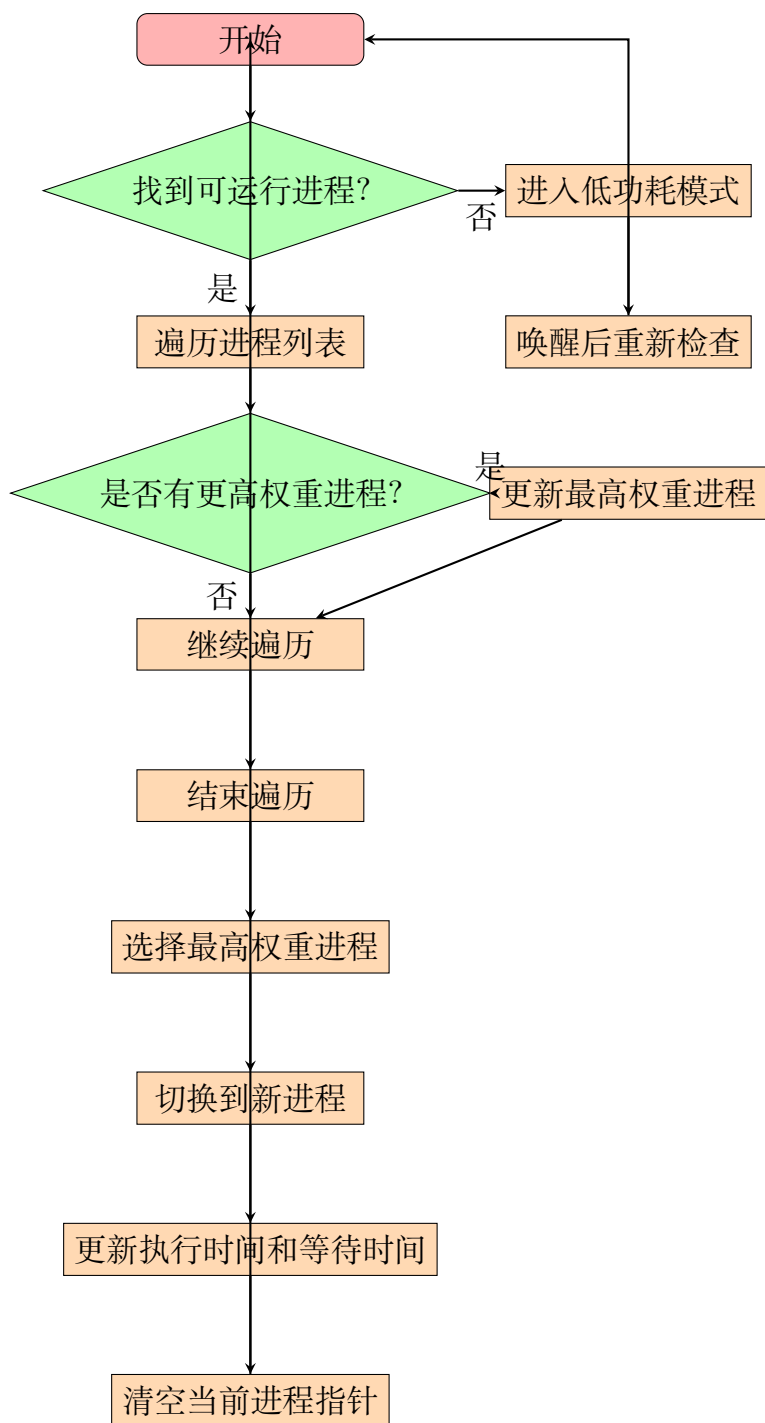


```
6     proc_t *p = NULL, *next_p = NULL;
7     while (1)
8     {
9         // 开中断以避免死锁的发生
10        intr_on();
11
12        found = 0;
13        next_p=NULL;
14        uint64 max_weight=0;
15        uint64 ticks = timer_get_ticks();
16        for (int idx = 0; idx < NPROC; ++idx)
17        {
18            p = &procs[idx];
19            spinlock_acquire(&(p->lk));
20            if (p->state == RUNNABLE)
21            {
22                p->total_wait_time += ticks - p->
                    begin_runnable_time; // 首先更新他们的等待
                    时间，然后再进行权重计算
23                p->begin_runnable_time=ticks;
24
25                // 下一个要执行的进程就是当前我们检查的proc
26                if(max_weight<get_weight(p))
27                {
28                    if(next_p)
29                    {
30                        spinlock_release(&(next_p->lk));
31                    }
32                    max_weight=get_weight(p);
33                    next_p=p;
```

```
34         }
35
36         found = 1;
37     }
38     if(next_p!=p)
39         spinlock_release(&(p->lk));
40 }
41
42 if (found == 0)
43 {
44     intr_on();
45     asm volatile("wfi"); // 进入低功耗模式等待可执行的进程
46 }
47 else
48 {
49     assert(next_p->state == RUNNABLE, "proc_scheduler
        : proc is not runnable");
50     next_p->state = RUNNING;
51     mycpu()->proc = next_p;
52     next_p->begin_running_time = ticks;
53                                     // 记录开始运行的时间
54
55     // 切换到next_p执行
56
57     // 更新进程的运行时间
58     ticks = timer_get_ticks();
59     mycpu()->proc->total_exec_time += ticks - mycpu()
        ->proc->begin_running_time; // 累计执行时间
```

```
60         mycpu()->proc->begin_runnable_time = ticks;
                                                // 记录等待的
                                                开始时间
61         spinlock_release(&(next_p->lk));
62
63         // 发生进程调度前首先将cpu的当前proc清空
64         mycpu()->proc = NULL;
65     }
66 }
67 }
```

上面的代码略微有些冗长, 我们使用流程图梳理一下



首先，我们从所有的进程中找出响应比最高的那个进程，然后，我们使用 `swtch` 函数切换上下文，跳转到相应的进程中执行。

此外，每当进程状态切换或进入调度器时，我们都需要设置 `proc_t` 中的相应字段，具体来说：

- SLEEPING/RUNNING->RUNNING 更新最近被调度的时间
- 调度器遍历累计等待时间
- RUNNING->RUNNABLE 更新最近进入就绪态的时间，累计运行时间

- **RUNNING->SLEEPING** 累计运行时间

这一系列规则略显复杂的原因在于，我们需要避免出现 **SLEEPING** 态使我们的累计等待时间不准确的情况。

## 5.5 进程间通信

为了完成进程间的通信，我们实现了一套系统调用，详见7.2节中的 `sleep`, `wait`, `exit` 等系统调用。

## 5.6 测试

### 5.6.1 高响应比优先 (HRRN) 调度算法

在测试时，我们向 `proc_scheduler` 函数中插入两条打印语句：

```
1   printf("Proc %d's current weight is %d\n", p->pid, (int)
    get_weight(p)); // 调试用
2   // 下一个要执行的进程就是当前我们检查的proc
3   if(max_weight < get_weight(p))
4   {
5       max_weight = get_weight(p);
6       next_p = p;
7   }
```

当找到一个 **RUNNABLE** 的进程时会执行这一语句。

```
1   if (found == 0)
2   {
3       intr_on();
4       asm volatile("wfi"); // 进入低功耗模式等待可执行的进
    程
5   }
6   else
7   {
8       printf("Proc %d running\n", next_p->pid);
9       spinlock_acquire(&(next_p->lk));
10      // ...
```

每一次调度最终执行的进程 id 将被打印出来

用户态进程代码如下：

```
1 #include "sys.h"
2
3 int main()
4 {
5     syscall(SYS_fork);
6     syscall(SYS_fork);
7
8     while(1);
9     return 0;
10 }
```

测试结果如图5-2

注意到，由于创建出来的几个新进程几乎是完全相同的忙等进程，因此他们的响应比几乎保持不变，从而此算法退化为一个时间片轮转调度算法。

我们使用一个更一般性的例子说明他与时间片轮转算法的不同点：

```
1 #include "sys.h"
2
3 int main()
4 {
5     if(syscall(SYS_fork)!=0)
6         syscall(SYS_sleep, 3);
7     if(syscall(SYS_fork)!=0)
8         syscall(SYS_sleep, 2);
9
10    while(1);
11    return 0;
12 }
```

这个例子中由于出现了进程进入阻塞态主动放弃 cpu 时间，从而导致响应比不同，实验结果如图5-3

```
Proc 1's current weight is 640
Proc 2's current weight is 640
Proc 3's current weight is 640
Proc 4's current weight is 621
Proc 1 running
Proc 1's current weight is 592
Proc 2's current weight is 658
Proc 3's current weight is 658
Proc 4's current weight is 640
Proc 2 running
Proc 1's current weight is 608
Proc 2's current weight is 608
Proc 3's current weight is 676
Proc 4's current weight is 658
Proc 3 running
Proc 1's current weight is 624
Proc 2's current weight is 624
Proc 3's current weight is 624
Proc 4's current weight is 676
Proc 4 running
Proc 1's current weight is 640
Proc 2's current weight is 640
Proc 3's current weight is 640
Proc 4's current weight is 624
Proc 1 running
Proc 1's current weight is 597
Proc 2's current weight is 656
Proc 3's current weight is 656
Proc 4's current weight is 640
Proc 2 running
Proc 1's current weight is 611
Proc 2's current weight is 611
Proc 3's current weight is 672
Proc 4's current weight is 656
Proc 3 running
Proc 1's current weight is 625
Proc 2's current weight is 625
Proc 3's current weight is 625
Proc 4's current weight is 672
Proc 4 running
```

图 5-2 高响应比优先算法测试结果

```
Proc 1's current weight is 256
Proc 2's current weight is 384
Proc 3's current weight is 512
Proc 4's current weight is 512
Proc 3 running
Proc 1's current weight is 384
Proc 2's current weight is 448
Proc 3's current weight is 426
Proc 4's current weight is 640
Proc 4 running
Proc 1's current weight is 512
Proc 2's current weight is 512
Proc 3's current weight is 469
Proc 4's current weight is 448
Proc 1 running
Proc 1's current weight is 384
Proc 2's current weight is 576
Proc 3's current weight is 512
Proc 4's current weight is 512
Proc 2 running
Proc 1's current weight is 448
Proc 2's current weight is 469
Proc 3's current weight is 554
Proc 4's current weight is 576
Proc 4 running
Proc 1's current weight is 512
Proc 2's current weight is 512
Proc 3's current weight is 597
Proc 4's current weight is 469
Proc 3 running
Proc 1's current weight is 576
Proc 2's current weight is 554
Proc 3's current weight is 512
Proc 4's current weight is 512
Proc 1 running
```

图 5-3 测试结果体现出不同点



## 6、文件系统模块

### 6.1 功能介绍

文件系统是操作系统中管理和存储数据的重要组件，其主要功能包括文件的创建、删除、读取和修改，提供灵活的命名规则和多种访问方式。此外，文件系统还负责组织文件的存储结构，支持目录管理以实现文件的分类存储，提供接口供用户和应用程序高效地操作文件，实现对存储设备的抽象管理。其中断层和驱动层位于 `/kernel/dev` 目录中，其余抽象层位于 `/kernel/fs` 目录中

### 6.2 中断层 (PLIC)

磁盘的 IO 操作是通过 PLIC 实现的，这需要在初始化时对 PLIC 寄存器进行写入，使其硬盘 IO 进入使能态。这主要涉及到 `kernel/plic/plic.c` 中的操作。

发生一个磁盘中断后，系统首先会进入 S 模式下的中断处理函数，随后，系统将调用 `external_interrupt_handler` 函数判断外部中断的 irq 号以区分串口 IO 和磁盘 IO 操作。然后，调用 `virtio_disk_intr` 函数来处理当前中断。这个函数主要涉及到驱动层中的操作。

### 6.3 驱动层

我们实现了 virtio 官方所实现的硬盘驱动接口，virtio 是一种虚拟化技术标准，用于为虚拟机提供高性能的虚拟设备接口<sup>[2]</sup>。

该层主要为上层提供了 `virtio_disk_rw` 函数来对磁盘进行写入，当其他代码想要进行直接的磁盘访问时，并不直接对这一层进行操作，而是使用缓冲层，这样可以提高性能。

### 6.4 缓冲层

类似在 CPU 寄存器和内存之间设计 cache 的操作，我们可以在内存中开辟一块空间用于缓存从磁盘中读取的数据，当需要访问磁盘时，首先检查目标数据是否存在于缓存中，若不存在则将数据从磁盘中读取到缓冲区中，再从缓冲区中读取数据。根据局部性原理，这样的操作显然可以提高系统性能。

我们将内存中的缓冲区组织为一个环形双链表，申明一个特殊的 `head_buf` 作为哨兵节点，从哨兵节点看去，其 `prev` 节点为最新的，当前未被引用的缓冲区，`next` 节点为最新的，已被引用的缓冲区。

为了实现高效的缓冲，我们需要针对磁盘块访问的特点设计合适的缓冲区替换算法。

我们的缓冲区寻找和替换算法如下：

```
1 buf_t* buf_read(uint32 block_num)
2 {
3     // printf("buf_read: buf %d allocated\n",block_num);
4     buf_t *target=NULL, *buf_node=head_buf.prev, *oldest_node=
        NULL;
5     spinlock_acquire(&lk_buf_cache);
6
7     // 局部性原理
8     while(target==NULL&&buf_node->buf_ref==0&&buf_node!=&
        head_buf)
9     {
10         // 找到一个块
11         if(buf_node->block_num==block_num)
12         {
13             insert_head(buf_node, 1);
14             target=buf_node;
15             target->buf_ref++;
16             spinlock_release(&lk_buf_cache);
17             sleeplock_acquire(&(target->slk));
18         }
19         buf_node=buf_node->prev;
20     }
21     // 避免多次遍历链表
22     oldest_node=buf_node->next;
23
24     buf_node=head_buf.next;
25     while(target==NULL&&buf_node!=oldest_node)
26     {
27         if(buf_node->block_num == block_num)
```

```

28     {
29         insert_head(buf_node, 1);
30         target=buf_node;
31         target->buf_ref++;
32         spinlock_release(&lk_buf_cache);
33         sleeplock_acquire(&(target->slk));
34     }
35     buf_node=buf_node->next;
36 }
37
38 if(target==NULL && oldest_node!=&head_buf)
39 {
40     target=oldest_node;
41     target->buf_ref=1;
42     target->block_num=block_num;
43
44     spinlock_release(&lk_buf_cache);
45     sleeplock_acquire(&(target->slk));
46     virtio_disk_rw(target, 0);
47 }
48
49 assert(target!=NULL, "buf_read: no buf available");
50 return target;
51 }

```

- 首先，从 head\_buf.prev 开始按照 prev 方向遍历。
- 如果找到包含目标块号的缓冲区，则使用
- 如果没找到，暂存最老的未使用缓冲区位置，再从 head\_buf.next 开始沿 next 方向遍历，找到了则同上处理
- 如果还是没找到，将最老的未使用缓冲区替换掉

经过实验（篇幅所限，对实验具体步骤的叙述不在此展开，我们将在答辩的 ppt 中详述之），我们发现，经常出现大量的磁盘读取操作读取的是同一个块，即 IO 操作具有极高的局部性，同时这也意味着，很多 buf 的 refcnt 在 0 和 1 之间来回切换，那么由我们在 Uno-OS 中的实现，显然 head.prev 是最有可能在下一次 IO 操作被访问的块。因此在算法设计中，我们首先沿着 prev 方向遍历而非 next 方向。我们的算法基于 xv6 的算法设计，但是最多从头到尾遍历一次链表，因此相较之下，我们的算法最坏情况优于 xv6<sup>[3]</sup>。

## 6.5 文件系统层

基于 Linux 系统下的 ext2 文件系统，我们设计了一套类似的，稍作简化的文件系统，详细介绍如下

### 6.5.1 superblock 的读取和文件系统初始化

文件系统磁盘可以理解为一个以 block 为读写单位的大数组，在我们的构想中，磁盘的布局是这样的：



图 6-1 磁盘结构

在初始化文件系统时，我们可以使用刚刚完成的 buf 层函数，在 fs\_init() 中读取磁盘里的 super block 填写到内存里的 super\_block\_t 数据结构中，并完成文件系统的初始化。

```

1 void fs_init()
2 {
3     buf_init();
4     inode_init();
5     buf_t* buf = buf_read(SB_BLOCK_NUM);
6     memcpy(&sb, buf->data, sizeof(sb));
7     buf_release(buf);
8     // 检查超级块的magic值是否正确，确保文件系统没有损坏
9     assert(sb.magic == FS_MAGIC, "fs_init: invalid super
    block magic number");

```

```

10     assert(sb.block_size == BLOCK_SIZE, "fs_init: invalid
        super block size");
11 }

```

由于 buf 层的操作涉及睡眠锁, 所以文件系统的初始化建立在进程的基础上, 同时为了避免多进程同时初始化文件系统, 我们只在第一个进程 `fork_return()` 中进行初始化。

```

1 static void fork_return()
2 {
3     proc_t* p = myproc();
4     spinlock_release(&p->lk);
5
6     // 仅在第一个进程初始化文件系统
7     if (p->pid == 1)
8         fs_init();
9     trap_user_return();
10 }

```

### 6.5.2 bitmap 的管理

在文件系统的管理中, 我们使用 **bitmap** 来维护文件系统空间的使用情况, 它使用一块区域的每一个 **bit** 标记某种资源是否被占用。当我们申请一个 **data block** 或 **inode** 时, 对应 **bitmap** 的一个 **bit** 被置为 1; 释放一个 **data block** 或 **inode** 时, 对应 **bitmap** 的对应 **bit** 就被置为 0。

在前面提到的磁盘布局中, 我们用到了两块 **bitmap** (出于简化考虑, 它们各占 1 个 **block**)

其位图和块之间的索引如下:

因为位图使用 **bit** 来记录每个块的分配状态, 所以在管理位图时, 我们必须按比特级别进行操作。

在分配块时, `buf_read` 会将相应的位图块读取到缓冲区中。然后, 函数遍历位图中的每一位, 找到第一个未分配的位置 (即值为 0 的位)。一旦找到空闲位置, 就将该位设置为 1, 表示该块已分配。由于修改了位图, 需要先调用 `buf_write` 将修改后

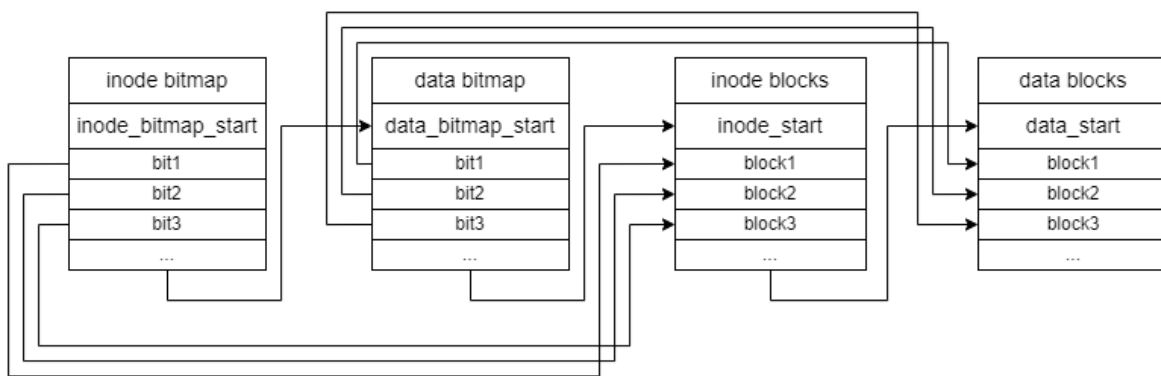


图 6-2 位图索引

的位图写回磁盘，然后再调用 `buf_release` 释放缓冲区。因为我们在遍历时得到的是块的偏移量，所以在返回时还要加上对应块结构的起始地址。

释放块时，操作与分配块类似。`bitmap_unset` 函数通过计算出对应块在位图中的位置，将其值清零，表示该块不再使用。首先，通过传入的块号计算出在位图中的具体位置，并使用按位与操作 (`&= m`) 将该位置对应的位清零。然后，调用 `buf_write` 将更新后的位图写回磁盘，最后调用 `buf_release` 释放缓冲区。

```

1 static uint32 bitmap_search_and_set(uint32 bitmap_block)
2 {
3     buf_t* bp = buf_read(bitmap_block);
4     uint32 m, block_num;
5
6     for (uint32 bi = 0; bi < sb.block_size * 8; ++bi) {
7         m = 1 << (bi % 8);
8
9         if ((bp->data[bi / 8] & m) == 0)
10        {
11            bp->data[bi / 8] |= m;
12
13            buf_write(bp);
14            buf_release(bp);
15
16            if(bitmap_block==sb.inode_bitmap_start)

```

```
17         block_num = bi + sb.inode_start;
18         else if(bitmap_block==sb.data_bitmap_start)
19             block_num = bi + sb.data_start;
20         else
21         {
22             panic("bitmap_search_and_set: invalid bitmap
                block %d\n", bitmap_block);
23             return -1;
24         }
25
26         buf_t *buf = buf_read(block_num);
27         buf_release(buf);
28         return block_num;
29     }
30 }
31
32 buf_release(bp);
33 return -1;
34 }
35
36 static void bitmap_unset(uint32 bitmap_block, uint32 num)
37 {
38     buf_t* buf = buf_read(bitmap_block);
39     num -= bitmap_block;
40     uint8 m = 1 << (num % 8);
41     buf->data[num / 8] &= ~m;
42     buf_write(buf);
43     buf_release(buf);
44 }
```

之后针对不同区域进行封装，就可以更简洁地管理位图的分配和释放。

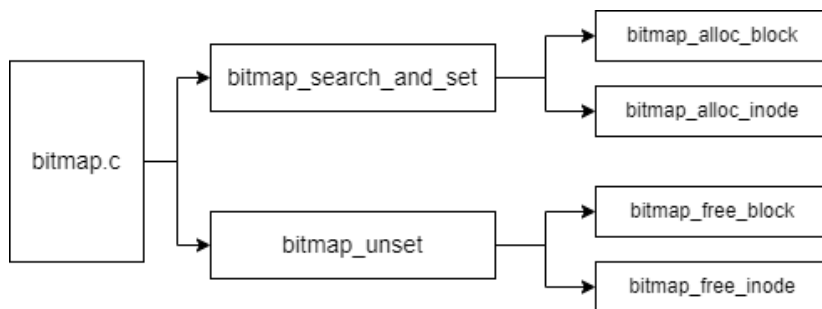


图 6-3 封装

### 6.5.3 inode 结构

磁盘中 inode 的结构由 `disk_inode_t` 定义

```

1 typedef struct inode_disk
2 {
3     short type; // inode 管理的文件类型
4     short major; // 设备文件使用：主设备号
5     short minor; // 设备文件使用：次设备号
6     short nlink; // 链接数量 (nlink个文件名链接到这个inode)
7     unsigned int size; // 文件大小 (字节)
8     unsigned int addrs[N_ADDRS]; // 文件存储在哪些block里 (分
    // 为一级 二级 三级)
9 } inode_disk_t;

```

注意到 `inode_disk_t` 的大小为 64 字节，而磁盘中一个块的大小为 512 字节，这意味着一个块可以存放  $512/64 = 8$  个 inode。

inode 管理文件采取 UNIX 系统中“直接 + 间接”的方法，分有 10 个直接索引，2 个一级间接索引和 1 个二级间接索引条目三部分，各部分的条目数定义于 `/include/dinode.h` 中：

```

1 #define N_ADDRS_1    10 // 管理 10 * BLOCK_SIZE = 10KB
2 #define N_ADDRS_2    2  // 管理 2 * (BLOCK_SIZE / 4) *
    BLOCK_SIZE = 512KB
3 #define N_ADDRS_3    1  // 管理 (BLOCK_SIZE / 4) * (
    BLOCK_SIZE / 4) * BLOCK_SIZE = 64MB
4 #define N_ADDRS      (N_ADDRS_1 + N_ADDRS_2 + N_ADDRS_3)

```



索引机制如图6-4所示

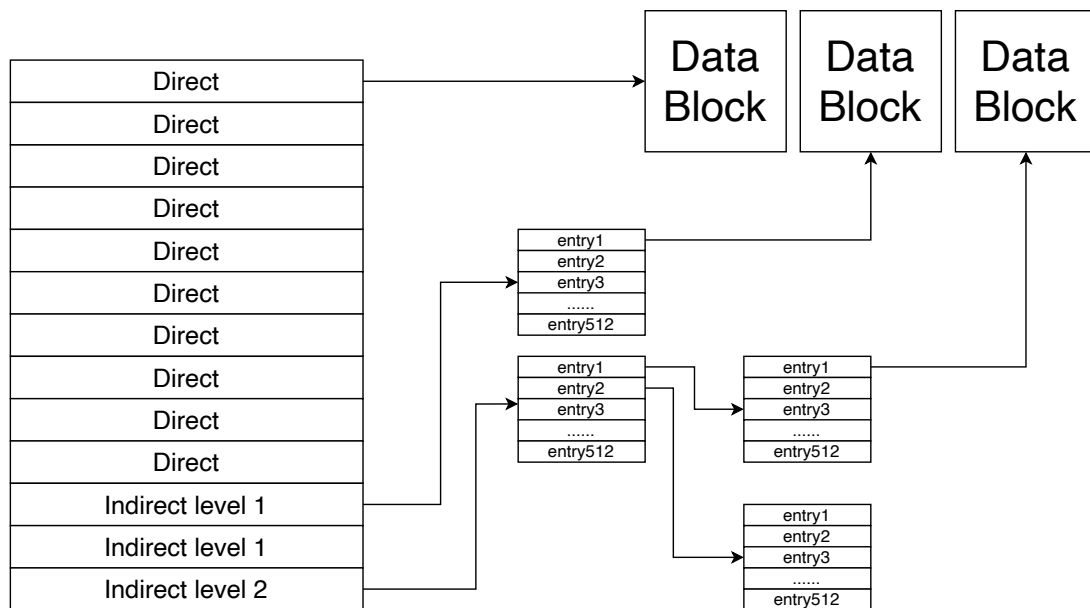


图 6-4 inode 结构示意图

#### 6.5.4 目录和路径管理

在开始实现目录和路径管理之前,我们先来梳理一下目录项 `dirent_t`、内存 `inode_t`、磁盘 `inode_disk_t` 之间的联系。根据这三个结构体的定义我们可以发现它们之间存在着以下联系:

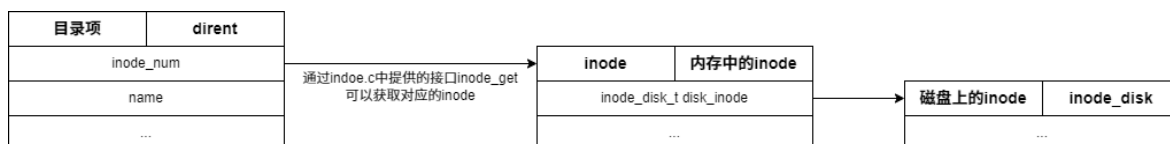


图 6-5 三种结构体之间的关联

在目录和路径的管理中,我们一共实现了这些函数:

- `dir_search_entry`: 查询一个目录项是否在目录里,成功返回这个目录项的 `inode_num`,失败返回 `INODE_NUM_UNUSED`。
- `dir_add_entry`: 在 `pip` 目录下添加一个目录项,成功返回这个目录项的偏移量,同时更新 `pip->size`,失败(没有空间或发生重名)返回 `BLOCK_SIZE`。
- `dir_delete_entry`: 在 `pip` 目录下删除一个目录项,成功返回这个目录项的 `inode_num`,失败返回 `INODE_NUM_UNUSED`。

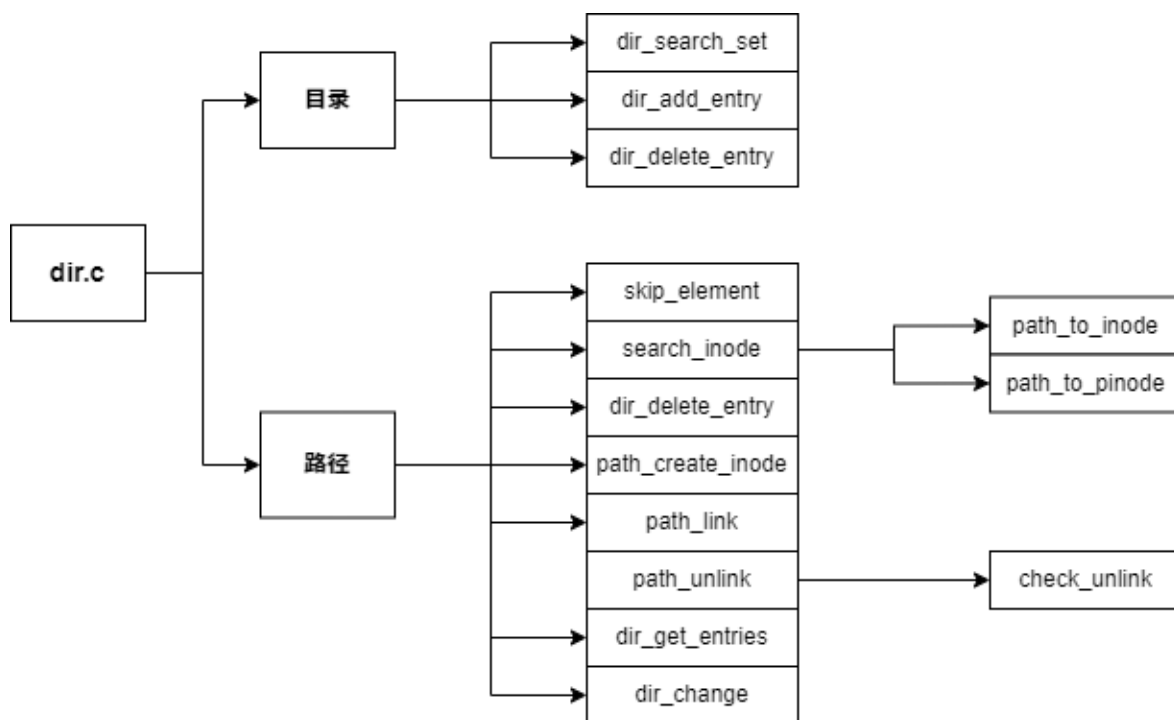


图 6-6 目录和路径管理

- skip\_element: 跳过路径中的一个元素，更新 name 为当前元素的名字，并返回下一个元素的路径。
- search\_inode: 查找路径 path 对应的 inode 或父节点。根据 find\_parent 参数决定是返回目标 inode 还是父目录 inode。返回 NULL 表示查找失败。
- path\_to\_inode: 查找路径 path 对应的 inode，成功返回 inode 指针，失败返回 NULL。
- path\_to\_pinode: 查找路径 path 对应的 inode 的父节点，将路径中的最后一个目录名存入 name，成功返回父目录的 inode 指针，失败返回 NULL。
- path\_create\_inode: 如果 path 对应的 inode 存在则返回 inode。如果 path 对应的 inode 不存在，则创建一个新的 inode。创建失败返回 NULL。
- path\_link: 创建一个硬链接，将旧路径的文件链接到新路径。成功返回 0，失败返回-1。
- path\_unlink: 删除路径 path 对应的文件或目录，成功返回 0，失败返回-1。该操作会检查目录是否为空（对于目录），并且不允许删除“.”和“..”。

- `check_unlink`: 检查一个 `unlink` 操作是否合理，主要用于检查目录是否为空。调用者需持有 `ip` 的锁。
- `dir_get_entries`: 读取目录下的有效目录项，将其复制到 `dst` 中，返回实际读取的字节数。调用者需要持有 `pip` 的锁。
- `dir_change`: 改变当前进程的工作目录。如果路径对应的 `inode` 不存在或不是目录，则返回-1；成功则返回 0。

目录管理我们先以 `dir_search_entry` 为例展开讲述：

之所以用 `pip->disk_inode.addrs[0] == 0` 来检验目录是否分配数据块，是因为在文件系统的设计中，目录的第一个数据块（即 `addrs[0]`）通常用于存储目录项。如果该字段为 0，表示该目录没有分配任何数据块，也就意味着目录是空的或尚未初始化。

接下来读取目录的首个数据块并遍历每个目录项。遍历过程中，跳过空的目录项（`name[0] == 0`），若找到匹配的目录项，释放缓存后返回对应的 `inode` 编号。若遍历结束未找到匹配项，则释放缓存并返回 `INODE_NUM_UNUSED`，表示未找到目标。

```

1 uint16 dir_search_entry(inode_t *pip, char *name)
2 {
3     assert(sleeplock_holding(&(pip->slk)), "dir_search_entry:
4         lock");
5
6     if (pip->disk_inode.addrs[0] == 0)
7         return INODE_NUM_UNUSED;
8
9     buf_t *buf = buf_read(pip->disk_inode.addrs[0]); // 读取
10        目录block
11
12     dirent_t *de;
13
14     for (uint32 offset = 0; offset < BLOCK_SIZE; offset +=
15        sizeof(dirent_t)) {
16
17         de = (dirent_t *) (buf->data + offset);
18
19         if (de->name[0] == 0) // 检测空目录项
20
21             continue;

```

```

14
15     if (strncmp(de->name, name, DIR_NAME_LEN) == 0) {
16         buf_release(buf);
17         return de->inode_num; // 返回匹配条目的inode_num
18     }
19 }
20
21 buf_release(buf);
22 return INODE_NUM_UNUSED; // 未找到
23 }

```

现阶段在整个目录和路径的管理中，一旦我们修改了 inode 中的东西——即，内存中的 inode 与磁盘中的不一样时，我们就要将这些改变写回磁盘，从而保持内存和磁盘的一致性，避免出现未知错误。

篇幅和时间有限，剩余函数的实现先省略。。。

## 6.6 测试

### 6.6.1 用缓冲层读写磁盘块

为了测试方便，我们用一个较小的，6 个 buf 块组成的 buf 链进行测试，每次分配完后打印整个链的状态检查。

测试目的写在注释中：

```

1 #include "sys.h"
2 #include "type.h"
3
4 int main()
5 {
6     char buf[128];
7     uint64 buf_in_kernel[10];
8
9     // 初始状态：读了sb并释放了buf
10    syscall(SYS_print, "\nstate-1:");
11    syscall(SYS_show_buf);

```

```
12
13 // 耗尽所有 buf
14 for(int i = 0; i < 6; i++) {
15     buf_in_kernel[i] = syscall(SYS_read_block, 100 + i,
16                               buf);
17     buf[i] = 0xFF;
18     syscall(SYS_write_block, buf_in_kernel[i], buf);
19 }
20 syscall(SYS_print, "\nstate-2:");
21 syscall(SYS_show_buf);
22
23 // 测试是否会触发buf_read里的panic, 测试完后注释掉(一次性)
24 // buf_in_kernel[0] = syscall(SYS_read_block, 0, buf);
25
26 // 释放两个buf-4 和 buf-1, 查看链的状态
27 syscall(SYS_release_block, buf_in_kernel[3]);
28 syscall(SYS_release_block, buf_in_kernel[0]);
29 syscall(SYS_print, "\nstate-3:");
30 syscall(SYS_show_buf);
31
32 // 申请buf, 测试LRU是否生效 + 测试103号block的lazy write
33 buf_in_kernel[6] = syscall(SYS_read_block, 106, buf);
34 buf_in_kernel[7] = syscall(SYS_read_block, 103, buf);
35 syscall(SYS_print, "\nstate-4:");
36 syscall(SYS_show_buf);
37
38 // 释放所有buf
39 syscall(SYS_release_block, buf_in_kernel[7]);
40 syscall(SYS_release_block, buf_in_kernel[6]);
```

```

41     syscall(SYS_release_block, buf_in_kernel[5]);
42     syscall(SYS_release_block, buf_in_kernel[4]);
43     syscall(SYS_release_block, buf_in_kernel[2]);
44     syscall(SYS_release_block, buf_in_kernel[1]);
45     syscall(SYS_print, "\nstate-5:");
46     syscall(SYS_show_buf);
47
48     while(1);
49     return 0;
50 }

```

测试结果较长，故不在此列出

### 6.6.2 用 inode 层读写磁盘

我们通过大量的读写磁盘来测试 inode 层的鲁棒性，该测试同时对直接索引区，一级索引区和二级索引区的 data block 进行了读写。

```

1     uint32 ret = 0;
2
3     for(int i = 0; i < BLOCK_SIZE; i++) {
4         str[i] = i;
5         empty[i] = 0;
6     }
7
8     // 创建新的inode
9     inode_t* nip = inode_create(FT_FILE, 0, 0);
10    inode_lock(nip);
11
12    // 第一次查看
13    inode_print(nip);
14
15    uint32 max_blocks = N_ADDRS_1 + N_ADDRS_2 *
        ENTRY_PER_BLOCK + 2 * ENTRY_PER_BLOCK;

```

```
16     // uint32 max_blocks =  N_ADDRS_1 + 1;
17
18     for(uint32 i = 0; i < max_blocks; i++)
19     {
20         ret = inode_write_data(nip, i * BLOCK_SIZE,
21                                BLOCK_SIZE, str, false);
22         assert(ret == BLOCK_SIZE, "inode_write_data fail, ret
23                %d", ret);
24     }
25
26     ret = inode_write_data(nip, (max_blocks - 2) * BLOCK_SIZE
27                            , BLOCK_SIZE, empty, false);
28     assert(ret == BLOCK_SIZE, "inode_write_data fail, ret %d"
29            , ret);
30
31     // 第二次查看
32     inode_print(nip);
33     get_free_buf();
34
35     // 区域-1
36     ret = inode_read_data(nip, BLOCK_SIZE, BLOCK_SIZE, tmp,
37                           false);
38     assert(ret == BLOCK_SIZE, "inode_read_data fail");
39     assert(strncmp(tmp, str, BLOCK_SIZE) == 0, "check-1 fail"
40            );
41     printf("check-1 success\n");
42
43     // 区域-2
44     ret = inode_read_data(nip, N_ADDRS_1 * BLOCK_SIZE,
45                           BLOCK_SIZE, tmp, false);
46     assert(ret == BLOCK_SIZE, "inode_read_data fail");
```

```
39     assert(strncmp(tmp, str, BLOCK_SIZE) == 0, "check-2 fail"
        );
40     printf("check-2 success\n");
41
42     // 区域-3
43     ret = inode_read_data(nip, (max_blocks - 2) * BLOCK_SIZE,
        BLOCK_SIZE, tmp, false);
44     assert(ret == BLOCK_SIZE, "inode_read_data fail");
45     assert(strncmp(tmp, empty, BLOCK_SIZE) == 0, "check-2
        fail");
46     printf("check-3 success\n");
47
48     // 释放inode管理的所有data block
49     inode_free_data(nip);
50     printf("free success\n");
51
52     // 第三次观察
53     inode_print(nip);
54
55     inode_unlock_free(nip);
```

测试结果见图6-7



```
inode information:
num = 3, ref = 1, valid = 1
type = INODE_FILE, major = 0, minor = 0, nlink = 0
size = 0, addrs = 0 0 0 0 0 0 0 0 0 0 0 0

inode information:
num = 3, ref = 1, valid = 1
type = INODE_FILE, major = 0, minor = 0, nlink = 0
size = 1058816, addrs = 132 133 134 135 136 137 138 139 140 141 142 399 656

free buf:65check-1 success
check-2 success
check-3 success
free success

inode information:
num = 3, ref = 1, valid = 1
type = INODE_FILE, major = 0, minor = 0, nlink = 0
size = 1058816, addrs = 0 0 0 0 0 0 0 0 0 0 0 0

Jump to initcode.
```

图 6-7 inode 测试结果

## 7、系统调用参考

### 7.1 内存分配

#### 7.1.1 brk

简介：扩展或收缩用户堆大小

参数：

uint64 new\_heap\_top 新的堆顶位置

返回值：成功返回新的堆顶，失败返回-1

示例：

```
1  uint64 new_heap_top = 0x10000;  
2  uint64 heap_top = brk(new_heap_top);  
3  printf("New heap top: %p\n", (void*)heap_top);
```

该段代码调整进程的堆顶地址为 new\_heap\_top。

测试：

```
1  void test_sys_brk() {  
2      // 测试查询堆顶  
3      uint64 current_heap_top = sys_brk(0);  
4      assert(current_heap_top != 0, "sys_brk:test1");  
5  
6      // 测试扩展堆  
7      uint64 new_heap_top = sys_brk(current_heap_top + 4096);  
8      assert(new_heap_top == current_heap_top + 4096, "sys_brk:  
          test2");  
9  
10     // 测试收缩堆  
11     new_heap_top = sys_brk(new_heap_top - 4096);  
12     assert(new_heap_top == current_heap_top, "sys_brk:test3")  
        ;  
13 }
```

### 7.1.2 mmap

简介：分配一个指定的 `mmap` 区域

参数：

`uint64 start` 分配内存区域的起始地址，当 `start` 为 0 时，起始地址由内核决定

`uint32 len` 分配区域的大小

返回值：成功返回映射空间的起始地址，失败返回-1

示例：

```
1  uint32 len = 4096; // 4KB
2  uint64 mmap_addr = mmap(0, len);
3  assert(mmap_addr != -1, "mmap error");
4  printf("Mmap address: %p\n", (void*)mmap_addr);
```

测试：

```
1 void test_sys_mmap() {
2     // 测试内存映射，由内核选择起始地址
3     uint64 start = sys_mmap(0, 4096);
4     assert(start != (uint64)-1, "sys_mmap:test1");
5
6     // 测试内存映射，指定起始地址
7     start = sys_mmap(MMAP_BEGIN+8192, 8192);
8     assert(start == MMAP_BEGIN+8192, "sys_mmap:test2");
9 }
```

### 7.1.3 munmap

简介：释放一个指定的 `mmap` 区域

参数：

`uint64 start`：释放内存区域的起始地址

`uint32 len`：释放区域的大小

返回值：成功返回 0 失败返回-1

示例：

```

1 // 接上一示例
2 assert(munmap(mmap_addr, len) != -1, "munmap failed");
3 printf("Munmap successful\n");

```

测试:

```

1 void test_sys_munmap() {
2     // 测试取消内存映射
3     uint64 start = sys_mmap(0, 4096);
4     assert(start != (uint64)-1, "sys_munmap:test1");
5     assert(sys_munmap(start, 4096) == 0, "sys_munmap:test2");
6
7     assert(sys_munmap(MMAP_BEGIN+8192, 8192) == 0, "
        sys_munmap:test3");
8 }

```

## 7.2 进程管理

### 7.2.1 exec

简介: 从磁盘中加载并执行一个 ELF 文件

参数:

char\* path: 可执行文件路径

char\*\* argv: 提供的参数

返回值: 成功返回提供的参数的个数失败返回-1

示例:

```

1 char* args[] = { "./ls", "-l", NULL };
2 assert(exec("./ls", args) != -1, "exec failed");

```

该段代码执行了 ./ls -l 命令

测试:

```

1 int main()
2 {
3     char path[] = "./test";
4     char* argv[] = {"hello", "world", 0};

```

```

5
6
7     int pid = syscall(SYS_fork);
8     if(pid < 0) { // 失败
9         syscall(SYS_write, 0, 20, "initcode: fork fail\n");
10    } else if(pid == 0) { // 子进程
11        syscall(SYS_write, 0, 22, "\n-----test start-----\n");
12        ;
13        syscall(SYS_exec, path, argv);
14    } else { // 父进程
15        int exit_state;
16        syscall(SYS_wait, &exit_state);
17        syscall(SYS_write, 0, 21, "\n-----test over-----\n");
18        while(1);
19    }
20    while(1);
21    return 0;
22 }

```

执行的 **test** 程序代码如下:

```

1 #include "userlib.h"
2
3 int main(int argc, char* argv[])
4 {
5     printf("\nchild arguments: ");
6     for(int i = 0; i < argc; i++)
7         printf("%s ", argv[i]);
8     printf("\n");
9     return 0;
10 }

```

执行结果见图7-1

```
-----test start-----  
  
child arguments: hello world  
  
-----test over-----
```

图 7-1 测试 exec 传参能力

### 7.2.2 fork

简介：完全复制当前进程的状态（除了 pid）

参数：无

返回值：成功调用后，对于父进程，返回子进程的 pid，对于子进程返回 0

示例：

```
1  int pid = fork();  
2  assert(pid != -1, "fork failed");  
3  if (pid == 0) {  
4      // 子进程  
5      printf("Child process\n");  
6      exit(0);  
7  } else if (pid > 0) {  
8      // 父进程  
9      uint64 exit_addr;  
10     int status = wait(&exit_addr);  
11     assert(status != -1, "wait failed");  
12     printf("Child exited with status: %d\n", status);  
13 }
```

我们可以通过返回值判断当前是父进程还是子进程

测试：同 exec，该函数为实现创建子进程的基础

### 7.2.3 wait

简介：进程进入休眠态，等待任一子进程的退出后将其唤醒

参数：uint64 addr：子进程退出时，退出状态码存于此处

返回值：成功返回退出的子进程的 pid，失败返回-1

示例：

```

1  int pid = fork();
2  assert(pid != -1, "fork failed");
3  if (pid == 0) {
4      // 子进程
5      printf("Child process\n");
6      exit(0);
7  } else if (pid > 0) {
8      // 父进程
9      uint64 exit_addr;
10     int status = wait(&exit_addr);
11     assert(status != -1, "wait failed");
12     printf("Child exited with status: %d\n", status);
13 }

```

该示例同 `fork`，父进程会等待子进程的结束

测试：同 `exec`，该函数为实现创建子进程的基础

#### 7.2.4 `exit`

简介：当前进程退出，如果有子进程，则将子进程交由 `proczero` 管理

参数：

`uint32 exit_state` 退出状态码

返回值：`exit_state`

示例：

```

1  int pid = fork();
2  assert(pid != -1, "fork failed");
3  if (pid == 0) {
4      // 子进程
5      printf("Child process\n");
6      exit(0);
7  } else if (pid > 0) {
8      // 父进程
9      uint64 exit_addr;

```

```

10     int status = wait(&exit_addr);
11     assert(status != -1, "wait failed");
12     printf("Child exited with status: %d\n", status);
13 }

```

该示例同 `fork`，子进程调用 `wait(0)` 退出，将返回的错误码（0）传递给父进程

测试：同 `exec`，该函数为实现创建子进程的基础

### 7.2.5 sleep

简介：令当前进程睡眠指定的系统时钟刻（10 个系统时钟刻约为 1 秒）

参数：

uint32 second 睡眠时间

返回值：成功返回 0，失败返回-1

示例：

```

1     uint32 sleep_time = 5; // 睡眠5秒
2     assert(sleep(sleep_time) != -1, "sleep failed");
3     printf("Slept for %d seconds\n", sleep_time);

```

这段代码令进程睡眠 5 秒

测试：

```

1 void test_sleep()
2 {
3     uint64 start=sys_time();
4     sys_sleep(5);
5     uint64 end=sys_time();
6     assert(end-start>=5, "test_sleep:test1");
7 }

```

### 7.2.6 pid

简介：获取当前进程的 pid

参数：无

返回值：返回当前进程的 pid

示例：



```
1 int pid=sys_pid();
```

测试:

```
1 void test_pid()
2 {
3     // 因为proczero的唯一任务就是加载当前进程，因此我们可以通过pid=2来判断是否成功
4     assert(sys_pid()==2, "test_pid:test1");
5 }
```

### 7.2.7 ppid

简介：获取父进程的 pid

参数：无

返回值：如果无父进程，返回-1 否则返回父进程的 pid

示例：

```
1 int ppid=sys_ppid();
```

测试:

```
1 void test_ppid()
2 {
3     assert(sys_ppid() == 1, "test_ppid:test1");
4 }
```

## 7.3 文件读写

### 7.3.1 open

简介：打开或创建文件

参数：

char \*path: 文件路径

uint32 open\_mode: 打开模式, 见表7-1

返回值：成功返回文件描述符（file descriptor）失败返回-1

示例：

```
1 char path[DIR_PATH_LEN] = "/path/to/file";
```

参数名称	值	意义
MODE_CREATE	1	创建指定的文件
MODE_READ	2	读取指定的文件
MODE_WRITE	4	写入指定的文件

表 7-1 打开文件模式

```

2  uint32 open_mode = MODE_CREATE; // 创建文件
3
4  uint64 fd = open(path, open_mode);
5  assert(fd != -1, "Failed to open file");
6
7  printf("File descriptor: %lu\n", fd);

```

创建了一个路径为`/path/to/file`的文件

测试:

```

1  void test_sys_open() {
2      // 测试创建一个新文件
3      int fd = sys_open("/newfile.txt", MODE_WRITE |
4          MODE_CREATE);
5      assert(fd >= 0, "sys_open:test1");
6      sys_close(fd);
7
8      // 测试打开一个存在的文件
9      fd = sys_open("/newfile.txt", MODE_READ);
10     assert(fd >= 0, "sys_open:test2");
11     sys_close(fd);
12 }

```

### 7.3.2 close

简介: 关闭指定的文件

参数:

int fd: 指定文件的文件描述符

返回值: 成功返回 0 失败返回-1

示例:

```
1  uint64 result = close(fd);
2  assert(result == 0, "Failed to close file");
3
4  printf("File closed successfully\n");
```

接上例, 关闭指定的文件

测试:

```
1 void test_sys_close() {
2     // 测试关闭一个有效的文件描述符
3     int fd = sys_open("/newfile.txt", MODE_READ);
4     assert(fd >= 0, "sys_close:test1");
5     assert(sys_close(fd) == 0, "sys_close:test2");
6 }
```

### 7.3.3 read

简介: 对于一个给定的文件, 读取不超过指定的长度个字节到数组中

参数:

int fd: 指定文件的文件描述符

uint32 len: 读取的最大长度

uint64 addr: 写入内存中的地址

返回值: 成功返回读取的字节数, 失败返回 0

示例:

```
1  uint64 bytes_read = read(fd, len, addr);
2  assert(bytes_read != 0, "Failed to read from file");
```

其中 fd 是一个打开的, 具有读权限的文件描述符。

测试:

```
1 void test_sys_read() {
2     // 测试从文件中读取数据
```

```

3   int fd = sys_open("/testfile.txt", MODE_READ);
4   sys_lseek(fd, 0, LSEEK_SET);
5   assert(fd >= 0, "sys_read:test1");
6   char buffer[100];
7   int bytes_read = sys_read(fd, 100, (void *)buffer);
8   assert(bytes_read > 0, "sys_read:test2");
9   sys_close(fd);
10 }

```

### 7.3.4 write

简介：对于一个给定的文件，写入不超过指定的长度个字节到文件中

参数：

int fd: 指定文件的文件描述符

uint32 len: 写入的最大长度

uint64 addr: 写入文件中的内存源地址

返回值：成功返回写入的字节数，失败返回 0

示例：

```

1   uint64 bytes_written = write(fd, len, addr);
2   assert(bytes_written != 0, "Failed to write to file");

```

其中 fd 是一个打开的，具有写权限的文件描述符。

测试：

```

1 void test_sys_write() {
2     // 测试向文件写入数据
3     int fd = sys_open("/testfile.txt", MODE_WRITE |
4         MODE_CREATE);
5     assert(fd >= 0, "sys_write:test1");
6     const char* data = "Hello, world!!!!!!!!!!!!!!";
7     int bytes_written = 0;
8     for(int i=0;i<1000;++i)
9         bytes_written += sys_write(fd, strlen(data), (void *)

```

参数名称	值	意义
LSEEK_SET	0	设置偏移量为指定值
LSEEK_ADD	1	偏移量增加指定值
LSEEK_SUB	2	偏移量减少指定值

表 7-2 lseek 标识

```

        data);
9    assert(bytes_written == strlen(data)*1000, "sys_write:
        test2");
10   sys_close(fd);
11 }

```

### 7.3.5 lseek

简介：设置文件偏移量。

参数：

int fd：文件描述符

uint32 offset：偏移量

int flags：标志，意义见表7-2

返回值：成功返回新的偏移量失败返回-1

示例：

```

1    uint32 offset = 1024; // 假设的偏移量
2    int flags = LSEEK_SET;
3
4    uint64 new_offset = lseek(fd, offset, flags);
5    assert(new_offset != -1, "Failed to set file offset");

```

将文件的偏移量设为 1024

测试：

```

1 void test_sys_lseek() {
2     // 测试设置文件偏移量 (LSEEK_SET)
3     int fd = sys_open("/testfile.txt", MODE_READ|MODE_WRITE);

```

```

4   assert(fd >= 0, "test_sys_lseek:open1");
5   int offset = sys_lseek(fd, 10, LSEEK_SET);
6   assert(offset == 10, "test_sys_lseek:lseek1");
7
8   // 测试添加到当前偏移量 (LSEEK_ADD)
9   offset = sys_lseek(fd, 5, LSEEK_ADD);
10  assert(offset == 15, "test_sys_lseek:lseek2");
11
12  // 测试从当前偏移量减去 (LSEEK_SUB)
13  offset = sys_lseek(fd, 5, LSEEK_SUB);
14  assert(offset == 10, "test_sys_lseek:lseek3");
15
16  sys_close(fd);
17 }

```

### 7.3.6 dup

简介：复制文件描述符。

参数：

int fd: 原始文件描述符

返回值：成功：新的文件描述符 (new\_fd) 失败：-1

示例：

```

1   uint64 new_fd = dup(fd);
2   assert(new_fd != -1, "Failed to duplicate file descriptor
   ");

```

测试：

```

1 void test_sys_dup() {
2     // 测试复制文件描述符
3     int fd = sys_open("/testfile.txt", MODE_READ);
4     assert(fd >= 0, "sys_dup:test1");
5     int new_fd = sys_dup(fd);

```

```

6     assert(new_fd >= 0, "sys_dup:test2");
7     sys_close(fd);
8     sys_close(new_fd);
9 }

```

### 7.3.7 fstat

简介：获取文件信息。

参数：

int fd: 文件描述符

uint64 addr: 文件信息存储地址

返回值：成功返回 0 失败返回 -1

示例：

```

1     uint64 result = fstat(fd, addr);
2     assert(result == 0, "Failed to get file status");

```

测试：

```

1 void test_sys_fstat() {
2     // 测试获取文件信息
3     int fd = sys_open("/testfile.txt", MODE_READ);
4     assert(fd >= 0, "sys_fstat:test1");
5     fstat_t st;
6     int result = sys_fstat(fd, &st);
7     assert(result == 0, "sys_fstat:test2");
8     sys_close(fd);
9 }

```

### 7.3.8 getdir

简介：获取目录中的目录项。

参数：

int fd: 文件描述符

uint64 addr: 目录项存储地址

uint32 len: 请求读取的字节数

返回值：成功返回实际读取的字节数失败返回-1

示例：

```
1  uint64 bytes_read = getdir(fd, addr, len);
2  assert(bytes_read != -1, "Failed to get directory entries
    ");
```

测试：

```
1 void test_sys_getdir() {
2     // 测试读取目录项
3     int fd = sys_open("/", MODE_READ);
4     assert(fd >= 0, "sys_getdir:test1");
5     dirent_t dirent[10];
6     int bytes_read = sys_getdir(fd, dirent, sizeof(dirent));
7     assert(bytes_read > 0, "sys_getdir:test2");
8     sys_close(fd);
9 }
```

### 7.3.9 mkdir

简介：根据指定的目录路径创建目录

参数：char\* path：目录路径

返回值：成功返回 0 失败返回-1

示例：

```
1  const char* path = "/path/to/newdir";
2
3  uint64 result = mkdir(path);
4  assert(result == 0, "Failed to create directory");
```

测试：

```
1 void test_sys_mkdir() {
2     // 测试创建目录
3     int result = sys_mkdir("/newdir");
4     assert(result == 0, "sys_mkdir:test1");
```



```

5
6
7     result=sys_mkdir("/newdir/subdir");
8     assert(result==0,"sys_mkdir:test2");
9
10    result=sys_mkdir("/newdir/subdir/subsubdir");
11    assert(result==0,"sys_mkdir:test3");
12 }

```

### 7.3.10 chdir

简介：修改当前工作目录。

参数：

char\* path: 新的工作目录路径

返回值：成功返回 0 失败返回-1

示例：

```

1     const char* path = "/path/to/newdir";
2
3     uint64 result = chdir(path);
4     assert(result == 0, "Failed to change directory");

```

测试：

```

1     void test_sys_chdir() {
2         // 测试切换到存在的目录
3         int result = sys_chdir("/newdir");
4         assert(result == 0,"sys_chdir:test1");
5     }

```

### 7.3.11 link

简介：创建文件链接。

参数：

char\* old\_path: 原始文件路径

char\* new\_path: 新的文件路径

返回值：成功返回 0 失败返回 -1

示例：

```
1  const char* old_path = "/path/to/oldfile";
2  const char* new_path = "/path/to/newfile";
3
4  uint64 result = link(old_path, new_path);
5  assert(result == 0, "Failed to create link");
```

测试：

```
1 void test_sys_link() {
2     // 测试创建文件链接
3     int result = sys_link("/testfile.txt", "/testfile_link.
      txt");
4     assert(result == 0, "sys_link:test1");
5 }
```

### 7.3.12 unlink

简介：删除文件链接，如果链接数为 0 则删除文件。

参数：

char\* path: 文件路径

返回值：成功返回 0 失败返回 -1

示例：

```
1  const char* path = "/path/to/file";
2
3  uint64 result = unlink(path);
4  assert(result == 0, "Failed to unlink file");
```

测试：

```
1 void test_sys_unlink() {
2     // 测试删除文件链接
3     int result = sys_unlink("/testfile_link.txt");
4     assert(result == 0, "sys_unlink:test1");
```

```
sys_open:test1 passed
sys_open:test2 passed
sys_close:test1 passed
sys_close:test2 passed
sys_close:test3 passed
sys_write:test1 passed
sys_write:test2 passed
sys_read:test1 passed
sys_read:test2 passed
test_sys_lseek:open1 passed
test_sys_lseek:lseek1 passed
test_sys_lseek:lseek2 passed
test_sys_lseek:lseek3 passed
sys_dup:test1 passed
sys_dup:test2 passed
sys_mkdir:test1 passed
sys_mkdir:test2 passed
sys_mkdir:test3 passed
sys_getdir:test1 passed
sys_getdir:test2 passed
sys_link:test1 passed
sys_unlink:test1 passed
```

图 7-2 文件相关系统调用测试结果

5 }

我们将先前的测试合并，测试结果见图7-2

## 参考文献

- [1] COMMITTEE P H. The RISC-V Instruction Set Manual Volume II: Privileged Architecture [M/OL]. 2024. [https://drive.google.com/file/d/17GeetSnT5wW3xNuAHI95-SI1gPGd5sJ\\_/view](https://drive.google.com/file/d/17GeetSnT5wW3xNuAHI95-SI1gPGd5sJ_/view).
- [2] Et AL. M S T. Virtual I/O Device (VIRTIO) Version 1.1[M/OL]. 2019. <https://docs.oasis-open.org/virtio/virtio/v1.1/virtio-v1.1.pdf>.
- [3] RUSS COX R M, Frans Kaashoek. xv6: a simple, Unix-like teaching operating system[M/OL]. 2021. <https://pdos.csail.mit.edu/6.828/2021/xv6/book-riscv-rev2.pdf>.