Readme

本文档为本次“操作系统大赛”的概括性文档，包含了本次大赛参赛作品的一些基本信息和TO-DO List。具体的教程文档，存放在对应的文件夹中。

因为指令集移植工作来不及做，所以部分代码仍然是x86风格

一、实验模块设计

实验节点要求满足模块化的特点。我们整个实验分为以下几个模块：

1、最小化内核：

本部分要求对操作系统的启动过程有足够的了解，并且能够编写并启动一个能够打印Hello World的最小化内核。包括了如下的步骤：

- 搭建开发环境。可以在Ubuntu等Linux发行版中进行。也可以在Windows操作系统中，配合WSLG等工具进行开发。
- 安装相应的开发工具，例如VSCode，rustc等。
- 了解rust的SBI，并学习借助sbi载入操作系统的方法。K210开发板中存在uboot。（待填坑）
- 利用如上的工具和机制，通过加载ELF文件的方式，进入操作系统内核。并且在内核中打印Hello World。这样可以大大减少汇编语言的使用。
- 如下是进入OS的主函数，这是移植之前的，所以使用的是Rust中与uefi有关的包。移植时将使用sbi完成类似的操作。这样做的一大好处就是可以最大程度减少汇编语言的使用。

解释待填坑，这一块我实在不太懂了（by brh）

```
#[entry]
fn efi_main(image: uefi::Handle, mut system_table: SystemTable<Boot>) -> Status {
    uefi_services::init(&mut system_table).expect("Failed to initialize utilities");

    info!("Running UEFI bootloader...");

    let bs = system_table.boot_services();
    let config = {
        let mut file = open_file(bs, CONFIG_PATH);
        let buf = load_file(bs, &mut file);
        config::Config::parse(buf)
    };

    let graphic_info = init_graphic(bs);
    // info!("config: {:#x?}", config);

    let acpi2_addr = system_table
        .config_table()
        .iter()
        .find(|entry| entry.guid == ACPI2_GUID)
        .expect("failed to find ACPI 2 RSDP")
        .address;
```

```
    info!("ACPI2: {:?}", acpi2_addr);

    let elf = {
        let mut file = open_file(bs, config.kernel_path);
        let buf = load_file(bs, &mut file);
        ElfFile::new(buf).expect("failed to parse ELF")
    };
    unsafe {
        ENTRY = elf.header.pt2.entry_point() as usize;
    }

    let max_mmap_size = system_table.boot_services().memory_map_size();
    let mmap_storage = Box::leak(
        vec![0; max_mmap_size.map_size + 10 *
max_mmap_size.entry_size].into_boxed_slice()
    );
    let mmap_iter = system_table
        .boot_services()
        .memory_map(mmap_storage)
        .expect("Failed to get memory map")
        .1;
    let max_phys_addr = mmap_iter
        .map(|m| m.phys_start + m.page_count * 0x1000)
        .max()
        .unwrap()
        .max(0x1_0000_0000); // include IOAPIC MMIO area

    let mut page_table = current_page_table();
    // root page table is readonly
    // disable write protect
    unsafe {
        Cr0::update(|f| f.remove(Cr0Flags::WRITE_PROTECT));
        Efer::update(|f| f.insert(EferFlags::NO_EXECUTE_ENABLE));
    }

    elf::map_elf(&elf, &mut page_table, &mut UEFIFrameAllocator(bs))
        .expect("Failed to map ELF");

    elf::map_range(
        config.kernel_stack_address,
        config.kernel_stack_size,
        &mut page_table,
        &mut UEFIFrameAllocator(bs),
        false
    ).expect("Failed to map stack");

    elf::map_physical_memory(
        config.physical_memory_offset,
        max_phys_addr,
        &mut page_table,
        &mut UEFIFrameAllocator(bs),
    );

    // recover write protect
    unsafe {
```

```
        Cr0::update(|f| f.insert(Cr0Flags::WRITE_PROTECT));
    }

    // FIXME: multi-core
    //  All application processors will be shutdown after ExitBootService.
    //  Disable now.
    // start_aps(bs);

    // for i in 0..5 {
    //     info!("Waiting for next stage... {}", 5 - i);
    //     bs.stall(100_000);
    // }

    info!("Exiting boot services...");

    let (rt, mmap_iter) = system_table
        .exit_boot_services(image, mmap_storage)
        .expect("Failed to exit boot services");
    // NOTE: alloc & log can no longer be used

    // construct BootInfo
    let bootinfo = BootInfo {
        memory_map: mmap_iter.copied().collect(),
        physical_memory_offset: config.physical_memory_offset,
        graphic_info,
        system_table: rt,
    };
    let stacktop = config.kernel_stack_address + config.kernel_stack_size *
0x1000;

    unsafe {
        jump_to_entry(&bootinfo, stacktop);
    }
}
```

2、控制台、日志和调试：

本部分是了解一些常见的调试方式，使得后续的实验轻松简单。

- 利用串口输出作为调试。这一步需要手动实现println等功能。在使用log crate的前提条件下，
- 利用VGA输出进行调试（可选）
- 编译时附带调试信息。编译时除了编译为Debug版本和Release版本，还可以选择编译为Release with debug info的版本，也即可以对Release版本进行调试。
- vscode调试。**本部分皓宇记得填坑，着重介绍**
- gdb调试：GDB调试在操作系统的编写中是相当重要的。借助于插件pwndbg，可以使得debug变得容易很多。
- LLDB：内置于XCode中的调试器，可以使得使用Mac的同学轻松地Debug。其他的平台也可以安装使用。

重点展示这一部分，皓宇记得写一下，这是亮点

3、内存管理（不含缺页中断）

- RISC-V SV39内存管理：对于64位的RISC-V架构，多使用三级页表，支持39位虚拟地址，也即每个地址池理论上最多支持512GB的内存（事实上被切分为两个256GB）。物理地址显然为64位。虚拟地址通过MMU转换为物理地址。分页模式选择和页表基地址保存在SARP寄存器中。
- 页面分配：使用BitMap管理资源的分配。先分配连续的虚拟页，再给每个虚拟页分配对应的物理页。
- 动态内存分配：在虚存的 `0xFFFFFF8000000000` 开始分配一个 32MB 的堆，从 Bootloader 传来的 mmap 中可以拿到可用的内存区域，在帧分配器中可以将它们切成 4KiB 的块，在我们需要的时候进行分配。同时，我们也可以给帧分配器附带一个动态数组，用于存储被释放的物理帧，以供再次使用

4、中断与陷阱

- RISCV的中断依赖于mtvec和mcause寄存器
- mtvec寄存器存放了中断处理程序的基地址和中断处理函数的寻址模式。我们选择直接寻址，并用模式匹配的方法去匹配到合适的函数。
- mcause则记录了中断发生的原因。
- Rust中的cpu crate有TrapFrame结构体，记录了一次中断的整型以及浮点寄存器、MMU、中断堆栈地址和内核线程编号。
- 对于不同类型的中断，我们需要单独编写不同的中断处理程序进行处理。
- 对于外部中断，则有赖于PLIC进行处理。这有赖于mie寄存器中的meie位

5、内核线程：

- 前面已经实现了动态内存分配，我们就可以为每个进程分配一块内存空间作为PCB。PCB中当然至少应该包括栈信息、栈信息、pid、计数、页目录基地址和状态。
- 要新生成一个线程，需要申请栈帧和页目录表，然后将新创建的内核线程加入到队列中

```
pub fn spawn_kernel_thread(entry: fn() -> !, name: String, data:
Option<ProcessData>) {
    x86_64::instructions::interrupts::without_interrupts(|| {
        let entry = VirtAddr::new(entry as u64);

        let stack = get_frame_alloc_for_sure().allocate_frame()
            .expect("Failed to allocate stack for kernel thread");

        let stack_top = VirtAddr::new(physical_to_virtual(
            stack.start_address().as_u64()) + FRAME_SIZE);

        let mut manager = get_process_manager_for_sure();
        manager.spawn_kernel_thread(entry, stack_top, name, ProcessId(0),
data);
    });
}

impl ProcessManager {
    pub fn spawn_kernel_thread(
        &mut self,
        entry: VirtAddr,
        stack_top: VirtAddr,
        name: String,
        parent: ProcessId,
        proc_data: Option<ProcessData>,
    ) -> ProcessId {
```

```
        let mut p = Process::new(
            &mut *crate::memory::get_frame_alloc_for_sure(),
            name,
            parent,
            self.get_kernel_page_table(),
            proc_data,
        );
        p.pause();
        p.init_stack_frame(entry, stack_top);
        info!("Spawn process: {}#{}", p.name(), p.pid());
        let pid = p.pid();
        self.processes.push(p);
        pid
    }
}
```

- 调度方法是时间片轮转调度。当记录到一定次数的时钟中断时，或者进程结束时，就启动调度程序。方法在于先保存目前栈帧的信息，然后修改PC寄存器和栈指针，恢复栈帧信息，就切换到了另一个进程执行。
    - 当然其他的进程调度算法也是可用的

```
pub fn switch(regs: &mut Registers, sf: &mut InterruptStackFrame) {
    x86_64::instructions::interrupts::without_interrupts(|| {
        let mut manager = get_process_manager_for_sure();

        manager.save_current(regs, sf);
        manager.switch_next(regs, sf);
    });
}

impl ProcessManager {
    pub fn save_current(&mut self, regs: &mut Registers, sf: &mut
InterruptStackFrame) {
        let current = self.current_mut();
        if current.is_running() {
            current.tick();
            current.save(regs, sf);
        }
        // trace!("Paused process #{}", self.cur_pid);
    }

    pub fn switch_next(&mut self, regs: &mut Registers, sf: &mut
InterruptStackFrame) {
        let pos = self.get_next_pos();
        let p = &mut self.processes[pos];

        // trace!("Next process {} #{}", p.name(), p.pid());
        if p.pid() == self.cur_pid {
            // the next process to be resumed is the same as the current one
            p.resume();
        } else {
            // switch to next process
            p.restore(regs, sf);
            self.cur_pid = p.pid();
```

```
        }
    }
}
```

- 进程间数据访问的互斥和同步可以通过互斥锁、信号量和管程实现，这些都不复杂。
- 进程间还可以通过管道通信。

6、I/O的处理

- 这里主要指的是获取键盘和串口的输入
- 输入的实现一般会有一个循环队列作为缓存，用来保存输入的键值，在需要输出或使用的时候取出，这样就可以保证输入的正常顺序。
- 对于键盘和串口的输入，可以提供一套统一的接口，也即键盘和串口输入的数据统统放进同一个队列，而需要输入的地方只需要从队列中取用、并在队列为空时等待即可。
- `InputStream`结构体的作用是，初始化一个输入队列、并承担弹出字符的作用。

```rust
pub struct InputStream;

impl InputStream {
    pub fn new() -> Self {
        init_INPUT_QUEUE(ArrayQueue::new(DEFAULT_BUF_SIZE));
        info!("Input stream Initialized.");
        Self
    }
}

impl Stream for InputStream {
    type Item = DecodedKey;

    fn poll_next(self: Pin<&mut Self>, cx: &mut Context) ->
Poll<Option<Self::Item>> {
        let queue = get_input_queue_for_sure();
        if let Some(key) = queue.pop() {
            Poll::Ready(Some(key))
        } else {
            INPUT_WAKER.register(&cx.waker());
            match queue.pop() {
                Some(key) => {
                    INPUT_WAKER.take();
                    Poll::Ready(Some(key))
                }
                None => Poll::Pending,
            }
        }
    }
}
```

- 需要注意的是，更多时候我们的输入并不是简单的字符，可能一串字节流在解析后是一个控制字符等等，因此我们也需要一个全局的键盘解释器，这样一个静态解释器也需要从两方同时获取输入.如果可以解析为Unicode字符那就直接输出，否则按照调试模式输出。`DecodedKey`这个枚举类型可以直接调用现有的库。

```rust
pub async fn get_key() {
    let mut input = InputStream::new();
    while let Some(key) = input.next().await {
        match key {
            DecodedKey::Unicode(c) => print!("{}", c),
            DecodedKey::RawKey(k) => print!("{:?}", k),
        }
    }
}
```

7、文件系统与页面交换

- 磁盘是一个块设备，可能含有多个分区，我们采用MBR分区表对其进行分区。分区表内部包含了磁盘大小、各分区大小等信息。MBR分区表支持至多四个物理分区。
- 每个分区可以有不同的文件系统。我们先实现了一个FAT16的文件系统，这个文件系统的实现相对简单，可以让做实验的同学不太困难地写出来。如果同学学有余力可以尝试更复杂的文件系统
-

8、系统调用

- 在RISC-V中，调用系统调用需要使用ecall指令，可以调用8号内部中断
- 我们可以在寄存器a0中存放系统调用号，a1~a7放参数。这样去使用系统调用。使用方法大致类似于C语言。
- 考虑到Rust强大的模式匹配能力，我们不需要再去编写系统调用表。
- 我们可以按照一定的规范（如POSIX），编写系统调用。
- 以下的代码仍然是x86-64下触发0x80中断的。但是两种架构下调用系统调用的思路大同小异。同样可以使用模式匹配来找到系统调用函数。

```rust
pub fn dispatcher(regs: &mut Registers, sf: &mut InterruptStackFrame) {
    let args = super::syscall::SyscallArgs::new(
        Syscall::try_from(regs.rax as u8).unwrap(),
        regs.rdi,
        regs.rsi,
        regs.rdx
    );

    trace!("{}", args);

    match args.syscall {
        // path: &str (arg0 as *const u8, arg1 as len) -> pid: u16
        Syscall::Spawn   => regs.set_rax(spawn_process(&args)),
        // pid: arg0 as u16
        Syscall::Exit    => exit_process(&args, regs, sf),
        // fd: arg0 as u8, buf: &[u8] (arg1 as *const u8, arg2 as len)
        Syscall::Read           => regs.set_rax(sys_read(&args)),
        // fd: arg0 as u8, buf: &[u8] (arg1 as *const u8, arg2 as len)
        Syscall::Write          => regs.set_rax(sys_write(&args)),
        // path: &str (arg0 as *const u8, arg1 as len), mode: arg2 as u8 -> fd: u8
        Syscall::Open           => regs.set_rax(sys_open(&args)),
        // fd: arg0 as u8 -> success: bool
```

```
        Syscall::Close          => regs.set_rax(sys_close(&args)),
        // None
        Syscall::Stat           => list_process(),
        // None -> time: usize
        Syscall::Time           => regs.set_rax(sys_clock() as usize),
        // path: &str (arg0 as *const u8, arg1 as len)
        Syscall::ListDir        => list_dir(&args),
        // layout: arg0 as *const Layout -> ptr: *mut u8
        Syscall::Allocate       => regs.set_rax(sys_allocate(&args)),
        // ptr: arg0 as *mut u8
        Syscall::Deallocate     => sys_deallocate(&args),
        // x: arg0 as i32, y: arg1 as i32, color: arg2 as u32
        Syscall::Draw           => sys_draw(&args),
        // pid: arg0 as u16 -> status: isize
        Syscall::WaitPid        => regs.set_rax(sys_wait_pid(&args)),
        // None -> pid: u16
        Syscall::GetPid         => regs.set_rax(sys_get_pid() as usize),
        // None -> pid: u16 (diff from parent and child)
        Syscall::Fork           => sys_fork(regs, sf),
        // pid: arg0 as u16
        Syscall::Kill           => sys_kill(&args, regs, sf),
        // op: u8, key: u32, val: usize -> ret: any
        Syscall::Sem            => sys_sem(&args, regs, sf),
        // None
        Syscall::None           => {}
    }
}
```

9、程序加载与用户进程

- 此时考虑从磁盘中加载程序到内存，这里一般指的是加载ELF文件
- ELF文件一般包括如下的部分：ELF Header、Program Headers和可执行程序部分
- 使用相应的结构体解析ELF Header、Program Headers。如果发现执行权限和架构都合适，那么就可以执行。
- 随后可以构建PCB，执行用户进程。
- 用户进程中的动态内存分配，可以在内核中创建一个共用的动态内存分配器
- ELF文件亦有相关的crate可供使用。

```rust
pub fn load_elf(
    elf: &ElfFile,
    physical_offset: u64,
    page_table: &mut impl Mapper<Size4KiB>,
    frame_allocator: &mut impl FrameAllocator<Size4KiB>,
    user_access: bool,
) -> Result<Vec<PageRangeInclusive>, MapToError<Size4KiB>> {
    trace!("Loading ELF file...{:?}", elf.input.as_ptr());
    elf.program_iter()
        .filter(|segment| segment.get_type().unwrap() == program::Type::Load)
        .map(|segment| load_segment(elf, physical_offset, &segment, page_table, frame_allocator, user_access))
        .collect()
}
```

10、shell

- shell的本质是，使用fork和exec运行一些特定的程序。Linux的shell支持管道，可以将上一次执行的结果作为下一次执行的输入
- 对于简单的shell实现，可以直接解析字符串，然后运行对应的函数来实现功能。

```rust
fn main() -> usize {
    let mut root_dir = String::from("/APP/");
    println!("            <<< Welcome to GGOS shell >>>            ");
    println!("                                    type `help` for help");
    loop {
        print!("[{}] ", root_dir);
        let input = stdin().read_line();
        let line: Vec<&str> = input.trim().split(' ').collect();
        match line[0] {
            "exit" => break,
            "ps" => sys_stat(),
            "ls" => sys_list_dir(root_dir.as_str()),
            "cat" => {
                if line.len() < 2 {
                    println!("Usage: cat <file>");
                    continue;
                }

                services::cat(line[1], root_dir.as_str());
            }
            "cd" => {
                if line.len() < 2 {
                    println!("Usage: cd <dir>");
                    continue;
                }

                services::cd(line[1], &mut root_dir);
            }
            "exec" => {
                if line.len() < 2 {
                    println!("Usage: exec <file>");
```

```
                    continue;
                }

                services::exec(line[1], root_dir.as_str());
            }
            "nohup" => {
                if line.len() < 2 {
                    println!("Usage: nohup <file>");
                    continue;
                }

                services::nohup(line[1], root_dir.as_str());
            }
            "kill" => {
                if line.len() < 2 {
                    println!("Usage: kill <pid>");
                    continue;
                }
                let pid = line[1].to_string().parse::<u16>();

                if pid.is_err() {
                    errln!("Cannot parse pid");
                    continue;
                }

                services::kill(pid.unwrap());
            }
            "help" => print!("{}", consts::help_text()),
            _ => println!("[=] you said \"{}\"", input),
        }
    }

    0
}
```

11、多核支持

待实现

12、图形界面

待实现

13、网络与套接字编程

待实现

14、指令集移植

- 因为我们的这个项目是基于一个自己编写的X86-64操作系统，我们在做这一步的时候其实是把x86-64移植到RISCV，然后教程反过来写就可以了。
- 要进行指令集移植，我们必须在如下方面进行更改：
    - 寄存器结构不同：riscv提供了大量的通用寄存器，但是x86-64的通用寄存器较少（虽然比x86还是多很多的），专用寄存器很多。而且一些特殊寄存器二者完全不同。
    - 汇编指令不同：这是显然的。主要注意一点：x86-64的外设是独立编址的，但是riscv的外设是统一编址的。这将涉及到所有的外设，比如硬盘和键盘。在这里，x86-64架构在rust中有很多可用的crate帮助减少汇编的使用，但是对于riscv架构，支持相对缺乏，可能使用汇编的地方相对较多。
    - 启动：uefi和rust-sbi的启动有所不同。**待填坑**
    - 页表：RISC-V多使用SV39三级页表（也有SV48四级页表可供使用），x86-64多使用四级页表。
    - 中断和系统调用：二者调用中断和系统调用的方式也有很大区别。对于x86-64需要借助中断描述符表但是riscv是不需要的

各个模块的依赖关系如下：

(一张图片)

理由（若干字）

二、已经实现的部分：

- 本组已经设计了一个基于x86-64和uefi的操作系统。且前九点绝大多数已经实现。文件系统的写入功能和交换文件还有待填坑。
- ……