

tatakOS 的设计与实现

一、比赛准备和调研

在准备比赛之初，我们有两个选择，一是从零开始实现一个操作系统，二是在已有开源的操作系统框架上进行更改，加入自己的想法与设计。经过讨论，由于完全从零开始工作量太大，而且一大部分工作属于重复造轮子，事倍功半，所以我们决定在一个开源操作系统的基础上进行更改。

经过调研，我们发现去年参赛的队伍有不少是在 `xv6` 和 `rCore` 的基础上更改而来的。前者是 `mit` 的教学用操作系统，而后者是清华的教学操作系统。

`rCore` 是使用 `rust` 语言编写的，我校另外一支队伍就选择了 `rCore`。由于我们想把更多精力更多的放在系统设计本身，对 `rust` 的不熟悉会导致在使用 `rust` 时引入许多不必要的语法噪音。所以我们最终选择了用 `ANSI-C` 编写的 `xv6` 作为我们的参考对象。

选择 `xv6` 的好处有：

1. 相比于重新开始学习 `rust`，我们对于 `c` 语言及其相应的工具链更加熟悉，对其行为能够更好的把握，出了问题更好调试；
2. 我们可以减少对语言特性的关注，更多的关注于项目本身的架构；
3. 少了前期语言的准备，时间上更加充分，我们可以实现更多的功能，做更多的优化。
4. 我们参考学习的对象 `linux` 主要使用 `c` 语言，在参考学习时可以减少语言的转换。
5. `Xv6` 在设计上相当简单，因此我们可以在这基础上做更多个性化的改造与设计，具有较大的创作空间。

此外 `xv6` 相应的课程实验可以帮助我们更快的理解操作系统本身，而且去年比赛的第二名 `xv6-k210` 正是使用 `xv6` 更改而来，可以说已经有了相关的成功经验。

二、系统框架和模块设计

系统框架

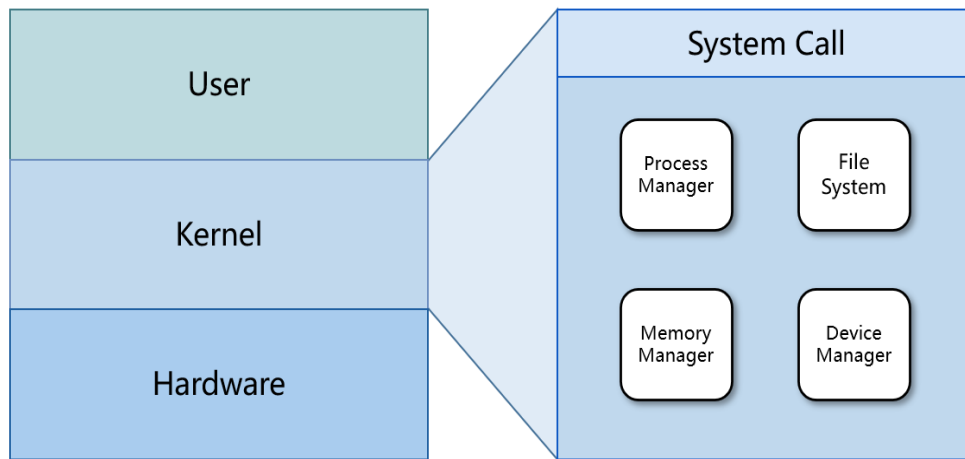


图 1 系统架构图

如图 1 所示，内核采用宏内核的设计架构，上层用户程序以及用户标准库通过系统调用的方式来要求内核提供功能服务，内核需要负责底层硬件的抽象以及相关资源的管理以便向上层提供统一的调用接口。为此，内核目前将服务划分为了四大四大子模块，分别为进程管理子系统、文件管理子系统、内存管理子系统、以及设备管理子系统等，对应抽象的资源为 CPU 资源、持久化资源、内存资源、设备资源等。

系统根据早期设计并不打算提供多指令集架构的支持，目前仅提供 riscv 的架构支持。不过，对于相同架构的平台开发板，能够针对性地开发不同 BSP 来支持上层的系统服务，目前系统已经能够支持在 qemu 以及 k210 的平台上正确运行。

模块设计

1. 内存管理模块

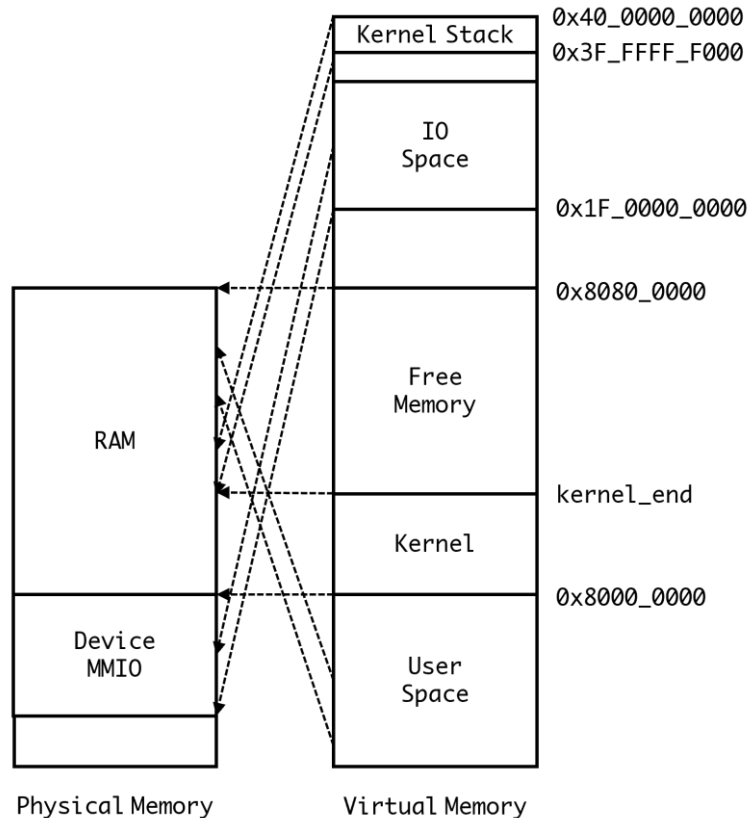


图 2 K210 内存映射图

系统的内存管理基于 riscv 虚拟地址的转换技术设计，用户与内核使用同一张页表来进行映射，以节省系统调用切换页表所带来的额外开销。如图 2 所示，内核空间位于 2GiB 虚拟地址，而用户空间则在这之下。为了给用户低地址空间腾出空间，我们需要将原来物理上处在低地址的 MMIO 通过页表“搬”至高地址的 IO 空间中。

2. 文件系统模块

系统提供了对 fat32 文件系统较为全面的支持，包括长短文件名支持等。考虑到当前仅仅只需支持 fat32 文件系统，引入 VFS 的意义不大，中间可能还会存在转换开销，因此目前系统仅支持 fat32 文件系统。不过出于对未来的长远考虑，在设计相关具体文件系统（fat32）API 时，我们也充分考虑了 VFS 这一方面，所以使其尽可能的简单、解耦，日后若想引进 VFS，实际的工作量其实相当之小。

此外，目前文件系统正在加入 `page cache` 缓存机制，用于加快文件的读写需求，此方面的文档有待补充。

3. 进程管理模块

系统目前仅仅支持最小以进程为单位的调度，不过为了方便和其他子系统配合，比如在文件系统中加入定期的页写回机制等等，我们在后续为维护中可能会加入对线程的支持，还有类似 `tasklet` 小任务的支持。

4. 设备管理模块

TODO

三、开发计划

我们计划开发后的操作系统，使用 `riscv` 架构，可以运行在 `k210` 板卡和 `qemu` 模拟器上；实现多种系统调用，支持上层用户程序；除此之外还要具有一定的效率和性能。综上所述我们的开发计划大致可以分为三种，分别是支持性开发，功能性开发和优化性开发。其中支持性开发是指让 `os` 正确运行在指定的架构和平台上，功能性开发是指给 `os` 添加对应的功能，比如添加更多的系统调用，优化性开发是指对于 `os` 的某个模块进行设计优化，以提高 `os` 的整体性能。支持性开发主要在开发前期进行，功能性开发主要在开发中期进行，而优化性开发主要在开发的中后期进行。

支持性开发计划

- 移植 `rustsbi`
- 移植 `xv6` 到 `k210` 板卡上
- 更改工程目录
- 自动化系统调用
- 性能分析工具
- 支持 `fat32` 文件系统

功能性开发计划

系统调用号	系统调用名	内核函数名
1	fork	sys_fork
93	exit	sys_exit
3	wait	sys_wait
6	kill	sys_kill
14	uptime	sys_uptime
101	nanosleep	sys_nanosleep
124	yield	sys_sched_yield
172	getpid	sys_getpid
173	getppid	sys_getppid
220	clone	sys_clone
221	exec	sys_exec
260	wait4	sys_wait4
17	getcwd	sys_getcwd
23	dup	sys_dup
24	dup3	sys_dup3
34	mkdirat	sys_mkdirat
35	unlinkat	sys_unlinkat
37	linkat	sys_linkat
39	umount2	sys_umount
40	mount	sys_mount
49	chdir	sys_chdir
56	openat	sys_openat
57	close	sys_close
59	pipe2	sys_pipe2
61	getdents64	sys_getdents64
63	read	sys_read

64	write	sys_write
80	fstat	sys_fstat
214	sbrk	sys_sbrk
215	munmap	sys_munmap
222	mmap	sys_mmap
153	times	sys_times
160	uname	sys_uname
169	gettimeofday	sys_gettimeofday
89	timetag	sys_timetag
90	bio_cache	sys_bio_cache
99	profile	sys_profile
100	memuse	sys_memuse

优化性开发计划

- 将 xv6 原来的用户内核双页表更改为单页表
- 优化设计文件缓存 (page cache)
- 添加 copy on write
- 添加 lazy allocation
- 优化内存分配方式 (buddy + slob)
- 优化磁盘驱动 (DMA)
- 优化 SD 卡驱动

四、 比赛过程中的重要进展

- 2022/3/26 装载 rustsbi
- 2022/3/31 重构项目工程目录
- 2022/4/3 引入 Linux 构建系统
- 2022/4/7 引入 SD 卡驱动
- 2022/4/16 添加对大页 (2MiB) 的支持
- 2022/4/23 添加对 COW 以及引入 Buddy 分配器
- 2022/5/2 添加 SLOB 分配器
- 2022/5/11 完成对 FAT32 文件系统的支持
- 2022/5/19 增加了对 DMA 的适配
- 2022/5/22 加入 Profiler 性能分析器
- ...未完待续

五、 遇到的主要问题和解决方法

- SBI 调用 setext 开启外部中断时, 会陷入死循环?

SBI 的问题，在 SBI 调用返回时没有给 PC+4，从而导致了循环进行 SBI 调用的死循环。

相关 PR: [fix: inc pc in emulate_xx_sext by yztz · Pull Request #2](#)

➤ 板子每次上电的运行行为不一致？

参考去年参赛队伍的经验以及自己的实际测试，定位问题为全局变量不会自动清零，手动初始化全局变量后问题得以解决。

➤ 用户态下发生 **illegal instruction**, SBI 的行为异常？

现象可以描述为：当在无论是用户态（U）下或是内核态下（S）执行到非法指令时，触发的都是内核态（S）下的错误。

发现此问题的背景为当错误地加载用户程序时，执行出错，结果最后触发了用户异常处理函数的“Not From User Mode”的 panic，当时也是困扰了许久，经过各种调试，最后确定了是 SBI 的坑。

在 SBI 转发指令错误的异常时，使用是如下设置：


```
mstatus::set_spp(SPP::Supervisor);
```

但是，实际情况不然，此异常可能还来自于用户态，所以我们可以稍加修改：

```
if ctx.mstatus.mpp() == MPP::User {  
    mstatus::set_spp(SPP::User);  
} else {  
    mstatus::set_spp(SPP::Supervisor);  
}
```

经过上述修改，以上的问题最终便得以解决了。

相关 issue: [关于非法指令的转发行为 · Issue #26](#)

➤ ioremap 引发的寄存器访问问题？

我们在将低地址的 MMIO 寄存器通过页表映射大页（2MB）的方式转移到高地址空间中时，发生了一些奇怪的**读取数值异常**的情况。经过排查，发现是因为映射的地址非页对齐的缘故。这算是一个小问题，但是也提醒了我们在处理内存页问题时，要特别注意地址的对齐问题。

➤ scause 读取错误问题？

在偶然一次调试输出中，我们发现我们触发一个理论上来讲根本就不会触发的异常。通过反复的 **review** 确认了不是其他代码的行为错误。通过比对 riscv 的中断异常表，我们发现我们期待的中断原因，变为了对应低位相同的异常原因，也就是说高位用于区分中断/异常的标志位变为了 0。

内核代码中我们引入了**单页表**的设计，从而导致用户态与核态之间不需要切换页表，也就是说不需要刷新 TLB 了，从而导致我们在写相关 PTE 后，需要手动刷新页表缓存才可。

➤ **潜在的 uvmmap 内存泄漏问题？**

在分析原有的 xv6 代码时，我注意到了 uvmmap 可能存的内存泄漏问题，在 map 失败时，之前映射成功的页没有被释放从而将会导致内存泄漏。

相关 PR: [\[fix\] mappages: potential memory leak by yztz · Pull Request #118](#)

六、 作品特征描述

- 内核空间与用户空间映射为在同一页表，系统调用不需要再刷新缓存，性能提高了 5 倍以上。
- 自己实现的 Buddy&Slob 内存分配器
- 支持 2MiB 大页映射
- COW&Lazy Allocation[TODO]
- 对 FAT32 长短文件名目录项的完整支持。
- 页缓存[TODO]&块缓存
- FAT 表专用缓存[TODO]
- DMA+SPI 全速传输
- 轻量又强大的 Profiler 内核性能分析器
- 自动化构建框架&系统调用自动化处理机制

七、 分工和协作

TODO

八、提交仓库目录和文件描述

工程目录树如下：

```
.
├── bootloader
├── build
├── doc
├── entry
├── include
│   ├── atomic
│   ├── driver
│   ├── kernel
│   ├── mm
│   └── fs
├── script
├── src
│   ├── atomic
│   ├── driver
│   ├── fs
│   ├── include
│   ├── kernel
│   ├── lib
│   ├── mm
│   └── platform
│       ├── k210
│       └── qemu
├── test
├── tools
└── user
```

下面对于每个工程目录下的文件进行一一讲解。

在 `bootloader` 目录下，主要存放的是 `rustsbi` 的相关内容，用作引导程序。分别有对应的 `k210` 和 `qemu` 版本。

`build` 目录在编译之前并不存在，是在编译后生成的新目录，主要用于存放编译后生成的中间文件。其目录结构和 `src` 类似。除此之外，编译生成的 `kernel` 镜像和 `qemu` 所使用的的文件镜像 `fs.img` 也存放在里面。

`doc` 目录存放文档。

`entry` 目录存放系统调用表，我们只需要在系统调用表里面加入系统调用号，系统调用名，函数名，即可自动化系统调用。相比于原来 `xv6` 添加系统调用的方式，这样做更加简单快捷。除此之外，`entry` 中还存放了分析函数表，用来进行性能分析。

`include` 目录下主要存放头文件，下面还对头文件依据其归属的内核模块进行了分类，分别为原子指令，驱动，文件系统，内存管理等。

`script` 目录下存放了各种脚本，比如链接脚本，`makefile` 脚本，`python` 脚本，这些脚本在内核编译时使用。

`src` 目录下存放内核源代码，这些源代码根据所属的内核模块进行了分类。其中 `platform` 下存放的是平台相关的源代码。在编译时会根据所属的平台进行选择编译。

`test` 目录下存放的是一些模块功能的测试代码，以及一些 `bug` 的修复记录。

`tools` 目录下存放的是烧录工具。

`user` 是用户程序所在的目录。

九、比赛收获

朱洋洋：这次比赛的收获是多个方面的。首先是专业知识上的收获，经过操作系统的理论学习，很多概念还是非常模糊，感觉自己似懂非懂，但是经过实际代码的实践，这些概念一个个的清晰起来了。之前从书本上学到的专业知识在项目工程中体现出来，做到了真正的理论与实践相结合；其次学会了合作开发，大型系统软件往往是一个团队合作开发的，我们要参与大型的项目，就必须学会和他人高效的沟通合作，这次和队友一起参加比赛，我们一起讨论解决bug，架构设计，一起想优化方案，让我体会到了团结合作的重要性，受益匪浅；再然后是提高了工程能力，参与一个较大型的项目的开发，需要综合考虑各个方面，不仅仅是写代码本身。这次比赛让我真正的理解了这一点。

杨宗振：此次比赛我收益良多。从大的系统的角度去看，此次比赛使得我能以更高的角度去俯视整个系统的运作，从一个系统的构建，开发到调试，我充分地认识到了系统开发生命周期各个阶段的难点，了解了系统在设计层面上面对具体场景时的取舍考量。从小的角度去看：硬件上，我熟悉了 riscv 的指令集以及特权级架构，学习实践了相关硬件驱动的开发；软件上，通过实际的编码工作，对平时操作系统课上学到的知识进行活学活用，深刻体会到了一系列算法与数据结构的意义与价值，并且举一反三，根据实际的场景，设计自己的数据结构与算法，不断地在原有的基础上进行迭代发展，从而取得更丰富的功能以及更出色的性能，这在我看来才是整个比赛最有意思的点。当然比赛也非一帆风顺，我们总能碰到各种各样诡谲古怪的bug，一次次的陷入失望、绝望，再一次次地从混沌失败的泥沼中艰难地爬出，享受着清脆悦耳的舒适感贯穿身心，就这样不断往复着，但就是在这些过程中，我才能一步步提高解决问题的能力，不断转变思考问题的角度，才能最终到体会比赛带给我的乐趣与价值。