StartOS Book

王辉宇

5 31，2021

目录

Chapter 1 整体架构

操作系统的一个关键作用就是同时支持几个支持运行多个进程。例如 shell，cat。操作系统必须要在这些进程之间共享计算机的资源。例如，即使进程的数量多于硬件 CPU 的数量，操作系统也必须保证所有进程都会执行。除此之外，操作系统也需要隔离不同的进程，以防止故障在进程之间传播。然而隔离性太强了也不行，因为进程之间也需要交互，例如管道。因此，一个操作系统需要满足下面 3 个要求：多路复用、隔离和交互。[1]

这一小节将介绍 StartOS 的如何实现多路复用，隔离与交互，以及进程的概述，包括进程的生命周期和多进程切换。

StartOS 运行在 RISC-V 处理器(K210)上，RSC-V 是 64 位 CPU，StartOS 主要使用 C++ 进行开发，并且包含一些必要的汇编代码。阅读本文档需要有以下基础：懂一些机器代码，了解 RISC-V 特权架构和虚拟内存实现，并且有对操作系统内核开发有过一定了解，本文档暂时不会对一些基本概念进行解释，所以这不是面向初学者的文档。

1.1 特权模式

如果应用程序发生错误，我们不希望操作系统崩溃，同样的也不希望其他程序崩溃，这就要求我们应该将应用程序与内核隔离，应用程序与应用程序之间进行隔离。为了实现隔离性，内核必须安排应用程序不能修改操作系统的内存，不能访问其他应用程序的内存。

RISC-V 提供实现隔离的硬件，它拥有三种模式，**机器模式、监督者模式和用户模式。**机器模式拥有最高的权限，它可以执行其他模式不能执行的指令，例如读取 hartid。监督者模式拥有仅此于机器模式的权限，用户模式的权限是最低的，StartOS 中 SBI 运行在机器模式下，kernel 运行在监督者模式下，应用程序运行在用户模式下。

CPU 在机器模式下启动，StartOS 中它首先会运行 sbi，sbi 主要用于配置 CPU，并向内核提供部分服务。Sbi 配置完成会就会装换为监督者模式，并跳转到内核代码。

监督者模式拥有部分特权指令的执行权力，例如，开关中断，读写页表寄存器（k210 不允许）。用户模式下不允许执行特权指令，当在用户模式的应用程序下执行特权指令时，会引起异常并交由内核处理，内核会 kill 这些“危险分子”，因为他们做了不该做的事情。

1.2 代码结构

内核主要包含以下模块:

```
kernel/
├─── boot  启动内核以及初始化其他模块
├─── common  一些工具函数，例如日志，格式化输出
├─── device  设备相关代码
├─── driver   驱动相关代码
├─── fs   文件系统，包含 vfs 和 FAT32
├─── include  包含所有代码的头文件，其目录结构和 kernel/保持一致
├─── memory 内存管理和虚拟内存实现
└─── os:     内核核心部分，包括进程调度，进程的创建以及退出
```

如果只是想快速了解 StartOS 的各个模块，只需要重点关注 include 目录，其下的头件包含着比较详细的注释。

1.3 进程

StartOS 中进程隔离性主要通过页表实现，页表为每个进程提供虚拟地址空间，进程的虚拟空间布局如下图所示:

进程的用户空间地址 是从虚拟地址 0 开始 。指令存放在最前面 ， 然后是全局变量，然后是栈，最后是堆区，进程可以根据需要扩展（sbrk、mmap）。在地址空间的最顶端，StartOS 保留了两页，trampoline 和 trapframe，前者存放着用来跳转到内核中的指令序列，后者保存进程的部分信息。

```
// 进程
class Task {
 public:
  SpinLock lock;                // 进程锁
  enum procstate state;         // 进程的状态
  Task *parent;                 // 父进程
  void *chan;                   // 如果非空，将在chan睡眠
  int killed;                   // 如果非空，将被杀死
  int xstate;                   // 返回给父进程的退出状态
  int pid;                      // 进程ID
  struct trapframe *trapframe;  // trampoline.S保存进程数据在这里
  pagetable_t pagetable;        // 用户页表
  // struct inode *current_dir;       // 当前目录
  struct file *openFiles[NOFILE];  // 用户打开文件，其下标为文件描述符。
  char currentDir[MAXPATH];
  uint64_t entry;
  int sticks;  // 程序在用户态下运行的时间
  int uticks;  // 程序在内核态下运行的时间
  struct vma *vma[NOMMAPFILE];
  uint64_t kstack;          // 进程的内核空间栈。
  struct context context;  // 被保存的寄存器，用于pswitch
  char name[16];            // 进程名
  int sz;                   // 进程使用空间的大小
};
```

StartOS 的代码中没有使用进程这一名称，而是使用了 Task，它们之间除了名称不同其他概念都是一致的，使用 Task 只是因为 Task 更适合作为一个类的名称。

StartOS 中进程拥有许多状态，其中比较重要的状态是 pagetable，kstack，context。这些决定了一个进程是否能够运行。

p->state 表示进程当前的运行状态：UNUSED，SLEEPING，RUNNABLE，RUNNING，ZOMBIE。

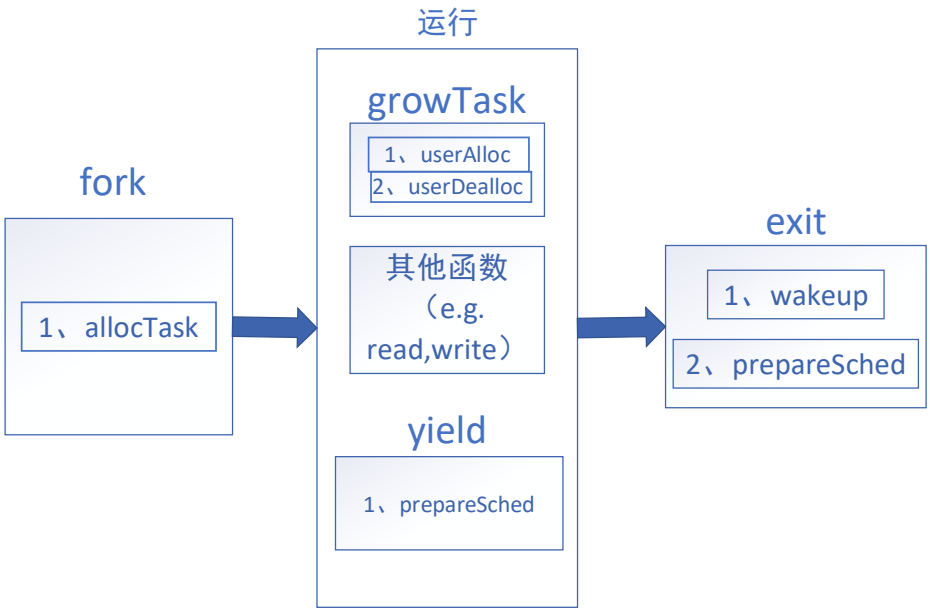
p->pagetable 保存着进程的页表，也就是它的地址空间，在运行某个进程前，会将页表寄存器 **stap** 设置为该进程。页表也会记录分配给该进程内存的物理内存的物理页地址。

1.4 进程的生命周期

为了使读者能够理解 StartOS 的进程模块，这一小节会将会概述进程的生命周期中涉及到的函数。

在 StartOS 进程的创建不是动态创建的，它定义了一个 Task 数组，内核在运行时分配的 Task，均从这里获取。

StartOS 中进程生命周期及其相关函数如下图。



1.4.1 Code: allocTask

allocTask 的主要作用是申请一个 Task，并为它分配必备的资源：trapframe，页表。

```
task->trapframe = (struct trapframe *)memAllocator.alloc();

// 为进程创建页表
task->pagetable = taskPagetable(task);

memset(&task->context, 0, sizeof(task->context));
memset(task->trapframe, 0, sizeof(*task->trapframe));

task->context.sp = task->kstack + PGSIZE;
task->context.ra = (uint64_t)forkret;
```

首先它会申请一页内存将其作为 **task->trapframe**，然后通过 **taskPagetable** 为该进程创建一个可用的页表。

taskPagetable 将页表的 **MAXVA** 下面的第一页映射到 **trampoline**，下面第二页映射到 **task->trapframe**.中。

然后就是将 **task->context** 和 **task->trapframe** 置零。接着就是为该进程设置内核栈，进程在内核中运行时会使用 **task->kstack**。最后就是设置进程运行的第一个函数为 **forkret**。StartOS 中所有进程运行的第一个函数都是 **forkret**。

1.4.2 Code: fork

fork 函数会创建一个新的子进程，并且子进程与父进程几乎完全一样，子进程返回 0，父进程返回-1。

Fork 函数首先会使用 **allocTask** 创建一个 **childTask**，然后就是复制父进程的资源和状态到子进程。其中包括页表，进程用户空间的寄存器，vma，打开的文件，到这里父进程和子进程就完全一致了。例如，父子进程的同一个虚拟地址的值相同（物理地址不同），同一个 fd 引用的文件相同(共享文件偏移量)。

接着设置 **childTask->trapframe->a0 = 0**，以保证子进程的 **fork** 函数返回 0。

最后就是将子进程的 **state** 设置为 **RUNNABLE**。这样就能够保证子进程可以被调度线程运行。

1.4.3 Code: growTask

growTask 函数用来增长和收缩进程内存，**growTask** 调用 **userAlloc** 或者 **userDealloc**，这取决对于 n 为正数或者负数。

userAlloc 通过 **memAllocator.alloc** 申请物理内存，并将其映射到进程相应的虚拟地址空间。**userDealloc** 通过调用 **userUnmap** 来释放相应的物理内存，**userUnmap** 会通过 **walk** 函数来查找对应的物理地址，并通过 **memAllocator.free** 来释放对应的物理内存。

1.4.4 Code: exit(int status)

exit 函数的的主要作用是记录退出状态码，释放部分资源，将自己的子进程交给 init 进程，设置自己的状态为 **zombie**，唤醒父进程回收自己，永久让出 CPU。

1.5 StartOS 启动序列

为了使 StartOS 更加具体，我们将概述内核如何启动和运行第一个进程。后面将会比较详细的 描述这个过程中出现的机制。

RISC-V 启动时，它首先会初始化自己，此时 CPU 处于机器模式下，完成后会运行 SBI，SBI 执行 一些 机器模式下才允许的配置，然后切换到监督者模式 ，并运行内核。为了计进入监督者模式，RISC-V 提供了指令 mret。这条指令通常用于从上一次的监督者到机器模式的调用返回。运行内核并不是从这样的调用返回，而是将寄存器设置的发生过这样调用一样：它在寄存器 mstatus 将上次的特权模式设置为监督则模式 ，然后通过把内核入口地址 0x80200000[1]写入到寄存器 mepc 中，将返回的地址设置为内核入口地址。然后会将部分中断和异常委托给特权模式[2]。

内核的入口地址为**_entry**，它的作用是设置一个栈，这样 StartOS 就可以运行 C++代码了。然后跳转到 **main**(main.cpp)函数，在 main 函数中初始化设备的和子系统后，它通过调用 **initFirstTask**(TaskScheduler.cpp) 来 创 建 第 一 个 进 程 。 第 一 个 进 程 执 行 程 序

[1] QEMU 中为 0x80200000，k210 中为 0x80020000

[2] K210 中时钟中断和外部中断只能在机器模式下处理。

init.c(user/init.c)，init.c 中会通过文件描述符 0 打开控制台设备，并复制它到文件描述符 1 和 2，将其作为标准输入，标准输出，标准错误输出。这样系统就启动了。

Chapter 2 时间管理

时间管理在内核中的内核中占有非常重要的地位，内核中的很多函数都需要定时执行(StartOS 中目前只有进程调度)。除了管理定时任务的时间之外，内核还需要管理当前日期和时间。StartOS 中主要使用 **Timer**(定时器)来管理定时任务，使用 RTC(实时时钟)来管理日期时间。在谈论这两个硬件设备之前，需要项理解 StartOS 中的 trap，方便读者理解后面的定时器中断。由于现在 StartOS 中 trap 机制沿用的 XV6 的机制，所以这里就先借用 XV6-book-2020 的第四节来说明 StartOS 中的 trap。

2.1 Trap（XV6-book-2020: chapter 4 section 1）

本小节使用 **trap** 作为系统调用、异常、中断的通用术语。

每个 RISC-V CPU 都有一组控制寄存器，内核写入这些寄存器来告诉 CPU 如何处理 trap，内核可以通过读取这些寄存器来识别已经发生的 trap。RISV-V 文档中包含的详细的描述。

S 态控制寄存器：

- **stvec**：这里存放着 trap 处理函数的地址，RISC-V CPU 会到这里来处理 trap。
- **sepc**：发生 trap 时，RISC-V CPU 会将 **PC 寄存器**保存在这里(因为 PC 会被 **stvec** 存放的地址覆盖掉)。sret(从 trap 中返回)指令将 **sepc** 复制到 **pc** 寄存器内核可以写 sepc 来控制 sret 返回的地址。
- **scause**：RISC-V CPU 会在这里存放一个数字，用于描述 trap 的原因
- **sscratch**：内核会使用这个寄存器来存放某些地址，这可以很方便的在 trap 恢复和储存用户上下文。
- **sstatus**：**sstatus** 中的 **SIE** 位控制设备中断是否被启用，如果内核清除 **SIE**，RISC-V 将推迟设备中断，直到内核设置 **SIE**。**SPP** 位表示 trap 是来自用户模式还是监督者模式，并控制 **sret** 返回到什么模式。

上述寄存器与监督者模式下处理的 trap 有关，用户模式不能读/写这些寄存器。对于及其模式下处理的 trap，有一组等效的控制寄存器；SBI 会使用这些寄存器为内核提供服务。

多核芯片上的每个 CPU 都有自己的一组这些寄存器，而且在任何时候都可能有多个 CPU 在处理 trap。

当需要执行 trap 时，RISC-V 硬件对所有的 trap 类型（除定时器中断外）进行以下操作：

1. 如果该 trap 是设备中断，且 **sstatus SIE** 位为 0，则不要执行以下任何操作。
2. 通过清除 SIE 来禁用中断。

3. 复制 pc 到 **sepc**
4. 将当前模式(用户或监督者)保存在 sstatus 的 **SPP** 位。
5. 在 **scause** 设置该次 trap 的原因。
6. 将模式转换为监督者。
7. 将 **stvec** 复制到 pc。
8. 执行新的 pc

注意，CPU 不会切换到内核页表，不会切换到内核中的栈，也不会保存 pc 以外的任何寄存器。内核软件必须执行这些任务。CPU 在 trap 期间做很少的工作的一个原因是为了给软件提供灵活性，例如，一些操作系统在某些情况下不需要页表切换，这可以提高性能。

你可能会想 CPU 的 trap 处理流程是否可以进一步简化。例如，假设 CPU 没有切换程序计数器（pc）。那么 trap 可以切换到监督者模式时，还在运行用户指令。这些用户指令可以打破用户空间/内核空间的隔离，例如通过修改 satp 寄存器指向一个允许访问所有物理内存的页表。因此，CPU 必须切换到内核指定的指令地址，即 **stvec**。

2.2 定时器

定时器的主要作用就是计算流逝的时间，它以某种指定的频率自行触发时钟中断，该频率可以通过内核写入相应寄存器来控制。由于时钟中断的发生间隔是受内核控制的，通常这个间隔被称为 tick。内核也将 tick 来作为时间的最小粒度。这个间隔时间是定义在 param.hpp 中，其值为 100ms。在 StartOS 中使用 **ticks**(Timer.cpp)来记录自系统启动后发生的 tick 数量。

StartOS 中使用定时器的主要原因是切换正在运行的进程；在 **usertrap**(trap.cpp)和 **kerneltrap**(trap.cpp)中会在发生时钟中断时调用 **yield** 函数来切换正在运行的进程。

由于 RISC-V 的时钟中断必须在机器模式下处理，因此 StartOS 会在 SBI 中处理定时器中断，SBI 在处理定时器中断只会触发一个软件中断，让内核进行处理。内核处理时钟中断会做下面 3 件事:

- 增加 **ticks**，并唤醒休眠在 **ticks** 的进程
- 若法时间中断之前运行的内核代码，则增加进程 **sticks**[3]，若运行的用户代码则增加进程 **uticks**。
- 配置下一次中断，由于只能在机器模式下配置定时器中断，所以这里需要调用 SBI 提供的 **timer_set_timer** 接口，来配置下一次中断。

[3] Uticks 和 sticks 分别代表进程在用户空间和内核空间的运行时间。

```
void init() {
  ticks = 0;
  spinLock.init("timer");
  uint32_t freq = sysctl_clock_get_freq(sysctl_clock_t((int)(timer::
TIMER_DEVICE_1) + (int)SYSCTL_CLOCK_TIMER0));
  uint64_t tmp = freq * INTERVAL;
 interval = tmp / 1000;
  setTimeout();
}
```

内核在初始化定时器时，会首先获取定时器硬件的频率，然后通过该频率计算一个 tick 的时钟周期数，并将计算得到周期数保存在 **interval** 中。然后调用 **setTimeout** 设置下一次发生时间中断的时间。**SetTimeout** 通过调用 SBI 提供的 **sbi_set_timer** 设置下一次时钟中断发生的时间。

2. 3 实时时钟

内核中的实际时间是通过 **RTC**(实时时钟)进行维护的，RTC 设置时间后本身就具有计时功能，内核只需要初始化为它设置一个初始时间就好了，后续只需要通过驱动读取实际时间就可以了，需要注意的是 RTC 的时间最小粒度是秒，如果需要 ms/us 级时间戳，是要额外设置时钟周期，并读取相应寄存器计算。

RTC 主要有以下功能：

- 可使用外部高频晶振进行计时
- 可配置外部晶振频率与分频
- 支持万年历配置，可配置的项目包含世纪、年、月、日、时、分、秒与 星期
- 可按秒进行计时，并查询当前时刻；
- 支持设置一组闹钟，可配置的项目包含年、月、日、时、分、秒，闹钟 到达时触发中断；
- 中断可配置，支持每日、每时、每分、每秒触发中断；
- 可读出小于 1 秒的计数器计数值，最小刻度单位为外部晶振的单个周期
- 上电/复位后数据清零。

RTC 通过 Clock.cpp 对内核提供时间服务，Clock.cpp 并没有做什么只是简单的调用驱动提供的接口，它只为内核提供两个接口，一个是获取当前的年月日时分秒格式的时间，另一个是获取当前时间戳。

Chapter 3 内存管理

本节与 xv6-book 第 3 节一致，只是将函数名替换为 StartOS 中的函数名，并重新绘制了 K210 下的内存布局。

页表是操作系统为每个进程提供私有地址空间和内存的机制。页表决定了内存地址的含义，以及物理内存的 那些部分可以被访问。它们运行 StartOS 隔离不同进程的地址空间，并将它们映射到物理内存上(trampoline 页)，以及使用未映射页来保护内核和用户的栈。本章将解释 StartOS 如何使用它们。

3.1 内核地址空间

StartOS 中会为内核维护一个地址空间的页表，内核会在启动时配置其地址空间的布局，使其能够通过虚拟地址访问到物理内存和各种硬件资源。下图展示了内核是如何将内核虚拟地址映射到物理地址的。文件(kernel/include/memlayout.hpp)声明了 StartOS 中内核内存布局的常量。

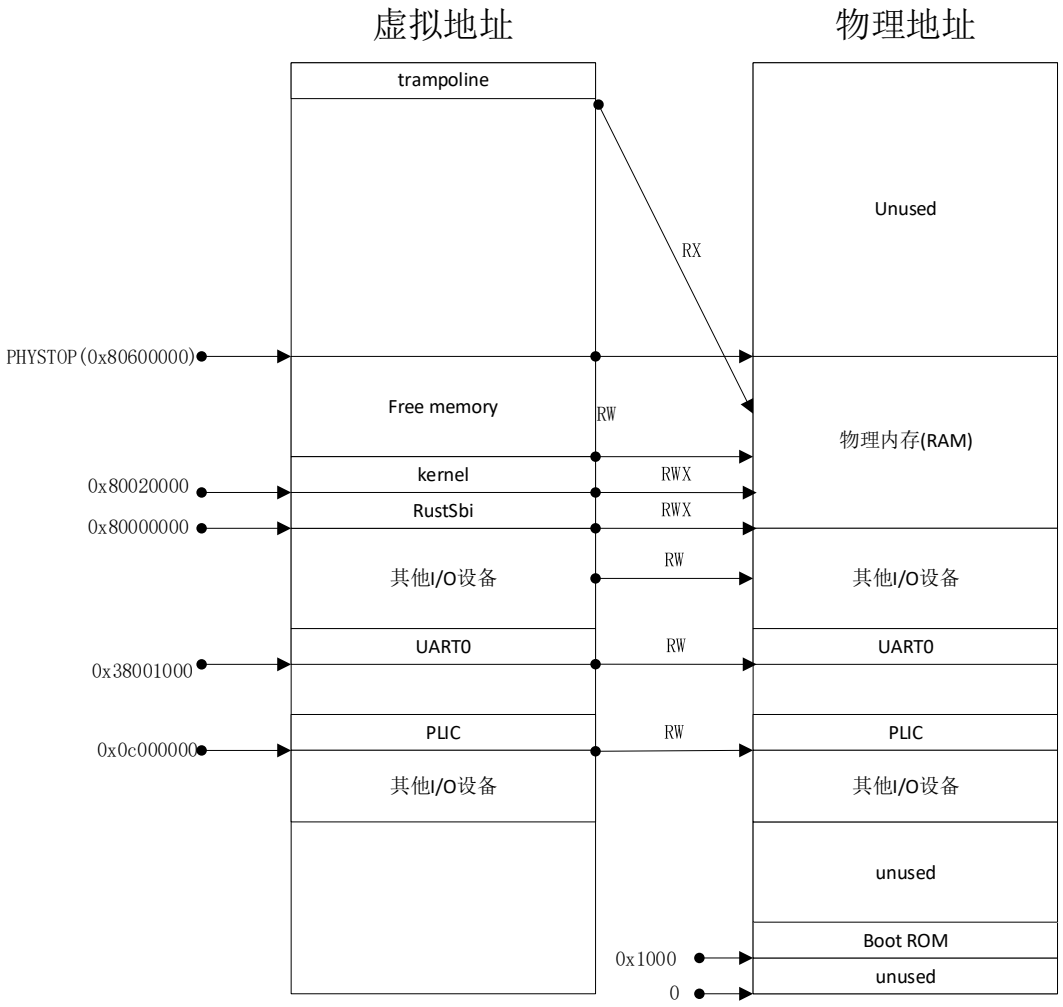


图 1 k210 内存布局

K210 拥有 8MB 的 RAM(物理内存)，从 0x80000000 到 0x80600000。K210 中将设备接口作为内存映射控制寄存器暴露给软件，这些 寄存器位于物理地址 0x80000000 之下。内核可以通过读取/写入这些特殊的物理地址与设备进行交互。

内核“直接映射“RAM 和 memory-mapped 设备寄存器，也就是 虚拟地址上映射硬件资源，这些地址与物理地址相等。例如，内核本身在虚拟地址空间和物理地址空间的位置都是 KERNBASE=0x80000000。这将使得内核可以很便利读/写物理内存。例如，当 fork 为子进程分配用户内存时，分配器返回 内存的 物理地址；fork 在将父进程的用户内存复制到子进程时，直接使用改地址作为虚拟地址 。

trampoline 页不是直接映射的。它被映射在虚拟地址空间的地顶端；用户页表也有这个映射。存放 **trampoline** 代码的物理页在内核的虚拟地址空间映射了两次：一次是虚拟地址空间的地段，一次是直接映射。

内核为 **trampoline** 和 **text**(可执行程序代码段)会有 **PTE_R** 和 **PTE_X** 权限。内核从这些页读取和执行指令。内核映射的其他页会有 **PTE_R** 和 **PTE_W**，以便内核读写这些页的内存 。

3.3 Code: 创建地址空间

大部分用于操作地址空间和页表的 StartOS 代码都在 **vmManager.cpp** 中。核心数据结构是 **pagetable_t**，它实际上是一个指向 RISC-V 根页表页的指针；**pagetable_t** 可以是内核页表，也可以是进程的页表。核心函数是 **walk** 和 **mappages**，前者通过虚拟地址得到 PTE，后者将虚拟地址映射到物理地址。以 **kernel** 开头的函数操作内核页表；以 **user** 开头的函数操作用户页表；其他函数用于这两种页表。**copyout** 可以将内核数据复制到用户虚拟地址，**copyin** 可以将用户虚拟地址的数据复制到内核空间地址，用户虚拟地址由系统调用的参数指定；它们在 **vmManager.cpp** 中，因为它们需要显式转换这些地址，以便找到相应的物理内存。

在启动序列的前面，**main** 调用 **initkernelVm** (kernel/vmManager.cpp)来创建内核的页表。这个调用发生在 xv6 在 RISC-V 启用分页之前，所以地址直接指向物理内存。**initkernelVm** 首先分配一页物理内存来存放根页表页。然后调用 **Kernel_vm_map** 将内核所需要的硬件资源映射到物理地址。这些资源包括内核的指令和数据，**KERNBASE** 到 **PHYSTOP**（0x80600000）的物理内存，以及实际上是设备的内存范围。

Kernel_vm_map (kernel/vmManager.cpp)调用 **mappages** (kernel/vmManager.cpp)，它将一个虚拟地址范围映射到一个物理地址范围。它将范围内地址分割成多页（忽略余数），每次映射一页的顶端地址。对于每个要映射的虚拟地址（页的顶端地址），mapages 调用 walk 找到该地址的最后一级 PTE 的地址。然后，它配置 PTE，使其持有相关的物理页号、所需的权限(**PTE_W、PTE_X 和/或 PTE_R**)，以及 **PTE_V** 来标记 PTE 为有效。

walk (kernel/vm.c:72)模仿 RISC-V 分页硬件查找虚拟地址的 PTE(见图 3.2)。walk 每次降低 3 级页表的 9 位。它使用每一级的 9 位虚拟地址来查找下一级页表或最后一级（kernel/vmManager.cpp）的 PTE。如果 PTE 无效，那么所需的物理页还没有被分配；如果 **alloc** 参数被设置 **true**，**walk** 会分配一个新的页表页，并把它的物理地址放在 PTE 中。它返回 PTE 在树的最低层的地址。

main 调用 **kvminithart** (kernel/vmManager.cpp) 来映射内核页表。它将根页表页的物

理地址写入寄存器 **satp** 中。在这之后，CPU 将使用内核页表翻译地址。由于内核使用唯一映射，所以指令的虚拟地址将映射到正确的物理内存地址。

Init_first_stack(kernel/proc.c:26)，它由 **main** 调用，为每个进程分配一个内核栈。它将每个栈映射在 **KSTACK** 生成的虚拟地址上，这就为栈守护页留下了空间。**Kvmmap** 栈的虚拟地址映射到申请的物理内存上，然后调用 **kvminithart** 将内核页表重新加载到 **satp** 中，这样硬件就知道新的 PTE 了。

每个 RISC-V CPU 都会在 ***Translation Look-aside Buffer(TLB)***中缓存页表项，当 xv6 改变页表时，必须告诉 CPU 使相应的缓存 TLB 项无效。如果它不这样做，那么在以后的某个时刻，TLB 可能会使用一个旧的缓存映射，指向一个物理页，而这个物理页在此期间已经分配给了另一个进程，这样的话，一个进程可能会在其他进程的内存上“乱写乱画“。RISC-V 有一条指令 **sfence.vma**，可以刷新当前 CPU 的 TLB。xv6 在重新加载 satp 寄存器后，在 **kvminithart** 中执行 **sfence.vma**，也会在从内核空间返回用户空间前，切换到用户页表的 trampoline 代码中执行 **sfence.vma**(kernel/trampoline.S:79)。

3.4 物理内存分配

内核必须在运行时为页表、用户内存、内核堆栈和管道缓冲区分配和释放物理内存。xv6 使用内核地址结束到 **PHYSTOP** 之间的物理内存进行运行时分配。它每次分配和释放整个 4096 字节的页面。它通过保存空闲页链表，来记录哪些页是空闲的。分配包括从链表中删除一页；释放包括将释放的页面添加到空闲页链表中。

Chapter 4 虚拟文件系统

虚拟文件系统模块为内核和用户程序提供了文件与文件系统相关的接口。内核中包含的全部文件系统都通过 VFS 对外提供服务。这样的好处是可以通过统一的接口访问储存在不同文件系统/不同储存设备的文件，调用者不用处理多个文件系统之间的差异，极大提升了编程体验。

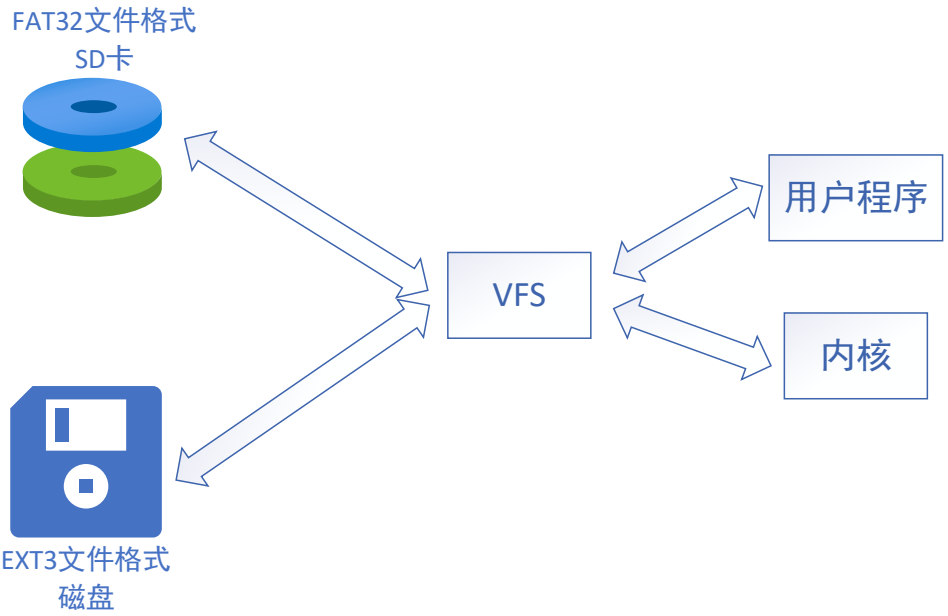


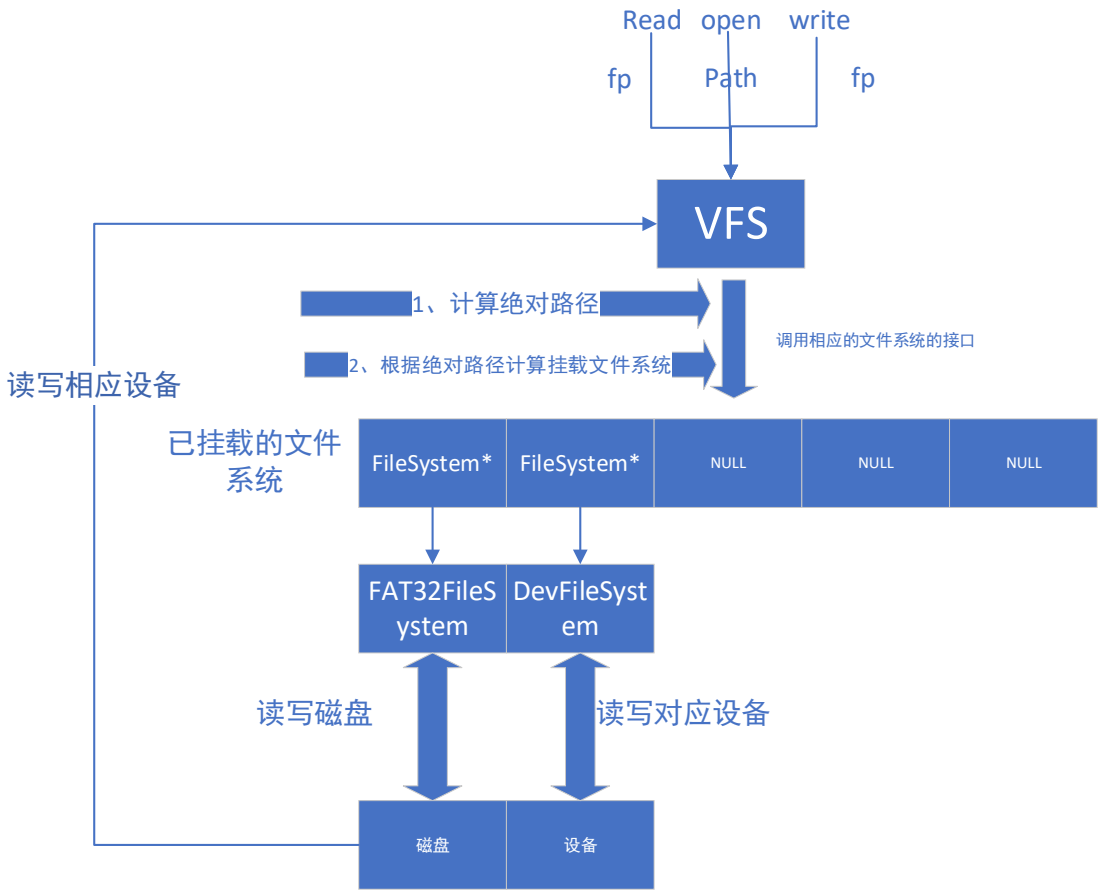
图 2 内核和用户程序通过调用 VFS 提供的接口来访问不同文件系统/不同储存介质中的文件

在 StartOS 中主要是通过**多态**来屏蔽不同类型文件系统的差异，VFS 模块中包含一个 **FileSystem**(vfs/FileSystem.hpp)抽象类，内核需要为每一种文件系统提供实现，目前 StartOS 中实现了 **DeviceFileSystem**(vfs/DeviceFileSystem.hpp) 和 **FAT32FileSystem** (vfs/Fat32File System.hpp)。

之所以可以多态来管理不同的文件系统，并提供一个统一的接口，因为 **FileSystem** 定义了所有文件系统都支持的、最基本的接口和数据结构。同时不同文件系统的实现也需要将自身的一些“如何打开文件“，“目录是什么“ 等概念与 VFS 的定义保持一致。因为文件系统的实现细节会在统一的接口和数据结构下被隐藏。其实也就是每个文件系统会为了兼容 VFS 会对本身的一些定义进行修改，这也就让使用者察觉不到不同文件文件系统的差异（差异被兼容了）。

VFS 不自己屏蔽不同储存介质的差异，这个差异应当由设备管理模块屏蔽，大部分的文件系统的储存介质都应该是块设备，例如 SD 卡，磁盘都属于块设备。设备管理应当为所有块设备提供一个统一的读写接口，让文件系统感知不到不同储存介质的差异。

下图较为详细的描述了 VFS 中包含的数据结构，以及如何处理 open,write 调用。VFS 中比较重要的数据是已经挂载到 VFS 上的文件系统数组。VFS 处理一个文件相关的调用时，会将首先计算出该文件的绝对路径，并根据绝对路径找出该文件所属的文件系统。然后再调用相关文件系统的接口，到这里，一个典型的文件相关调用就完成了。



现在举一个 open 调用的例子，来说明 VFS 具体是如何工作的。

在 StartOS 启动的时候会默认 挂载两个文件系统，挂载文件系统需要 3 个参数，文件系统类型，挂载点，挂载设备。

```
    mount(FileSystemType::DEVFS, "/dev", "");
    mount(FileSystemType::FAT32, "/", "/dev/hda1");
```

假设用户程序通过下面的方式打开一个文件:

```
    int fd = open("test.txt", O_RDONLY);
```

首先用户程序会通过 ecall 进入内核，并通过系统调用控制器，进入 **sys_open** 函数，该函数会首先对本次调用进行必要的参数校验(e.g. 文件名长度)。然后它会调用 VFS 提供的 **vfs::open** 函数，该函数首先会计算文件的绝对路径，计算绝对路径的方法比较简单，这里就不解释了。计算得到绝对路径为"/test.txt"(进程 pwd="/")，然后遍历所有挂载点判断哪一个文件系统是最匹配的，很明显"/test.txt"是属于根目录"/"，然后就调用对应挂载点上的文件系统即可。文件系统会通过挂载时设置的挂载设备路径来读写相应的储存介质。

Reference：

[1] xv6-book.
[2] The RISC-V instruction set manual: privileged architecture.
[3] The RISC-V instruction set manual: user-level ISA.
[4] Robert Love, Linux Kernel Development Third Edithion.