

**本科生毕业设计(论文)**

# 中期报告

**rCore模块化改进的设计与实现**

**Design and implementation of modular improvement of rCore**

|  |  |
| --- | --- |
| 学院： | 计算机学院 |
| 专业： | 计算机科学与技术 |
| 班级： | 07111703 |
| 学生姓名： | 石文龙 |
| 学号： | 1120173592 |
| 指导教师： | 陆慧梅 |

|  |
| --- |
| **一、毕业设计（论文）主要研究内容、进展情况及取得成果**  主要研究内容：  在实验室工作的基础上，实现对于应用于Rust语言的rcore内核模块化的 改进与优化主要工作包括增加系统中的每一个模块的单元测试。  进展情况：   1. kernel-context模块实现了内核上下文的控制，主要的结构包括：   LocalContext： 线程上下文。  PortalCache：传送门缓存。  ForeignContext：异界线程上下文即不在当前地址空间的线程上下文。  PortalText： 传送门代码。  MultislotPortal: 包含多个插槽的异界传送门。  lib.rs  pub struct LocalContext {  sctx: usize,  x: [usize; 31],  sepc: usize,  /// 是否以特权态切换。  pub supervisor: bool,  /// 线程中断是否开启。  pub interrupt: bool,  }  该结构包含14个方法，其分别为： pub const fn empty() 该方法的作用是创建空白上下文。 pub const fn user(pc: usize) 该方法的作用是初始化指定入口的用户上下文，切换到用户态时会打开内核中断。 pub const fn thread(pc: usize, interrupt: bool) 该方法的作用是初始化指定入口的内核上下文。 pub fn x(&self, n: usize) pub fn a(&self, n: usize) pub fn ra(&self) pub fn sp(&self) pub fn pc(&self) 该方法的作用分别是读取用户通用寄存器；读取用户参数寄存器；读取用户栈指针；读取用户栈指针；读取当前上下文的 pc。 pub fn x\_mut(&mut self, n: usize) pub fn a\_mut(&self, n: usize) pub fn sp\_mut(&self) pub fn pc\_mut(&self) 该方法的作用分别是修改用户通用寄存器;修改用户参数寄存器;修改用户栈指针;修改上下文的 pc。 pub fn move\_next(&mut self) 该方法的作用是将 pc 移至下一条指令。 pub unsafe fn execute(&mut self) 该方法的作用是执行此线程，并返回 sstatus,将修改 sscratch、sepc、sstatus 和 stvec。  foreign/mod.rs  pub struct PortalCache {  a0: usize, // (a0) 目标控制流 a0  a1: usize, // 1\*8(a0) 目标控制流 a1 （寄存，不用初始化）  satp: usize, // 2\*8(a0) 目标控制流 satp  sstatus: usize, // 3\*8(a0) 目标控制流 sstatus  sepc: usize, // 4\*8(a0) 目标控制流 sepc  stvec: usize, // 5\*8(a0) 当前控制流 stvec （寄存，不用初始化）  sscratch: usize, // 6\*8(a0) 当前控制流 sscratch（寄存，不用初始化）  }  该结构是传送门缓存，即映射到公共地址空间，在传送门一次往返期间暂存信息。该结构的方法一共有2种，分别是 pub fn init(&mut self, satp: usize, pc: usize, a0: usize, supervisor: bool, interrupt: bool) 该方法的作用是初始化传送门缓存。 pub fn address(&mut self) 该方法的作用是返回缓存地址。  pub struct ForeignContext {  /// 目标地址空间上的线程上下文。  pub context: LocalContext,  /// 目标地址空间。  pub satp: usize,  }  该结构的作用是异界线程上下文，即不在当前地址空间的线程上下文。该结构一共有1中方法，其是 pub unsafe fn execute(&mut self, portal: &mut impl ForeignPortal, key: impl SlotKey) 该方法的作用是执行异界线程。  struct PortalText(&'static [u16]);  该结构的作用是传送门代码。该结构一共有3种方法，分别是： pub fn new() pub fn aligned\_size(&self) pub unsafe fn copy\_to(&self, address: usize)  foreign/multislot\_portal.rs  pub struct MultislotPortal {  slot\_count: usize,  text\_size: usize,  }  该结构的含义是包含多个插槽的异界传送门。该结构有2个方法，分别是： pub fn calculate\_size(slots: usize) 该方法的作用是计算包括 slots 个插槽的传送门总长度。 pub unsafe fn init\_transit(transit: usize, slots: usize) 该方法的作用是初始化公共空间上的传送门。其中参数transit 必须是一个正确映射到公共地址空间上的地址。  kernel-context模块的测试方法就是调用LocalContext结构,然后一次调用LocalContext结构的几个方法:测试empty().测试user()函数,初始化指定入口的用户上下文。thread()函数,初始化指定入口的内核上下文。测试读取类函数.测试move\_next,将 pc 移至下一条指令。测试修改类函数.然后使用assert\_eq()函数来比较方法的返回值和预期的结果是否一样,若结果一样则说明函数方法没有问题,测试通过.   1. linker板块为内核提供链接脚本的文本，以及依赖于定制链接脚本的功能。   build.rs 文件可依赖此板块，并将 [SCRIPT] 文本常量写入链接脚本文件  定义内核入口，即设置一个启动栈，并在启动栈上调用高级语言入口。macro\_rules! boot0  KernelLayout结构：代表内核地址信息；  KernelRegion结构：内核内存分区。  KernelRegionIterator结构：内核内存分区迭代器。  KernelLayout的结构为： pub struct KernelLayout { text: usize, rodata: usize, data: usize, sbss: usize, ebss: usize, boot: usize, end: usize, } 该结构有6个方法，分别为 pub fn locate() 该方法作用为定位内核布局。 pub const fn start(&self) 该方法作用为得到内核起始地址。 pub const fn end(&self) 该方法作用为得到内核结尾地址。 pub const fn len(&self) 该方法作用为得到内核静态二进制长度。 pub unsafe fn zero\_bss(&self) 该方法作用为清零 .bss 段。 pub fn iter(&self) 该方法作用为得到内核区段迭代器。  KernelRegion结构为： pub struct KernelRegion { /// 分区名称。 pub title: KernelRegionTitle, /// 分区地址范围。 pub range: Range, } 该结构的含义是内核内存分区。该结构存在fmt方法。 fn fmt(&self, f: &mut fmt::Formatter<'\_>) 该方法的作用是使用给定的格式化程序格式化值。  KernelRegionIterator结构为： pub struct KernelRegionIterator<'a> { layout: &'a KernelLayout, //内核内存分区名称 next: Option, } 该结构的含义是内核内存分区迭代器。该结构存在next方法。 fn next(&mut self) 该方法的作用是得到迭代器中下一位的值。  app.rs  AppMeta：应用程序元数据。  AppIterator：应用程序迭代器。  AppMeta结构为： pub struct AppMeta { base: u64, step: u64, count: u64, first: u64, } 该结构的含义是应用程序元数据。该结构有2个方法，分别为： pub fn locate() 该方法的作用是定位应用程序。 pub fn iter(&'static self) 该方法的作用是遍历链接进来的应用程序。  AppIterator结构为： pub struct AppIterator { meta: &'static AppMeta, i: u64, } 该结构的含义是应用程序迭代器。该结构有一个next方法： fn next(&mut self) 该方法的作用是对应用程序进行迭代。  定义了实现VmMeta特征的SV39结构和实现了PageManager特征的Sv39Manager结构体。  需要赋值一个物理页range: Range；  linker模块测试了KernelLayout结构的集中方法,首先需要构建依赖环境,然后调用KernelLayout结构的方法,并且对内核内存分区迭代器KernelRegionIterator结构的测试，最后使用assert\_eq()函数将方法的返回值和预期结果进行比较,如果比较通过,则测试也通过.   1. kernel-vm模块的主要内容是内核虚拟存储的管理。   space/mod.rs  AddressSpace：地址空间结构。  pub struct AddressSpace { /// 虚拟地址块 pub areas: Vec>, page\_manager: M, } 该结构共有7个方法，分别为： pub fn new() 该方法的作用是创建新地址空间。 pub fn root\_ppn(&self) 该方法的作用是得到地址空间根页表的物理页号。 pub fn root(&self) 该方法的作用是得到地址空间根页表 pub fn map\_extern(&mut self, range: Range, pbase: PPN, flags: VmFlags) 该方法的作用是向地址空间增加映射关系。 pub fn map( &mut self, range: Range, data: &[u8], offset: usize, mut flags: VmFlags, ) 该方法的作用是分配新的物理页，拷贝数据并建立映射。 pub fn translate(&self, addr: VAddr, flags: VmFlags) 该方法的作用是检查 flags 的属性要求，然后将地址空间中的一个虚地址翻译成当前地址空间中的指针。 pub fn cloneself(&self, new\_addrspace: &mut AddressSpace) 该方法的作用是遍历地址空间，将其中的地址映射添加进自己的地址空间中，重新分配物理页并拷贝所有数据及代码。  space/mapper.rs  pub(super) struct Mapper<'a, Meta: VmMeta, M: PageManager> {  space: &'a mut AddressSpace,  range: Range>,  flags: VmFlags,  done: bool,  }  该结构有5个方法，分别是： pub fn new( space: &'a mut AddressSpace, range: Range, flags: VmFlags, ) 该方法的作用是创建一个新的Mapper。 pub fn ans(self) 该方法的作用是得到Mapper结构的done值。 fn arrive(&mut self, pte: &mut Pte, target\_hint: Pos) fn meet( &mut self, \_level: usize, pte: Pte, \_target\_hint: Pos, ) fn block(&mut self, \_level: usize, pte: Pte, \_target\_hint: Pos)  kernel-vm模块的测试,首先需要构建依赖环境,即要定义并实现一个满足VmMeta特征的结构SV39,然后要定义并实现满足PageManager特征的结构SV39Manage;最后，还需要定义内核地址信息KernelLayout，并且实现KernelLayout结构的方法；至此,kernel-vm模块测试所需要的的依赖环境构建完成.  测试的时候，首先自己初始化一个KernelLayout结构，然后调用AddressSpace结构和Mapper结构的方法，最后将预期结果和实际结果进行比较，如果两者相符，则说明kernel-vm模块中的mapper模块测试成功；在测试space是可以依次测试：创建新地址空间。地址空间根页表的物理页号。地址空间根页表。向地址空间增加映射关系。检查 flags 的属性要求，然后将地址空间中的一个虚地址翻译成当前地址空间中的指针。遍历地址空间，将其中的地址映射添加进自己的地址空间中，重新分配物理页并拷贝所有数据及代码。并且在每次调用之后，将预期结果和实际结果进行比较，如果结果哦相同，则space模块测试完成。   1. task-manage模块的作用是实现父进程,子进程和线程之间的调度管理.   任务 id 类型，自增不回收，任务对象之间的关系通过 id 类型来实现  ProcId  ThreadId  CoroId结构ProcId的方法有3个，分别是new(),from\_usize(),get\_usize;  new()方法创建了一个进程编号自增的进程id类型，  from\_usize()根据输入的usize类型参数可以获得一个以参数为id的ProcId，  get\_usize()方法需要输入的参数是一个ProcId的引用，返回该ProcId结构对应的id。 结构ThreadId的方法与ProcId的方法相同。  任务对象管理 manage trait，对标数据库增删改查操作  insert  delete  get\_mut  任务调度 schedule trait，队列中保存需要调度的任务 Id  add：任务进入调度队列  fetch：从调度队列中取出一个任务  封装任务之间的关系，使得 PCB、TCB 内部更加简洁  ProcRel：进程与其子进程之间的关系  ProcThreadRel：进程、子进程以及它地址空间内的线程之间的关系 ProcRel结构包含父进程id，子进程列表和已经结束的进程列表。ProcRel结构有new()方法，add\_child(),del\_child(),wait\_any\_child(),wait\_child()5种方法。  new()方法需要输入父进程ProcId，返回一个ProcRel结构，其中父进程ProcId是输入的参数，子进程列表和已经结束的进程列表使新创建的动态数组。  add\_child()方法的参数是一个ProcRel的可变引用和一个子进程ProcId，该方法的作用是将参数子进程id放入到输入的ProcRel的子进程列表中。  del\_child()方法的参数有一个ProcRel的可变引用、一个子进程ProcId和一个退出码exit\_code，该方法的作用是：令子进程结束，子进程 Id 被移入到 dead\_children 队列中，等待 wait 系统调用来处理。  wait\_any\_child()方法的参数是一个ProcRel的可变引用，该方法的作用是：等待任意一个结束的子进程，直接弹出 dead\_children 队首，如果等待进程队列和子进程队列为空，返回None，如果等待进程队列为空、子进程队列不为空，则返回 -2。  wait\_child方法的参数有有一个ProcRel的可变引用和一个子进程ProcId，该方法的作用是：等待特定的一个结束的子进程，弹出 dead\_children 中对应的子进程，如果等待进程队列和子进程队列为空，返回None，如果等待进程队列为空、子进程队列不为空，则返回 -2。  封装任务之间的调度方法  PManager：管理进程以及进程之间的父子关系  PThreadManager：管理进程、子进程以及它地址空间内的线程之间的关系 PManager结构为： pub struct PManager + Schedule> { // 进程之间父子关系 rel\_map: BTreeMap, // 进程对象管理和调度 manager: Option, // 当前正在运行的进程 ID current: Option, phantom\_data: PhantomData  , }  该结构办函的方法共有9个，分别为： pub const fn new() -> Self 此方法用于新建 PManager pub fn find\_next(&mut self) -> Option<&mut P> 此方法用于找到下一个进程 pub fn set\_manager(&mut self, manager: MP) 此方法用于设置 manager pub fn make\_current\_suspend(&mut self) 此方法用于阻塞当前进程 pub fn make\_current\_exited(&mut self, exit\_code: isize) 此方法用于结束当前进程，只会删除进程的内容，以及与当前进程相关的关系 pub fn add(&mut self, id: ProcId, task: P, parent: ProcId) 此方法用于 添加进程，需要指明创建的进程的父进程 Id pub fn current(&mut self) -> Option<&mut P> 此方法用于获取当前进程 #[inline] pub fn get\_task(&mut self, id: ProcId) -> Option<&mut P> 此方法用于获取某个进程 pub fn wait(&mut self, child\_pid: ProcId) -> Option<(ProcId, isize)> 此方法用于wait 系统调用，返回结束的子进程 id 和 exit\_code，正在运行的子进程不返回 None，返回 (-2, -1)  测试id模块ProcId结构时，只需要依次调用ProcId模块里各结构的方法，然后将预期结果和实际结果进行比较就可以了。proc\_rel模块和id模块的测试方法基本一样，只需要依次：创建一个进程时同时创建进程关系；添加子进程前测试等待子进程结束的函数；添加子进程；测试等待子进程结束的函数；子进程结束，子进程 Id 被移入到 dead\_children 队列中；测试等待子进程结束的函数；最后将与其结果和实际结果进行比较就可以了。  测试proc\_manage模块里的PManager结构的时候，首先需要构建需要的依赖环境，即定义一个进程结构Process，并且实现Process的new()方法，然后定义一个ProcManager结构，ProcManager结构需要实现Manage特征和Schedule特征，至此，proc\_manage模块的依赖环境完成了，然后就可以依次：新建 PManager；设置manager；添加进程；获取指定进程；并且将预期结果和实际结果进行比较。 |
| **二、存在的问题和拟解决方案**   1. console模块： 需要实现put\_char函数； 在Cargo.toml里直接添加sbi-rt会在测试的时候直接关于寄存器报错，自己实现sbi的console\_putchar在asm中关于寄存器 x10会报错：invalid register x10: unknown register无效寄存器 x10：未知寄存器。 2. kernel-alloc模块： 完成了init函数的测试 测试transfer()时，参数正确还是失败，准备分开测试一下其中的代码。 Heap::new()创建的是空分配器，总容量为0,所以分配失败。 测试alloc()时，会运行handle\_alloc\_error(layout)。 3. sync模块： 测试up.rs模块时，调用sstatus::read().sie()函数会出错，测试exit方法使要避免nested\_level为0的情况，否则会发生错误。 测试condvar.rs时测试signal()方法前需要向新建的Condvar里添加数据否则会panic。 4. easy-fs模块 对于实现BlockDevice特征的结构定义成功了，但是在测试的时候会报错：无效内存引用。 5. signal-imple模块：在进行测试的时候会对kernel-context的线程切换核心部分报错 unsafe extern "C" fn execute\_naked() { core::arch::asm!() } |
| **三、下一步研究任务与进度安排**  4.6-4.15 使完成了部分的4个模块能够进行演示。  4.16-4.23 完成easy-fs模块的测试。  4.24-4.30 完成syscall和signal-imple模块的测试。 |
| **四、指导教师意见**  石文龙同学按照毕设的工作安排有序地推进论文工作。后续希望更加有效地完成剩余任务。  **签字：**  2023年4月22日 |
| **成绩： ，占比：0.00%** |
| **五、中期审核负责人意见**  通过  签字：  2023年5月22日 |