**StartOS Book**

王辉宇

5 31，2021

目录

[Chapter 1 整体架构 3](#_Toc73368817)

[1.1特权模式 3](#_Toc73368818)

[1.2代码结构 4](#_Toc73368819)

[1.3进程 4](#_Toc73368820)

[1.4进程的生命周期 5](#_Toc73368821)

[1.5 StartOS启动序列 7](#_Toc73368822)

[Chapter 2 时间管理 8](#_Toc73368823)

[2.1 Trap（XV6-book-2020: chapter 4 section 1） 8](#_Toc73368824)

[2.2 定时器 9](#_Toc73368825)

[2. 3实时时钟 10](#_Toc73368826)

[Chapter 3 内存管理 11](#_Toc73368827)

[3.1 内核地址空间 11](#_Toc73368828)

[3.3 Code: 创建地址空间 12](#_Toc73368829)

[3.4 物理内存分配 13](#_Toc73368830)

[Chapter 4 虚拟文件系统 14](#_Toc73368831)

## 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。



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]](#footnote-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]](#footnote-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**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]](#footnote-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。

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.](https://pdos.csail.mit.edu/6.828/2020/xv6/book-riscv-rev1.pdf)

[2] [The RISC-V instruction set manual: privileged architecture.](https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMFDQC-and-Priv-v1.11/riscv-privileged-20190608.pdf)

[3] [The RISC-V instruction set manual: user-level ISA.](https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf)

[4] Robert Love, Linux Kernel Development Third Edithion.

1. QEMU中为0x80200000，k210中为0x80020000 [↑](#footnote-ref-1)
2. K210中时钟中断和外部中断只能在机器模式下处理。 [↑](#footnote-ref-2)
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。 [↑](#footnote-ref-3)