xv6-rust设计报告

2021全国大学生计算机系统能力大赛操作系统设计赛

Ko-oK战队

一、设计简介

本项目为使用Rust语言对MIT的`xv6-riscv`项目的重新设计与实现，这次实验既可以看做Rust语言在操作系统开发的探索，也可用作高校在操作系统这门课程上的实践作业。在初赛阶段，我们除了文件系统基本实现了操作系统内核的所有功能，并支持多核；同时，我们也开发了一部分网络协议栈的内容以及针对`e1000`网卡写的设备驱动，目标是为`xv6-rust`支持网络功能。除此之外，我们也开发了Buddy System Allocator来代替`xv6-riscv`的朴素的内存分配算法。

二、 设计思路

Rust作为更具现代化的语言，相对于传统的系统开发语言C语言来说具有更强的抽象性，这使得Rust语言具有更强的表意性。同时，Rust为了保证开发的安全性，它强制用户使用所有权机制来保证系统的安全性，对于未实现`Copy`特性的类型，则赋值操作均按照移动语意来处理。同时，Rust设置不可变引用与可变引用，Rust的引用是对裸指针的封装，在编译期会进行一些检查：例如，在一个作用域内可以同时出现多个不可变引用但只能出现一个可变引用，且不可变引用与可变引用不能在同一作用域内出现。同时，Rust对于直接操作裸指针视作是`unsafe`的，如果我们想去直接读写裸指针，则必须将代码块标识为`unsafe`。Rust的这些特性可以将大部分使用传统系统语言难以察觉的错误拦截在编译期，极大地方便了我们的调试过程。但这些也使得我们充分利用Rust特性对OS进行重新开发。以下我将挑选几点使用Rust的特性对`xv6-riscv`进行重新实现的部分进行详细说明。

1. 锁的实现

在`xv6-riscv`中，对于需要锁的结构，仅仅在其中的域里放入锁结构的指针。而在获取锁的过程中，仅仅对于变量的`lock field`进行检查从而判断其是否可以acquire。这种写法对于程序员有极高的要求，因为他在不获取锁的情况下依然可以变量的内容，或者由于程序员忘记了去release锁都将会造成程序死锁且难以调试。

Rust具有较为完善的类型系统，支持泛型。因此我们可以将锁设计为智能指针的形式，将变量具体内容包裹在锁内部，当我们调用`acquire`方法的时候返回一个守卫变量，在变量中可以访问原数据的各个域。除此之外，Rust有`Drop`特性，只要我们为我们的锁实现了`Drop`特性，当变量离开作用域时会自动release锁变量，真正意义上实现了`RAII`，从而避免死锁。

例如，在`xv6-riscv`中，它使用如下结构来上锁：

```
struct {
  struct spinlock lock;
  struct run *freelist;
} kmem;
```

而我们使用将lock作为指针包裹在变量外面的方式进行上锁:

```
pub struct KernelHeap(Spinlock<BuddySystem>);
```

在实现上，我们调用`acquire`方法并返回一个`SpinlockGuard`类型的变量，并为`SpinlockGuard`实现了`Drop`和`DeferMut`特性，从而使其更不容易发生死锁：

```
    pub fn acquire(&self) -> SpinlockGuard<'_, T> {

        push_off();
        if self.holding() {
            panic!("acquire");
        }

        while self.locked.swap(true, Ordering::Acquire){
            // Now we signals the processor that it is inside a busy-wait spin-
loop
            spin_loop();
        }
        fence(Ordering::SeqCst);
        unsafe {
            self.cpu_id.set(cpuid() as isize);
        }

        SpinlockGuard{spinlock: &self}
    }

    pub fn release(&self) {
        if !self.holding() {
            panic!("release");
        }
        self.cpu_id.set(-1);
        fence(Ordering::SeqCst);
        self.locked.store(false, Ordering::Release);

        pop_off();
    }
```

```
impl<T> Deref for SpinlockGuard<'_, T>{
    type Target = T;

    fn deref(&self) -> &Self::Target {
        unsafe{
            &*self.spinlock.data.get()
        }
    }
}

impl<T> DerefMut for SpinlockGuard<'_, T>{
    fn deref_mut(&mut self) -> &mut Self::Target{
        unsafe{
            &mut *self.spinlock.data.get()
        }
    }
}

impl<T> Drop for SpinlockGuard<'_, T>{
    fn drop(&mut self){
        self.spinlock.release()
    }
}
```

2. 静态变量

在Rust中，静态变量在编译之后就拥有了精确的内存地址，这就意味着不能在运行时为静态变量进行地址空间的分配。同样一件事，在C语言中可以将其作为全局变量声明，之后在程序运行时再对其进行初始化，然而这在Rust语言中是不允许的。因此我们对于特定类型的变量都要为其提供`new()`方法进行初始化；对于需要在堆上分配的变量，需要使用`lazy_static!`宏对其进行懒加载从而实现动态内存分配。

以`xv6-rust`中的程序来举例，我们需要在操作系统启动过程中为内核分配页表从而通过页表项记录物理地址到虚拟地址的转换：

```
pub static mut KERNEL_PAGETABLE:PageTable = PageTable::empty();
```

```
impl PageTable{
    pub const fn empty() -> Self{
        Self{
            entries:[PageTableEntry(0); PGSIZE/8]
        }
    }
}
```

同时，在`e1000`网卡驱动程序中，根据Intel的标准，我们需要为发送消息与接收消息分配消息缓冲队列，并将队列的头地址写入寄存器中，当我们实现发送消息或者接收消息时，则网卡会从这些队列中根据其他寄存器的信息来发送和接收分组。因此我们使用`lazy_static`来为我们的全局变量来分配空间：

```
lazy_static! {
    static ref RECEIVE_MBUF:Spinlock<[MBuf;RECEIVE_RING_SIZE]> =
Spinlock::new(array![_ => MBuf::new();RECEIVE_RING_SIZE], "receive_mbuf");
    static ref TRANSMIT_MBUF:Spinlock<[MBuf;TRANSMIT_RING_SIZE]> =
Spinlock::new(array![_ => MBuf::new();TRANSMIT_RING_SIZE], "transmit_mbuf");
}
```

对静态变量做写操作是不安全的，因此我们使用锁来包裹变量内容，从而简化我们的程序。

3. 所有权机制与RAII

所有权机制很好，但对于程序员来说却是一种折磨，在实现`xv6-rust`的过程中遇到了许多由所有权机制带来的问题，尤其是在写调度算法的时候。举例来说，在实现`alloc_proc()`函数时（即为进程分配一块地址空间）：

```
pub fn alloc_proc(&mut self) -> Option<&mut Process> {
        for p in self.proc.iter_mut() {
            let mut guard = p.data.acquire();
            if guard.state == Procstate::UNUSED {
                .
                .
                .
                // An empty user page table
                if let Some(page_table) = unsafe { extern_data.proc_pagetable()
} {
                    .
                    .
                    .
```

```
                } else {
                    p.freeproc();
                    drop(guard);
                    return None
                }
            }else {
                p.freeproc();
                drop(guard);
            }
        }
        None
    }
```

可以看到，在最初的实现中，我们首先遍历调度器内部所有的进程，然后找到一个状态为未使用的空进程，并为它分配页表，如果失败的话就释放它。但是由于此时在if与else连个代码块中同时持有了对于`p`的可变引用，这在rust的编译器中是不允许的，所以rust编译器不予通过，因此，我们只能通过修改`proc_pagetable()`函数使其在内部进行检查是否分配成功当失败时调用`freeproc()`释放页表的物理内存分配。修改后的代码如下：

```
pub fn alloc_proc(&mut self) -> Option<&mut Process> {
        for p in self.proc.iter_mut() {
            let mut guard = p.data.acquire();
            if guard.state == Procstate::UNUSED {
                .
                .
                .
                // An empty user page table
                unsafe{
                    extern_data.proc_pagetable();
                }
                .
                .
                .
            }else {
                drop(guard);
            }
        }

        None
    }
```

此时我们在`proc_pagetable()`完成了检查的过程，只需在`alloc_proc()`调用其方法即可避免所有权的错误。

4. 进程的设计

在使用Rust重新实现`xv6-riscv`的过程中，在进程与调度的设计与实现中遇到了很大的麻烦。在C语言中尽管要求程序员具有很高的自律性，但其具有很高的灵活性来设计操作系统。例如在进程的实现时，一个进程的某些内容需要在线程访问时上锁，有些内容不需要上锁。例如进程的状态、通道(channel)、是否被杀死、退出状态、进程ID这些可读写变量需要上锁访问；而对于内核虚拟内存、进程内存大小、用户页表、中断帧、上下文这些变量则不需要加锁访问。在`xv6-riscv`中可以很容易的实现：

```
// Per-process state
struct proc {
  struct spinlock lock;
```

```
  // p->lock must be held when using these:
  enum procstate state;        // Process state
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  int xstate;                  // Exit status to be returned to parent's wait
  int pid;                     // Process ID

  // proc_tree_lock must be held when using this:
  struct proc *parent;         // Parent process

  // these are private to the process, so p->lock need not be held.
  uint64 kstack;               // Virtual address of kernel stack
  uint64 sz;                   // Size of process memory (bytes)
  pagetable_t pagetable;       // User page table
  struct trapframe *trapframe; // data page for trampoline.S
  struct context context;      // swtch() here to run process
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

在C中，只需要在进程的结构中放入一个锁的域，当需要访问需要上锁的变量时只需要调用锁的`acquire()`和`release()`方法来获取和释放锁来访问其中的域；当访问不需要上锁的内容时只需要直接访问其中的域即可。

而由于Rust的锁形式是以智能指针的方式包裹在变量的外层，因此如果我们直接像原有实现一样直接在变量外层包裹锁。那么无论我们想去访问公有域或者私有域都要将其`acquire()`和`release()`。那么试想一下，当我们去访问用户页表、打开文件、当前目录这些比较耗时的变量时将会使当前进程被锁住，这会导致效率极大的下降！

因此我们的实现是将公有域与私有域分开实现并将其放入`Process`结构中，并将公有域结构进行上锁，那么当我们需要访问公有域时只需将公有域部分上锁就可以了。

```
pub struct Process {
    pub data: Spinlock<ProcData>,
    pub extern_data: UnsafeCell<ProcExtern>,
}

pub struct ProcData {
    // p->lock must be held when using these
    pub state:Procstate,
    pub channel:usize, // If non-zero, sleeping on chan
    pub killed:usize, // If non-zero, have been killed
    pub xstate:usize, // Exit status to be returned to parent's wait
    pub pid: usize,   // Process ID
}

pub struct ProcExtern {
    // these are private to the process, so p->lock need to be held
    pub kstack:usize,  // Virtual address of kernel stack
    pub size:usize, // size of process memory
    pub pagetable: Option<Box<PageTable>>, // User page table
    pub trapframe: *mut Trapframe, // data page for trampoline.S
    pub context: Context, // swtch() here to run processs

    pub name: &'static str,   // Process name (debugging)
```

```
    // proc_tree_lock must be held when using this:
    pub parent: Option<NonNull<Process>>,

    // TODO: Open files and Current directory
}
```

5. 中断的优化

在原版`xv6-riscv`中，对于内核中断信息给出的信息较少，仅仅给出了`scause`、`sepc`、`sscause`等寄存器的值，具体的中断类型还需要去查看RISC-V来确定。因此我们参考`rust-embedded/riscv`的实现为我们寄存器做了优化：

```
use bit_field::BitField;
use core::mem::size_of;

// Supervisor Trap Cause
#[inline]
pub unsafe fn read() -> usize {
    let ret:usize;
    llvm_asm!("csrr $0, scause":"=r"(ret):::"volatile");
    ret
}

#[inline]
pub unsafe fn write(x:usize){
    llvm_asm!("csrw scause, $0"::"r"(x)::"volatile");
}

// scause register
#[derive(Clone, Copy)]
pub struct Scause{
    bits: usize
}

// Trap Cause
#[derive(Copy, Clone, Debug, Eq, PartialEq)]
pub enum Trap{
    Interrupt(Interrupt),
    Exception(Exception),
}

// Interrupt
#[derive(Copy, Clone, Debug, Eq, PartialEq)]
pub enum Interrupt{
    UserSoft,
    SupervisorSoft,
    UserTimer,
    SupervisorTimer,
    UserExternal,
    SupervisorExternal,
    Unknown
}

// Exception
#[derive(Copy, Clone, Debug, Eq, PartialEq)]
pub enum Exception {
    InstructionMisaligned,
```

```
    InstructionFault,
    IllegalInstruction,
    Breakpoint,
    LoadFault,
    StoreMisaligned,
    StoreFault,
    UserEnvCall,
    KernelEnvCall,
    InstructionPageFault,
    LoadPageFault,
    StorePageFault,
    Unknown
}


impl Interrupt{
    pub fn from(nr: usize) -> Self {
        match nr{
            0 => Interrupt::UserSoft,
            1 => Interrupt::SupervisorSoft,
            4 => Interrupt::UserTimer,
            5 => Interrupt::SupervisorTimer,
            8 => Interrupt::UserExternal,
            9 => Interrupt::SupervisorExternal,
            _ => Interrupt::Unknown
        }
    }
}

impl Exception{
    pub fn from(nr: usize) -> Self {
        match nr {
            0 => Exception::InstructionMisaligned,
            1 => Exception::InstructionFault,
            2 => Exception::IllegalInstruction,
            3 => Exception::Breakpoint,
            5 => Exception::LoadFault,
            6 => Exception::StoreMisaligned,
            7 => Exception::StoreFault,
            8 => Exception::UserEnvCall,
            9 => Exception::KernelEnvCall,
            12 => Exception::InstructionPageFault,
            13 => Exception::LoadPageFault,
            15 => Exception::StorePageFault,
            _ => Exception::Unknown
        }
    }
}

impl Scause{

    // new a Scause Object by usize
    #[inline]
    pub fn new(scause: usize) -> Self{
        Self{
            bits: scause
        }
    }
```

```
    // Returns the contents of the register as raw bits
    #[inline]
    pub fn bits(&self) -> usize{
        self.bits
    }

    // Returns the code field
    pub fn code(&self) -> usize{
        let bit =  1 << (size_of::<usize>() * 8 -1);
        self.bits & !bit
    }

    // Trap Cause
    #[inline]
    pub fn cause(&self) -> Trap{
        if self.is_interrupt() {
            Trap::Interrupt(Interrupt::from(self.code()))
        }else{
            Trap::Exception(Exception::from(self.code()))
        }
    }


    // Is trap cause an interrupt
    #[inline]
    pub fn is_interrupt(&self) -> bool{
        self.bits.get_bit(size_of::<usize>()*8 - 1)
    }

    // Is trap cause an exception
    #[inline]
    pub fn is_exception(&self) -> bool{
        !self.is_interrupt()
    }


}
```

而在我们的中断中，我们通过分析`scause`来判断不同的中断类型来进行特定的处理，具体见中断文档。

三、实现描述

请见该目录下其他文档，除此之外，我们也在代码中提供了详细的注释。由于时间原因，文档内容有空缺或者未更新，请见谅！

- 环境搭建
- gdb调试
- 启动
- 锁
- 中断
- 物理内存分配
- 虚拟内存
- 进程与线程
- 调度

四、遇到的问题及解决办法

1. 内核栈

由于在操作系统的引导启动阶段，我们只为每个核分配了4KB的内核栈大小：

```
 # qemu -kernel starts at 0x1000. the instructions
    # there seem to be provided by qemu, as if it
    # were a ROM. the code at 0x1000 jumps to
    # 0x80000000, the _entry function here,
    # in machine mode. each CPU starts here.
    .text
    .globl _entry
_entry:
    # set up a stack for Rust.
    # stack0 is declared below,
    # with a 4096-byte stack per CPU.
    # sp = stack0 + (hartid * 4096)
    la sp, stack0
    li a0, 1024*4
    csrr a1, mhartid
    addi a1, a1, 1
    mul a0, a0, a1
    add sp, sp, a0
    # jump to start() in start.rs
    call start

    .section .data
    .align 4
stack0:
    .space 4096 * 8 # 8 is NCPU in param.rs
```

于是当我们传递的参数，就会出现莫名其妙的死机与输出很奇怪东西。当我在实现虚拟内存与物理内存的映射时，曾经将一个函数的返回参数设置为页表，而页表的内存大小正好为4KB，这导致在运行时一直出现bug，不管打断点还是使用`GDB`调试都没查到问题所在。最后求助项目导师陈恒杰学长一起解决了这个问题。

2. 死锁

由于想去测试时间中断，所以我在`clockintr()`函数进行输出，发现程序不输出并且发生死锁了。经过查询发现原来是自旋锁没有实现`push_off()`和`pop_off()`方法。`push_off()`和`pop_off()`是对`intr_on()`和`intr_off()`方法的封装，在获取锁的时候应当去关闭中断。如果中断处理程序在尝试获取锁时阻塞，则可能迫使高优先级线程进入睡眠状态，以等待完全不相关的线程。更糟糕的情况是，当前线程可能已经持有该锁，在这种情况下，这将导致单线程死锁。

五、参考实现

[xv6-riscv]

[xv6-riscv-rust]

[rCore-Tutorial-v3]