



華東師範大學

East China Normal University

## 基于 xv6 的 aabcb 内核系统

## aabcb kernel system based on xv6

队伍名称: aabcb

队伍编号: T202410269994328

参赛队员: 李嘉烨

学 院: 数据科学与工程学院

指导教师: 石亮

职 称: 教授

2024 年 12 月

## 目录

摘要 .....	I
ABSTRACT.....	II
<b>1、 aabcb 内核简介 .....</b>	<b>1</b>
1.1 简介 .....	1
1.2 运行环境与依赖 .....	1
1.3 代码架构 .....	1
<b>2、 Boot 模块 .....</b>	<b>3</b>
2.1 功能介绍 .....	3
2.2 实现分析 .....	3
2.3 测试 .....	4
<b>3、 中断处理模块 .....</b>	<b>6</b>
3.1 功能介绍 .....	6
3.2 中断处理的过程 .....	6
3.3 时钟中断 .....	7
3.4 测试 .....	10
<b>4、 内存管理模块 .....</b>	<b>13</b>
4.1 功能介绍 .....	13
4.2 物理内存 .....	13
4.3 虚拟内存 .....	14
4.4 测试 .....	16
<b>5、 进程调度模块 .....</b>	<b>20</b>
5.1 功能介绍 .....	20
5.2 PCB 块.....	20
5.3 初始进程 .....	21
5.4 调度算法 .....	22
5.5 测试 .....	23
<b>6、 文件系统模块 .....</b>	<b>26</b>
6.1 功能介绍 .....	26
6.2 磁盘布局 .....	26
6.3 Inode 管理.....	27
6.4 目录项管理.....	28
6.5 文件路径处理 .....	28
6.6 测试 .....	29
<b>参考文献 .....</b>	<b>32</b>

## 基于 xv6 的 aabcb 内核系统

### 摘要:

本文基于 xv6 框架设计并实现了一套完整的内核实验系统，命名为 aabcb 内核系统。通过九个递进版本，系统实现了从机器启动到文件系统整合的全流程内核功能开发。具体而言，Version-1 完成了机器启动与初始化；Version-2 实现了内存管理的初步设计；Version-3 建立了中断和异常处理机制；Version-4 实现了第一个用户态进程的运行；Version-5 构建了用户态虚拟内存管理和系统调用流程；Version-6 开发了进程管理模块；Version-7 和 Version-8 实现了文件系统的底层设备支持与上层抽象；Version-9 最终完善文件系统功能并完成系统整合。实验过程中，aabcb 系统重点解决了内存管理、进程调度、系统调用和文件系统等核心模块的设计与实现问题。实验结果表明，该系统具备良好的稳定性和高效性，为深入理解操作系统原理和开发流程提供了有力支持。

**关键词：**xv6, c, kernel, 虚拟内存管理

## **aabcb kernel system based on xv6**

### **Abstract:**

This paper presents the design and implementation of the aabcb kernel system, developed based on the xv6 framework. Through nine incremental versions, the system achieves a complete kernel development workflow, from machine bootstrapping to file system integration. Specifically, Version-1 focuses on machine bootstrapping and initialization; Version-2 implements basic memory management; Version-3 establishes interrupt and exception handling; Version-4 achieves the execution of the first user-mode process; Version-5 builds user-mode virtual memory management and system call mechanisms; Version-6 develops process management modules; Version-7 and Version-8 implement the file system's underlying device support and upper-level abstractions; and Version-9 completes the file system functionality and system integration. The aabcb kernel addresses key challenges in memory management, process scheduling, system calls, and file system design. Experimental results demonstrate the system's stability and efficiency, providing strong support for understanding operating system principles and development processes.

**Keywords:** xv6, C Programming, kernel, Virtual Memory Management

## 1、aabcb 内核简介

### 1.1 简介

aabcb 是一个基于 **xv6** 架构开发的实验性操作系统，使用 **C 语言** 实现，运行于 **RISC-V** 架构的模拟环境。系统参考了 xv6 的简洁设计，进行了部分扩展，以支持文件系统的路径解析和基本的目录操作。

### 1.2 运行环境与依赖

- **操作系统与工具链**：建议在 **Linux** 环境（如 **Ubuntu 20.04.2**）下运行，需安装以下工具：
  - **GNU Make**：用于编译。
  - **Git**：便于版本管理。
- **RISC-V 工具链**：需安装 `riscv64-unknown-elf-gcc` 等工具，用于将代码编译为 RISC-V 指令并进行调试。
- 使用 **QEMU** 模拟 RISC-V 虚拟硬件平台，运行编译后的内核镜像。可通过以下命令启动：

```
make qemu
```

需要注意的是，在运行 `make qemu` 之前，需要先对全体代码进行 `make` 操作，这会根据 `Makefile` 中给定的规则编译内核源码，并构建 xv6 的整个运行环境之后才可以启动内核。

### 1.3 代码架构



图 1-1 项目文件结构

## 2、Boot 模块

本章详细介绍 aabcb 系统在启动时的关键过程，依次分析从 ‘entry.S’ 到 ‘start.c’ 再到 ‘main.c’ 的执行流，帮助理解系统初始化的核心逻辑。

boot 模块内容位于 /kernel/boot 中，主要任务是实现系统的加载以及初始化。简单来说就是实现了系统的启动工作。

### 2.1 功能介绍

Boot 模块负责完成系统的早期启动过程，包括以下功能：

- 初始化处理器运行环境（如堆栈指针、分页机制）。
- 加载和解析操作系统镜像，切换到内核执行上下文。
- 启动多核系统并为每个内核分配执行任务。

### 2.2 实现分析

#### 2.2.1 ‘entry.S’：系统的入口点

‘entry.S’ 是系统启动的第一步，它完成了以下工作：

1. 设置各处理器的运行环境，包括初始化内核栈指针。
2. 切换至内核模式，配置必要的寄存器。
3. 跳转到 ‘start.c’ 的入口函数以继续初始化流程。

关键代码片段：

```
.globl _start
_start:
    la sp, boot_stack    # 设置内核栈指针
    csrw mscratch, zero  # 清空寄存器
    call start            # 跳转到 start.c
```

图 2-1 文件 entry.S 中的核心代码

#### 2.2.2 ‘start.c’：早期初始化阶段

‘start.c’ 负责进一步初始化内存管理单元（MMU）并为内核代码准备运行环境：

1. 初始化分页机制，设置页表映射。

2. 加载多核信息，并启动除主核外的所有核。
3. 跳转到 ‘main.c’ 的主内核入口点。

关键代码片段：

```
void start()
{
    // 配置 S 模式并准备寄存器状态
    unsigned long x = r_mstatus();
    x = (x & ~MSTATUS_MPP_MASK) | MSTATUS_MPP_S; // 设置 MPP 位为 S 模式
    w_mstatus(x);

    // 设置返回地址为 main, 禁用分页
    w_mepc((uint64)main); // 设置返回地址
    w_satp(0);           // 关闭分页机制

    // 委托中断和异常处理给 S 模式
    w_medeleg(0xffff);    // 委托机器模式异常处理
    w_mideleg(0xffff);    // 委托机器模式中断处理

    // 启用 SEIE、STIE、SSIE 中断
    w_sie(r_sie() | SIE_SEIE | SIE_STIE | SIE_SSIE);

    // 初始化时钟
    timer_init();

    // 设置 hart ID
    w_tp(r_mhartid());

    // 切换到 S 模式并跳转到 main
    asm volatile("mret");
}
```

图 2-2 文件 start.c 中的核心代码

## 2.3 测试

为了验证 Boot 模块的正确性，我们设计了以下测试场景并进行验证。

### 2.3.1 测试场景与代码

1. 核实系统在单核和多核环境下是否正确进入 main.c。

测试代码如图 2-3 所示：

### 2.3.2 测试结果

测试运行的效果如图 2-4 所示：



```
#include "riscv.h"
#include "lib/print.h"
#include "proc/cpu.h"
#include "dev/uart.h"

volatile static int started = 0;

int main()
{
    // 通过定义在cpu.h中的mycpuid(), 初始化系统。
    if(mycpuid() == 0){
        printf("cpu 0 is booting!\n");
    }else{
        printf("cpu 1 is booting!\n");
    }

    while (1); // 保证系统不会掉线
}
```

图 2-3 文件 main.c 的测试代码，用于验证是否能够进入 main.c

```
make[1]: Leaving directory '/home/zzr/quantum_mind_os/kernel'
qemu-system-riscv64 -machine virt -bios none -kernel kernel-qemu -m 128M -smp 2 -nographic
CPU 0 is booting
CPU 1 is booting
```

图 2-4 运行 make qemu 之后的效果图

### 2.3.3 结论

测试结果表明，aabcb 的启动模块在单核和多核环境下均表现稳定，能够正确完成初始化流程并顺利进入 main.c。

### 3、中断处理模块

#### 3.1 功能介绍

中断处理模块是操作系统核心功能之一，负责在中断发生时进行有效的响应和处理。本模块的功能包括：

- 处理来自硬件和软件的中断。
- 在不同特权模式（M 模式和 S 模式）下完成中断委托和处理。
- 在用户态（U 模式）和内核态（S 模式）之间进行安全上下文切换。

#### 3.2 中断处理的过程

##### 3.2.1 M 模式下的中断处理

在 RISC-V 架构中，M 模式是最高特权模式。中断处理的主要流程如下：

1. 在中断发生时，跳转到 ‘trap.S’ 中定义的中断入口。
2. 保存当前的处理器状态到栈中。
3. 检查中断类型并委托非 M 模式中断至 S 模式。

关键代码摘录如下：

```
.globl m_trap_vector
m_trap_vector:
    addi sp, sp, -32      # 保存寄存器
    sd ra, 0(sp)
    sd t0, 8(sp)
    csrr t0, mcause      # 获取中断原因
    ...
    csrr t0, mtval
    sd t0, 24(sp)
    call handle_m_interrupt # 调用中断处理函数
```

图 3-1 文件 trap.S 的核心代码

##### 3.2.2 S 模式下的中断处理

S 模式主要处理用户态程序的中断。其流程如下：

1. 从 M 模式跳转到 trampoline.S，完成中断的上下文切换。

2. 执行 `trap_kernel.c` 或 `trap_user.c` 中定义的具体中断处理逻辑。
3. 根据中断类型执行相应的中断服务例程（ISR）。

S 模式中断处理的关键代码摘录如下：

```
.globl s_trap_vector
s_trap_vector:
    csrr t0, scause          # 获取中断原因
    ...
    call handle_s_interrupt # 调用 S 模式中断处理逻辑
```

图 3-2 trap.S 中处理 S 中断的关键代码

在 S 模式中，内核对用户态程序的中断进行处理和管理的例子代码：

```
void handle_s_interrupt() {
    uint64 cause = r_scause();
    if (cause == SCAUSE_TIMER) {
        timer_handler();
    } else if (cause == SCAUSE_EXTERNAL) {
        external_device_handler();
    }
}
```

图 3-3 S 模式中内核态对用户态程序中断处理的样例代码

### 3.3 时钟中断

#### 3.3.1 原理

时钟中断的核心原理是基于物理时钟的 `MTIME` 和 `MTIMECMP` 寄存器。其工作原理如下：

- 物理时钟每个滴答增加 `MTIME` 寄存器的值。
- `MTIMECMP` 寄存器存储一个预定值，与 `MTIME` 寄存器进行比较。
- 当 `MTIME` 的值等于 `MTIMECMP` 时，触发一次时钟中断。
- 中断触发后，更新 `MTIMECMP` 为 `MTIMECMP += INTERVAL`，以确保下次时钟中断按间隔触发。

通过设置适当的 INTERVAL 值，可以将物理滴答划分为逻辑时间单元 tick，使得  $1 \text{ tick} = \text{INTERVAL}$  个滴答。

### 3.3.2 初始化过程

时钟中断初始化主要分为两部分：M 模式下的初始化和 S 模式下的初始化。

#### 3.3.2.1 M 模式初始化

在 M 模式下，timer\_init() 函数的主要任务是：

1. 设置  $\text{MTIMECMP} = \text{MTIME} + \text{INTERVAL}$ ，确保第一次时钟中断触发。
2. 配置 M 模式的中断入口地址，存储在 mtvec 寄存器中。
3. 配合 trap.S 中的 timer\_vector 函数处理时钟中断。

```
void timer_init() {
    int id = r_mhartid(); // 获取当前 CPU 的 ID

    // 设置 mscratch 的初始值，存储 CLINT_MTIMECMP 地址和 INTERVAL
    mscratch[id][3] = (uint64)CLINT_MTIMECMP(id);
    mscratch[id][4] = INTERVAL;

    // 将 mscratch 数组地址写入 mscratch 寄存器
    w_mscratch((uint64)mscratch[id]);

    // 设置 mtvec 寄存器为 timer_vector 地址
    w_mtvec((uint64)timer_vector);

    // 设置初始的 CLINT_MTIMECMP 值，使中断在 INTERVAL 后触发
    *(uint64*)CLINT_MTIMECMP(id) = r_time() + INTERVAL;

    // 启用 M-mode 的全局中断
    w_mstatus(r_mstatus() | MSTATUS_MIE);

    // 启用 M-mode 的定时器中断
    w_mie(r_mie() | MIE_MTIE);
}
```

图 3-4 timer\_init 函数主要初始化内容

#### 3.3.2.2 S 模式初始化

在 S 模式下，trap\_kernel\_inithart() 完成如下工作：

1. 将 S 模式中断入口地址存储到 stvec 寄存器中。
2. 启用 S 模式中断。

```
void trap_kernel_inithart() {
    w_stvec((uint64)kernel_vector); // 设置 S 模式中断入口
    intr_on(); // 启用 S 模式中断
    ...
}
```

图 3-5 trap\_kernel\_inithart 函数中关于 S 模式的部分

### 3.3.3 处理函数

时钟中断的处理逻辑位于 timer\_interrupt\_handler() 函数中，分为以下步骤：

1. 清除 sip 寄存器中的 SSIP 位，确认中断已被处理。
2. 如果是主核（CPU 0），更新系统时间滴答数并打印日志。

关键代码如下：

```
void timer_interrupt_handler() {
    w_sip(r_sip() & ~2); // 清除 sip 中的 SSIP 位
    if (mycpuid() == 0) {
        timer_update(); // 更新系统时间滴答数
        printf("ticks = %d\n", timer_get_ticks());
    }
    printf("cpu %d: dida\n", r_tp());
}
```

图 3-6 timer\_interrupt\_handler() 部分代码，其中包含了测试文件

### 3.3.4 返回用户态

在 trap\_kernel\_handler() 函数中，完成从内核态返回用户态的步骤：

1. 保存当前的寄存器状态，包括 sepc 和 sstatus。
2. 检查 scause 判断中断类型，调用相应的中断处理函数。
3. 恢复 sepc 和 sstatus，并返回用户态。

关键代码如下：

```

void trap_kernel_handler() {
    uint64 sepc = r_sepc();
    uint64 sstatus = r_sstatus();
    uint64 scause = r_scause();
    uint64 stval = r_stval();

    int trap_id = scause & 0xFF;
    bool is_interrupt = (scause & (1ULL << 63)) != 0;

    ...

    if (is_interrupt) {
        switch (trap_id) {
            case 5: // S-mode timer interrupt
                timer_interrupt_handler();
                break;
            default:
                printf("Unexpected interrupt: %d\n", trap_id);
        }
    }
    w_sepc(sepc);
    w_sstatus(sstatus);
}

```

图 3-7 timer\_kernel\_handler() 部分代码，保留了由内核态返回用户态的过程

## 3.4 测试

### 3.4.1 时钟中断的 Dida 测试

可以通过在时钟处理函数中插入打印语句，判断时钟中断是否实现，以及判断函数是否正确调用了时钟中断。

```

// 时钟中断处理 (基于CLINT)
void timer_interrupt_handler()
{
    w_sip(r_sip() & ~2); // 通过清除sip中的SSIP位确认软件中断

    if(mycpuid() == 0)
    {
        timer_update(); // 增加滴答数
    }
    printf("cpu %d: dida\n", r_tp());
}

```

图 3-8 在时钟处理函数中插入打印语句

运行结果表明，系统能够不间断地产出时钟中断：

```

:pu 1: dida
:pu 0: dida
:pu 1: dida
:pu 0: dida
:pu 1: dida
:pu 0: dida
:pu 1: dida
:pu 0: dida
:pu 1: dida
:pu 0: dida
:pu 1: dida
:pu 0: dida
:pu 1: dida
:pu 0: dida
:pu 1: dida
:pu 0: dida
:pu 1: dida
:pu 0: dida
:pu 1: dida
:pu 0: dida
:pu 1: dida
:pu 0: dida

```

图 3-9 系统不间断产出时钟中断的运行结果

### 3.4.2 时钟中断的 Tick 测试

在时钟处理函数中存在 `timer_update` 函数，能够不断更新系统时间。通过在函数中增加一行输出系统时间 `tick` 的打印语句，可以测试系统时间是否正常更新以及时间流逝的速度。

```

// 时钟中断处理 (基于CLINT)
void timer_interrupt_handler()
{
    w_sip(r_sip() & ~2); // 通过清除sip中的SSIP位确认软件中断

    if(mycpuid() == 0)
    {
        timer_update(); // 增加滴答数
        printf("ticks = %d\n", timer_get_ticks());
    }
}

```

图 3-10 在时钟处理函数中插入打印语句测试系统时间更新

运行结果显示，系统时间能够持续更新，测试效果如下：

```
emu-system-riscv64 -machine virt -bios none -kernel kernel-qemu -m 1G  
-nographic  
pu 0 is booting!  
icks = 1  
pu 1 is booting!  
icks = 2  
icks = 3  
icks = 4  
icks = 5  
icks = 6  
icks = 7  
icks = 8  
icks = 9  
icks = 10
```

图 3-11 系统时间不断更新的运行结果



## 4、内存管理模块

### 4.1 功能介绍

内存管理模块是操作系统的核心模块之一，负责管理物理内存和虚拟内存。主要功能包括：

- 初始化物理内存，管理内存分配与回收。
- 维护内核页表 and 用户页表，实现虚拟内存管理。
- 支持内存映射（mmap）机制，为用户程序提供灵活的内存使用方式。

### 4.2 物理内存

#### 4.2.1 物理内存布局

物理内存管理是操作系统内核的关键组件，其核心任务是有效地分配和回收物理内存资源，确保内存的高效利用和系统的稳定运行。以下是 aabcb 系统物理内存模块的设计思路：

- **内核代码区**：用于存放内核代码和常驻数据。该区域通常在系统启动时加载，并不会被动态分配。
- **设备映射区**：为硬件设备分配固定的物理内存区域，用于直接访问设备寄存器或内存。
- **空闲内存区**：剩余的物理内存被视为空闲区域，用于动态分配给用户进程或其他内核数据结构。

通过这种分区设计，系统可以确保内核和设备的关键数据得到保护，同时最大化利用空闲内存。

#### 4.2.2 物理内存初始化

物理内存的初始化包括以下步骤：物理内存初始化通过文件 `pmem.c` 完成。关键函数如下：

```
void pmem_init() {
    for (uint64 addr = MEM_START; addr < MEM_END; addr += PAGE_SIZE) {
        free_list_add(addr); // 添加空闲内存页
    }
}
```

图 4-1 该函数遍历所有空闲物理页，并将其加入空闲列表供分配使用。

通过这种初始化方式，系统能够在启动后快速找到可用的内存页面，并确保内存分配的高效性。

#### 4.2.3 处理内存分配与回收

物理内存的分配和回收是通过链表结构实现的：

- **分配**：从空闲链表中移除一个节点并返回其地址。
- **回收**：将物理页重新加入链表，供后续使用。

通过 `pmem.c` 提供的 `alloc_page` 和 `free_page` 函数，这种链表结构的优点在于简单且高效，适合嵌入式或实验性操作系统的设计。以下是关键代码：

```
void* alloc_page() {
    if (free_list == NULL) return NULL;
    void* page = free_list;
    free_list = free_list->next;
    return page;
}

void free_page(void* addr) {
    free_list_add(addr);
}
```

图 4-2 处理分配与回收的核心逻辑

#### 4.2.4 设计思路的关键问题与优化方向

物理内存管理模块是操作系统内核的基础部分，设计合理的物理内存布局 and 分配策略是系统稳定运行的关键。在 aabcb 系统中，我们使用简单高效的链表管理内存，同时为后续扩展优化提供了可能性。

### 4.3 虚拟内存

#### 4.3.1 内核页表布局

虚拟内存是操作系统管理内存的高级机制，它为每个进程提供了一个连续的虚拟地址空间，使得程序可以专注于逻辑地址的使用而无需考虑底层的物理地址分布。

- **地址隔离**：确保每个进程的地址空间互不干扰，提高系统的安全性。
- **灵活映射**：允许动态分配和回收虚拟内存，支持大规模应用程序的运行。
- **高效管理**：提供页表缓存机制以加速地址转换。

aabcb 系统通过 RISC-V 的分页机制实现了虚拟内存管理。

内核页表布局定义在 `kvm.c` 中，主要包括：

- **内核代码段**：映射内核代码的虚拟地址和物理地址。
- **设备映射段**：将设备的物理地址映射到虚拟地址。

关键代码片段如下：

```
void setup_kernel_page_table() {  
    ...  
    map_pages(KERNEL_VMA, KERNEL_PMA, KERNEL_SIZE, PTE_R | PTE_X);  
    map_pages(DEVICE_VMA, DEVICE_PMA, DEVICE_SIZE, PTE_R | PTE_W);  
    ...  
}
```

图 4-3 关于内核页表布局的部分代码

#### 4.3.2 用户页表布局

用户页表的设计目标是为每个进程提供独立的虚拟地址空间，并支持动态扩展。用户页表包括以下区域：

- **代码段**：存放可执行指令。
- **数据段**：存放全局变量和静态变量。
- **堆区**：用于动态内存分配。
- **栈区**：为函数调用提供临时存储。
- **内核映射**：只读映射部分内核地址，以支持系统调用。

核心代码如下：

```
void setup_user_page_table(pagetable_t pagetable, proc_t* proc) {  
    ...  
    map_pages(pagetable, USER_CODE_VMA, proc->code_pa, proc->code_size, PTE_R | PTE_X);  
    map_pages(pagetable, USER_DATA_VMA, proc->data_pa, proc->data_size, PTE_R | PTE_W);  
    map_pages(pagetable, USER_STACK_VMA, proc->stack_pa, proc->stack_size, PTE_R | PTE_W);  
    ...  
}
```

图 4-4 关于用户页表布局的部分代码

### 4.3.3 虚拟内存的关键实现模块

地址转换通过多级页表实现，每级页表负责管理一部分虚拟地址空间。系统通过以下函数完成地址的映射和查找：

- **map**：将虚拟地址映射到物理地址。
- **unmap**：解除虚拟地址与物理地址的映射。
- **walk**：递归查找虚拟地址对应的物理页。

## 4.4 测试

内存管理模块的正确性测试旨在验证以下功能是否能够正确实现：

- **初始化内核页表**：设置基本的虚拟内存布局。
- **页表映射测试**：将虚拟地址 TEST\_VA 映射到物理地址 TEST\_PA。
- **页表打印验证**：调用 print\_page\_table 函数，递归输出内核页表结构。
- **映射关系清理**：解除映射并释放内存，确保无残留。

### 4.4.1 测试代码

测试代码实现了上述测试逻辑，具体代码如下：

```
#include "riscv.h"
#include "lib/print.h"
#include "lib/str.h"
#include "mem/pmem.h"
#include "mem/vmem.h"
#include "mem/mmap.h"
#include "proc/proc.h"
#include "trap/trap.h"

volatile static int started = 0;

#define TEST_VA 0x400000 // 测试虚拟地址
#define TEST_PA 0x800000 // 测试物理地址
```

```

#define PAGE_SIZE 4096      // 页大小

// 打印页表内容
void print_page_table(pagetable_t pagetable, int level) {
    for (int i = 0; i < 512; i++) {
        pte_t pte = pagetable[i];
        if (pte & PTE_V) { // 判断页表项是否有效
            uint64 pa = PTE2PA(pte);
            printf("%*slevel-%d pgtbl %d: pa = 0x%016lx flags = %lx\n",
                level * 4, "", level, i, pa, pte & 0x3FF); // 格式化输出
            if ((pte & PTE_RWX) == 0) { // 如果是非叶子节点，递归打印子页表
                print_page_table((pagetable_t)pa, level + 1);
            }
        }
    }
}

int main()
{
    int cpuid = r_tp();

    if (cpuid == 0) {
        // 主核初始化
        print_init();
        printf("cpu %d is booting!\n", cpuid);
        pmem_init();
        kvm_init();
        kvm_inithart();
        trap_kernel_init();
        trap_kernel_inithart();
        mmap_init();
    }
}

```

```
proc_make_first();

// 测试虚拟内存页表功能
test_virtual_memory();

__sync_synchronize();
started = 1;

} else {
    // 其他核等待主核初始化完成
    while (started == 0);
    __sync_synchronize();

    printf("cpu %d is booting!\n", cpuid);
    kvm_inithart();
    trap_kernel_inithart();
}

while (1);
}
```

#### 4.4.2 测试结果

测试运行的结果如图 4-5 所示，成功验证了页表的递归结构和映射关系：

```

look: heap_top = 0x0000000000002000
level-2 pgtbl: pa = 0x0000000080055000
.. level-1 pgtbl 0: pa = 0x0000000080058000
.. .. level-0 pgtbl 0: pa = 0x0000000080055000
.. .. .. physical page 1: pa = 0x000000008040b000 flags = 95
.. level-1 pgtbl 255: pa = 0x0000000080056000
.. .. level-0 pgtbl 511: pa = 0x0000000080055000
.. .. .. physical page 509: pa = 0x000000008040a000 flags = 215
.. .. .. physical page 510: pa = 0x0000000080054000 flags = 199
.. .. .. physical page 511: pa = 0x0000000080003000 flags = 75

grow: heap_top = 0x000000000000c000
level-2 pgtbl: pa = 0x0000000080055000
.. level-1 pgtbl 0: pa = 0x0000000080058000
.. .. level-0 pgtbl 0: pa = 0x0000000080055000
.. .. .. physical page 1: pa = 0x000000008040b000 flags = 95
.. .. .. physical page 2: pa = 0x000000008040c000 flags = 23
.. .. .. physical page 3: pa = 0x000000008040d000 flags = 23
.. .. .. physical page 4: pa = 0x000000008040e000 flags = 23
.. .. .. physical page 5: pa = 0x000000008040f000 flags = 23
.. .. .. physical page 6: pa = 0x0000000080410000 flags = 23
.. .. .. physical page 7: pa = 0x0000000080411000 flags = 23
.. .. .. physical page 8: pa = 0x0000000080412000 flags = 23

```

图 4-5 内存管理模块页表测试结果

#### 4.4.3 结论

通过本次测试，确认以下功能正常实现：

- **内核页表初始化：**页表初始化无异常，虚拟内存布局正确设置。
- **页表递归打印：**打印函数能够准确输出多级页表的映射关系。
- **映射关系清理：**映射关系解除后，内存回收功能无异常。

测试结果表明，aabcb 系统的内存管理模块功能稳定，能够正确完成内存分配、页表映射和内存回收任务，为用户态和内核态程序提供了可靠的内存管理支持。

## 5、进程调度模块

### 5.1 功能介绍

进程调度模块是操作系统的核心组成部分之一，用于管理多个进程在多核处理器上的运行。它通过高效的调度算法协调进程的运行，使得每个进程都能在合理的时间内获得 CPU 的执行权。这一个部分的文件在 `/kernel/proc` 目录下，主要功能包括：

- **进程管理**：负责进程的创建、状态转换（如就绪、运行、阻塞等）、销毁等生命周期管理。
- **上下文切换**：在进程切换时保存当前进程的 CPU 状态，并加载下一个进程的状态。
- **调度决策**：根据调度算法选择合适的进程运行，保证 CPU 使用效率。
- **支持多核环境**：在多核环境下均匀分布进程，充分利用系统资源。

进程调度的目标是在保证系统稳定性的前提下，实现进程的公平竞争、高效利用 CPU 资源，以及实时响应关键任务。

### 5.2 PCB 块

**进程控制块 (Process Control Block, PCB)** 是操作系统描述和管理进程的核心数据结构。PCB 是调度器对进程操作的主要依据，其主要内容如下：

- **进程标识符 (PID)**：每个进程都有一个唯一的标识符，用于区分不同的进程。
- **进程状态**：标识进程当前所处的状态，包括运行中 (Running)、就绪 (Runnable)、阻塞 (Blocked) 等。
- **上下文信息**：保存进程的寄存器值、程序计数器 (PC) 和栈指针 (SP) 等信息，以便在上下文切换时能恢复运行状态。
- **内存管理信息**：记录进程的页表地址，用于实现进程的虚拟内存管理。
- **进程调度信息**：包括优先级、时间片剩余时间等信息，用于调度算法的决策。

以下是 PCB 数据结构的代码定义，由 `/kernel/proc/proc.h` 定义：



```
// 进程定义
typedef struct proc {

    spinlock_t lk;                // 自旋锁

    /* 下面的六个字段需要持有锁才能修改 */

    int pid;                      // 标识符
    enum proc_state state;        // 进程状态
    struct proc* parent;          // 父进程
    int exit_state;               // 进程退出时的状态(父进程可能关心)
    void* sleep_space;           // 睡眠是为在等待什么

    pgtbl_t pgtbl;               // 用户态页表
    uint64 heap_top;             // 用户堆顶(以字节为单位)
    uint64 ustack_pages;         // 用户栈占用的页面数量
    mmap_region_t* mmap;         // 用户可映射区域的起始节点
    trapframe_t* tf;            // 用户态内核态切换时的运行环境暂存空间

    uint64 kstack;               // 内核栈的虚拟地址
    context_t ctx;               // 内核态进程上下文

    inode_t* cwd;                // 当前目录
    file_t* filelist[FILE_PER_PROC]; // 打开文件列表
} proc_t;
```

### 5.3 初始进程

当操作系统启动后，必须创建一个初始进程（通常称为 `init` 进程），它是所有其他用户进程的祖先。创建初始进程的步骤包括：

1. 调用 `proc_init` 函数初始化所有的 **PCB** 数据结构。
2. 分配初始进程的 **PCB** 并设置其状态为 `RUNNABLE`。

3. 分配进程的页表和内存空间，加载必要的初始化程序。
4. 设置初始进程的上下文信息，并将其加入调度器的就绪队列。

代码示例如下：

```
void proc_make_first() {
    struct proc *p = alloc_proc(); // 分配 PCB
    p->pid = 1;                     // 设置初始进程的 PID 为 1
    strncpy(p->name, "init", sizeof(p->name)); // 进程名称
    p->state = RUNNABLE;            // 设置为就绪状态
    printf("Initial process created with PID = %d\n", p->pid);
}
```

初始进程的创建是操作系统启动的重要步骤，它为后续的用户进程提供了一个基础运行环境。

## 5.4 调度算法

aabcb 系统采用简单的时间片轮转调度算法，该算法实现了进程的公平调度，同时保证了调度的高效性。其主要步骤如下：

1. 从就绪队列中选择下一个进程，基于先进先出的原则。
2. 通过上下文切换保存当前进程的 CPU 状态，并加载选定进程的状态。
3. 如果当前进程的时间片用尽，将其重新放入队尾并选择下一个进程。
4. 在多核系统中，调度器为每个核分配一个独立的就绪队列，以减少竞争。

以下是调度器的核心代码：

```
void scheduler() {
    struct proc *p;
    while (1) {
        for (p = proc_table; p < &proc_table[MAX_PROC]; p++) {
            if (p->state == RUNNABLE) {
                p->state = RUNNING; // 设置为运行中
                swtch(&scheduler_context, &p->context); // 上下文切换
            }
        }
    }
}
```

```

        p->state = RUNNABLE; // 切换回来后恢复就绪状态
    }

}

}
}

```

## 5.5 测试

为了验证进程管理模块的正确性，设计了如下测试场景，主要包括多进程管理、内存分配区域测试以及父子进程之间的协作。测试的核心目标如下：

- 验证进程的创建和销毁功能是否正确。
- 测试内存映射区域（MMAP）和堆（HEAP）区域的正确性。
- 验证父子进程之间的通信及同步是否正常。

### 5.5.1 测试逻辑

测试代码的核心逻辑如下：

1. 首先输出测试开始的标志性信息：user begin。
2. 在内存映射（MMAP）区域分配一页空间，并在其中存储字符串 MMAP。
3. 在堆（HEAP）区域通过系统调用扩展内存空间，并在其中存储字符串 HEAP。
4. 调用系统调用 `SYS_fork` 创建一个子进程：
  - 子进程打印出预先存储在 MMAP 和 HEAP 区域的字符串。
  - 子进程调用 `SYS_exit` 正常退出，并返回退出状态码。
5. 父进程通过 `SYS_wait` 系统调用等待子进程退出，并根据退出状态码打印结果。

以下是测试代码：

```

#include "sys.h"

#define VA_MAX      (1ul << 38)
#define PGSIZE      4096

```

```

#define MMAP_END      (VA_MAX - 34 * PGSIZE)
#define MMAP_BEGIN    (MMAP_END - 8096 * PGSIZE)

char *str1, *str2;

int main()
{
    syscall(SYS_print, "\nuser begin\n");

    str1 = (char*)syscall(SYS_mmap, MMAP_BEGIN, PGSIZE);
    long long top = syscall(SYS_brk, 0);
    str2 = (char*)top;
    syscall(SYS_brk, top + PGSIZE);

    str1[0] = 'M'; str1[1] = 'M'; str1[2] = 'A'; str1[3] = 'P'; str1[4] = '\n';
    str2[0] = 'H'; str2[1] = 'E'; str2[2] = 'A'; str2[3] = 'P'; str2[4] = '\n';

    int pid = syscall(SYS_fork);

    if(pid == 0) {
        for(int i = 0; i < 100000000; i++);
        syscall(SYS_print, "child: hello\n");
        syscall(SYS_print, str1);
        syscall(SYS_print, str2);
        syscall(SYS_exit, 1);
        syscall(SYS_print, "child: never back\n");
    } else {
        int exit_state;
        syscall(SYS_wait, &exit_state);
        if(exit_state == 1)
            syscall(SYS_print, "parent: hello\n");
    }
}

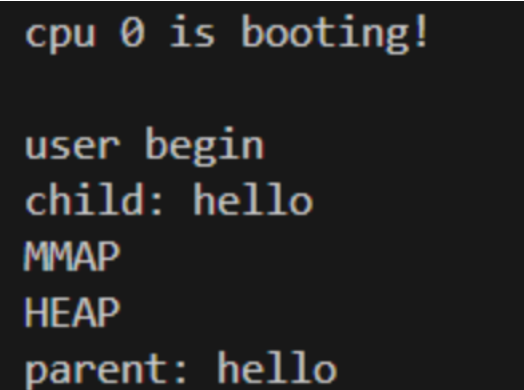
```

```
        else
            syscall(SYS_print, "parent: error\n");
    }

    while(1);
    return 0;
}
```

### 5.5.2 测试结果

测试在内核调试环境下运行，观察到的输出如下：



```
cpu 0 is booting!
user begin
child: hello
MMAP
HEAP
parent: hello
```

图 5-1 进程管理模块的测试结果

测试结果分析：

- 在 user begin 输出后，系统成功分配 MMAP 和 HEAP 区域，并正确存储测试字符串。
- 父进程调用 SYS\_fork 成功创建子进程，子进程正确读取并打印出 MMAP 和 HEAP 中的内容。
- 子进程调用 SYS\_exit 返回状态码 1，父进程通过 SYS\_wait 成功接收状态码并根据状态码输出正确的结果。
- 测试结果表明，aabcb 系统能够正确地实现多进程的创建、运行、同步与销毁，且内存管理模块（包括 MMAP 和 HEAP 区域）功能正常。

## 6、文件系统模块

### 6.1 功能介绍

文件系统模块是操作系统的重要组成部分，承担着管理磁盘存储、支持文件与目录操作的关键任务。其主要功能包括：

- **磁盘管理**：抽象和管理磁盘块，包括块的分配与回收，保证存储资源的高效利用。
- **Inode 管理**：负责文件和目录的元数据存储，支持文件大小、权限、数据块分布等信息的管理。
- **目录项管理**：实现目录的操作，包括目录项的添加、删除和查询，为文件系统提供路径解析的基础。
- **文件操作支持**：提供对文件的基本操作，包括文件的创建、读取、写入、链接和删除。

通过这些功能，文件系统模块为用户程序提供了一个简单易用的接口，同时隐藏了底层复杂的磁盘操作逻辑，确保数据的持久化和访问效率。

### 6.2 磁盘布局

文件系统的磁盘布局定义了磁盘空间的逻辑分区方式，决定了文件系统的效率和稳定性。在 aabcb 系统中，磁盘布局设计如下：

- **Super Block**：存储磁盘和文件系统的全局元数据，包括磁盘大小、块大小、inode 数量等关键信息。它是文件系统的核心元数据。
- **Inode Bitmap**：记录 inode 的分配情况，采用位图的方式，每个 bit 表示一个 inode 是否已分配。
- **Inode Blocks**：存储所有文件和目录的元数据信息，每个 inode 对应一个文件或目录。
- **Data Bitmap**：记录数据块的分配情况，与 inode bitmap 类似，每个 bit 表示一个数据块的使用状态。
- **Data Blocks**：存储文件和目录的数据内容，所有的文件数据和目录项信息均存放在这里。

磁盘布局的逻辑表示如下：

```
[disk layout: [super block | inode bitmap |
inode blocks | data bitmap | data blocks]]
```

这种设计将文件系统的元数据和实际数据分离，既便于管理又提高了访问效率。

## 6.3 Inode 管理

### 6.3.1 Inode 的结构与管理原理

**Inode**（索引节点）是文件系统元数据管理的核心，用于描述文件或目录的属性。每个文件或目录都由一个 **inode** 表示，其结构如下：

```
typedef struct inode {
    uint16 type;           // 文件类型（普通文件、目录、设备文件等）
    uint16 major;          // 主设备号（针对设备文件）
    uint16 minor;          // 次设备号
    uint16 nlink;          // 链接数量，表示文件名到此 inode 的引用数量
    uint32 size;           // 文件大小（字节数）
    uint32 addrs[N_ADDRS]; // 数据块的地址数组
    uint16 inode_num;      // inode 的编号
    uint32 ref;            // 引用计数，用于管理 inode 的生命周期
    bool valid;            // inode 是否有效，标记是否从磁盘读取
    sleeplock_t slk;       // 睡眠锁，用于保护 inode 数据的一致性
} inode_t;
```

#### 设计原理：

1. **磁盘与内存分离**：Inode 结构中同时包含磁盘上的元数据（如类型、大小）和内存中的管理信息（如引用计数、有效性标记）。这种设计既支持快速访问，也能保证数据的一致性。

2. **层次化管理**：通过 `addrs` 数组存储文件数据块的地址。对于小文件，直接存储数据块地址；对于大文件，通过索引块实现多级地址映射，支持大文件的存储。

3. **锁机制保障并发安全**：通过 `sleeplock_t` 锁，确保多个进程同时访问 inode 时的数据一致性。

### 6.3.2 Inode 的核心函数实现

为了管理 inode，文件系统模块实现了一组核心函数：

- `inode_alloc()`：分配新的 inode，更新 inode bitmap 并返回 inode 对象。
- `inode_lock()` 和 `inode_unlock()`：对 inode 加锁和解锁，确保并发安全。
- `inode_read_data()` 和 `inode_write_data()`：实现文件数据的读取和写入，支持多级地址映射。

## 6.4 目录项管理

### 6.4.1 目录项结构

目录是文件系统的一种特殊文件，其作用是将文件名映射到对应的 inode。目录项的结构定义如下：

```
typedef struct dirent {
    uint16 inode_num;      // 目录项对应的 inode 编号
    char name[DIR_NAME_LEN]; // 目录项的名称
} dirent_t;
```

每个目录由一组目录项组成。目录项通过 inode 引用实际文件或子目录。

### 6.4.2 核心管理函数

为了实现目录项的操作，文件系统提供了以下函数：

- `dir_add_entry()`：向目录中添加一个新的目录项，更新目录文件的数据。
- `dir_delete_entry()`：从目录中删除指定的目录项。
- `dir_search_entry()`：根据文件名查找目录项，返回对应的 inode 编号。

这些函数利用文件系统的 inode 和数据块管理功能，实现了目录的基本操作。

## 6.5 文件路径处理

文件路径是用户访问文件和目录的主要方式。文件系统通过以下机制支持路径解析：

- **路径解析函数：**
  - `path_to_inode()`：解析路径并返回目标文件的 inode。



- `path_to_pinode()`: 解析路径并返回目标文件的父目录 `inode`, 同时返回文件名。
- **路径解析逻辑**: 路径解析的核心是逐级处理路径中的目录名, 通过递归或迭代调用 `dir_search_entry()`, 逐步找到目标 `inode`。

路径解析实现了用户视角的文件系统访问方式, 是文件系统的重要接口之一。

## 6.6 测试

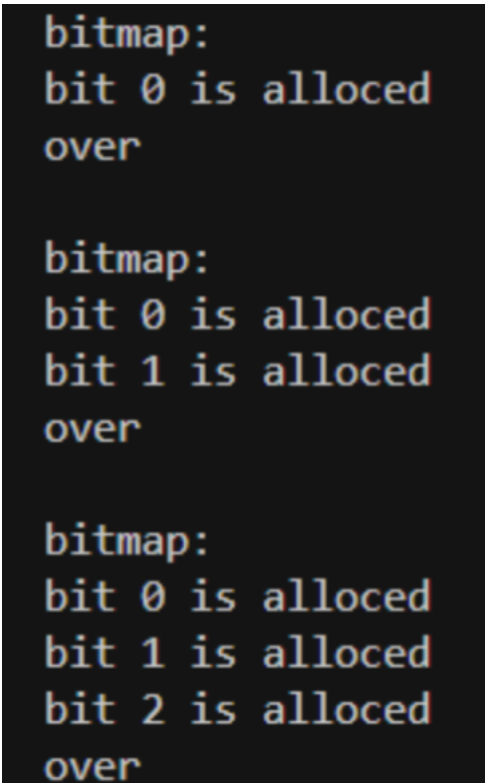
测试的核心目标是验证文件系统模块的正确性。以下是部分测试及其结果。

### 6.6.1 测试 bitmap 的正确性

测试代码:

```
int main() {
    uint32 block_num_1 = syscall(SYS_alloc_block);
    uint32 block_num_2 = syscall(SYS_alloc_block);
    uint32 block_num_3 = syscall(SYS_alloc_block);
    syscall(SYS_free_block, block_num_2);
    syscall(SYS_free_block, block_num_1);
    syscall(SYS_free_block, block_num_3);
    while (1);
    return 0;
}
```

测试结果显示 `bitmap` 的分配与回收逻辑正常, 具体效果如图



```
bitmap:
bit 0 is allocated
over

bitmap:
bit 0 is allocated
bit 1 is allocated
over

bitmap:
bit 0 is allocated
bit 1 is allocated
bit 2 is allocated
over
```

图 6-1 bitmap 的测试结果

### 6.6.2 测试 buf 层功能

测试代码:

```
int main() {
    char buf[128];
    uint64 buf_in_kernel[10];

    syscall(SYS_print, "\nstate-1:");
    syscall(SYS_show_buf);

    for (int i = 0; i < 6; i++) {
        buf_in_kernel[i] = syscall(SYS_read_block, 100 + i, buf);
        buf[i] = 0xFF;
        syscall(SYS_write_block, buf_in_kernel[i], buf);
    }

    syscall(SYS_print, "\nstate-2:");
    syscall(SYS_show_buf);
}
```

```

while (1);

return 0;

}

```

测试结果表明 buf 层功能正常，具体效果如图

```

state-1:
buf_cache:
buf 1: ref = 0, block_num = -1
0 0 0 0 0 0 0 0
buf 2: ref = 0, block_num = -1
0 0 0 0 0 0 0 0
buf 3: ref = 0, block_num = -1
0 0 0 0 0 0 0 0
buf 4: ref = 0, block_num = -1
0 0 0 0 0 0 0 0
buf 5: ref = 0, block_num = -1
0 0 0 0 0 0 0 0
buf 0: ref = 0, block_num = 0
120 86 52 18 0 4 0 0

```

图 6-2 buf 层的测试结果 1

```

state-2:
buf_cache:
buf 0: ref = 1, block_num = 105
0 0 0 0 0 255 0 0
buf 5: ref = 1, block_num = 104
0 0 0 0 255 0 0 0
buf 4: ref = 1, block_num = 103
0 0 0 255 0 0 0 0
buf 3: ref = 1, block_num = 102
0 0 255 0 0 0 0 0
buf 2: ref = 1, block_num = 101
0 255 0 0 0 0 0 0
buf 1: ref = 1, block_num = 100
255 0 0 0 0 0 0 0

```

图 6-3 buf 层的测试结果 2

### 6.6.3 测试目录项与路径解析

测试代码：

```

// in fs_init()

inode_init();

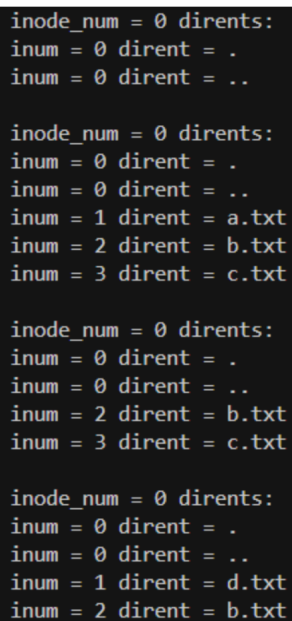
inode_t* ip = inode_alloc(INODE_ROOT);

inode_lock(ip);

```

```
dir_print(ip);
dir_add_entry(ip, 1, "a.txt");
dir_add_entry(ip, 2, "b.txt");
dir_add_entry(ip, 3, "c.txt");
dir_print(ip);
assert(dir_search_entry(ip, "b.txt") == 2, "error-1");
dir_delete_entry(ip, "a.txt");
dir_print(ip);
dir_add_entry(ip, 1, "d.txt");
dir_print(ip);
assert(dir_add_entry(ip, 4, "d.txt") == BLOCK_SIZE, "error-
2");
inode_unlock(ip);
printf("over");
while (1);
```

测试结果验证了目录项管理功能，输出如图



```
inode_num = 0 dirents:
inum = 0 dirent = .
inum = 0 dirent = ..

inode_num = 0 dirents:
inum = 0 dirent = .
inum = 0 dirent = ..
inum = 1 dirent = a.txt
inum = 2 dirent = b.txt
inum = 3 dirent = c.txt

inode_num = 0 dirents:
inum = 0 dirent = .
inum = 0 dirent = ..
inum = 2 dirent = b.txt
inum = 3 dirent = c.txt

inode_num = 0 dirents:
inum = 0 dirent = .
inum = 0 dirent = ..
inum = 1 dirent = d.txt
inum = 2 dirent = b.txt
```

图 6-4 目录项管理的测试结果

## 参考文献

- [1] Xia Haodong. ECNU OSLab: 操作系统实验代码与资料 [EB/OL]. Gitee, [2024-12-25]. Available: <https://gitee.com/HaoDong-Xia/ecnu-oslab>.
- [2] MIT PDOS. xv6-public: Unix-like teaching operating system [EB/OL]. GitHub, [2024-12-25]. Available: <https://github.com/mit-pdos/xv6-public>.