南开大学
Nankai University

南　　开　　大　　学

ShouChun OS **设计文档**

队伍名称：寿春队

队伍成员：林坤　邹先予　郑博月

指导老师：宫晓利　张金

2022 年 8 月 20 日

目录

一、 概述

ShouChun OS 是面向 2022 **全国大学生计算机系统能力大赛操作系统设计赛**(https://os.educg.net/2022CSCC) 的比赛要求进行设计与实现的，该系统为使用 C **语言**进行实现、基于 RISCV 的**多核操作系统**，可运行在 QEMU（RISCV64 System）模拟器以及 HiFive Unmatched 开发板上。

在比赛之前，小队成员系统学习分析了清华大学的 uCore 教学实验，并进行了操作系统相关知识的学习。结合比赛要求与团队能力情况，我们决定将清华大学陶天骅所开发的 uCore-SMP (https://github.com/TianhuaTao/uCore-SMP)源代码作为开发基础，移植到 HiFive Unmatched 开发板上进行完善与改进。

(一) 项目背景及意义

操作系统作为计算机系统的管家，为用户程序提供了一个简单清晰的计算机模型。由于操作系统所管理的计算机系统较为复杂，但又对于理解计算机运行原理来说至关重要，所以在本科教学过程中，操作系统是一个困难而重要的学科。

南开大学十分重视操作系统知识的教授，故十分鼓励学生进行课程知识学习之外的拓展与实践。本科学习阶段注重于基本理论知识的学习，由于课时限制的原因，操作系统课程停留在单核操作系统阶段，学生初步了形成一个较为系统清晰的知识体系。然而，现代计算机硬件技术发展飞速，多核以及异构架构的硬件日新月异，操作系统作为软件运行中重要的一环也需要不断更新拓展。

因此，理论知识也应当不断更新，与时俱进，课程之外的学习实践十分重要。**以赛促学**，于是项目小组决定在参加操作系统大赛的过程中不断学习，拓展理论知识，并通过编程实践提高动手能力和计算机系统设计能力，锻炼自身解决复杂工程问题的能力。

(二) 项目工作概述

1. 开发框架

该项目的开发均基于 uCore-SMP（**清华大学，陶天骅**）。uCore-SMP 大量引用了 MIT 的 xv6(https://github.com/mit-pdos/xv6-riscv)，但是同时也存在许多不同之处。相比于 xv6，uCore-SMP 具有不同的系统调用、不同的内存布局、不同的代码风格等等，此外 uCore-SMP 使用了一个 SBI 来做硬件的抽象化。经过学习研究后，项目组将 uCore-SMP 与比赛要求进行了比较，发现**需要实现更多的系统调用，并且需要调整文件系统、进程管理、内存管理等功能的实现**。

uCore-SMP 是基于 riscv64 而实现的操作系统，支持多处理器工作，符合比赛方所要求的开发板条件。同时 uCore-SMP 主要使用了 C 语言进行开发，更易上手。**于是项目小组选择了** uCore-SMP **作为基础进行开发**，从而将其应用于不同于原 K210 开发板的 HiFive Unmatched 开发板上，从而满足比赛所需求的功能。

2. 实现功能

本项目主要是面对操作系统大赛的要求进行开发的，比赛要求实现**需要实现的功能**如下：

- 启动和系统初始化
- 内存管理
- 进程管理和中断异常机制

- 系统调用
- FAT32 文件系统
- 命令解释程序

3. **开发环境**

项目开发过程中使用的硬件模拟器为 QEMU。本项目选择的硬件实验设备为 HiFive Unmatched 开发板，其主要参数如下：

（1）CPU: 64 位 SiFive FU740 SoC，集成四个 1.5GHz U74-MC 内核 + 一个 S7 嵌入式内核

（2）内存：16GB DDR4 RAM

（3）板载 32MB SPI 闪存芯片，提供了 4×USB 3.2 Gen 1 端口、一个 PCI Express x16 插槽（x8 速率）、一个 NVME M.2 插槽、microSD 读卡器、以及千兆以太网

二、 任务分析总结

本项目开发所基于的 uCore-SMP 是基于 MIT 的 xv6 进一步实现的、支持多核的操作系统。但是其原支持硬件平台为 K210 开发板，且只实现了部分系统调用。为成功移植 uCore-SMP 到 HiFive Unmatched 开发板，并且实现比赛所要求实现的功能，需要对原代码进行修改调整。从比赛要求所需要的**系统调用**出发进行需求的分析总结，总的来说需要的工作为：

- 实现 SD 卡驱动
- 参照 Linux 操作系统进行 FAT32 文件系统的移植
- 调整文件管理结构
- 实现加载文件使用的 loader
- 实现系统调用，根据要求调整内存、进程和文件相关代码
- 实现静态链接与动态链接

按照比赛所设定的阶段，任务也可以按照阶段进行划分，具体表现为不同阶段实现的系统调用。初赛阶段需要实现的系统调用如下：

初赛系统调用

```
#define SYS_getcwd 17
#define SYS_dup 23
#define SYS_dup3 24
#define SYS_mknod 33
#define SYS_mkdirat 34
#define SYS_unlinkat 35
#define SYS_linkat 37
#define SYS_umount2 39
#define SYS_mount 40
#define SYS_chdir 49
#define SYS_openat 56
#define SYS_close 57
```

```
#define SYS_pipe2 59
#define SYS_getdents64 61
#define SYS_read 63
#define SYS_write 64
#define SYS_fstat 80
#define SYS_exit 93
#define SYS_waitpid 95
#define SYS_sched_yield 124
#define SYS_kill 129
#define SYS_setpriority 140
#define SYS_getpriority 141
#define SYS_times 153
#define SYS_uname 160
#define SYS_gettimeofday 169
#define SYS_settimeofday 170
#define SYS_getpid 172
#define SYS_getppid 173
#define SYS_sysinfo 179
#define SYS_brk 214
#define SYS_munmap 215
#define SYS_clone 220
#define SYS_execve 221
#define SYS_mmap 222
#define SYS_wait4 260
#define SYS_sharedmem 282
#define SYS_spawn 400
#define SYS_mailread 401
#define SYS_mailwrite 402
```

随着比赛阶段的深入，需要实现更多的系统调用

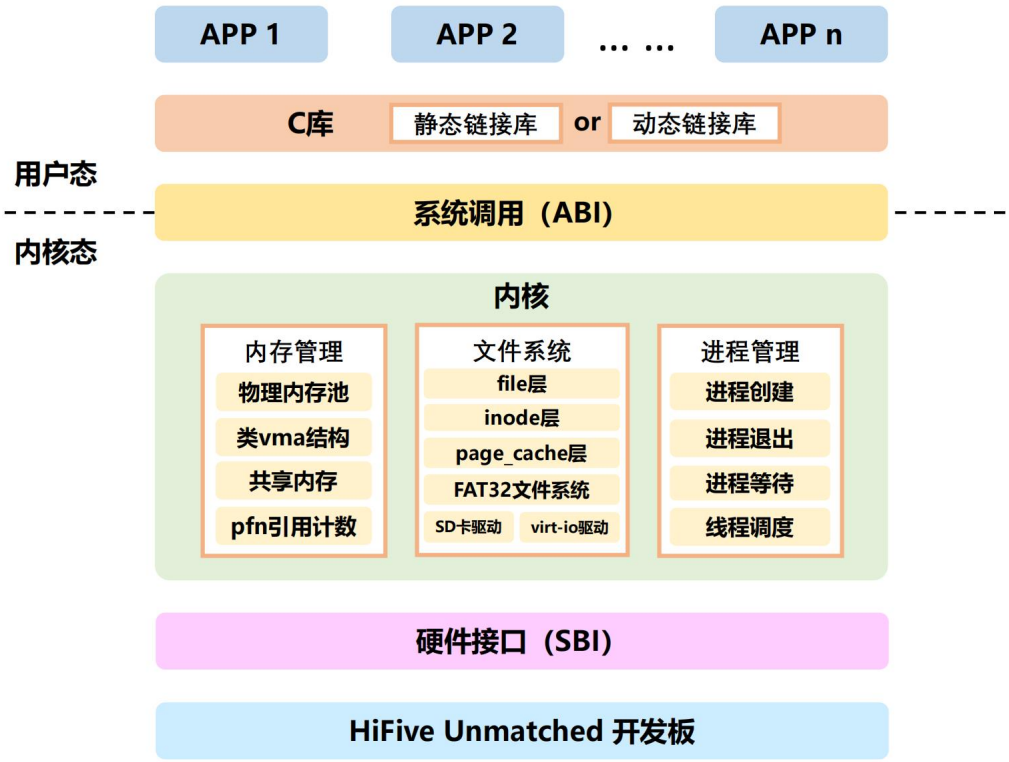
决赛系统调用

```
#define SYS_lseek 62
#define SYS_readv 65
#define SYS_writev 66
#define SYS_fstatat 79
#define SYS_futex 98
#define SYS_nanosleep 101
#define SYS_mprotect 226
```

三、 系统框架设计

(一) 整体框架

ShouChun OS 的设计框架图如下所示：



ShouChun OS 实现框架设计具有**模块化**的特点，这一点正是对应 RISCV 架构的特点实现的。RISCV 不同模块组织在一起，成为统一架构，用户可灵活选择模块组合应对不同的应用场景。对于本项目中较为复杂的操作系统场景而言，同时应用了 MachineMode（机器模式）和 UserMode（用户模式）。对应这样的模块组合，操作系统工作的大致流程即：用户应用程序通过库函数（静态或动态链接）发起系统调用请求，从而实现自身特权级无法完成的操作，内核层作为高特权级在更为安全的情况下完成对应请求。请求的具体实现是通过硬件接口使用 HiFive Unmatched 开发板实现的，其中还需要驱动 SD 卡。接下来对框架中每一个模块的大致功能实现进行阐述。

HiFive Unmatched **开发板与** SBI **（硬件接口）**。HiFive Unmatched 开发板具体情况在前文已提及，不作赘述。开发板是具体实现所使用的硬件实现对于操作系统开发者而言并非擅长之处。为了方便开发者工作，SBI（Supervisor Binary Interface）发挥了较大的作用。SBI 在规定的行业标准下将硬件抽象化，为开发者提供了便捷使用硬件的接口，从而减少了开发者的工作量，又很好的实现了模块化，使得不同部分的移植更为便捷。ShouChun OS 所使用的 RISCV 运行规范为 OpenSBI。

项目工作主要集中在**内核部分**，为了方便项目开发与文档理解，项目将内核部分的设计大致分为三个部分：进程管理、内存管理和文件系统。在成功启动内核后，三个部分协同工作，完成用户应用程序所要求的功能。其中进程管理主要负责用户进程创建、删除、切换等工作，实现进程部分对用户进程相关请求的相应。内存管理部分实现内存的分配、释放功能，其中包含有页表相关功能以及堆栈的管理。关于文件系统部分，项目按照比赛要求实现了 FAT32 格式文件系统，此外由于比赛要求使用 SD 卡读取烧写好的测试文件进行测试，ShouChun OS 还实现了 SD 卡驱动的功能。驱动 SD 卡功能需要文件系统和 SPI 共同实现，文件系统模块概念稍后再做详细介绍。SPI （Serial Peripheral Interface）是一种高速的、全双工的、同步的通信总线，它遵从主-从模式的控制方式，由主设备来控制次设备，为开发者提供接口来实现外设功能。最终实现了能够对硬盘进行驱动、初始化、读取和写入的功能。FAT32 文件系统对于文件实现了抽象化、规范化，使得内核能够更加清晰便捷地管理文件。

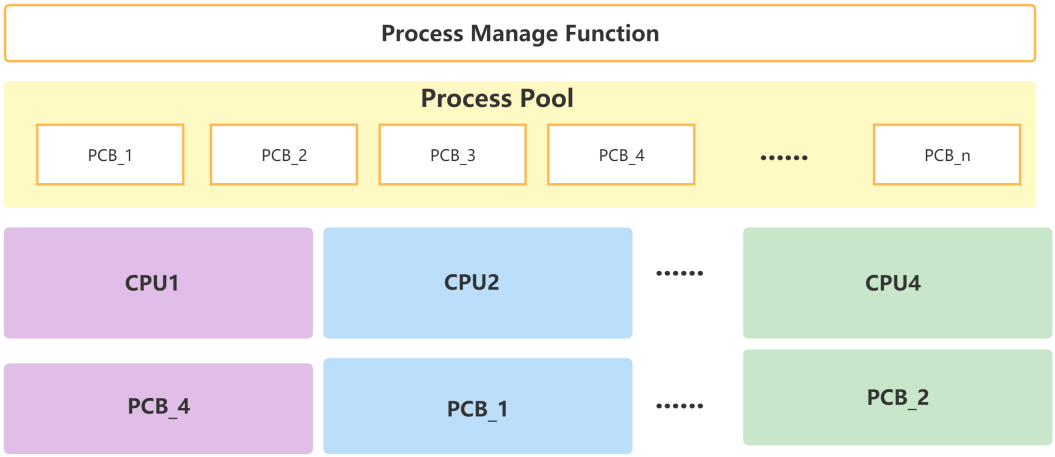
用户层各模块相互配合，利用内核所实现的上述工作，使得**用户应用程序**能够间接实现更高权限的操作，同时又更加具备安全性。这一过程是通过**系统调用**实现的：用户应用程序通过 API **层的库函数**来执行所需功能，库函数可以发起系统调用，通过数据的传递等发起系统调用请求，从而发起中断使操作系统内核执行相关操作并作出响应。

(二) 内核模块设计

项目组的工作主要在于内核部分，故接下来针对内核部分的三大模块：进程管理、内存管理、文件系统进行设计展示。

1. 进程管理

进程管理部分功能涉及到了**进程的创建与回收**、**进程切换**等方面，项目使用统一的结构体**进程控制块**（Process Control Block，简称 PCB）来进行管理。总的来说本项目中的进程管理可以使用下图来阐述：



自下而上进行解释：每一个进程都会对应一个进程控制块，用以记录进程基本信息用于管理。每个核在某一时刻只能运行一个进程，于是核对应着某一个确定的进程控制块。核所运行的进程来自于进程池。每载入一个进程，就需要放入进程池中等待调度。进程池中的进程都受进程管理函数的管理。

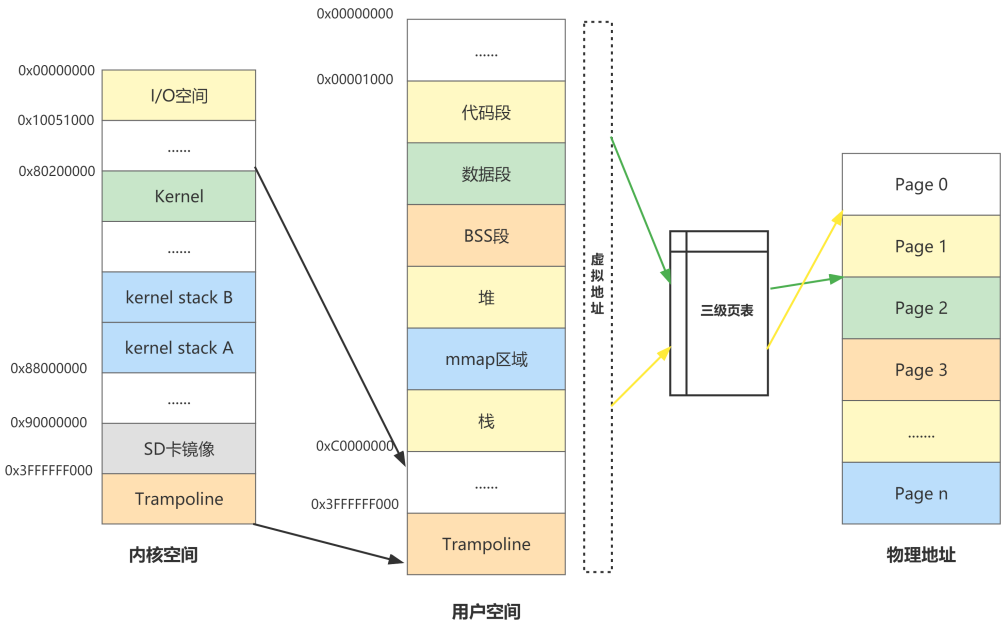
其中**进程管理块**中记录了进程运行、管理所需要的信息，包括有进程信息、内存信息、文件信息等。其中进程信息需要记录的有进程上下文、进程标识符等；内存信息需要记录进程内存大小、堆栈情况等；文件信息需要记录文件描述符表、路径信息等。其他一些重要信息还包括时钟、定时器、锁以及实现文件映射所需要的信息等。

核管理器中会记录当前运行进程的进程控制块。进程控制块也可以通过汇编指令来获取当前核的编号。核管理器和运行进程一一对应。

进程池用于存储所有需要运行的进程。其中的进程会通过**进程管理函数**进行创建或回收，进程间的调度切换也是通过这一组管理函数来实现的。

2. 内存管理

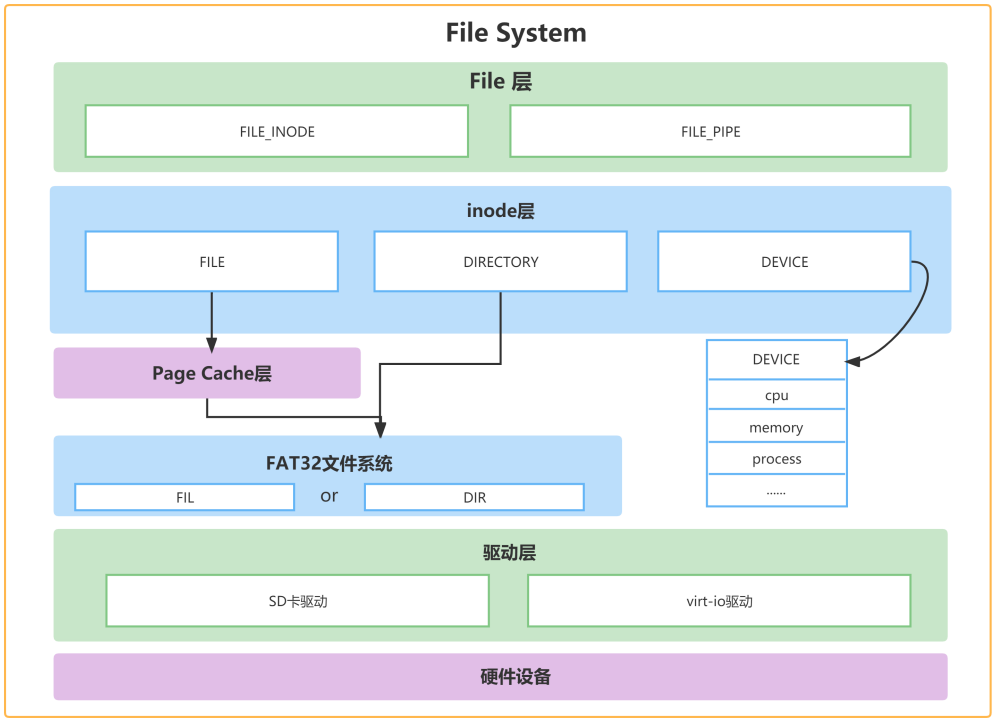
内存管理部分主要分为内核地址空间、用户地址空间及页表三个部分。其布局如下图所示。



系统启动、进程管理都在**内核地址空间**内，每个进程都会分配一个内核栈用以执行内核态下才能够实现的功能。程序数据存储、动态内存存储等都在**用户地址空间**内。内核地址空间和用户地址空间都使用虚拟地址进行实现，虚拟地址和物理地址之间的转换关系使用**页表机制**进行实现。ShouChun OS 中使用了三级页表进行实现。

3. 文件系统

文件系统模块设计图如下：



文件系统模块中，文件抽象为 FILE 结构进行管理。FILE **层会分为** pipe **类型和** inode **类型**进行管理，维护引用计数、读写权限等信息。其中 pipe 类型的 file 可用于管道管理，从而进行进程间的通信。

inode **层**中，使用 inode 组织管理文件类型、目录类型和设备类型的文件，inode 不被具体进程绑定，且可被多个进程共享。目前设计中，inode 层于 FAT32 文件系统一层绑定，尚未实现 vfs 层，故仅支持目前一种文件系统。普通文件类型和目录类型使用 FAT32 文件系统进行读写操作。设备类型的文件包括处理器、内存、进程等类型，当判断文件类型为设备类型时，即可根据设备号调用对应处理函数进行管理。inode 层中还实现了有关符号链接的部分，目前实现了普通文件和设备文件的符号链接，用于链接 libc 库的动态链接库文件。

page cache 用于将磁盘上的数据加载入页缓存当中，结合 mmap 实现共享内存并提高读写文件的效率，减少空间浪费。

FAT32 **文件系统**部分采用了嵌入式系统常用的 FatFs 文件系统驱动，由于其提供的接口更加接近于 Windows，因此需要进行包装来适应类 Unix 文件系统的接口。

驱动层通过 SPI 实现 SD 卡驱动，从而对 SD 卡上的数据进行读写。

四、 具体实现

ShouChun OS 主要使用 C 语言实现，其基于的框架为清华大学陶天骅的 uCore-SMP。

(一) 内核的启动

内核运行平台为 HiFive Unmatched 开发板，其继承了多个内核。其启动过程入下：

首先是使用一个 python 脚本（scripts/kernelld.py）生成内核镜像，其中指定了内核的入口为 0x80200000，另外规定了存储布局情况，将 shell 程序和测试点运行程序附加到内核镜像末尾。在执行完该脚本后，就会得到 kernel_app.ld 用于内核的加载。而后会运行 scripts/pack.py 来生成链接脚本，该程序用于将用户程序链接打包。

内核加载脚本中指定的入口地址 0x80200000 即为内核镜像的加载地址，在内核镜像加载完成时即跳转到该入口点处执行相应内容。我们可以通过汇编文件 entry.S 来观察这个入口点：

内核镜像入口

```
  .section .text.entry
  .globl _entry
_entry:
  # a0: hartid
  # every core has a boot stack of 4 KB
  la sp, boot_stack
  li t1, 1024*4   # t1 = 4KB
  addi t0, a0, 1  # t0 = hartid + 1
  mul t0, t0, t1  # 4K * (hartid + 1)
  add sp, sp, t0
  call main

  .section .bss.stack
  .globl boot_stack
  .globl boot_stack_top
boot_stack:
```

```
  .space 1024 * 4 * 8
  .globl boot_stack_top
boot_stack_top:
```

该汇编文件具体功能即为给当前核分配一个大小为 4KB 的内核栈空间，并且将内核栈的地址存储到 sp 寄存器中，最终调用 main() 函数启动内核。在多核系统中，每一个核都会执行该段代码，获取自身的内核栈空间，使得不同核运行在不同的位置处（这一位置是与内核编号相关的），互不影响，并行工作。

在跳转到 main 函数后，会进行操作系统的初始化等操作。此时尚未成功在 FU740 上成功启动多核。项目具体的实现方法为：在完成初始化操作后，**由当前启动的核来启动其他核**，该用于启动其他核的核为第一次启动。启动是通过 SBI **接口**来实现的，需要传参所需启动核的编号。以下为具体启动核的 SBI 接口函数：

内核镜像入口

```
void start_hart(uint64 hartid,uint64 start_addr, uint64 a1) {
    a_sbi_ecall(0x48534D, 0, hartid, start_addr, a1, 0, 0, 0);
}
```

启动内核使用的 main 函数大致如下（省略信息展示与初始化部分）。

内核启动的 main 函数

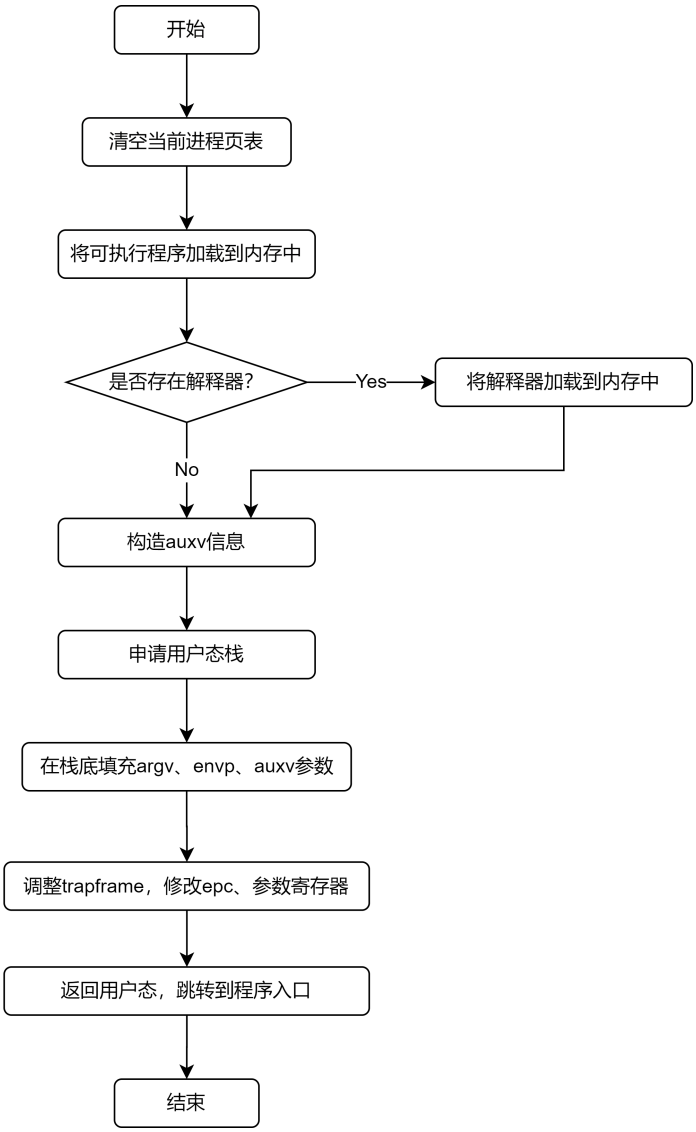
```
  void main(uint64 hartid, uint64 a1) {
    if (first_hart) {
      // 计算机信息展示...
      // 初始化工作...
        if (hartid >= NCPU){
            panic("unexpected hartid");
        }
        int CPU_START=1;    // core 0 is not usable
        for (int i = CPU_START; i < NCPU; i++) {
            if (i != hartid && i!=0) // not this hart
            {
                printf("[ucore] start hart %d\n", i);
                start_hart(i, (uint64)_entry, 0);
                while (booted[i] == 0){
                    // wait
                }
            }
        }
        wait_all_boot();
    } else {
        hart_bootcamp(hartid, a1);
    }
    while (!all_started) {
        ; // wait until all hard started
    }
    // 内核运行
}
```

可以看到，当前启动核为**第一次启动**时，就会用于遍历启动其他所有核；当判断到当前核并非初次启动时，就说明无需帮忙其他核，初始化与其他核的独立部分即可。等待所有核启动后，各个核再进入运行状态。**需要注意的是**，FU740 具有四个大核以及一个小核，小核并不能通过 SBI 启动，所以启动时内核编号需要从 1 开始。

(二) elf loader：可执行程序加载

elf loader 用于将 ELF 格式的可执行程序加载到内存中，在该系统中支持静态链接和动态链接两种程序的加载。

其简化的流程图如下：



对于**静态链接程序**，将可执行程序按照 ELF 头中的信息加载到内存中即可。

对于**动态链接程序**，除了需要加载可执行程序外，还需要将解释器加载到内存中。解释器的作用是根据系统加载器提供的 AUXV 信息，通过 mmap 系统调用，加载动态链接程序依赖的所有动态链接库，并且对动态链接库进行重定向，最终跳转到动态链接程序入口，此时解释器的工作完成。

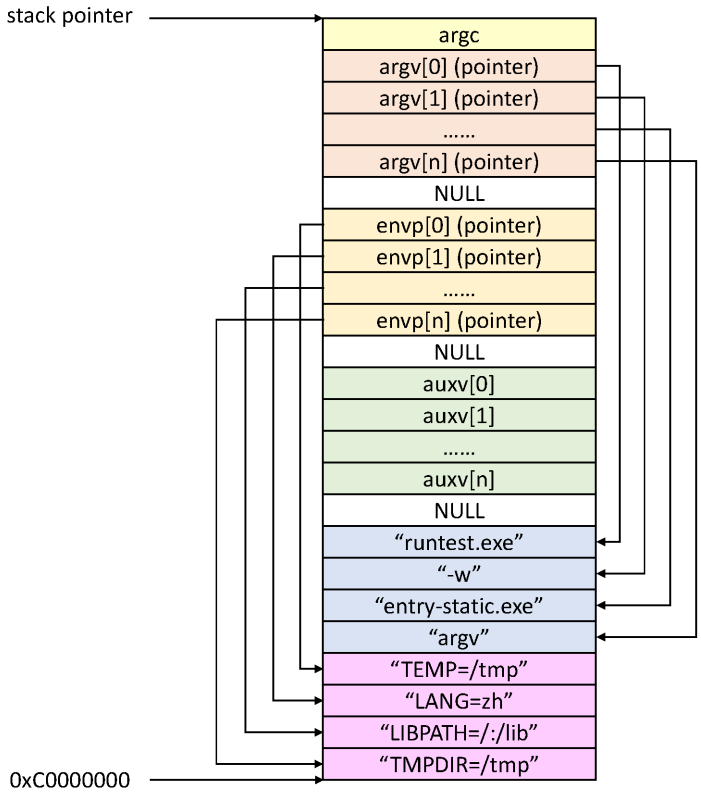
AUXV 中的信息主要包括以下几个部分，这些信息对于解释器来说是必要的：

AUXV 信息

```
#define AT_PHDR     3     /* Program headers for program */
#define AT_PHENT    4     /* Size of program header entry */
#define AT_PHNUM    5     /* Number of program headers */
#define AT_PAGESZ   6     /* System page size */
#define AT_BASE     7     /* Base address of interpreter */
#define AT_ENTRY    9     /* Entry point of program */
```

其中 AT_PHDR、AT_PHENT、AT_PHNUM、AT_ENTRY 用于给解释器提供可执行程序的各种信息，解释器会解析可执行程序的 ELF 头，找找到依赖的动态链接库文件并加载到内存中。AT_BASE 是解释器的加载地址，用于解释器获取自身信息并进行重定向工作。AT_PAGESZ 是系统页面大小，在该系统中该数值为常数 0x1000。

在系统加载器加载完毕后，会将 argv、envp、auxv 参数存放在用户态栈中，提供给应用程序与 C 库使用，栈结构如下：



(三) 中断异常机制

中断机制在系统中起着**通信网络**的作用，可以协调系统对各种外部事件的响应和处理。中断是 CPU 对系统发生的某个事件作出的一种反应。引起中断的事件称为**中断源**。中断源向 CPU 提出处理的请求称为**中断请求**。发生中断时被打断程序的暂停点称为**断点**。CPU 暂停现行程序而转为响应中断请求的过程称为**中断响应**。处理中断源的程序称为**中断处理程序**。CPU 执行有关的中断处理程序称为**中断处理**。

1. 中断处理

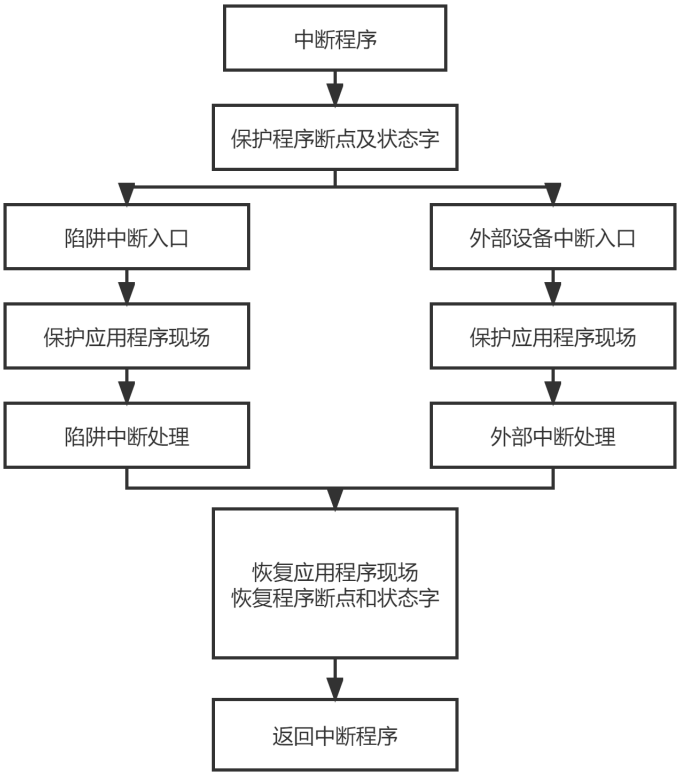
中断处理所使用的到结构体为 trapframe：

trapframe 结构体

```
struct trapframe {
  /*   0 */ uint64 kernel_satp;   // kernel page table
  /*   8 */ uint64 kernel_sp;     // top of process's kernel stack
  /*  16 */ uint64 kernel_trap;   // usertrap()
  /*  24 */ uint64 epc;           // saved user program counter
  /*  32 */ uint64 kernel_hartid; // saved kernel tp
  // ..... 寄存器
  /* 280 */ uint64 t6;
};
```

trapframe 中主要内容为**发生中断异常时的现场情况**，其中 kernel_satp 给出了内核页表的地址；kernel_sp 指向了进程内核栈的顶部；kernel_trap 指向 usertrap 函数，用于在用户状态下发生中断时跳转到处理函数。

用户状态下中断处理流程图如下：



中断的处理需要使用的函数为 usertrap，在 trap.c 文件中。每次出现中断或者异常时，都需要跳转到中断入口处，执行 trampoline.S 汇编代码的内容，即先保存当前用户程序的现场到 trapframe 当中，最后取出 kernel_trap 中所存储的 usertrap 函数的地址，最终跳转到该函数处进行中断的处理。在 usertrap 中，会首先将 kernelvec 的地址写入 stvec 中，作为中断处理的基址，此时的中断为需要**升高权限为内核态**所处理的中断了。usertrap 处理中断时，分为了中断和异常两种情况。其中中断包括有计时器发起的中断和外部设备发起的中断；异常包含了指令异常和页加载异常等等，本项目中心工作所集中的系统调用便包含在异常当中，具体处理异常的代码如下：

系统调用异常

```
case UserEnvCall:    // 8
if (p->killed)
    exit(-1);
trapframe->epc += 4;
intr_on();
syscall();
break;
```

在用户态发起系统调用异常后，会使用 syscall **函数**作为接口，读取 trapframe 中所记录的系统调用号以及参数，从而在 syscall 函数中找到对应的中断处理函数。将参数传给相应的中断处理函数并执行结束后，将返回值存储在 trapframe 的 a0 中，最终返回时会通过存储的地址返回调用位置，恢复触发中断前的现场。本项目中内核态下的中断主要是**时钟中断与外部中断**，其他情况下的中断与异常均作为意外情况，使用 panic 关闭系统。

(四) 进程管理

1. 相关系统调用

进程管理部分相关的系统调用如下：

进程管理系统调用

```
#define SYS_clone 220
#define SYS_execve 221
#define SYS_wait4 260
#define SYS_exit 93
#define SYS_getppid 173
#define SYS_getpid 172
```

2. 核心结构体 proc

进程管理的实现主要是通过核心的**结构体** proc 实现的。进程管理的核心结构体为 proc，其定义如下：

proc 结构体

```
struct proc {
  struct spinlock lock;

  // PUBLIC: p->lock must be held when using these:
  enum procstate state;  // Process state
  int pid;               // Process ID
  int killed;            // If non-zero, have been killed
  pagetable_t pagetable; // User page table
  void *waiting_target;  // used by sleep and wakeup, a pointer of anything
  uint64 exit_code;      // Exit status to be returned to parent's wait

  // proc_tree_lock must be held when using this:
  struct proc *parent; // Parent process
```

```
14
15      // PRIVATE: these are private to the process, so p->lock need not be held
            .
16      uint64 ustack_bottom;         // Virtual address of user stack
17      uint64 kstack;                // Virtual address of kernel stack
18      struct trapframe *trapframe; // data page for trampoline.S, physical
            address
19      struct context context;       // swtch() here to run process
20      uint64 total_size;            // total memory used by this process
21      uint64 heap_start;            // start of heap
22      uint64 heap_sz;
23      uint64 stride;
24      uint64 priority;
25      uint64 user_time;             // us, user only
26      uint64 kernel_time;           // us, kernel only
27      uint64 last_start_time;       // us
28      struct file *files[FD_MAX]; // Opened files
29      struct inode *cwd;            // Current directory
30      struct shared_mem * shmem[MAX_PROC_SHARED_MEM_INSTANCE];
31      void * shmem_map_start[MAX_PROC_SHARED_MEM_INSTANCE];
32      void* next_shmem_addr;
33      struct mapping maps[MAX_MAPPING];
34      char name[PROC_NAME_MAX]; // Process name (debugging)
35  };
```

进程管理相关的操作大部分都是基于此结构体进行实现的。proc 结构体的成员从**性质**层面来看，大致可以分为**公开资源和私有资源**两个部分，可用以实现多进程安全并行；从**功能**方面来看，proc 成员分为四类：进程的基本信息、进程的内存信息、进程文件信息和其他信息。

进程的基本信息包括了进程编号、进程上下文、父进程等。其中 state 为进程的状态，包括有 UNUSED、USED、SLEEPING、RUNNABLE、RUNNING、ZOMBIE 六种情况，利用这几种状态就可以实现进程的调度和回收等功能；pid 为进程编号，是进程重要的成员属性，大多进程管理相关操作都基于此实现；parent 记录了父进程，用于进程之间关系的管理；context 记录了进程的上下文，用于进程的切换；trapframe 陷阱帧用于异常中断机制的运行现场恢复；exit_code 用于在子进程退出时将退出状态码放回给父进程。

进程的内存信息包括进程占用的地址空间、堆以及共享内存信息。ustack_bottom、kstack、heap_start 分别记录了用户栈、内核栈、堆的地址，这里的地址指虚拟地址；maps 用来记录文件载入后映射的情况；pagetable 记录了当前进程所使用的页表，从而进行虚拟地址和物理地址之间的转换。

进程的文件信息包括了文件描述符数组和当前位于目录。files 记录了所有已经打开的文件描述符；cwd 可获取当前所处的目录。

其他信息包含了运行时间、信号、优先级、调度权等，辅助进行进程的管理。结构体中的成员 lock 用于实现进程锁，从而达成多进程并行执行而互不影响。进程锁的使用对进程公开资源的修改来说十分重要。

3. 进程管理方法

一个核可以同时运行多个进程，内核使用了**进程池**来对进程进行管理。结合 proc 结构、pool 进程池和其他一些重要辅助信息，可以完成进程的创建、销毁、调度和调度等功能。

（1）**进程的创建与销毁**：在创建进程时，会在进程池中找到一个可用的进程号（pid），进行初始化。proc 唯一对应一个进程，创建时将根据程序内容构建处用户地址空间并分配物理空间。当进程退出时，需要对进程进行销毁，释放其占用的进程池中的可用 pid，并重新初始化结构体。

（2）**进程调度方法**：进程的调度使用 Stride **算法**。该种算法为不同的进程分配了不同的优先级，每个进程得到的时间资源预期优先级成正比。以下是调度所使用的代码（只列出了调度相关的核心部分）：

进程调度方法

```
void scheduler(void)
{
    // ...
    for (;;){
        uint64 min_stride = ~0ULL;
        struct proc *next_proc = NULL;
        int any_proc = FALSE;

        for (struct proc *p = pool; p < &pool[NPROC]; p++){
            if (!p->state == UNUSED){
                any_proc = TRUE;
                // debugcore("state=%d", p->state);
            }
            if (p->state == RUNNABLE && !p->lock.locked){
                if (p->stride < min_stride){
                  //找到stride最小的进程
                    min_stride = p->stride;
                    next_proc = p;
                }
            }
        }
      // ...
        if (next_proc != NULL)
        {
            //...
            //计算步长pass并更新stride
            uint64 pass = BIGSTRIDE / (next_proc->priority);
            next_proc->stride += pass;
            //进行进程的切换
            swtch(&mycpu()->context, &next_proc->context);
            //...
        }
        else{
          //...
        }
       //...
```

```
    }
}
```

算法基本思想为：

1. 为处于 RUNNABLE **状态**的进程设置优先级，表示进程当前的调度权。并定义 pass 值，表示相应进程在调度后，stride 需要累加的值。

2. 在每次需要调度时，从当前 RUNNABLE 状态的进程中选择 stride **最小**的进程进行调度。

3. 获得调度的进程，将其 stride 加上对应的步长 pass（pass 值只与**进程的优先权**有关）。

4. 在一段固定时间后，回到步骤 2，重新进行调度，运行当前进程池中 stride 最小（非 0）的进程。

其中，调度进程时的**切换过程**为：首先保存当前进程的上下文（context），而后再通过所需调度进程的上下文恢复所需调度进程的现场，从而实现调度。具体内容在汇编文件 switch.S 中。

（3）**创建子进程**：创建子进程实际也是实现 fork 的方法，在执行 fork 操作时，当前进程，即父进程，会构造一个与自己完全相同的用户空间，并分配对应的空间、pid 以及内核栈，新得到的进程即为子进程。

（五） 内存管理

1. 相关系统调用

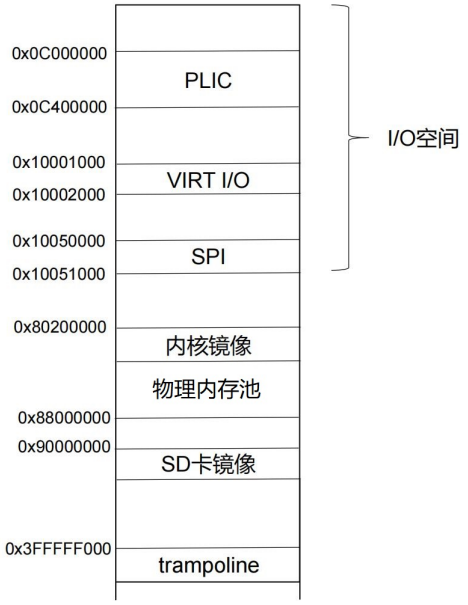
内存管理系统调用

```
#define SYS_brk 214
#define SYS_munmap 215
#define SYS_mmap 222
```

2. 内核态内存管理

在系统启动后，首先处于没有开启分页的状态，访问的地址全部是物理地址，因此首先需要构建**内核态页表**，该任务由 kvminit 函数完成。在内核态中，几乎保持所有映射均为**虚拟地址 = 物理地址**，便于管理内存。

内核态内存布局如下：



其中 I/O **空间部分的映射用于与外设交互**。其中 PLIC 为中断控制器地址，用于控制中断开关，获取中断信息。VIRT I/O 为 qemu 的 mmio 地址，用于在 qemu 中读取磁盘文件。SPI 地址为 fu740 的 SPI 端口地址，用于读写 SD 卡。

从 0x80200000 开始为**内核镜像**，其中代码段映射访问权限为 RX，数据段映射访问权限为 RW。这部分内容由上一启动阶段的 U-Boot 加载到内存中。

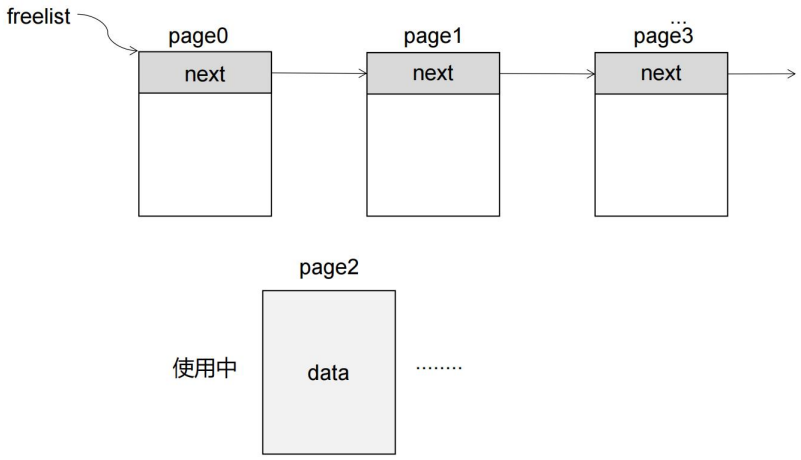
从内核镜像末尾到 0x88000000 均作为**物理内存池**，用于给内核中的部分动态数据结构、所有用户态进程分配内存。

另外，评测机将 SD 卡镜像加载到了 0x90000000 位置，我们也对此地址范围进行了映射，用于**读写 SD 卡内容**。

除此之外，在虚拟地址为 0x3FFFFFF000 的位置还映射了 trampoline，在用户态内存中的相同虚拟地址同样也映射了 trampoline，trampoline 是一小段代码，这段代码用于**衔接用户态和内核态之间的切换**，在由用户态进入到内核态时，需要保存上下文，构造 trapframe，并跳转到异常处理函数，在由内核态退出到用户态时，需要根据 trapframe 还原上下文。

（1）物理内存池

物理内存池结构如下：



空闲页面依靠**单向链表**连接，每个空闲页面的头部是一个 list_entry，指向下一个空闲页面。

每次需要分配页面时，则从链表头部摘下一个节点，当页面释放时，则重新加入空闲页面链表。

（2）pfn 引用计数

由于该系统支持文件 mmap 映射，因此会存在多个进程的虚拟地址映射到相同物理页的情况，因此为了正确实现物理页面资源的管理，需要记录**物理页面引用计数**，记录的数据结构如下：

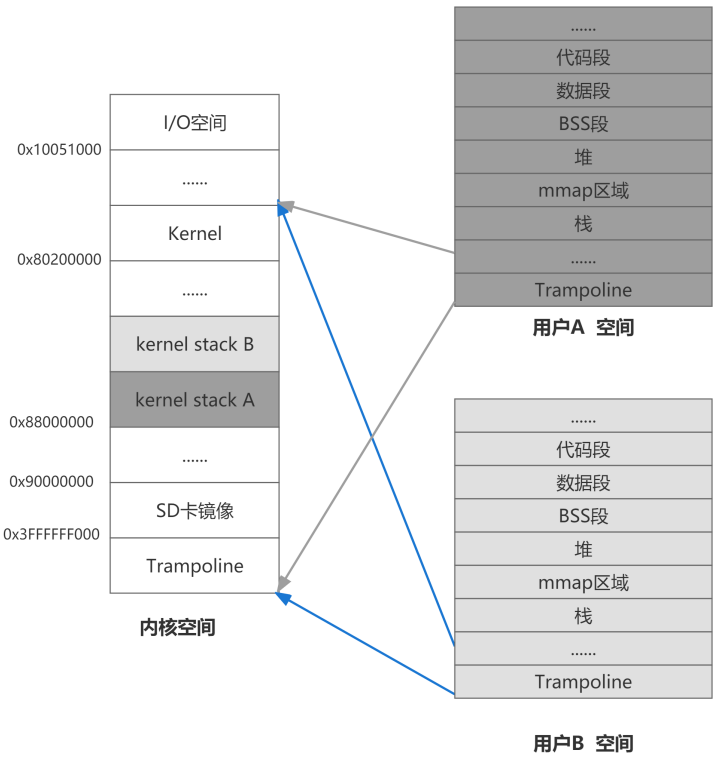
页面引用计数

```
uint8 pfn_ref[(PHYSTOP - KERNBASE) >> PGSHIFT];
```

每个页面分配时，在 pfn_ref 中将对应位置置为 1，当页面被再次引用时，则引用计数加 1，当页面被释放时，引用计数减 1，当引用计数变为 0 时，则该页面变为空闲状态，放入空闲链表中。

（3）多进程内核栈分配

在多进程运行过程中，内核地址空间被划分为了多个内核栈空间，对应不同的用户进程：



每个进程对应着自己唯一的内核栈。当新建一个进程时，都需要为其分配内核栈空间。以下是新建进程时给进程分配内核栈的代码：

分配内核栈空间

```
struct proc *alloc_proc(void) {
    // ......
    memset(&p->context, 0, sizeof(p->context));
    p->kstack = (uint64)kstack[p - pool];
    memset((void *)p->kstack, 0, KSTACK_SIZE);
```

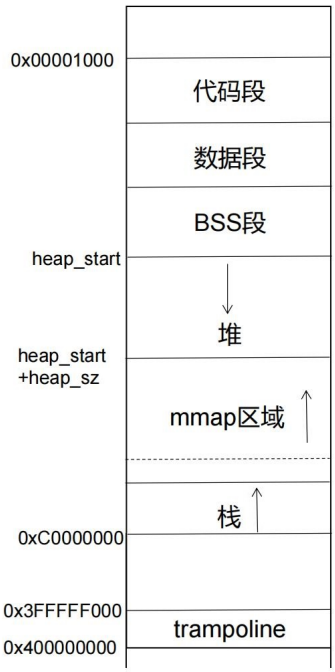
```
6
7       p->context.ra = (uint64)forkret; // used in swtch()
8       p->context.sp = p->kstack + KSTACK_SIZE;
9       // ......
10  }
```

可以看到，**内核栈空间的分配方法**为：从划分好的内核栈分配中选取对应进程池位置的内核栈。也就是说，每个进程对应着进程池中唯一的一个 proc 指针，每一个 proc 都对应着固定的内核栈地址。**内核栈设置方法**为定义 NPROC×KSTACK_SIZE 大小的二位字符数组，即最多可同时给 NPROC（256）个进程分配内核栈，且内核栈大小为 KSTACK_SIZE（8192）。

3. 用户地址空间

用户态所有数据所占内存全部由内核中的**物理内存池**提供。

用户态内存布局如下：



本项目使用了 SV39 虚拟内存，支持高达 256GB 的用户内存空间。由 ELF 装载地址（默认为 0x1000）到 heap_start 之间的地址空间用于**装载** ELF **可执行程序**的各个段。由 heap_start 到 heap_start + heap_sz 之间的地址空间是堆空间，其作为动态分配的空间从堆栈底部向高地址增长。栈空间从 0xC0000000 向低地址增长，目前设定栈大小为 25 个页面大小。在堆空间和栈空间之间的是 mmap **区域**，用于动态申请内存或映射文件，向低地址方向增长。用户空间的顶部为 trampoline，将虚地址和实地址对应了起来，可用于在多进程并行时切换进程。Trampoline 下存储着 4KB 大小的 TrapFrame 用于中断异常处理。

其中 mmap 区域内存映射信息使用了 mapping **结构**进行记录：

mapping 结构

```
1   struct mapping {
2     uint64 va; // must be PGSIZE aligned
3     uint npages;
```

```
    bool shared;
};
```

每个进程都有一个 mapping 结构体数组，作用类似于 Linux 中的 vma，用于存放 mmap 映射信息，其中 shared 记录用于判断在进程克隆时该映射是否与子进程共享。

4. 页表机制

虚拟地址的管理使用了**页表机制**，进程在访问到记录好的页表的物理地址后，可以通过页表转换找到虚拟地址所对应的物理地址。

（1）**页表机制的使能**。虚拟地址和物理地址之间的映射关系记录在了页表当中，可以通过 mmap 函数和 munmap 函数进行这种关系的管理。本系统中，所有进程内核态下虚拟地址与物理地址之间的对应关系都是相同的。进程在一开始启动时就会初始化内核空间的页表，内核空间的映射关系为固定的直接映射，使用 kvmmap 函数将其记录到了内核页表当中。注意在这一初始化内核页表的过程中不刷新块表和开启页表机制，在进程加载好内核页表后，才可以使能页表机制。

（2）**三级页表机制**。Sv39 页表机制是具有三级页表的，虚拟地址将分为五个部分（下图为示意图）：39-63 位使用 0 进行填充，30-38 位为页目录索引，21-29 位为一级页表索引，12-20 位为二级页表索引，0-11 位为页内偏移地址。

其寻址方法流程如图所示，首先访问记录好的进程对应页表位置，取 30-38 位偏移找到对应页表的物理位置，而后取 21-29 位在页表内偏移找到对应页表项所记录的二级页表的物理地址，而后访问二级页表根据 12-20 位找到对应物理页的位置，最终根据页内偏移即可访问正确的地址。该部分对应了函数 walk（位于 virtual.c 文件当中）。

5. 系统调用实现

该部分系统调用的实现主要体现在 brk 和 mmap、munmap。

（1）brk 的实现

brk 系统调用用于**调整堆内存大小**，首先将新栈顶指针与堆起始地址比较，防止新地址低于堆起始地址。

检查 brk 参数

```
if ((uint64)addr < p->heap_start) {
    infof("sys_brk: addr is below heap start");
    return -1;
}
```

然后根据新栈顶指针的位置，决定申请内存/释放内存。

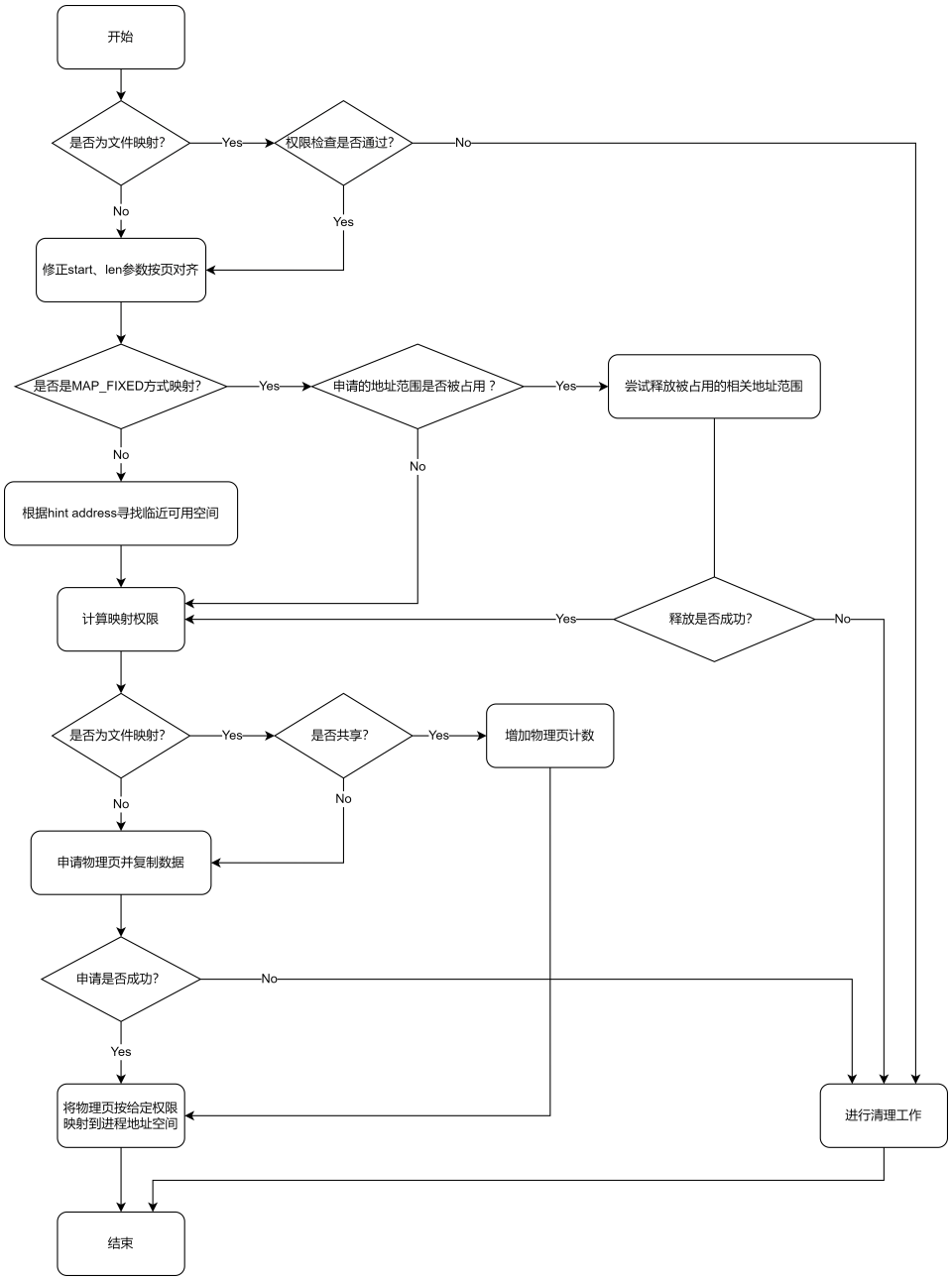
brk 实现

```
uint64 old_pos = p->heap_start + p->heap_sz;
uint64 new_pos = (uint64)addr;
```

```
if (new_pos > old_pos) {
    // allocate memory
    new_pos = uvmalloc(p->pagetable, old_pos, new_pos);
} else {
    // deallocate memory
    new_pos = uvmdealloc(p->pagetable, old_pos, new_pos);
}
```

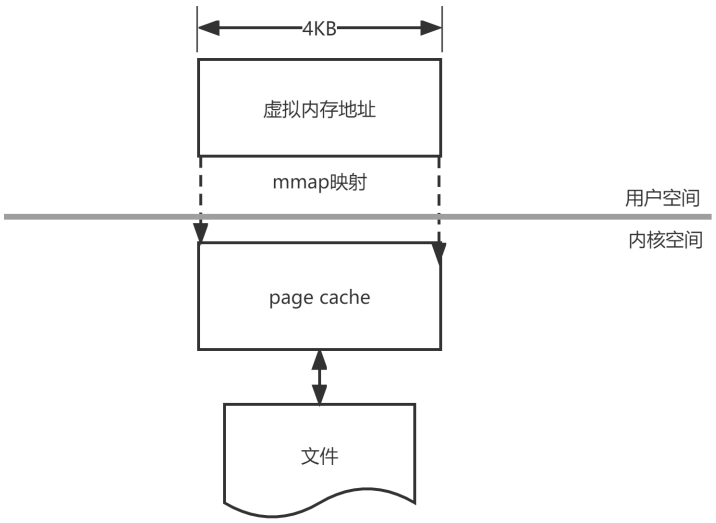
（2）mmap 和 munmap 的实现

mmap **用于申请内存、映射文件、实现共享内存**，其内部实现较为复杂，目前支持 MAP_SHARED、MAP_PRIVATE、MAP_FIXED、MAP_ANONYMOUS、MAP_FILE 类型的请求，简化的流程图如下：

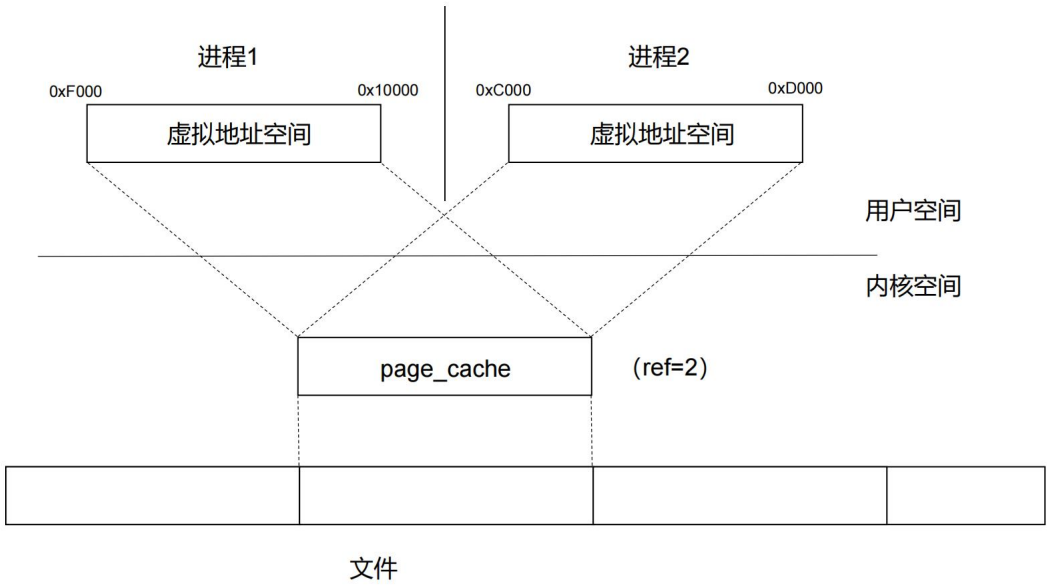


对于**文件映射类请求**，由于在文件系统的设计中实现了 page cache 层，因此可以很容易的

获得文件对应的物理页面，并将其映射到进程地址空间中。当某进程需要使用磁盘上的某一部分内容时，需要先在内核空间中为其分配一定大小的空间作为缓存，调用 mmap **函数**来建立该部分内容到虚拟内存的映射，并使用页表记录这一关系。当有其他进程要求访问该部分内存时，可以通过 sys_mmap 调用来直接使用这段已经加载入内核空间的内容，同时更新该段内容的使用情况。当进程结束访问后，就会调用 sys_munmap 函数来释放其对该部分内存的引用，当其引用数清零时，即可删除这一映射关系，释放对应部分的内存。



文件映射除了可以用于**加快文件读写速度**，还可以用于实现**共享内存**，当两个进程以 MAP_SHARED 方式映射了同一个文件的同一位置后，则该 page_cache 物理页映射到了两个进程地址空间中，即可实现共享内存。



对于内存申请类请求，实现的难点主要在于对 MAP_FIXED 的处理，根据 MAP_FIXED 标志的说明，对于新申请的内存区间与已有内存区间存在交集时，需要将 overlap 的区间释放掉，因此就涉及到对 mapping **数据结构的拆分与合并**问题，因此新加入了以下三个函数用于解决该

问题。

mapping 数据结构的拆分与合并

```
int mapping_add(struct proc *p, uint64 va, uint npages, bool shared);
int mapping_remove_fixed(struct proc *p, uint64 va, uint npages);
int mapping_try_remove_page(struct proc *p, uint64 check_va);
```

在处理 MAP_FIXED 请求时，首先需要对 overlap 的区间调用 mapping_try_remove_page 和 uvmunmap 进行释放，若成功释放掉了 overlap 区间，则继续后续处理，否则映射失败。

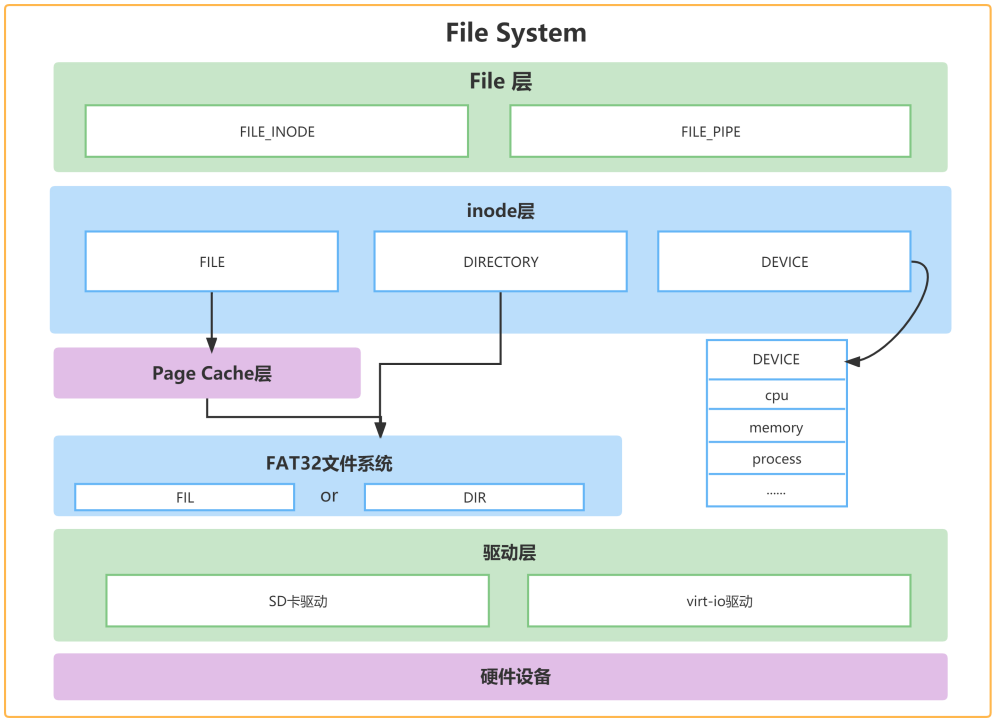
在处理普通的**内存申请请求**时，若给定 address 不是 NULL，则将给定 address 作为 hint，基于此地址向高地址搜索空闲区间，否则则按照 mmap 增长方向，由高地址向低地址搜索空闲区间。

munmap 实现较为简单，只需将参数按照页面大小对齐，然后对给定区间调用 mapping_try_remove_page 和 uvmunmap 释放内存。

(六) 文件系统

在系统中，对 fat32 文件系统进行了多层封装，并且对应用程序提供了类 Unix 的文件系统操作接口。

其层次结构如下：



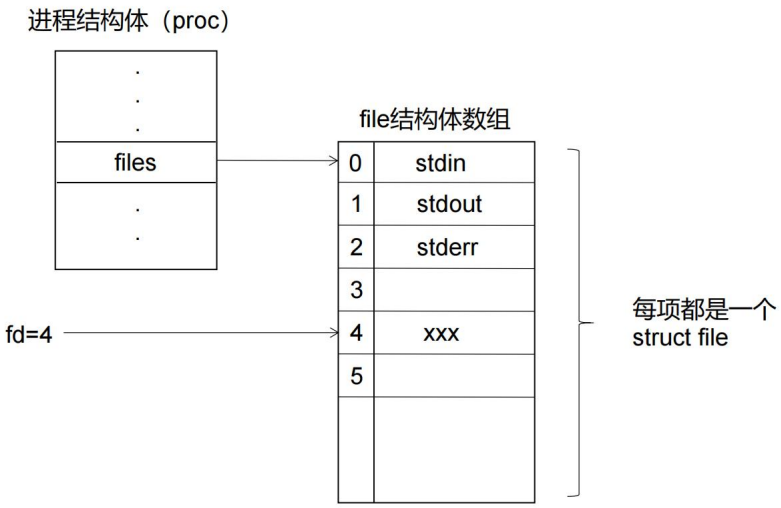
1. 相关系统调用

系统中实现的文件系统相关 syscall 如下，相关函数位于 syscall/syscall_impl.c 中。除比赛要求的 syscall 外，为了支持 libc-test 的多个测试点，还对 SYS_lseek、SYS_readv、SYS_writev、SYS_fstatat 提供了支持。

文件系统相关系统调用

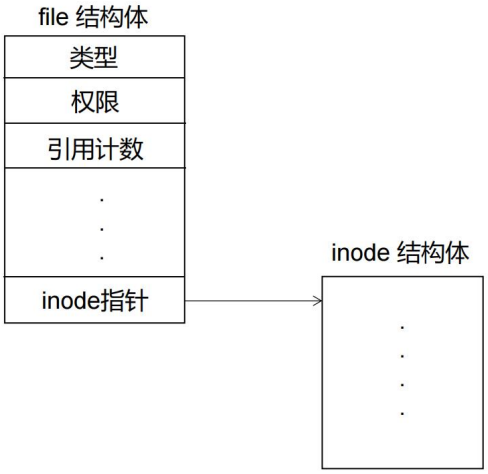
```
#define SYS_getcwd 17
#define SYS_mknod 33
#define SYS_mkdirat 34
#define SYS_linkat 37
#define SYS_unlinkat 35
#define SYS_chdir 49
#define SYS_openat 56
#define SYS_close 57
#define SYS_pipe2 59
#define SYS_getdents64 61
#define SYS_lseek 62
#define SYS_read 63
#define SYS_write 64
#define SYS_readv 65
#define SYS_writev 66
#define SYS_fstatat 79
#define SYS_fstat 80
```

文件系统相关 syscall 的主要作用是接收用户传入的参数，将字符串等参数从进程的地址空间复制到内核空间，便于后续的层进一步处理，然后对参数进行合法性检验，过滤掉一些不合法、不支持的系统调用请求。在这一层中还需要将用户传入的 fd 转换成 struct file*，然后将请求发送给下一层：file 层。



2. file 层

file 层操作的对象为 file **结构体**，每个 file 结构体是进程**独有**的，因此不需要使用锁防止对该数据结构的并行操作，主要工作是**维护** file **结构体内的各个属性**，包括引用计数、读写权限等信息，并取出内部的 struct inode 指针，将请求转发给 inode 层。



举例打开文件所需要使用的函数 fileopen，用以说明 file 层的功能。根据传入文件路径 path 查找对应的 inode，这里需要注意对于文件路径的处理，例如文件路径中包含./或者. ，则需要通过 fix_cwd_slash 函数进行处理。处理后根据 flags 选择对应操作，例如若是创建模式，则会根据路径创建对应的 inode, 若打开已存在的文件或目录则依据 inode_by_name 函数获取文件路径对应的 inode，再依据获取 inode 类型进行对应的一些错误处理操作。同时再利用 filealloc() 函数获取可用的 file 结构体，而后对 file 结构体的信息进行补充并完善。

3. inode **层**

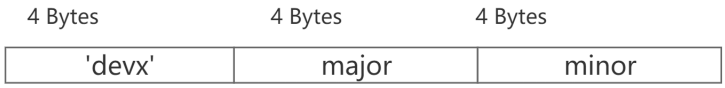
inode 层负责**管理磁盘上的文件**、**目录**，每一个 inode 结构体都是对普通文件、设备文件或目录的抽象，由于 inode 是对某个文件/目录的抽象，而不与具体进程绑定，inode 可以被多个进程共享，因此存在多线程冲突问题，需要使用锁来避免对 inode 数据结构的并行操作。大部分 inode 层的函数要求获得 inode 的锁以后才可以使用。

在目前的设计中，由于**仅**需要**支持** FAT32 **文件系统**，因此 inode 结构体和 fatfs 中的文件、目录数据结构绑定，更好的设计是在此处插入虚拟文件系统层（vfs），使得 inode 与 fatfs 解绑，从而降低耦合性，便于支持多种文件系统。

（1）设备文件

已支持的设备文件包括：console、cpu、mem、proc、null、zero

设备文件的存储格式如下：



从文件中读取出上述信息后，根据 major 号调用相应 handler 函数进行处理。

（2）符号链接

目前仅支持普通文件/设备文件类型的符号链接，不支持目录的符号链接，主要用于链接 libc 库的动态链接库文件。

符号链接文件的存储格式如下：

4 Bytes	n-4 Bytes
'slnk'	链接文件的绝对路径

从文件中读取出上述信息后，根据链接文件的绝对路径打开连接的文件。

（3）普通文件

若判断类型为文件，但并不是设备文件、符号链接，则认定该文件为普通文件，通过 fatfs 接口的 f_open 打开该文件。

（4）目录

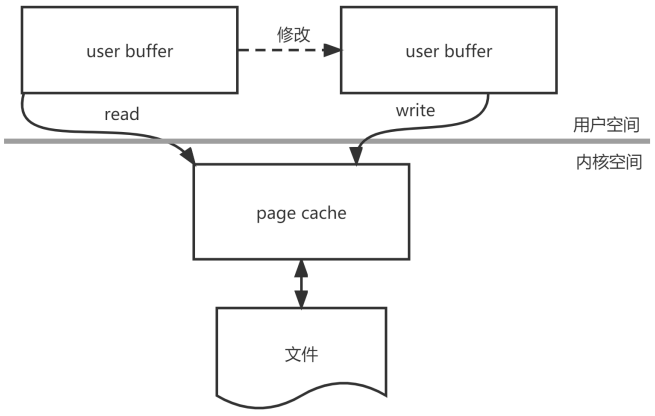
若判断类型为目录，则通过 fatfs 接口的 f_opendir 函数打开该目录。

inode 层的功能可以通过举例 readi 函数的实现方式来展示。readi 函数利用 inode 指针，负责将文件对应的信息读取到 dst 地址上。在保证读取文件内容中的偏移位置合法的情况下（不能小于文件的大小，以及读取长度不能超出从 off 开始文件的剩余内容）。根据获取内容长度，对应利用 either_copyout 从内核态将对应信息逐个复制到对应地址上。

4. page_cache **层**

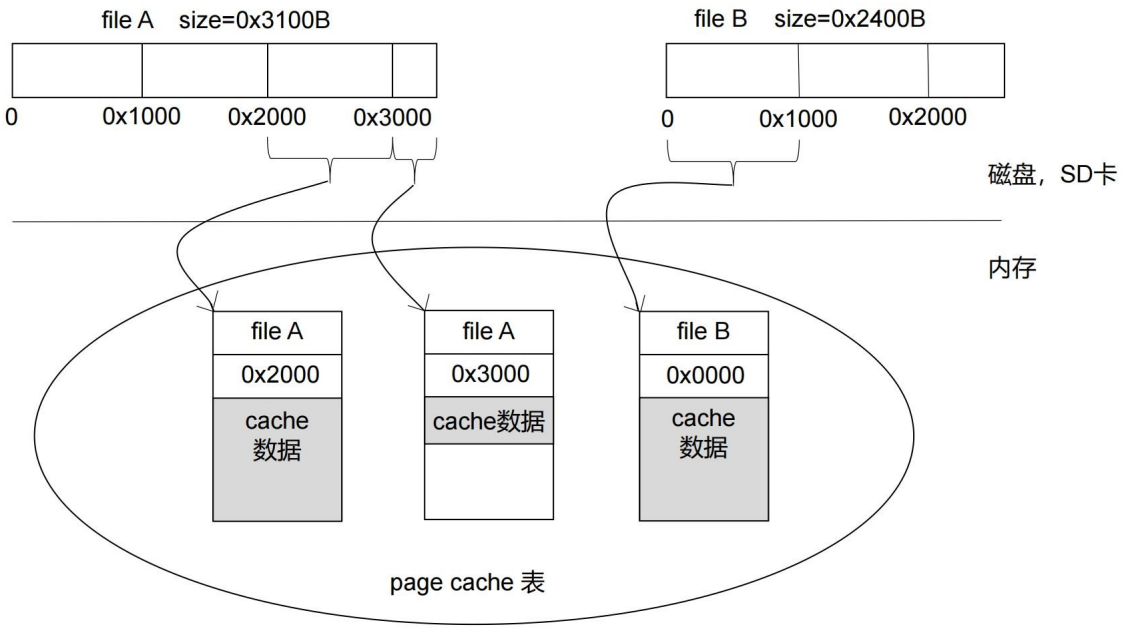
为了**提高** I/O **速度**，同时为了**支持** mmap，因此在 inode 下层加入 page_cache 层，在该系统中 page_cache 的设计思想与 Linux 类似，但是具体实现有所不同。

在 Linux 2.4 版本的内核之前，page cache 与 buffer cache 是完全分离的。但是，块设备大多是磁盘，磁盘上的数据大多通过**文件系统**来组织，这种设计使得读写文件需要经历五个步骤（如下图）：将文件数据块加载入页缓存中，页缓存中文件内容读入内存，修改内存对应部分，将修改后的内容写入到页缓存，页缓存写回磁盘。这就导致很多数据被缓存了两次，浪费内存。



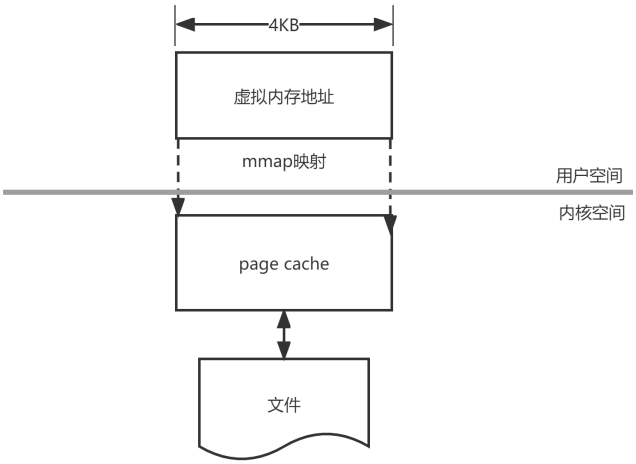
为了提高效率，在该系统的设计中剔除了 buffer cache，**仅使用** page cache，从而减少了数据的缓冲次数。

由于磁盘 I/O 对系统性能影响较大，因此在 page cache 的设计中选择了 cache 命中率较高的 LRU **淘汰算法**，而没有选择 random 淘汰算法。



由于大多数操作可以由处在内存中的 page cache 满足，而不需要进行磁盘 I/O 操作，因此在普通读写中可通过 page cache 提高 I/O 速度。

为了再次减少内存拷贝的次数，可以使用 mmap 将 page cache 的物理页面**直接映射**到应用程序的地址空间内，从而实现更加高效的文件 I/O 操作。



5. FAT32 文件系统驱动

FAT32 文件系统驱动采用的是嵌入式系统常用的 FatFs **文件系统**驱动：http://elm-chan.org/fsw/ff/00index_e.html

FatFs 提供了完备的 FAT 系列文件系统操作接口，支持 FAT16、FAT32、exFAT。但是该文件系统驱动提供的接口接近 Windows 驱动的接口，因此在该系统中对该驱动进行了多层包装，最终对应用程序**提供类 Unix 的文件系统操作接口**。

6. sd 卡驱动

sd 卡驱动移植自 fu540 bootloader：https://github.com/sifive/freedom-u540-c000-bootloader，**支持通过 SPI 以忙等待的方式与 SD 卡通信，暂不支持 DMA 方式读写**。

SD 卡通信协议参考了 SD Specifications 的物理层部分。

SPI 地址参考自 fu740 手册。

代码位置：os/driver/sd.c

（1）sd 卡驱动初始化

首先将 SPI **内存区域**映射进内核地址空间，然后通过 SPI 给 SD 卡**发送启动命令**，最后重新设置分频，提高 I/O 性能。

初始化部分代码如下：

sd 卡驱动初始化

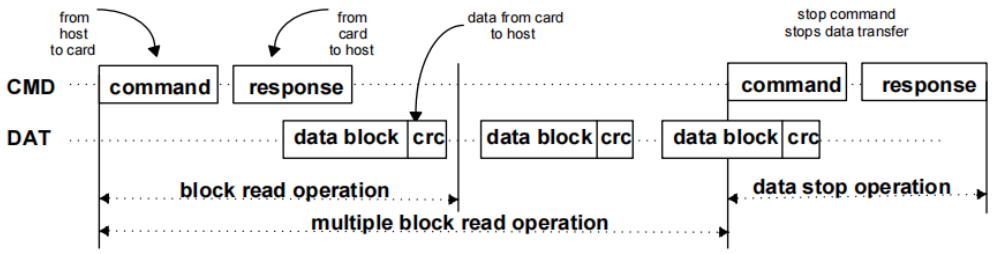
```
int sd_init(spi_ctrl* spi, unsigned int input_clk_khz, int
    skip_sd_init_commands)
{
  // Skip SD initialization commands if already done earlier and only set
      the
  // clock divider for data transfer.
  if (!skip_sd_init_commands) {
    sd_poweron(spi, input_clk_khz);
    if (sd_cmd0(spi)) return SD_INIT_ERROR_CMD0;
    if (sd_cmd8(spi)) return SD_INIT_ERROR_CMD8;
    if (sd_acmd41(spi)) return SD_INIT_ERROR_ACMD41;
    if (sd_cmd58(spi)) return SD_INIT_ERROR_CMD58;
    if (sd_cmd16(spi)) return SD_INIT_ERROR_CMD16;
  }
  // Increase clock frequency after initialization for higher performance.
  spi->sckdiv = spi_min_clk_divisor(input_clk_khz, SD_POST_INIT_CLK_KHZ);
  return 0;
}
```

（2）block I/O 方法

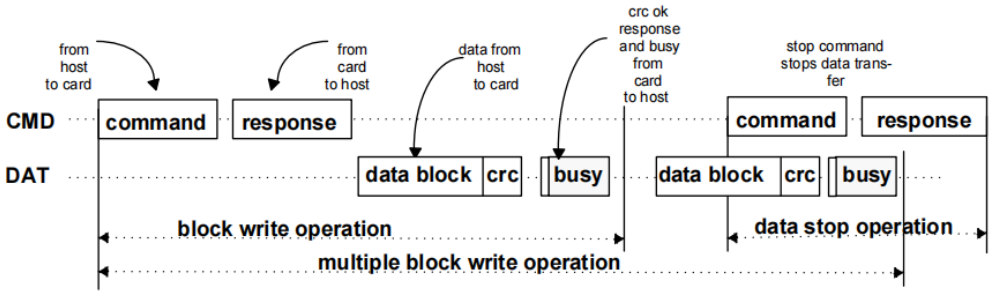
参考 SD 卡手册，通过 SPI **发送相应的指令序列**，通过**忙等待**的方式实现 block I/O。

相关函数为：sd_read_blocks、sd_write_blocks

读操作：



写操作：



需要注意以下两点：

- 为了防止对该段内存的读写操作被 cache，从而使得 SPI 接收不到数据，需要使用 fence 指令刷新 cache。
- 经过测试，该驱动工作正常，但是由于评测机将 sd 卡数据映射到了内存的 0x90000000 位置，内存 I/O 速度远高于 SD 卡 I/O 速度，因此为了提升测评速度，在评测机中并没有使用到该驱动。

7. rootfs 布局

lib 目录及其内部符号链接、dev 目录及其内部设备文件、tmp 目录，均由系统启动后的首个进程：test_runner 建立，test_runner 在建立好以下目录结构后，会逐一运行 libc-test 测试点。

在 lib 目录存储的是符号链接，指向根目录的动态链接库文件。
在 dev 目录存储的是设备文件，libc-test 中使用的设备文件主要是 null 和 zero。
tmp 目录用于给 libc-test 创建临时文件使用。
布局示意图如下：

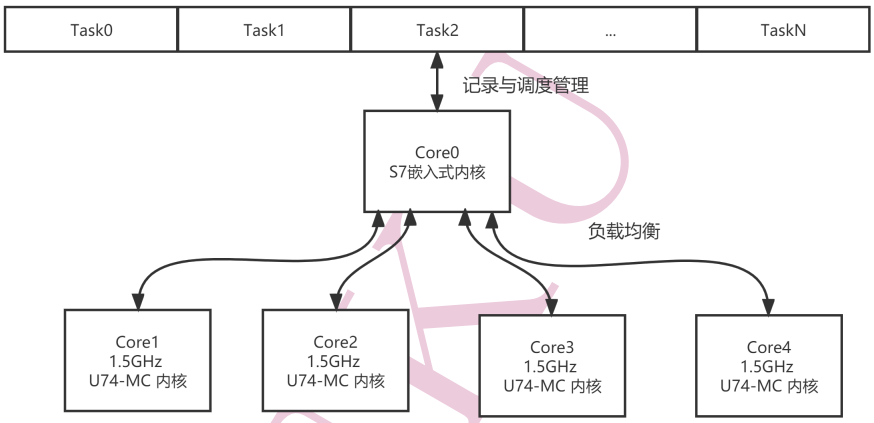
rootfs 布局

```
/
|-- /lib
|     |-- ld-musl-riscv64-sf.so.1     -> /libc.so
|     |-- libdlopen_dso.so            -> /libdlopen_dso.so
|     |-- libtls_align_dso.so         -> /libtls_align_dso.so
|     |-- libtls_get_new-dtv_dso.so   -> /libtls_get_new-dtv_dso.so
|     \-- libtls_init_dso.so          -> /libtls_init_dso.so
|
|-- /dev
|     |-- console   (major:1, minor:0)
|     |-- cpu       (major:2, minor:0)
|     |-- mem       (major:3, minor:0)
|     |-- proc      (major:4, minor:0)
|     |-- null      (major:5, minor:0)
|     \-- zero      (major:6, minor:0)
|
|-- /tmp
|     \-- libc-test生成的临时文件
|
```

```
|-- runtest.exe
|-- entry-static.exe
|-- entry-dynamic.exe
|-- libc.so
|-- libdlopen_dso.so
|-- libtls_align_dso.so
|-- libtls_get_new-dtv_dso.so
\-- libtls_init_dso.so
```

五、 核之间的进程调度

随着工作的不断进行，我们意识到目前核间的进程调度具有较大的弊端。由于时间因素，考虑到可能难以在规定赛程顺利实现，故这里我们提出内核间进程调度的设计方案：



总体而言，该设计方案的核心思想即为利用小核来运行进程调度管理相关的算法以及记录各个核的状态，不用于实际执行用户程序。

由于当前的调度算法基本可以概括为只要产生一个新的进程，各个核心都会抢占式调度该进程的内容执行，但是如此忽略了各个核心之间的负载均衡，很有可能会导致某些核心出现不断切换进程，进行访存等长时期操作，导致核心算力浪费，而当前考虑利用 5 个核中的简单核心负责调度其余四个复杂核心上的进程内容，因为进程调度以及切换主要是在进行访存切换等操作，对于核心的算力要求并不强，所以采取小核进行进程调度是一个可行的方法。同时还可以利用小核的调度来平衡四个核的负载均衡，而每一个核的负载计算程度，则可以借助 LLCMR 该参数来进行衡量，有的核可能相较于其他的核而言是 I/O 密集型，而有的核则相对而言是计算密集型。

利用变量：LLCMR 可以表示任务的行为特征（LLCMR），LLCMR 表示缺失次数在最后一级 cache 总的访问次数比例，LLCMR 越低，任务内存访问时间更短，代表计算密集程度更高，更受益于复杂核心。LLCMR 越高，任务访问内存越多，因此则代表其为 I/O 密集型。

若某核计算密集程度较高，则说明该核负载的计算过于紧密，需要适当与别的核调度进程协调，在新进程创建时，则不应该继续向计算密集型过高的核心安排对于新进程内容的执行。

利用核的历史任务记录情况判断核为计算密集型或者 I/O 密集型，LLCMR 直接利用未命中次数进行计算 (简单考虑可以利用 page cache)，简单进行衡量，若 LLCMR 值相对较低，则该核心相对于别的核为计算密集型，应适当将进程调度至其余 LLCMR 值较高的核心。

算法基本描述：

对于四个复杂核心，都对应维护其各自的 LLCMR 值，同时调度算法运行在简单核心之上，目的是维持四个复杂核心的 LLCMR 值基本保持在一致水平，若某核心 LLCMR 值过高，则产生新进程时，应该由小核心优先将该新进程分配给 LLCMR 值较大的核心。反之，若 LLCMR 值较小，则尽量避免将新进程分配给该核心，以此保证四个核心的 LLCMR 值基本维持在一个相对稳定的水平之上。

六、 工作总结

ShouChun OS 使用 C 语言进行实现，可运行于多核硬件平台和 QEMU 模拟器上。已经该系统目前已经实现的功能有：

- 启动和系统初始化
- 内存管理
- 中断异常机制
- 进程管理
- FAT32 文件系统
- SD 卡驱动
- 系统调用
- 动态链接

其中系统调用并未完全实现比赛需求，仍需继续改进。

注：本设计文档的撰写参考了 HiFive Unmatched **产品手册**及清华大学陶天骅的 uCore-SMP **设计文档**。