评委老师们，专家们，大家好。我们是西北工业大学NPUcore-IMPACT小组。我是组长冯宜湑，主要负责系统设计与功能添加。这是我的组员张瀚宸，负责开发板验证。组员张逸飞因为学校课程无法来现场，他主要负责测例修复与EXT4适配。

我们将从以下四点进行介绍：

首先是内核概要，我们的内核IMPACT使用Rust编写，基于龙芯架构支持2K1000开发板及其虚拟机。我们另起炉灶从头写起，在初赛决赛均最高分，唯一libc-bench得分队伍，最终提交得分第一。

在我们的内核中，有四个主模块，分别命名为...

这个内核在LA赛道的得分前提，就是我们在kernel中独自实现了2K1000龙架构的移植，并封装为一个arch包。

我们也根据大赛要求成功支持FAT32/EXT4文件系统。

我们的内核优势是有着完善的功能，可以基本支持106个系统调用，这也是为什么我们在最终决赛上拿到了最高的分数。

在硬件上，我们适配支持了PCI驱动，可以从QEMU和开发板中读到测例。

我们也在仓库中提供了六个总计约100页的详细文档。

左上图是我们组与其它版本NPUcore的对比，我们有着不同的系统调用，不同的优化策略，不同的内存映射，以及不同的文件系统。接下来我们也会重点讲我们的这些区别。

左下图是我们内核成功进入命令行的截图

在从3月15日至今的开发中，我们的内核过程量代码约70000行，内核代码总数目前约为35000行，其中包括了5000行测试代码。总计在Github和gitlab中有188次commit。

接下来我会讲一些我们内核的优势和一些有意思的点。

首先是LA2K1000的适配。我们注意到，在目前网上公开的由Rust编写的仓库中，还没有基于龙芯指令集的crate包，这对于那些想用rust写操作系统的人提供了不小的难度。因此我们开源了我们的IMPACT版2K1000 ARCH包。在包中，我们提供了三个接口，分别是board、register和trap，对应龙芯文档中2K1000的串口，寄存器，和内核态陷入方面的规定。

第二个keypoint，是我们内核有着较为完善的功能。我们实现了106个syscall，采用静态查找的映射方式。利用rust register简化代码，对每一个syscall注册。有边的两个大表是我们根据龙芯Linux命名方式所整理的已实现的系统调用。每一行代表改调用所在的文件包。

特别说明一下，为什么我们要进行这么多的功能性实现。是因为在LA测例编译的时候，因为其特殊的LLVM，会将老的系统调用链接到很多全新且冷门的系统调用上。

这里我们会具体针对三个我们认为比较有意思的添加过程进行讲解。

第一我们要讲指令集相关的系统调用statx。我们初次发现这个系统调用是在初赛测例mmap与munmap中。我们在一个网盘链接中查找到，龙芯将fstat或newfstatat统一映射到statx中。我们将mmap与munmap反汇编后，也发现statx可以在静态字符段中找到，这说明龙芯的LLVM的确做了这个映射。于是我们仿照了RISCV的stat来实现statx。有图是我们实现的statx结构体，其与stat的区别在于支持更大的rdev和dev（major和minor），支持abcm四种time，并加入attributes支持了文件更多的状态信息。

第二是指令集相关系统调用Clone3。它在libc-bench用例中出现并大量使用。在最开始我们认为它无从下手。因为Clone只在LA-LLVM的闭源测试版中，在目前所有的发行版linux中均不存在。在man手册中可以看到一点端倪，就是Clone3和Clone视为同一个系统调用，但是我们仍然不知道它与clone的区别。继续寻找，我们在一个博客上找到Clone3 是为了克服flag bit用完，无法满足多种新增 Flags 而诞生的系统调用。也就是说我们需要添加新的FLAGs来将clone适配成clone3。最终，我们在Linux 内核开发人员 Michael Kerrisk to Christian Brauner 的一封 email查找到需要增加的FLAGs，并复用clone函数。这也可能是我们在LA赛道唯一通过libc-bench的原因。

第三是指令集无关系统调用。我们这里给出我们优化Openat系统调用的过程。Openat的目的是分配该进程还未使用的最小的文件描述符。根据需求，我们就能得到三种得到最小的解决方案。其一，在回收文件描述符后进行一次排序，这样就可以保证拿到的第一个fd是最小的。其二，回收时不进行排序，重新分配时使用 O(n) 时间寻找最小的文件描述符。这样也满足Openat系统调用。其三是我们认为最好的方法，使用二叉平衡树替换线性表，从而在 O(log n) 时间进行回收与重新分配，但也许会带来内存分配的额外开销。但使用lmbench的open测例，我们发现三者无明显差距。经比对测试，我们发现该测例的fd数值太小，且查找次数太少，无法反应openat速度。于是我们进行压力测试，将fd最大设置到1024，保证每次查找fd不小于512。最终发现，方法三比前两种快30倍与7倍。而最终在决赛提交时，我们仍选用方法一，因为他是基于这个测试最快的。左图是我们的实现。

第三个Keypoint，是我们组在内核中EXT4的适配过程。与其说是适配，不如说是重写。我们在网上虽然找到了一个几乎已经是配好的xv6-LA模板，但是为了体验这个过程，我们选择不基于该模板，而是做出匹配 NPUcore-IMPACT内核独立的轻量化EXT4。我们在调研后，最终根据RISCV往届优秀作品，选择了已有的C语言第三方包，使用bindgen制作FFI接口进行转译。Bindgen在llvm上层，因此接口形式无需考虑平台架构。我们前三个版本均适配失败，理由我们后面会分别讲述。而最终，我们参考LWEXT4的形式，自行开发了LWEXT4-NPUcore。

第一次适配，我们使用LA工具链编译了2016版LWEXT4-C，并用bindgen生成接口。但是遇到了no standard没有标准库问题。我们自己参考龙芯标准库，为其重写了libc库中的12个函数，大约2000行代码。但是可能是该LA标准库的宏信息与LWEXT4的宏不匹配，或者是接口不对应，导致我们并没有适配成功。

第二次适配，我们选择直接使用RUST版LWEXT4进行适配。该版本是2019年的lwext4在rust上的重写版，本质上实现的功能与C语言版相同，但是接口与NPUcore内衔接更友善。但是我们在其编译工具链中，发现明确说明只支持x86\RISCV\AARCH的情况。我们尝试只修改编译工具链为LA，发现宏信息不同，甚至在最开始编译都无法通过。我们也尝试修改了很多细节，但最终无果只好放弃了本仓库。

第三次适配，我们使用2024年由谷歌内部开发并开源的LWEXT4-VIEW仓库。该仓库由三个基本属性：只读、不崩溃、不panic、不无限循环、以及没有unsafe代码。我们在适配过程中发现，由于该项目仍在开发中，里面的实现没有well-test，不完善。但是由于刚才的属性都是LWEXT4没有的，因此它值得我们后续阶段的高度重视。

最终，我们决定，自己动手丰衣足食。我们参考LWEXT4，自己写了一版LWEXT4 - NPUcore。在这个过程中，我们也逐渐挖掘出了前面适配不成功的原因。第一步，我们需要生成满足LA 架构的LWEXT4文件系统。这里的生成我们仍然使用mkfs.ext4，但是区别在于，我们要设置很多特有的features以便能在这个特殊的LWEXT4上跑起来（如图）。而BLOCK SIZE必须设置为1024的倍数，且要和内核中的BLOCK SIZE存在二倍差异。

第二步，我们实现自定义的File trait，它可以理解为一层文件系统抽象，而这个trait相当于vfs，但它会比实现一个vfs代码量要少，节约成本，性价比高。 我们将文件系统从原先的 FAT32 转换为 lwext4 中的对应的 Inode 和FileSystem。根据这个trait，我们便可以修改根目录，适配PCI、mem和sata。

第三步也是最难的一步，我们要根据刚才的File trait接口，重构LWEXT4，并为我们NPUcore-IMPACT内核提供合适的接口。我们将 LWEXT4 中的 File 和 Directory 合并为 DirTree，来反向适配 NPUcore 中的 DirectoryTreeNode 数据结构。

我们的整个适配过程就结束了，这个总适配过程性代码量大约30000行。最下面是我们进入命令行界面的截图。

接下来由张瀚宸同学来讲解最后的外设支持部分。

各位评审老师好，接下来由我为各位讲解我们进行开发板验证的流程

首先我想先为大家介绍一下我们组上板的基本方案：

我们将操作系统镜像准备为文件系统镜像与内核镜像，首先将文件系统烧录至介质当中，再将内核镜像烧录入开发板内存。系统在内核初始化时自动识别我们的文件系统镜像，从而完成整个系统的初始化

下面是我们总结的，对于龙芯2K1000星云板的上板操作流程图。

大家可以看到，龙芯2K1000星云板配备了NAND子系统与SCSI子系统，我们共有三种方式进行文件的传输与修改，首先我们可以通过uboot对NAND与SCSI进行烧写，同时可以通过上位机驱动的方式对SSD进行操作。如若我们实现了网络协议栈，则可以通过tftp与初始化完成的内核进行通信

由此，我们列举了两种上板方案，有别于去年2K500的SD卡，今年的方案分别为NAND方案与SCSI方案，其对比可以看这张图，NAND方案通过uboot自动在NAND安装文件系统镜像并将内核烧入内存启动，SCSI方案则需要先烧写文件系统镜像入SSD再启动uboot烧写内核，且为了令内核读取SSD需要实现PCI与SATA驱动。这样固然会繁琐一些，但是大大增加了内核的可扩展性，我们经过思考，认为现场赛时间有限，优先选取第一种方案

但是，在进行NAND方案的尝试时，我们遇到了以下问题：

- NAND子系统缺失：大家可以看这张图，在执行BootMenu的指令宏时，开发板报错“Unknown command 'nand'”，这意味着有些开发板并没有实现NAND子系统

- NAND设备缺失与大小空间不足：各位可以看这张图，报错信息为“Partition not found or too large”，这意味着NAND设备分区并不存在，或者设备空间太小。我们发现了很有意思的一点，在uboot开机自检的时候，扫描出的NAND设备大小为零，且经过我们查证，预留给NAND子系统的地址空间仅有64MB，而复赛后测例大小已经达到120MB+，更不用提决赛的2GB的LTP测例。

因此，我们放弃了NAND方案

SCSI方案的进行也并非一帆风顺，我们上来就遇到了PCI设备读取不到的问题，这个问题十分隐蔽，是由于QEMU在运行时为了防止内存溢出与保证性能，会将一部分地址映射至另一部分地址，而这个问题仅能在QEMU.log中发现。因为这个原因，PCI MEM区域的地址就由0x80区域映射到了0x00区域，这也导致我们找不到PCI设备。在支持PCI&SATA后，我们利用硬盘盒将文件系统挂载至SSD中，随后启动内核进入系统。这样做时间开销大，但是支持的存储空间达到了32GB，在提升扩展性的同时，告别了uboot对于SCSI设备的繁琐操作（读入内存，逐地址修改），且上位机拥有更多调试用工具，便于Debug。

我们最终成功在板子上启动我们的Kernel，并进入命令行。下面我们播放自动执行决赛第一阶段测例的视频演示。

我们来总结一下龙芯赛道与RISCV赛道的区别。

我们组一致认为龙芯赛道是比RISCV赛道难的。难度主要来源于三方面：

首先是测例不同。同一段代码，LA工具链会比RISCV工具链多编译出很多新的系统调用。同一个测例的strace不同，出现了很多很难参考的冷门系统调用。比如我们前面提到的Clone3。

其次是我们的可参考资料少，可用成熟包少，高质量博客与文档更少：RISCV早于LA发布约10年，生态环境完善。我们在github上分别搜索两个指令集的关键词，可以看到两个赛道第三方包与工具数量差距约20倍。

第三点是刚才提到的硬件设备区别，2K1000 去掉了 NAND，让上板测试宛如带着镣铐跳舞。

但是我们仍然认为这是一个机遇与挑战并存的赛道，并且我们相信未来基于大家的努力也一定会越做越完善。

最后我们总结一下NPUcore-IMPACT的特点，它是由我们独立使用 Rust 开发，在 2K1000 中适配内核，并且未使用xv6-loongarch模板。它功能齐全，实现了106个系统调用。

我们借这个机会也来分享一下我们的答辩心得：

其一，本次答辩只讲了与其它 NPUcore 和往届高分获奖队伍与众不同的地方。

其二，操作系统大赛是我接触最优雅、本科阶段工程技术性很高的比赛

其三，本次答辩因为篇幅原因，忽略了很多戏剧性与故事性的 Debug 过程。因为这之中有了太多的收获，是我体会到的而说不清的。因此，我们组特地选择以展示大部分成果为主的方式进行PPT讲解，追求尽量全面的内核讲解。

最后，我们组一致认为，对于LA赛道的OS实现，我们坚持到最后就是胜利。

我们的答辩到此结束，请各位专家评委批评指正。