ArceOS兼容linux程序的宏内核扩展

1. 开发流程

在开始实验时，ArceOS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

我们希望在它的基础上实现一个可以兼容linux的各种syscall的操作系统。

1.1 第一步：从unikernel到宏内核

1. rcore：RustSBI运行在M态，通过切换到内核所在start代码进入S态。内核通过调用`__restore`函数转换到U态，运行应用程序。因此应用程序运行在U态，而内核运行在S态。 应用程序通过调用`ecall`触发CPU切换到S态，进入`trap_handler`。
2. acreos：RustSBI运行在M态，通过切换到内核所在start代码进入S态。内核执行应用程序时也未改变`sstatus`，因此跳转到应用程序之后仍然在S态。给定的应用程序示例(如`yield`程序)并未调用`ecall`，而是直接调用内核的函数接口，故未发生特权级切换。

> `exception`程序的`ebreak`并未导致特权级切换，只是触发了一个S态下的异常。

- 为了实现初赛要求的各种系统调用，我们如果想要保证应用程序运行在U态下，因此就需要实现Trap切换。

  在`ArceOS`中，关于`trap`的切换实现在`modules/axhal/src/riscv/trap.S`，这里实现的是一个完整的切换流程，即包括了从用户态进入内核态与从内核态进入用户态。

  但是我们最开始启动内核并且运行应用程序时，需要放置初始化的`Trap`上席文，并且从内核的S态进入到U态，此时原先的`trap.S`无法满足要求，因此需要手动写一个切换函数。

  `ArceOS`的启动任务逻辑如下：

  1. 每一个`task`在初始化时都会指定一个入口函数`task_entry`，同时在`entry`中指定其初始运行的函数。

```
pub fn new<F>(
    entry: F,
    name: &'static str,
    stack_size: usize,
    process_id: u64,
    page_table_token: usize,
) -> AxTaskRef
where
F: FnOnce() + Send + 'static,
{
    let mut t = Self::new_common(TaskId::new(), name, process_id,
page_table_token);
    t.set_leader(true);
    let kstack = TaskStack::alloc(align_up_4k(stack_size));
    t.entry = Some(Box::into_raw(Box::new(entry)));
    t.ctx.get_mut().init(task_entry as usize, kstack.top());
```

```
    t.kstack = Some(kstack);
    if name == "idle" {
        t.is_idle = true;
    }
    Arc::new(AxTask::new(t))
}
```

2. 当任务被调度准备执行时，会跳转到`task_entry`。此时需要根据任务的性质进行判断。
    - 若任务是运行在内核态，即是gc或idle任务，或是微内核架构下的用户程序，则此时内核不需要切换特权级，可以直接通过运行`entry()`来启动该任务。
    - 若任务运行在用户态，即是宏内核架构下的用户程序，则此时需要手写一个特权级切换函数，在`sepc`中写入启动函数的地址，通过`sret`启动，即是下文的`first_into_user`函数。

```
///modules/axtask/src/task.rs

/// 初始化主进程的trap上下文
#[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}
            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
            mv      tp, t0                      // tp：线程指针
            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!")
}

#[no_mangle]
/// 本线程将会执行的函数
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.process_id == KERNEL_PROCESS_ID {
            // 是内核态下运行任务，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.trap_frame.get();
            // 切换页表已经在switch实现了
            first_into_user(kernel_sp, frame_address as usize);
        }
    }
    // 任务执行完成，释放自我
    unreachable!("test!");
    // crate::exit(0);
```

2. 分离页表，引入虚拟地址

   为了性能和方便性起见，`StarryOS`实现的是单页表机制，即当用户程序`trap`入内核态时不会切换地址空间。

   在这种架构下，用户态下的地址空间建立流程如下：

   1. 拷贝原先内核地址空间的全部内容，使得在用户地址空间下，内核依旧可以访问自身的代码与数据。
   2. 通过用户程序的ELF文件插入对应的代码和数据，从而完成用户地址空间创建。
   3. 当初始化进入用户程序，即上文的`first_into_user`时，则切换`satp`寄存器，切换到对应用户地址空间。
   4. 当在内核态进行任务调度切换时，检查`satp`是否要进行切换。

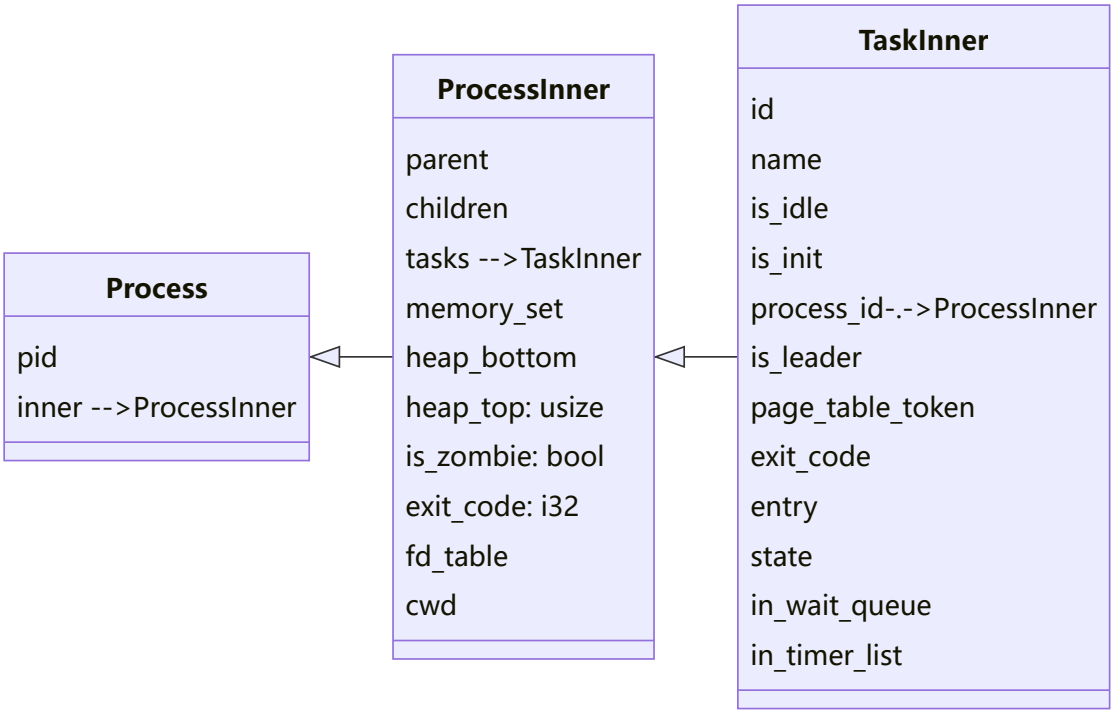
3. 建立进程与线程的抽象

   `Arceos`的原有设计指向`unikernel`，并未引入进程的概念，原有的`axtask`模块承担的更像是线程的任务。为了适配宏内核，需要引入进程的概念。而在如何引进这方面，我们团队成员有两种不同的意见：

   1. 为了最大程度保留`ArceOs`原有内容，计划将进程和线程分离，在`axtask`上额外加入一个模块`axprocess`，作为进程模块，即类似于`rCore`的进程控制块与线程控制块分离的结构。
   2. 为了尽可能贴近`linux`的设计理念，决定将进程和线程统一在一个数据结构中，类似于`pthread`的设计理念。因此设计过程中会避开原有的`axtask`模块，自行设计一个新的模块，承担进程与线程的功能。

   我们在两个方向上都有推进，最终选择了第一种方案。

   在原有module的基础之上，添加了`axprocess`模块，维护了一个进程管理结构体，而线程则是在原有的axtask基础上简单修改。它们的抽象大致如下：



1.2 第二步 支持初赛的syscall并通过在线测试

1. 初赛syscall

> syscall_openat, syscall_close, syscall_read, syscall_write, syscall_exit等三十多个系统调用

文件系统相关系统调用：
SYSCALL_CLOSE
SYSCALL_READ
SYSCALL_WRITE
SYSCALL_EXIT
SYSCALL_GETCWD
SYSCALL_PIPE2
SYSCALL_DUP
SYSCALL_DUP3
SYSCALL_MKDIRAT
SYSCALL_CHDIR
SYSCALL_GETDENTS64
SYSCALL_UNLINKAT
SYSCALL_MOUNT
SYSCALL_UNMOUNT

文件系统相关系统调用：
SYSCALL_FSTAT
…
SYSCALL_MMAP
SYSCALL_MUNMAP

1. 关于文件的硬连接：

   fat32不支持硬连接的问题，我们做的所有相关的工作实际上都是在内存里模拟硬连接的过程。在目前预想的实现里，每个对于文件的访问传入的“文件路径”实际上都是一个虚拟的路径，只是作为KEY存入模拟硬连接的BTreeMap里，映射到它在文件镜像中的实际位置，包括恒等映射。

   这样有两个问题：

   1. 所有文件都必须存入这个全局BTreeMap里（建立一个自身路径到自身的映射），那么就要求在初始化文件系统时将文件镜像中的所有文件遍历一遍，否则在之后读文件时就存在歧义。如果以后文件数量增多，可能会影响系统初始化的速度。
   2. A、B指向A文件，删除A后因为有B存在，实际上只会减少A文件的硬连接数，不会删除实际文件。这时再创建一个新的A文件时会报错。

   参考了maturin的实现，在BTreeMap中只保存非指向自身的映射， 而在查询路径时对指向自身的映射做更高优先级的特判。

   ```
   TruePath = {
       if path_exists(VirtPath) VirtPath
       else BTreeMap(VirtPath)
   }
   ```

   这样依然存在一些问题，例如上面的例子，删除A时会直接导致B的硬连接失效。

   对每个实际的文件维护一个记录指向它的硬连接的列表，删除“自身映射”时把实际文件重命名为列表首项并从BTreeMap中删除该项，让它成为新的“自身映射”。

2. 关于目录项遍历syscall_getdent64：

   结合硬连接后原有axfs模块中关于目录项的部分无法使用，axfs中提供的遍历目录项的迭代器ReadDir遍历的是真实的文件，而加入硬连接映射表后我们实际需要遍历的是模拟的路径。

   权宜的解决方法是对ReadDir做一层包装，ReadDir遍历结束后再在BTreeMap中遍历一次查找前缀。

3. 关于mmap

   主要用到了陈嘉钰同学实现的new_lazy功能，大致是在内存映射MapArea中添加一个backend项用于记录文件，在调用mmap时将backend指向目标文件，并利用ArceOS的new_fault_page实现内存的懒惰分配。目前这个模块存在一个小问题，即在mmap一个文件之后实际操作前会将映射到的内存的整个页初始化为0，结束后多余的0会保留在文件中，改变了文件大小（向4K对齐）。

4. 其他

   目前axfs的一些实现不太适用于我们的OS。

例如，我对于文件系统最早的思路是延续现有的实现，将axfs中所有的操作集成到api的mod.rs形成接口，在外部只允许直接调用这些接口，并因此在api/mod.rs中添加了一些需要的接口；根据这个思路，我还将文件描述符表做了简化，只记录路径，方便与硬连接配合、并直接调用api/mod.rs的接口(参数大都是路径)。但后来发现，为了保证线程、进程间安全， 我们对文件的操作很多都要带锁，这就使得我们要记录一个加锁的File实例，并跳过api中提供的接口直接调用File实例的方法，因此在fd中还是需要保留一个File实例；这时候就出现了新的问题，也就是我们上面对于硬连接的操作还要考虑到对于各个进程的File实例的更新。

另一个例子是目录项迭代器ReadDir，它是我们前期一个很奇怪的bug的根源。它用一个长度为31个目录项的数组做缓存区，用两个指针标记每次读取到缓存区中的目录项起始点。我把缓存区长度增加到128后bug消失，但我们看不出来ReadDir有什么问题。

2. 线上测试

    1. 自己写脚本

    线上评测环境里自带的第三方依赖只有log和bit_flag，其他的都需要本地化，因为编译环境不能联网。参考了包括maturin、untitled-project(1)(2)在内的往届初赛作品，它们都是将所需要的依赖直接下载到本地，然后将cargo.toml文件中的依赖修改为本地依赖。例如

Name	Last commit	Last update
..		
aho-corasick	哈哈,我把依赖全部加进来	1 year ago
bare-metal	哈哈,我把依赖全部加进来	1 year ago
bit_field	哈哈,我把依赖全部加进来	1 year ago
bitflags	哈哈,我把依赖全部加进来	1 year ago
buddy_system_allocator	哈哈,我把依赖全部加进来	1 year ago
cfg-if	哈哈,我把依赖全部加进来	1 year ago
embedded-hal	哈哈,我把依赖全部加进来	1 year ago
lazy_static	哈哈,我把依赖全部加进来	1 year ago
log	哈哈,我把依赖全部加进来	1 year ago
memchr	哈哈,我把依赖全部加进来	1 year ago
nb-0.1.3	哈哈,我把依赖全部加进来	1 year ago
nb	哈哈,我把依赖全部加进来	1 year ago
regex-syntax	哈哈,我把依赖全部加进来	1 year ago
regex	哈哈,我把依赖全部加进来	1 year ago
riscv-target	哈哈,我把依赖全部加进来	1 year ago
riscv	哈哈,我把依赖全部加进来	1 year ago
rustc_version	哈哈,我把依赖全部加进来	1 year ago
semver-parser	哈哈,我把依赖全部加进来	1 year ago
semver	哈哈,我把依赖全部加进来	1 year ago
spin-0.5.2	哈哈,我把依赖全部加进来	1 year ago
spin	哈哈,我把依赖全部加进来	1 year ago
vcell	哈哈,我把依赖全部加进来	1 year ago
virtio-drivers	哈哈,我把依赖全部加进来	1 year ago
volatile	哈哈,我把依赖全部加进来	1 year ago
xmas-elf	哈哈,我把依赖全部加进来	1 year ago
zero	哈哈,我把依赖全部加进来	1 year ago

    但是ArceOS的依赖项很多，删掉.cargo下的缓存并执行一次make run，.cargo下缓存的第三方库有65个，也就是说我们至少要把这65个库本地化到项目目录中（当然这些库中有一部分可能是log或bit_flag引入的，实际上不用本地化）。但在后续评测时发现，即使是没有执行到的modules、crates甚至apps，它的依赖项也会在编译时检验，所以需要本地化的不止这65个，第一次跑通测例时本地化的第三方库总数大概有130个左右。

    主要是通过python脚本实现

```
-scripts
    -localize.py
    -rename.py
    -oneclick.sh
```

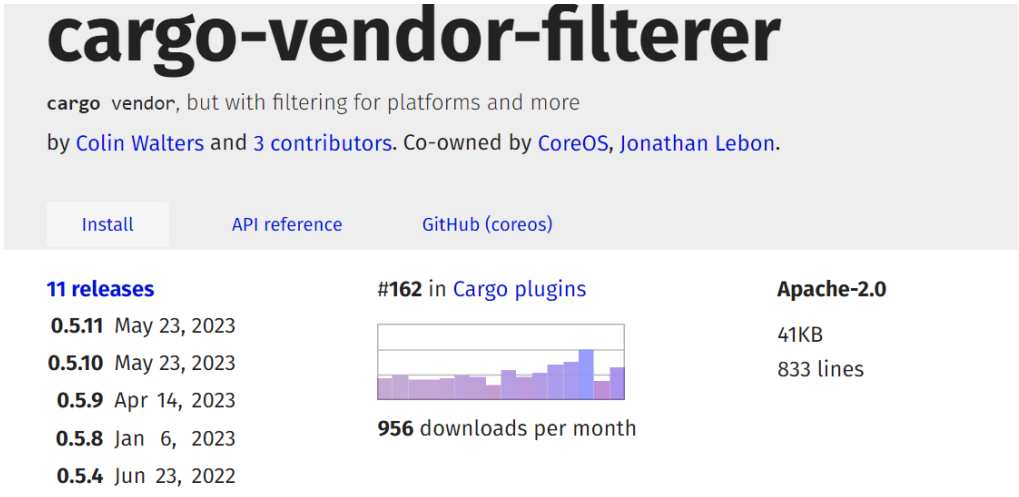
1. 将所有本地库保存到./extern_crates目录下；
2. `python3 localize.py [dep_name]`：

   遍历./extern_crates下所有库目录，对每一个库，遍历项目中的所有cargo.toml文件，将对应的依赖重定向到本地；如果遇到重名不同版本的库，写入一个json文件。添加参数也可以用来本地化或者回复单个依赖库。
3. `python3 rename.py old_name new_name version`

   手动将重名的库的其中一个修改名称后，用这个脚本将所有cargo.toml中对应的依赖项重命名。

2. vendor
   1. 在google上查找类似的工具，发现这个功能实际上被集成在了rust自身的cargo工具里，即`cargo vendor`，它会将项目中所有的依赖下载到一个vendor文件夹中，这时只要将crates.io重定向到这个vendor文件夹(像我们常用的重定向到清华源或中科大源ustc一样)，项目在构建依赖时就会将缺失的依赖库从这个文件夹“下载”到.cargo中。
   2. 这样做之后成功在本地以及本地的docker容器中跑通，但是上传时发现gitlab报错`Remote: fatal: pack exceeds maximum allowed size`。
   3. vendor自动下载到本地228个库，总大小560MB，比ArceOS本身要大很多。检查后发现里面用于支持windows的库占掉了绝大多数体积。
   4. 希望只留下linux平台的依赖，但是简单将windows-*删除会报错。cargo项目的issue中有很多类似的需求，但是rust官方没有推出类似的功能。

3. vendor-filterer



   1. 一个很新的第三方工具，用于在`cargo vendor`时将一部分的无用依赖“删除”。

```
[package.metadata.vendor-filter]
platforms = ["*-unknown-linux-gnu"]
tier = "2"
all-features = true
exclude-crate-paths = [ { name = "curl-sys", exclude = "curl" },
                        { name = "libz-sys", exclude = "src/zlib" },
                        { name = "libz-sys", exclude = "src/zlib-ng" },
                      ]
```

2. 有一些问题，比如它把`compiler_builtins`等rust工具链自身依赖的库也过滤掉了，对这些而言可以在项目中显式指定。
3. vendor目录缩小到了67MB。

1.3 第三步 重构+支持更多功能拓展

1.3.1 重构

1. 将批量测试从OS的内核中独立出来。现在导入各种测例进行测试的模块也被作为一个APP`syscall`放在了APPS目录下，这样一定程度上实现了原本的ArceOS和现有项目的并行，make build时如果选择`syscall`就会生成宏内核并执行选择的测例，如果选择其他APP就会正常生成原本的unikernel内核。
2. 取消axsyscall模块，将ArceOS对于linux syscall的处理解释为一个用户库，和c_libax、libax一起放在ulib目录下，在执行`syscall`APP时调用对应的用户库。
3. 将原有的临时模块axfs_os拆解，关于进程文件描述符表的部分添加到axprocess模块中，关于axfs接口的封装添加到axfs模块中。
4. 尽可能通过条件编译使得原有的ArceOS功能不受影响。

1.3.2 推进

1. 添加更多syscall

文件系统相关syscall
syscall_utimensat
syscall_fcntl64
syscall_writev
syscall_readv
syscall_access
syscall_lseek
syscall_renameat

还需要的系统调用
SYSCALL_IOCTL
SYSCALL_STATFS
SYSCALL_CHMOD
SYSCALL_PREAD
SYSCALL_SENDFILE64
SYSCALL_PSELECT6
SYSCALL_PPOLL

还需要的系统调用
SYSCALL_READLINKAT
SYSCALL_FSTATAT
SYSCALL_FSYNC
SYSCALL_FDATASYNC

2. 支持动态链接

   修改文件系统硬连接部分的实现。

   依旧是前面提到的fat不支持硬连接的问题。在支持动态链接时需要先将动态链接库的路径添加到硬连接的BTreeMap中，如/lib/ld-musl-riscv64-sf.so.1 |--->libc.so，这时候如果左侧实际指向的路径也有一个同名文件的话这个硬连接就无效，算是一个小Bug(因为这个文件只能是文件镜像中本来就存在的)，依旧做了一点小的修改，即在映射的函数中把/lib/ld-musl-riscv64-sf.so.1放在最前。

3. 其他

   在计划外添加了一些内容，比如

   1. 将rCore中实现的semaphore和死锁检测移植到axsync中；
   2. 尝试实现一个文件的读写锁；
   3. 实现了一些简化版的shell程序：

```
>>echo abc
abc

>>ls
brk
chdir
clone
close
dup
dup2
execve
exit
fork
fstat
getcwd
getdents
getpid
getppid
gettimeofday
mkdir_
mmap
mount
munmap
open
openat
pipe
read
sleep
times
umount
uname
unlink
wait
waitpid
write
yield
shell
>>help
exit: exit Starry
help: print this help message
echo: print the arguments
ls: list files in the current directory
cat: print the content of a file
wc: count the number of characters in a file
pwd: print the current working directory
cd: change the current working directory
mkdir: create a new directory
rmdir: remove an empty directory
rm: remove a file
mv: move a file
cp: copy a file
clear: clear the screen
```

```
echo: print the arguments
ls: list files in the current directory
cat: print the content of a file
wc: count the number of characters in a file
pwd: print the current working directory
cd: change the current working directory
mkdir: create a new directory
rmdir: remove an empty directory
rm: remove a file
mv: move a file
cp: copy a file
clear: clear the screen
date: print the current date and time
cal: print the calendar of a month
test: test the system calls
>>exit
Bye!
[ 17.062192 0:4 axruntime::lang_items:5] panicked at 'All test finish!', modules/axprocess/src/test.rs:320:5
```