Starry 设计文档

陈嘉钰、郑友捷、王昱栋

Starry是一个可以在`qemu`上运行的多核模块化OS（对于`sifive-fu740`的支持还在工作中）

背景

Starry是在`ArceOs`rcore-os/arceos: An experimental modular OS written in Rust.基础上进行开发的、以宏内核架构运行Linux应用的内核。原有的ArceOS设计架构为`Unikernel`，后续计划在原有代码结构的基础上设计宏内核架构和微内核架构，而Starry即是ArceOS宏内核架构化的一个成果。

目前测例支持

当前测例支持如下：

- musl-libc：静态链接与动态链接均已支持，实现的特性有动态库加载、线程、信号、futex等
- lua：已经支持
- busybox：已经支持大部分指令，通过了比赛的测例
- lmbench：已经支持，可以使用lmbench测算内核性能
- iperf/netperf：支持大部分测例，可以实现网络的基本功能
- UnixBench：已经支持，可以用来测算内核在文件读取、数据基本操作等方面的性能
- libc-bench：已经支持

使用方式

详见主页面的README.md

相关依赖库均已本地化在`vendor`文件夹中

架构设计介绍

ArceOS介绍

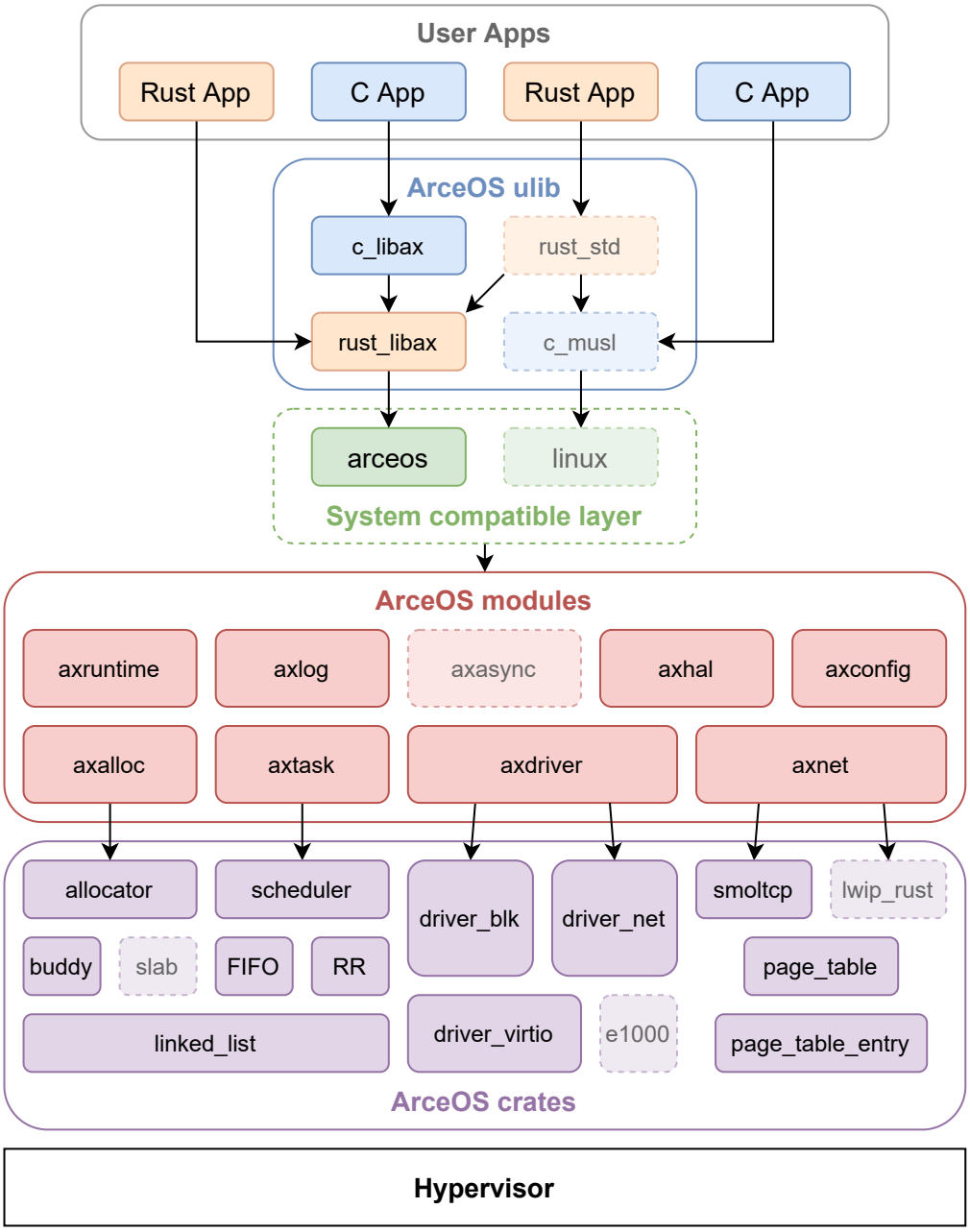
我们的Starry是基于ArceOS生成的，因此需要简单介绍一下ArceOS实现的内容。

ArceOS采用模块化组件化的设计思维，通过使用内核组件 + 组件化的OS框架 来得到 不同形态的OS kernel。

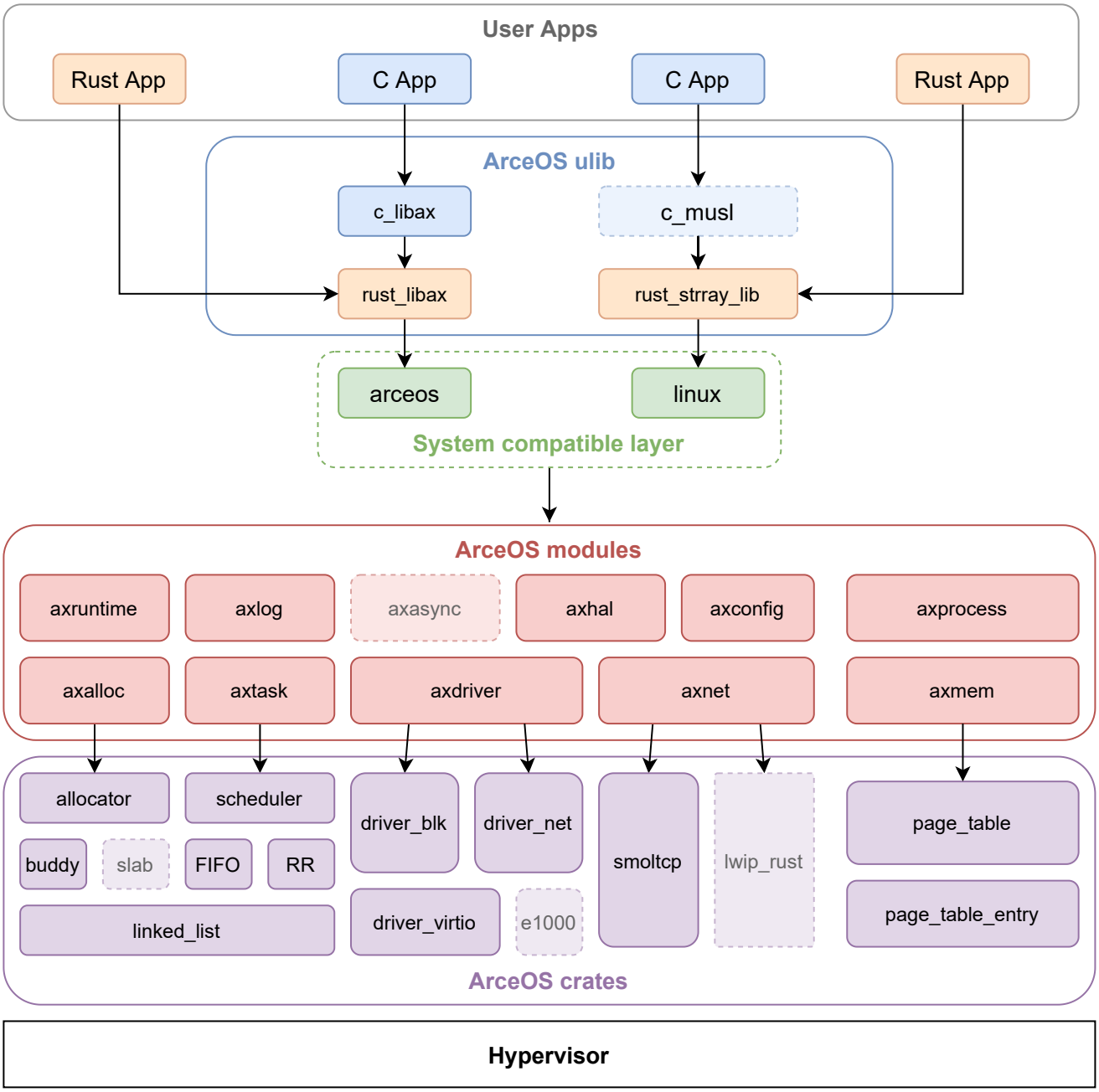
- 提供了一套组件化的操作系统框架
- 提供各种内核组件的实现，各种内核组件可在没有OS kernel的情况下独立运行
    - 如filesystem, network stack等内核组件可以在裸机或用户态以库的形式运行
    - 各种设备驱动等内核组件可以在裸机上运行
- 理想情况下可以通过选择组件构成unikernel/宏内核/微内核
- 实际上在我们开始实验时它还只支持unikernel
    - 只运行一个用户程序
    - 用户程序与内核链接为同一镜像
    - 不区分地址空间与特权级
    - 安全性由底层 hypervisor 保证

结构图对比

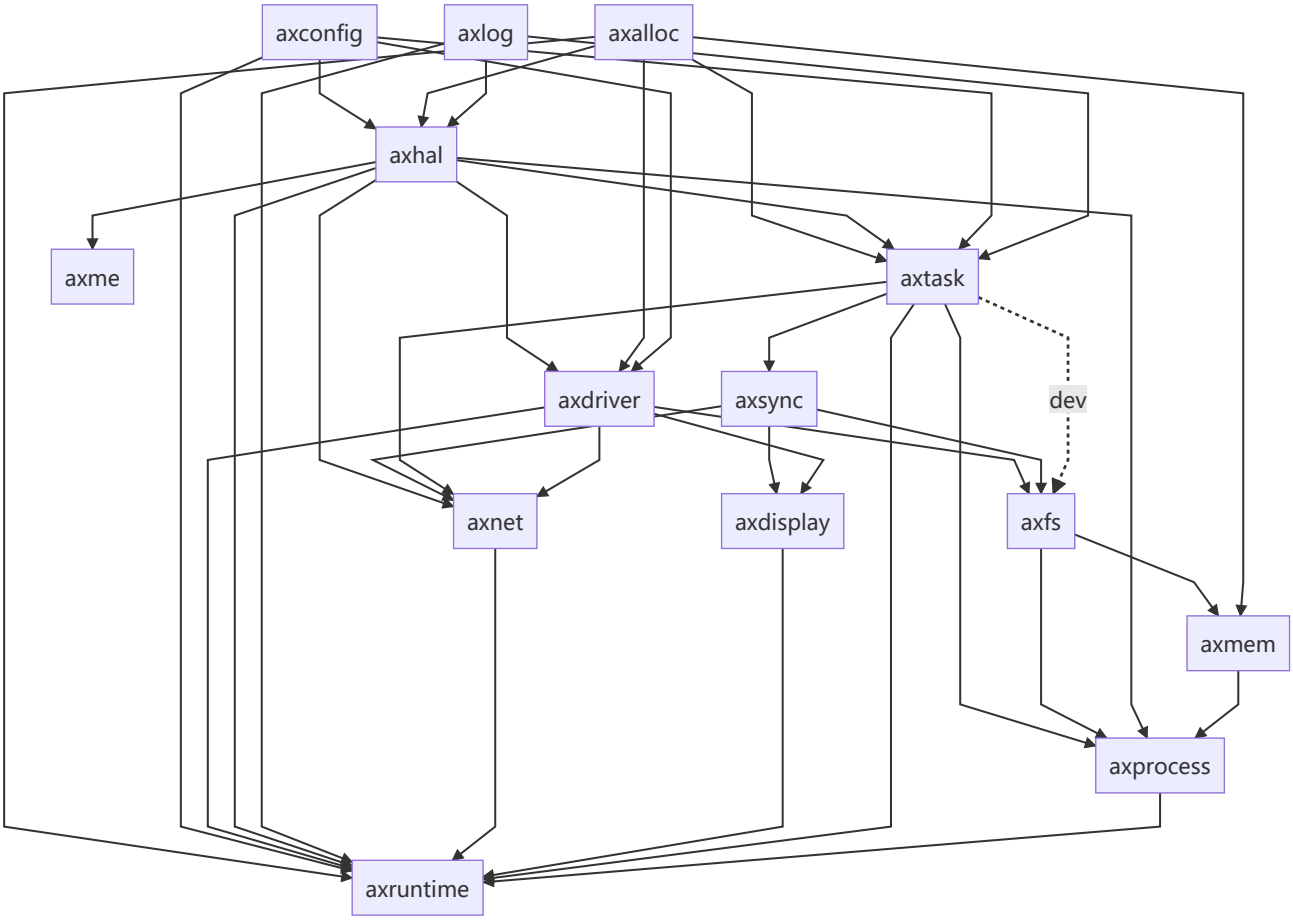
原先Arceos结构图：



重构后StarryOS架构图：



Starry的模块依赖图如下：



Starry结构说明

- crates：与OS设计无关的公共组件
- modules：与OS设计更加耦合的组件，各个模块功能简要介绍如下：
  - axalloc：实现全局分配器
  - axconfig：定义内核参数
  - axdisplay：简单的图形化实现
  - axdriver：设备驱动管理
  - axfs：文件系统支持
  - axhal：硬件抽象层，定义了一系列的平台API，如trap入口等
  - axlog：log输出层
  - axnet：网络模块

    - axruntime：运行库，定义了内核的启动逻辑
    - axsync：实现同步模块
    - axmem：地址空间模块
    - axprocess：进程模块，也实现了动态加载
    - axsignal：信号模块
    - axtask：定义了任务与调度序列的操作
- apps：unikernel架构下的用户程序，继承原有ArceOS
- ulib：用户库，继承原有ArceOS，并添加了starry_libax部分作为Linux兼容层。

1. 为了实现宏内核架构体系，需要对原有Arceos的部分核心模块（如axtask）进行修改。为了防止合并时冲突过多，因此在对应模块下建立`monolithic*`为前缀的文件夹，存放为宏内核架构实现的内容。同时使用条件编译来选择是宏内核架构还是unikernel架构。
2. 为了实现linux APP兼容，需要实现一系列面向linux的系统调用。我们将系统调用的具体实现部分放在`starry_libax`部分，即以用户库的形式形成一个linux兼容层。通过调用上述模块提供的一系列接口，实现对应的linux 系统调用，并暴露给外界。这个系统兼容层与原有的`libax`进行对应，分别提供不同的接口与服务。
3. 模块部分放置可以为宏内核与unikernel尽可能共享的内容，通过条件编译等方式做到尽可能地为不同架构下的兼容层所调用。

结构优势

1. 用相同的代码组件，利用条件编译等方式可以组建出不同架构的OS内核，从而可以达到使用不同的启动参数来启动不同架构的内核，大大提高内核的泛用性。
2. 在利用unikernel可插拔的特性的基础上，可以使得实现某一个功能的crate和整体OS进一步解耦，从而利于更新换代。
3. 解耦的特性利于开发模式的分工，不同开发人员可以负责不同的module或者crate内容，只要实现了对应的接口便可以较好地适配本内核，从而方便其他开发人员参与其中不断完善。

设计思路

从unikernel架构的ArceOS转变为宏内核的Starry，需要仔细考虑两者之间的不同，并且在原有代码的基础上去进行调整改造。

两者较为重要的不同有以下几点：

- unikernel不区分特权级，全过程不会进入U态，大部分情况下在S态，而宏内核有区分用户态和内核态，不同特权级对代码的安全性做出了一定的保障
- unikernel不区分地址空间
- unikernel没有进程的概念，只能运行一个应用，应用可以通过spawn生成不同任务，不同任务之间的关系始终是平等的（调度任务除外）

而在这些差异的基础上，我们做出了如下设计：

进程引入

为了运行Linux相关的应用，我们需要让不同任务之间存在父子等关系，因此我们引入了进程的概念。在标准的Linux中，进程和线程统一用pthread结构体代替，但我们为了保证原有arceos的任务调度结构不受过大影响，因此选择将进程和线程进行分离，进程保存在独立的结构体`Process`中。

依据模块化的思想，我们可以将进程视为一个容器，存储了各类运行时资源，包括虚存、文件描述符、
线程、信号等。
在该种设计理念下，进程仅是对上述资源的一个统一与包装。因此可以通过添加 feature 等方式将进程作为一个可插拔模块，使得内核在宏内核架构与微内核架构中随时进行切换。

进程结构设计如下：

```
pub struct ProcessInner {
    /// 父进程的进程号
    pub parent: u64,
    /// 子进程
    pub children: Vec<Arc<Process>>,
    /// 子任务
    pub tasks: Vec<AxTaskRef>,
    /// 地址空间，由于存在地址空间共享，因此设计为Arc类型
    pub memory_set: Arc<SpinNoIrq<MemorySet>>,
    /// 用户堆基址，任何时候堆顶都不能比这个值小，理论上讲是一个常量
    pub heap_bottom: usize,
    /// 当前用户堆的堆顶，不能小于基址，不能大于基址加堆的最大大小
    pub heap_top: usize,
    /// 进程状态
    pub is_zombie: bool,
    /// 退出状态码
    pub exit_code: i32,
    /// 文件管理器，存储如文件描述符等内容
    #[cfg(feature = "fs")]
    pub fd_manager: FdManager,
    /// 进程工作目录
    pub cwd: String,
    #[cfg(feature = "signal")]
    /// 信号处理模块
    /// 第一维代表线程号，第二维代表线程对应的信号处理模块
    pub signal_module: BTreeMap<u64, SignalModule>,

    /// robust list存储模块
    /// 用来存储线程对共享变量的使用地址
    /// 具体使用交给了用户空间
    pub robust_list: BTreeMap<u64, FutexRobustList>,
}
```

任务结构如下：

```rust
/// The inner task structure.
pub struct TaskInner {
    id: TaskId,
    name: String,
    is_idle: bool,
    is_init: bool,
    /// 任务的入口函数，仅在内核态下有效
    entry: Option<*mut dyn FnOnce()>,
    state: AtomicU8,

    in_wait_queue: AtomicBool,
    #[cfg(feature = "irq")]
    in_timer_list: AtomicBool,

    #[cfg(feature = "preempt")]
    need_resched: AtomicBool,
    #[cfg(feature = "preempt")]
    pub preempt_disable_count: AtomicUsize,

    exit_code: AtomicI32,
    wait_for_exit: WaitQueue,
    /// 内核栈，对于unikernel不需要
    #[cfg(feature = "monolithic")]
    kstack: Option<TaskStack>,

    ctx: UnsafeCell<TaskContext>,

    #[cfg(feature = "monolithic")]
    // 对应进程ID
    process_id: AtomicU64,

    #[cfg(feature = "monolithic")]
    /// 是否是所属进程下的主线程
    is_leader: AtomicBool,

    #[cfg(feature = "monolithic")]
    // 所属页表ID，在宏内核下默认会开启分页，是只读的所以不用原子量
    page_table_token: usize,

    #[cfg(feature = "monolithic")]
    /// 初始化的trap上下文
    pub trap_frame: UnsafeCell<TrapFrame>,
    // 时间统计
    #[cfg(feature = "monolithic")]
    time: UnsafeCell<TimeStat>,

    #[allow(unused)]
    #[cfg(feature = "monolithic")]
    /// 子线程初始化的时候，存放tid的地址
```

```
    set_child_tid: AtomicU64,

    #[cfg(feature = "monolithic")]
    /// 子线程初始化时，将这个地址清空；子线程退出时，触发这里的 futex。
    /// 在创建时包含 CLONE_CHILD_SETTID 时才非0，但可以被 sys_set_tid_address
修改
    clear_child_tid: AtomicU64,

    #[cfg(feature = "monolithic")]
    /// 退出时是否向父进程发送SIG_CHILD
    pub send_sigchld_when_exit: bool,
}
```

该种设计的优势如下：

- 保留了 ArceOS 的结构，可以较为方便地与其他同学开发结果进行结合
- 耦合度低，因此可以使内核较为方便地在不同模式间进行切换

在该种设计架构下，接受外来系统调用时，需要将部分对线程进行操作的系统调用转发给进程。进程收
到该系统调用之后，再对当前进程下正在运行的线程进行相应的操作。实例为 yield ， exit 等。

在生成新的任务时，由于是通过Linux的clone调用生成新的任务，因此可以根据clone的参数判断生成的是新的进程还是线程，从而确定线程所属的进程是哪一个，进程与线程之间形成父子关系，而同一进程下的线程形成兄弟关系，从而可以更加方便地进行管理。

地址空间引入

任务切换

引入了进程之后，由于进程是资源容器集合，因此地址空间相关的存储结构也存放在这里，不同进程之间可以共享或者独享地址空间，因此在切换任务时，只需要额外判断当前所属进程的地址空间的token是否发生改变，就可以完成多地址空间的引入。

特权级切换

目前内核和用户态使用的是同一个地址空间，可以避免trap时更改页表，减少时空损耗。

特权级切换

在Starry中，各种测例运行在用户态下，从内核态进入到用户态的方式有两个：用户程序初始化进入和trap返回。

初始化进入

对于用户程序初始化进入部分，即是在原有ArceOS基础上添加了额外的判断：

判断的原则如下：若要执行的任务的入口函数在内核态，则直接调用即可。否则需要通过手写汇编代码保存寄存器，以类似trap返回的机制调用sret进入用户态执行对应的函数。

```
extern "C" fn task_entry() {
    // release the lock that was implicitly held across the reschedule
    unsafe { crate::RUN_QUEUE.force_unlock() };
    axhal::arch::enable_irqs();
    let task: CurrentTask = crate::current();
    if let Some(entry) = task.entry {
        if task.get_process_id() == KERNEL_PROCESS_ID {
            // 对于unikernel，这里是应用程序的入口，由于都在内核态所以可以直接调用
函数
            // 对于宏内核，这是初始调度进程，也在内核态，直接执行即可
            unsafe { Box::from_raw(entry)() };
        } else {
            // 需要通过切换特权级进入到对应的应用程序
            let kernel_sp = task.get_kernel_stack_top().unwrap();

            let frame_address = task.get_first_trap_frame();
            // 切换页表已经在switch实现了
            first_into_user(kernel_sp, frame_address as usize);
        }
    }
    // only for kernel task
    crate::exit(0);
}

/// 初始化主进程的trap上下文
#[no_mangle]
fn first_into_user(kernel_sp: usize, frame_base: usize) -> ! {
    let trap_frame_size = core::mem::size_of::<TrapFrame>();
    let kernel_base = kernel_sp - trap_frame_size;
    // 在保证将寄存器都存储好之后，再开启中断
    // 否则此时会因为写入csr寄存器过程中出现中断，导致出现异常
    axhal::arch::disable_irqs();
    // 在内核态中，tp寄存器存储的是当前任务的CPU ID
    // 而当从内核态进入到用户态时，会将tp寄存器的值先存储在内核栈上，即把该任务对应的
CPU ID存储在内核栈上
    // 然后将tp寄存器的值改为对应线程的tls指针的值
    // 因此在用户态中，tp寄存器存储的值是线程的tls指针的值
```

```
    // 而当从用户态进入到内核态时，会先将内核栈上的值读取到某一个中间寄存器t0中，然后将tp的值存入内核栈
    // 然后再将t0的值赋给tp，因此此时tp的值是当前任务的CPU ID
    // 对应实现在axhal/src/arch/riscv/trap.S中
    unsafe {
        asm::sfence_vma_all();
        core::arch::asm!(
            r"
            mv      sp, {frame_base}
            .short  0x2432                      // fld fs0,264(sp)
            .short  0x24d2                      // fld fs1,272(sp)
            LDR     gp, sp, 2                   // load user gp and tp
            LDR     t0, sp, 3
            mv      t1, {kernel_base}
            STR     tp, t1, 3                   // save supervisor tp，注意是存储到内核栈上而不是sp中，此时存储的应该是当前运行的CPU的ID
            mv      tp, t0                      // tp：本来存储的是CPU ID，在这个时候变成了对应线程的TLS 指针
            csrw    sscratch, {kernel_sp}       // put supervisor sp to scratch
            LDR     t0, sp, 31
            LDR     t1, sp, 32
            csrw    sepc, t0
            csrw    sstatus, t1
            POP_GENERAL_REGS
            LDR     sp, sp, 1
            sret
        ",
            frame_base = in(reg) frame_base,
            kernel_sp = in(reg) kernel_sp,
            kernel_base = in(reg) kernel_base,
        );
    };
    core::panic!("already in user mode!")
}
```

trap切换

trap切换对应的汇编代码在`axhal/src/arch/riscv`，值得关注的是其嵌套trap的处理。在第一次进入trap时，是从用户态进入到内核态，此时会将内核栈的地址赋给sp，将用户栈的地址存在内核栈上，并将sscratch清零。若发生内核嵌套trap，则此时sscratch的值为0，与sp交换之后，sp为0，即发生了内核嵌套trap。

因此可以通过交换之后sp是否为0来判断是否发生了内核嵌套trap。

实现重点

依赖问题

由于ArceOS自身的unikernel架构，不同模块需要保持一定的依赖关系，从而可以方便地通过条件编译等操作来解耦某些模块，使用某些指定的模块来启动内核，从而增强OS的泛用性。

查看项目的依赖关系

项目的依赖关系可以通过对应的toml配置文件进行查看。如下列为axmem模块的toml：

```
# modules/axmem/Cargo.toml
[dependencies]
log = "0.4"
axhal = { path = "../axhal", features = ["paging"] }
axalloc = { path = "../axalloc" }
axconfig = { path = "../axconfig" }
axerrno = { path = "../../crates/axerrno" }
axfs = { path = "../axfs" }
axio = { path = "../../crates/axio" }
spinlock = { path = "../../crates/spinlock" }
xmas-elf = { path = "../../extern_crates/xmas-elf-0.9.0" }
riscv = { path = "../../extern_crates/riscv-0.10.1" }
page_table_entry = { path = "../../crates/page_table_entry" }
```

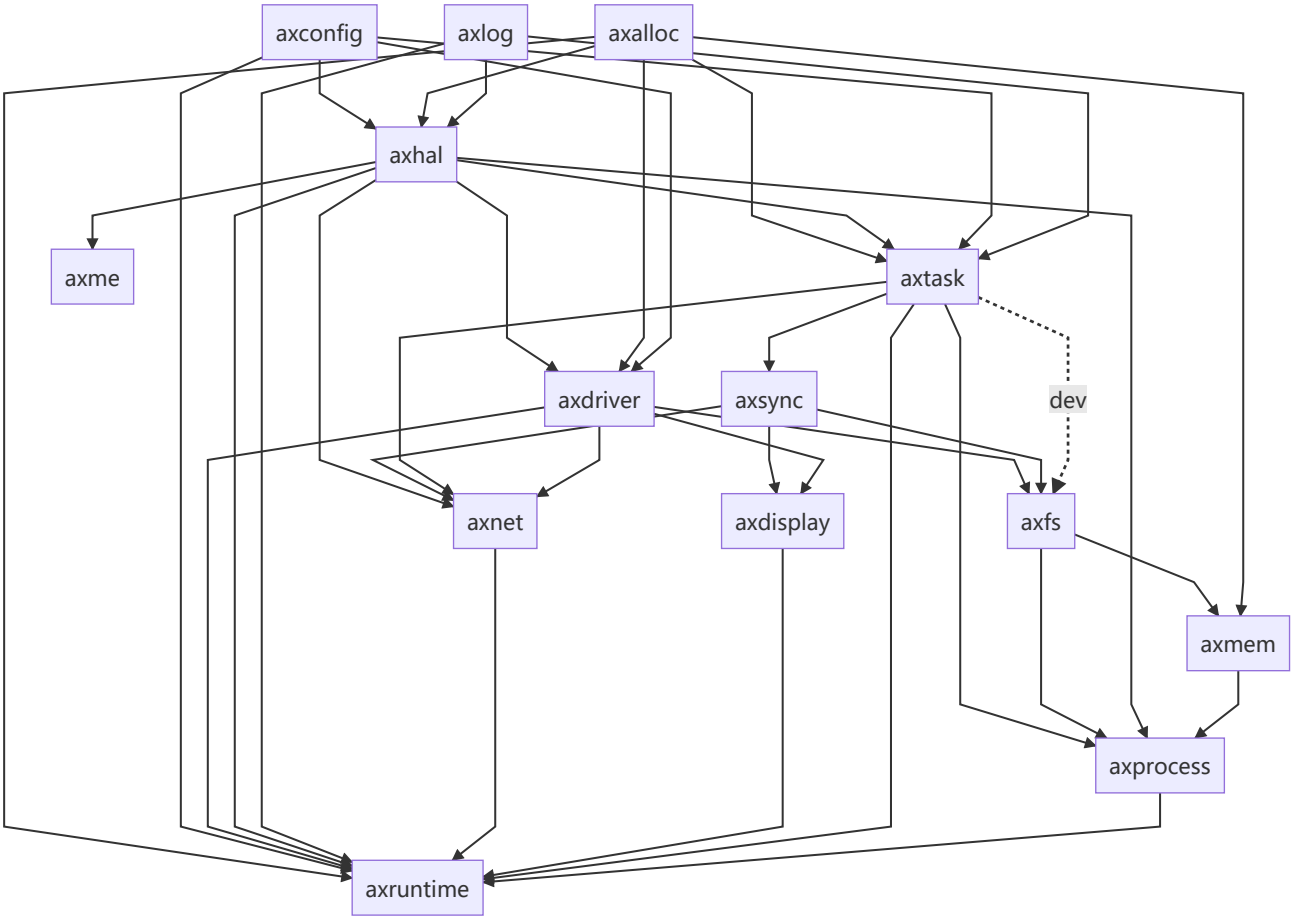
以上就可以看出axmem依赖了axhal/axfs/axconfig等模块。

循环依赖问题

而Starry虽然是宏内核架构，但仍然保持了这一泛用特性，但这也为我们开发带来了一些问题，即循环依赖问题。

> 由于*ArceOS*的模块化设计，不同的*modules*之间会形成以*module*为单位的依赖关系，相较于以文件为
> 单位的依赖关系而言更容易产生循环依赖的问题。
>
> 一个例子：假如一个项目中有*A*、*B*、*C*三个文件，*A*依赖*B*、*B*依赖*C*，不会有任何问题；但如果三
> 个文件被解耦到两个不同的项目 *M*和*N*中，*M*中有*A*和*C*，*N*中有*B*，那么*M*和*N*之间就会发生相互依
> 赖。
> 这种情况在我们的开发过程中并不少见。

当前Starry的模块依赖图如下：



如axhal需要定义trap入口，而trap实现需要很多模块的支持如axmem的地址空间等，此时就可能出现循环依赖的情况，即axhal依赖于axmem，而axmem依赖于axhal。

为了解决这个问题，有以下几种方法：

1. 优化结构设计，即尽可能将实现的功能进行划分，如地址空间内容独立出来放在axmem，而不是和进程控制一起放在axprocess。
2. 通过ArceOS提供的模块crate_interface中的call_interface和def_interface，在底层模块定义好相关的函数之后，交给上层模块去实现。

   如在axhal中定义了TrapHandler如下：

```
#[def_interface]
pub trait TrapHandler {
    /// Handles interrupt requests for the given IRQ number.
```

```
    fn handle_irq(irq_num: usize);
    // more e.g.: handle_page_fault();
    // 需要分离用户态使用
    #[cfg(feature = "monolithic")]
    fn handle_syscall(syscall_id: usize, args: [usize; 6]) -> isize;

    #[cfg(feature = "paging")]
    fn handle_page_fault(addr: VirtAddr, flags: MappingFlags, tf:
&mut TrapFrame);

    #[cfg(feature = "paging")]
    fn handle_access_fault(addr: VirtAddr, flags: MappingFlags);

    /// 处理当前进程的信号
    #[cfg(feature = "signal")]
    fn handle_signal();

    /// 为了lmbench特判，即在出现未能处理的情况，不panic，而是退出当前进程
    #[cfg(feature = "monolithic")]
    fn exit();
}
```

而在starry_libax/trap.rs完成了对TrapHandler的实现：

```
#[crate_interface::impl_interface]
impl axhal::trap::TrapHandler for TrapHandlerImpl {
    fn handle_irq(irq_num: usize) {
        /// ..
    }
    fn handle_syscall(syscall_id: usize, args: [usize; 6]) -> isize
{
        /// ..
    }

    #[cfg(feature = "paging")]
    fn handle_page_fault(addr: VirtAddr, flags: MappingFlags, tf:
&mut TrapFrame) {
        /// ..
    }

    fn handle_access_fault(addr: VirtAddr, flags: MappingFlags) {
        /// ..
    }

    #[cfg(feature = "signal")]
    fn handle_signal() {
        /// ..
    }
```

```
    fn exit() {
        /// ..
    }
}
```

而在axruntime/src/trap.rs中定义了ArceOS原有的unikernel架构下的trap实现：

```
/// 仅用作非宏内核下的trap入口

struct TrapHandlerImpl;

#[crate_interface::impl_interface]
impl axhal::trap::TrapHandler for TrapHandlerImpl {
    fn handle_irq(_irq_num: usize) {
        #[cfg(feature = "irq")]
        {
            let guard = kernel_guard::NoPreempt::new();
            axhal::irq::dispatch_irq(_irq_num);
            drop(guard); // rescheduling may occur when preemption
is re-enabled.
        }
    }
}
```

通过不同的TrapHandler的实现，可以实现宏内核和unikernel架构下不同trap实现的支持。

兼容问题

为了保证Starry在未来的泛用性，我们在比赛开发过程中便有意识地去注意不同架构下的实现兼容，并采用条件编译等方式进行区分。如process部分的结构体的定义为：

```
pub struct ProcessInner {
    /// 父进程的进程号
    pub parent: u64,
    /// 子进程
    pub children: Vec<Arc<Process>>,
    /// 子任务
    pub tasks: Vec<AxTaskRef>,
    /// 地址空间，由于存在地址空间共享，因此设计为Arc类型
    pub memory_set: Arc<SpinNoIrq<MemorySet>>,
    /// 用户堆基址，任何时候堆顶都不能比这个值小，理论上讲是一个常量
    pub heap_bottom: usize,
    /// 当前用户堆的堆顶，不能小于基址，不能大于基址加堆的最大大小
    pub heap_top: usize,
    /// 进程状态
```

```
    pub is_zombie: bool,
    /// 退出状态码
    pub exit_code: i32,
    // /// 文件描述符表
    // pub fd_table: Vec<Option<Arc<SpinNoIrq<dyn FileIO>>>>,
    // /// 文件描述符上限，由prlimit设置
    // pub fd_limit: usize,
    #[cfg(feature = "fs")]
    pub fd_manager: FdManager,
    /// 进程工作目录
    pub cwd: String,
    #[cfg(feature = "signal")]
    /// 信号处理模块
    /// 第一维代表线程号，第二维代表线程对应的信号处理模块
    pub signal_module: BTreeMap<u64, SignalModule>,

    /// robust list存储模块
    /// 用来存储线程对共享变量的使用地址
    /// 具体使用交给了用户空间
    pub robust_list: BTreeMap<u64, FutexRobustList>,
}
```

其额外限定了fs和signal的feature，规定了信号模块和文件系统模块的条件编译，可以根据编译参数来决定内核是否支持fs和信号模块。

问题与解决

操作CSR寄存器时关闭时钟中断

由于要运行多个测例，每一个测例都是单独的可执行文件，而当测例量加大时，会经常性出现准备开始运行某个测例的时候卡死或者循环发生内核trap的现象，错误的内容是随机的，可能呈现为store fault、卡死甚至unknown trap等内容。

通过gdb调试，定位了第一次发生错误的地址均在下列函数中：

```
#[no_mangle]
// #[cfg(feature = "user")]
fn first_into_user(kernel_sp: usize, frame_base: usize) -> ! {
    let trap_frame_size = core::mem::size_of::<TrapFrame>();
    let kernel_base = kernel_sp - trap_frame_size;
    unsafe {
        asm::sfence_vma_all();
        core::arch::asm!(
            r"
            mv      sp, {frame_base}
```

```
            .short  0x2432                          // fld fs0,264(sp)
            .short  0x24d2                          // fld fs1,272(sp)
            LDR     gp, sp, 2                       // load user gp and tp
            LDR     t0, sp, 3
            mv      t1, {kernel_base}
            STR     tp, t1, 3                       // save supervisor tp，注
意是存储到内核栈上而不是sp中，此时存储的应该是当前运行的CPU的ID
            mv      tp, t0                          // tp：本来存储的是CPU ID，
在这个时候变成了对应线程的TLS 指针
            csrw    sscratch, {kernel_sp}           // put supervisor sp to
scratch
            LDR     t0, sp, 31
            LDR     t1, sp, 32
            csrw    sepc, t0
            csrw    sstatus, t1
            POP_GENERAL_REGS
            LDR     sp, sp, 1
            sret
        ",
            frame_base = in(reg) frame_base,
            kernel_sp = in(reg) kernel_sp,
            kernel_base = in(reg) kernel_base,
        );
    };
    core::panic!("already in user mode!")
}
```

继续定位gdb汇编代码，将错误地址进一步确定为：

```
csrw    sstatus, t1
```

当执行这一条汇编代码，会出现不可预测的错误。

考虑到sstatus的功能，其可以控制时钟中断、特权级等一系列的内容，通过查阅资料了解到当使用sstatus屏蔽内核中时钟中断时，并非是阻止了中断发出，而是阻止对中断进行处理，一旦内核中时钟中断屏蔽关闭，原先积累的时钟中断会被用来处理。sstatus是控制中断的关键寄存器，查阅资料得知，riscv要求在修改sstatus信息的时候需要保证时钟中断使能关闭，否则会产生不可预料的行为。

因此在调用`first_into_user`函数前需要手动调用`axhal::arch::enable_irqs()`关闭中断使能。

读取长度不足

当读取`strings.lua`测例时，常会有报错：`strings.lua:1: unexpected symbol`。但其他lua测例运行结果正常且正确。

strings.lua内容如下：

```
local str = "Jelly Think"

result = 0

-- string.len可以获得字符串的长度

if string.len(str) ~= 11 then
    result = -1
end

-- string.rep返回字符串重复n次的结果

str = "ab"

if string.rep(str, 2) ~= "abab" then
    result = -1
end

-- string.lower将字符串小写变成大写形式，并返回一个改变以后的副本

str = "Jelly Think"

if string.lower(str) ~= "jelly think" then
    result = -1
end

-- string.upper将字符串大写变成小写形式，并返回一个改变以后的副本

if string.upper(str) == "JELLY THINK" then
    result = -1
end

return result
```

考虑是否是文件编码问题：将strings.lua测例内容复制到其他lua文件，运行对应文件，报错不变。将其他lua文件内容复制到strings.lua，运行strings.lua，结果正常，因此排除了编码问题。

再考虑内核是否成功读入文件，输出`read`系统调用内容，发现read系统调用确实正确读入并返回了文件长度591。

考虑到strings.lua测试内容由多个断言形成，考虑逐个断言逐个断言验证，但发现一旦删除了strings.lua的部分内容之后，运行结果就正常了。即当缩短了strings.lua的长度之后，结果正常。

考虑输出read系统调用中读入的buf的内容，发现读入的892位buf中，前512位buf被修改为了0，后面的buf仍然正常。而当限制buf长度为512时，此时前512位buf读取结果正常。

arceos原先在fat32节点的读取函数定义如下：

```
fn read_at(&self, offset: u64, buf: &mut [u8]) -> VfsResult<usize> {
    let mut file = self.0.lock();
    file.seek(SeekFrom::Start(offset)).map_err(as_vfs_err)?; // TODO:
more efficient
    file.read(buf).map_err(as_vfs_err)
}
```

依据上述debug结果，发现starry的文件系统驱动中fat32的块大小定义为512。查询read语义知，每一次实际读取长度不一定等于传入的buf长度，因此不可以直接通过传入buf来实现文件的读入，而需要进行循环判断：

```
n read_at(&self, offset: u64, buf: &mut [u8]) -> VfsResult<usize> {
    let mut file = self.0.lock();
    file.seek(SeekFrom::Start(offset)).map_err(as_vfs_err)?; // TODO:
more efficient
    // file.read(buf).map_err(as_vfs_err)
    let buf_len = buf.len();
    let mut now_offset = 0;
    let mut probe = buf.to_vec();
    while now_offset < buf_len {
        let ans = file.read(&mut probe).map_err(as_vfs_err);
        if ans.is_err() {
            return ans;
        }
        let read_len = ans.unwrap();

        if read_len == 0 {
            break;
        }
        buf[now_offset..now_offset +
read_len].copy_from_slice(&probe[..read_len]);
        now_offset += read_len;
        probe = probe[read_len..].to_vec();
    }
    Ok(now_offset)
}
```

依据上述写法，即可实现大文件的读入。

文件的链接与初始化

fat32文件系统自身不支持符号链接，因此需要在内核中手动维护一个链接映射。但是不同的文件名称字符串可能指向同一个文件， 因此不能单纯地将传入的文件名作为映射的键值。

比如我们建立了一个从a.out到b.out的链接，此时传入的文件名叫做./a.out，此时它应该被连接到b.out，但在链接中找不到对应程序。

为了规范化起见，starry引用了arceos提供的canonicalize函数，将文件名转化为统一格式的绝对路径，并以此建立文件名到链接实际文件的映射。

因此从a.out到b.out的链接，会被转化为./a.out到./b.out的链接，通过规范的字符串使得误判的情况可以被减少。

实现busybox、lua、lmbench过程中，需要用到一系列的链接，对应实现在starry_libax/test.rs的fs_init函数中。如程序会寻找/lib/tls_get_new-dtv_dso.so，而它会被定向到./tls_get_new-dtv_dso.so文件，这个过程需要我们手动建立链接。

另外，busybox等测例也需要我们手动建立一系列的文件系统与文件夹，如dev_fs与ram_fs，其中dev_fs并不允许动态增删文件内容，需要初始化时就确定好。相关的实现在axfs/src/root.rs的init_rootfs函数，需要添加dev/shm、dev/misc等文件夹。

lmbench测例的结束

运行lmbench测例时发现时，程序总是会访问0x2,0x4,0x6,0x8等地址，导致page fault。gdb进行debug无果，发现程序已经输出了预期输出，之后会直接访问该非法地址。

询问往年参加比赛的学长，了解到去年的lmbench会在每个测例结束的时候直接让pc跳到堆栈上，从而触发I fault，通过内核捕获该信号并进行特判，从而手动调用exit结束当前的lmbench测例，进入到下一个lmbench测例。

而在今年编译得到的lmbench版本，pc不再跳转到堆栈，而是跳转到低地址如0x6，此时也是要求内核直接做出特判，结束当前任务。

查阅riscv规范得知，非法访问内存，内核处理失败之后应当发送SIGSEGV信号到对应线程，从而结束当前任务。因此修改代码如下：

```
#[cfg(feature = "paging")]
fn handle_page_fault(addr: VirtAddr, flags: MappingFlags, tf: &mut
TrapFrame) {
    use axprocess::handle_page_fault;
```

```
        use axsignal::signal_no::SignalNo;
        use axtask::current;
        axprocess::time_stat_from_user_to_kernel();
        use crate::syscall::signal::{syscall_sigreturn, syscall_tkill};
        if addr.as_usize() == SIGNAL_RETURN_TRAP {
            // 说明是信号执行完毕，此时应当执行sig return
            tf.regs.a0 = syscall_sigreturn() as usize;
            return;
        }

        if handle_page_fault(addr, flags).is_err() {
            // 如果处理失败，则发出sigsegv信号
            let curr = current().id().as_u64() as isize;
            axlog::error!("kill task: {}", curr);
            syscall_tkill(curr, SignalNo::SIGSEGV as isize);
        }
        axprocess::time_stat_from_kernel_to_user();
    }
```

这种情况不仅可以处理pc跳转到低地址的情况，也可以处理跳转到堆栈的情况，更加地规范化。