



Eonix 项目汇报

sudo_pacman_Syu队

T202510247995532

汇报人：那晋宁

成员：郭冰放 邵卓炜 那晋宁



0

Eonix是...?

Eonix

基于**Rust**编写

多处理器 多架构 的 宏内核操作系统

基于rust async语法的有/无栈异步任务管理方案

简单的内核设计

足够复杂的特性：

虚拟内存管理、多进程支持、POSIX兼容.....



Eonix

当前进展

截止6月30日，可以通过初赛的 basic, busybox, libctest 的大部分测试用例，lua 的全部测试用例，还有较高的 iotzone 得分

15	T202510247995532	sudo_pacman_Syu/ 同济大学	9	2025-06-30 04:17:16	92.0	98.0	92.0	97.0	49.0	49.0	49.0	49.0
----	------------------	--------------------------	---	------------------------	------	------	------	------	------	------	------	------

iotzone	0.0	36.505248244566516	38.256298345720325	36.71625646113301	111.47780305141984
---------	-----	--------------------	--------------------	-------------------	--------------------

还可以手动gcc编译程序

```
~ # /mnt/i486-linux-musl-cross/bin/i486-linux-musl-gcc test.c -static
~ # ls
a.out test.c
~ # ./a.out
Hello, world!
Please input a number:
42
You typed a positive number.
~ # ./a.out
Hello, world!
Please input a number:
0
You input a zero.
~ #
```



目录

CONTENTS

1 内存管理

3 任务管理

5 多架构支持

2 文件系统

4 设备管理

6 未来的改进



1

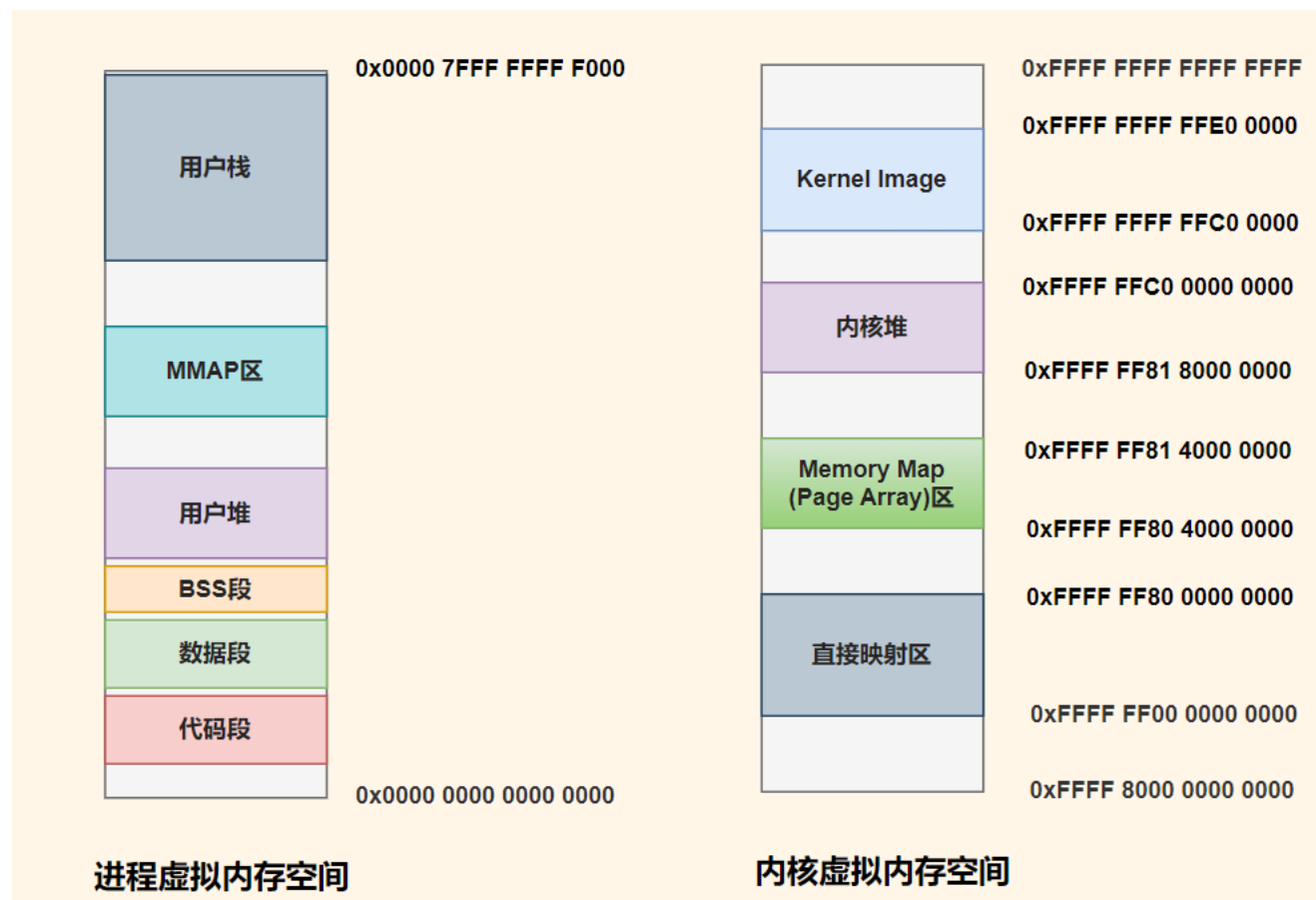
内存管理

物理页框管理



物理页框管理——buddy系统

整体内存布局



物理页框管理——buddy系统

来自Linux

维护大小为2的n次方的队列

优点：

分配：

直接取一页

如果没有能用的，拆一页更大的

高效

碎片化问题较小

释放：


放回去对应的队列

顺便合并相邻的页为一个更大的页



物理页框管理

双向侵入式链表

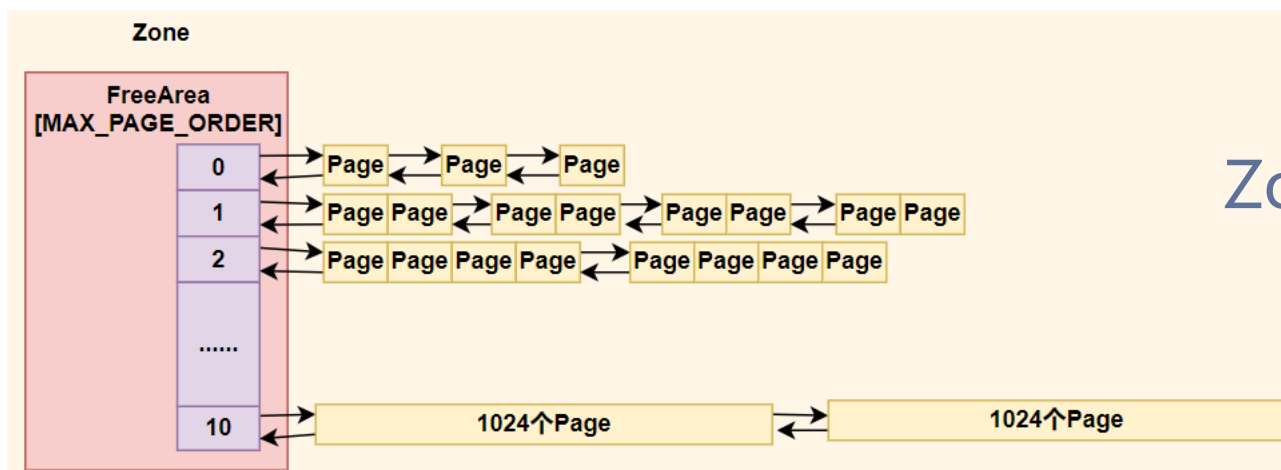


```
1 implementation
pub(super) struct Page {
    // Now only used for free page links in the buddy system.
    // Can be used for LRU page swap in the future.
    link: Link,
    flags: PageFlags, // TODO: This should be atomic.
    /// # Safety
    /// This field is only used in buddy system, which is protected by the global lock.
    order: u32,
    refcount: AtomicU32,
}
```

每一个物理页对应一个struct Page

在buddy系统内部使用

物理页框管理



2 implementations

```
struct Zone {  
    free_areas: [FreeArea; MAX_PAGE_ORDER as usize + 1],  
}
```

Zone: 管理一堆不同大小的页

1 implementation

```
struct FreeArea {  
    free_list: Link,  
    count: usize,  
}
```

FreeArea: 管理一堆同样大小的页

对应大小的FreeArea里拿一页

可能FreeArea里也没有.....

Zone负责继续从更大的Area中拆一页出来

物理页框管理——多CPU环境改进

获取锁很**昂贵**，大多数的页分配都是**小页**

每个核拿着一些小页.....

定义一个「昂贵」的大小（比如 2^3 个页）

小于这个大小的尝试从本地的页缓存中分配
不够的时候再从全局的Zone里要一些

本地的Zone**不需要锁**



物理页框使用

struct Page

真正外部看到的页

刚才buddy的细节外部看不到.....

遵循RAII原则分配、释放

```
impl Drop for Page {  
    fn drop(&mut self) {  
        match unsafe { self.page_ptr.decrease_refcount() } {  
            0 => panic!("In-use page refcount is 0"),  
            1 => free_pages(self.page_ptr, self.order),  
            _ => {}  
        }  
    }  
}
```

7 implementations

```
pub struct Page {  
    page_ptr: PagePtr,  
    order: u32,  
}
```





1

内存管理

内核内存分配



内核内存分配——slab系统

也是来自Linux系统的想法.....

内核内存使用：

对象大小比较固定，缓存行要对齐.....

快



内核内存分配——slab系统

一个页分成好多块.....

页头上用来管理这个页上的所有对象

next_free指向下一个可用对象：直接用

这个页满了，找下一个

```
pub struct SlabAllocator<T, A, const SLAB_CACHE_COUNT: usize> {  
    slabs: [Spin<SlabCache<T, A>>; SLAB_CACHE_COUNT],  
    alloc: A,  
}
```

```
pub trait SlabRawPage: RawPage {  
    /// Get the container raw page struct of the list link.  
    ///  
    /// # Safety  
    /// The caller MUST ensure that the link points to a `RawPage`.  
    unsafe fn from_link(link: &mut Link) -> Self;  
  
    /// Get the list link of the raw page.  
    ///  
    /// # Safety  
    /// The caller MUST ensure that at any time, only one mutable reference  
    /// to the link exists.  
    unsafe fn get_link(&self) -> &mut Link;  
  
    fn slab_init(&self, first_free: Option<NonNull<usize>>);  
    // zhuowei shao, 上个月 * refactor: refactor slab allocator  
    // which slab page the ptr belong  
    fn in_which(ptr: *mut u8) -> Self;  
  
    fn real_page_ptr(&self) -> *mut u8;  
  
    fn allocated_count(&self) -> &mut u32;  
  
    fn next_free(&self) -> &mut Option<NonNull<usize>>;  
}
```

内核内存分配——slab系统

slab_cache: 固定大小的slab串起来

满的、半满、空的

8、16、32、64、128.....

可能会有浪费，但是值得

```
pub(crate) struct SlabCache<T, A> {  
    empty_list: List,  
    partial_list: List,  
    full_list: List,  
    object_size: u32,  
    _phantom: PhantomData<(T, A)>,  
}
```

```
zhuowei shao, 上个月 | 1 author (zhuowei shao) | 1 implementation  
trait SlabRawPageExt {  
    fn alloc_slot(&self) → Option<NonNull<usize>>;  
    fn dealloc_slot(&self, slot_ptr: *mut u8);  
    fn is_full(&self) → bool;  
    fn is_empty(&self) → bool;  
    fn slab_page_init(&self, object_size: u32) →  
        Option<NonNull<usize>>;  
}
```




1

内存管理

用户空间内存管理



用户空间内存——低开销访问

内核访问用户空间：UserBuffer系接口

简单验证地址范围

直接访问！

权限违反、页不存在

page fault中处理

2 implementations

```
pub struct UserBuffer<'lt> {  
    ptr: CheckedUserPointer,  
    size: usize,  
    cur: usize,  
    _phantom: core::marker::PhantomData<&'lt ()>,  
}
```

用户空间内存——低开销访问

写入/读取时：使用专门指令写入

```
#[cfg(target_arch = "x86_64")]
asm!(
    "2:",
    "rep movsb",
    "3:",
    "nop",
    ".pushsection .fix, \"a\", @progbits",
    ".align 32",
    ".quad 2b",           // instruction address
    ".quad 3b - 2b",     // instruction length
    ".quad 3b",          // fix jump address
    ".quad 0x3",         // type: load
    ".popsection",
    inout("rcx") total => error_bytes,
    inout("rsi") self.ptr => _,
    inout("rdi") buffer => _,
);
```

```
#[cfg(target_arch = "riscv64")]
asm!(
    "2:",
    "lb t0, 0(a1)",
    "sb t0, 0(a2)",
    "addi a1, a1, 1",
    "addi a2, a2, 1",
    "addi a0, a0, -1",
    "bnez a0, 2b",
    "3:",
    "nop",
    ".pushsection .fix, \"a\", @progbits",
    ".8byte 2b",          // instruction address
    ".8byte 3b - 2b",    // instruction length
    ".8byte 3b",         // fix jump address
    ".8byte 0x3",        // type: load
    ".popsection",
    inout("a0") total => error_bytes,
    inout("a1") self.ptr => _,
    inout("a2") buffer => _,
    out("t0") _,
);
```

```
#[cfg(target_arch = "loongarch64")]
asm!(
    "2:",
    "ld.bu $t0, $a1, 0",
    "st.b $t0, $a2, 0",
    "addi.d $a1, $a1, 1",
    "addi.d $a2, $a2, 1",
    "addi.d $a0, $a0, -1",
    "bnez $a0, 2b",
    "3:",
    "nop",
    ".pushsection .fix, \"a\", @progbits",
    ".8byte 2b",          // instruction address
    ".8byte 3b - 2b",    // instruction length
    ".8byte 3b",         // fix jump address
    ".8byte 0x3",        // type: load
    ".popsection",
    inout("$a0") total => error_bytes,
    inout("$a1") self.ptr => _,
    inout("$a2") buffer => _,
    out("$t0") _,
);
```

用户空间内存——低开销访问

写入/读取时：page fault中特殊处理

```
for entry: &FixEntry in entries.iter() {  
    if ip >= entry.start && ip < entry.start + entry.length {  
        int_stack.rip = entry.jump_address as u64;  
        return;  
    }  
}
```

```
kernel_page_fault_die(vaddr: addr, ip as usize)
```

跳过剩余字节
表示写入失败

```
if error_bytes != 0 {  
    Err(EFAULT)  
} else {  
    Ok(())  
}
```

用户空间内存——低开销映射

fork时：只复制页表，设置copy on write

mmap：CoW + 按需读取

page fault：处理CoW

写入时才分配新页，拷贝数据

如果是mmap的，读取这一页数据





2

文件系统

文件系统抽象

目录树结构——Dentry

内存中，缓存目录结构

加速路径查找

DCache: Dentry缓存
hash list + RCU链表

7 implementations

```
pub struct Dentry {  
    // Const after insertion into dcache  
    parent: Arc<Dentry>,  
    name: Arc<[u8]>,  
    hash: u64,  
  
    // Used by the dentry cache  
    prev: AtomicPtr<Dentry>,  
    next: AtomicPtr<Dentry>,  
  
    // RCU Mutable  
    data: RCUPointer<DentryData>,  
}
```

```
static ref DCACHE: [RCUList<Dentry>; 1 << DCACHE_HASH_BITS] =  
    core::array::from_fn(|_| RCUList::new());
```

目录树结构——Dentry

目录查找：先快找，再慢找

快找：hash list里看一圈

慢找：文件系统去找
为其创建Dentry（找到或没找到）

```
match name {  
    b"." => Ok(self.clone()),  
    b".. " => Ok(self.parent.clone()),  
    _ => {  
        let dentry: Arc<Dentry> = Dentry::create(parent: self.clone(), name);  
        Ok(dcache::d_find_fast(&dentry).unwrap_or_else(|| {  
            dcache::d_try_revalidate(&dentry);  
            dcache::d_add(&dentry);  
  
            dentry  
        })))  
    }  
}
```


文件索引——Inode

每个文件的索引数据、操作
实现文件系统只要实现这个trait
tmpfs、fatfs等

非继承

```
pub trait Inode: Send + Sync + InodeInner {
    fn is_dir(&self) -> bool {
        self.mode.load(order: Ordering::SeqCst) & S_IFDIR != 0
    }

    fn lookup(&self, dentry: &Arc<Dentry>) -> KResult<Option<Arc<dyn Inode>>> {
        Err(if !self.is_dir() { ENOTDIR } else { EPERM })
    }

    fn creat(&self, at: &Arc<Dentry>, mode: Mode) -> KResult<()> {
        Err(if !self.is_dir() { ENOTDIR } else { EPERM })
    }

    fn mkdir(&self, at: &Dentry, mode: Mode) -> KResult<()> {
        Err(if !self.is_dir() { ENOTDIR } else { EPERM })
    }

    fn mknod(&self, at: &Dentry, mode: Mode, dev: DevId) -> KResult<()> {
        Err(if !self.is_dir() { ENOTDIR } else { EPERM })
    }

    fn unlink(&self, at: &Arc<Dentry>) -> KResult<()> {
        Err(if !self.is_dir() { ENOTDIR } else { EPERM })
    }

    fn symlink(&self, at: &Arc<Dentry>, target: &[u8]) -> KResult<()> {
        Err(if !self.is_dir() { ENOTDIR } else { EPERM })
    }
}
```

文件系统元数据——Vfs

不同的文件系统，Vfs可以有各自的定义实现良好的扩展性（Vfs、Inode）

实现Vfs接口

```
use super::DEV_ID;  
  
#[allow(dead_code)]  
3 implementations  
pub trait Vfs: Send + Sync + AsAny {  
    fn io_blksize(&self) -> usize;  
    fn fs_devid(&self) -> DevId;  
    fn is_read_only(&self) -> bool;  
}
```



```
▼ fs  
  ▼ fat32  
    Ⓡ dir.rs  
    Ⓡ file.rs  
    Ⓡ fat32.rs  
    Ⓡ mod.rs  
    Ⓡ procfs.rs  
    Ⓡ tmpfs.rs
```

文件系统元数据——Vfs

procfs, 提供内核中部分数据

问题 2 调试控制台 输出 端口 GITLENS 注释 终端

```
/ # ls
bin  dev  etc  mnt  proc  root
/ # cd proc
/proc # ls
ahci-p0-stats  mounts
/proc # cat ahci-p0-stats
AdapterPortStats { cmd_sent: 1772, cmd_error: 0, int_fired: 4 }
/proc # cat mounts
rootfs / tmpfs rw,noatime 0 0
/dev/sda /mnt fat32 ro,nosuid,nodev,noatime 0 0
proc proc procfs rw,relatime 0 0
/proc # mount
rootfs on / type tmpfs (rw,noatime)
/dev/sda on /mnt type fat32 (ro,nosuid,nodev,noatime)
proc on proc type procfs (rw,relatime)
/proc #
```

1 Launch Kernel (greatbridf_os) [Debug] [GCC 9.2.0 x86_64-li...] [all]

挂载点——Mount

将Vfs挂载到Dentry上
覆盖挂载.....

3 implementations

```
pub struct Mount {  
    vfs: Arc<dyn Vfs>,  
    root: Arc<Dentry>,  
}
```

```
/ # ls  
bin  dev  etc  mnt  proc  root  
/ # mkdir new_proc  
/ # ll new_proc/  
total 0  
/ # mount -t procfs procfs new_proc/  
/ # ll new_proc/  
total 0  
-r--r--r--    1 root    root          0 Jan  1 00:00 ahci-p0-stats  
-r--r--r--    1 root    root          0 Jan  1 00:00 mounts  
/ # mount -t tmpfs none new_proc/  
/ # ll new_proc/  
total 0  
/ # cat > new_proc/test_file  
Hello  
/ # cat new_proc/test_file  
Hello  
/ #
```

页缓存——PageCache

PageCache 提供统一的缓存层，管理页面映射集合和对应存储结构的后端引用从而实现高效IO

PageCacheBackend 定义了 read_page、write_page 和 size 等基本存储操作接口

```
pub struct PageCache {  
    pages: Mutex<BTreeMap<usize, CachePage>>,  
    backend: Weak<dyn PageCacheBackend>,  
}
```

```
// with this trait, "page cache" and "block cache" are unified,  
// for fs, offset is file offset (floor align to PAGE_SIZE)  
// for blkdev, offset is block idx (floor align to PAGE_SIZE / BLK_SIZE)  
// Oh no, this would make unnecessary cache  
3 implementations  
pub trait PageCacheBackend {  
    fn read_page(&self, page: &mut CachePage, offset: usize) -> KResult<usize>;  
  
    fn write_page(&self, page: &CachePage, offset: usize) -> KResult<usize>;  
  
    fn size(&self) -> usize;  
}
```

页缓存——PageCache

以Read为例

1. 获取页面锁: 确保对缓存页的并发访问安全。
2. 计算 page_id: 根据 offset 计算所属的页面ID。
3. 缓存查找:
命中:
 - 从缓存页中读取数据到 buffer, 并根据 inner_offset 和 PAGE_SIZE 更新 offset 以处理跨页读取。

未命中:

- 创建一个新的 CachePage。
- 通过 backend 调用 read_page 方法从实际存储后端读取数据到这个新的 CachePage。
- 将新读取的页面插入到 pages 中, 以便后续访问。

```
pub async fn read(&self, buffer: &mut dyn Buffer, mut offset: usize) -> KResult<usize> {
    let mut pages: MutexGuard<'_, BTreeMap<usize, ...> = self.pages.lock().await;

    loop {
        let page_id: usize = offset >> PAGE_SIZE_BITS;
        let page: Option<&CachePage> = pages.get(key: &page_id);

        match page {
            Some(page: &CachePage) => {
                let inner_offset: usize = offset % PAGE_SIZE;

                if page.valid_size() == 0
                || buffer &mut (dyn Buffer + 'static)
                    .fill(&page.valid_data()[inner_offset..])? FillResult
                    .should_stop()
                || buffer.available() == 0
                {
                    break;
                }

                offset += PAGE_SIZE - inner_offset;
            }
            None => {
                let mut new_page: CachePage = CachePage::new();
                self.backend Weak<dyn PageCacheBackend + 'static>
                    .upgrade() Option<Arc<dyn PageCacheBackend + 'static>>
                    .unwrap() Arc<dyn PageCacheBackend + 'static>
                    .read_page(&mut new_page, offset.align_down(power_of_two: PAGE_SIZE))?;
                pages.insert(key: page_id, value: new_page);
            }
        }
    }

    Ok(buffer.wrote())
} fn read
```



3

任务管理



任务管理抽象

将任务调度的运行时层 (Task) 与POSIX规范中的线程与进程资源抽象分离

运行时层负责管理任务

POSIX资源抽象层负责管理具体的线程、进程、进程组等资源



任务管理——Task

最小的调度单位

调度相关信息

既支持协作式调度,
也支持抢占式调度

通过 park 和 unpark 方法进行暂停和恢复

```
pub struct Task {  
    pub id: TaskId,                // 唯一标识符  
    pub(crate) on_rq: AtomicBool, // 是否在就绪队列中  
    pub(crate) unparked: AtomicBool, // 是否被唤醒  
    pub(crate) cpu: AtomicU32,      // 亲和性CPU  
    pub(crate) state: TaskState,    // 任务状态  
    pub(crate) execution_context: ExecutionContext, // 执行上下文  
    executor: AtomicUniqueRefCell<Option<Pin<Box<dyn Executor>>>>, // 执行器  
    link_task_list: RBTREEAtomicLink, // 全局任务列表链接  
}
```

任务管理——Task的生命周期

1. 初始状态为 RUNNING，表示任务可以执行
2. 当任务调用 park 方法时，状态变为 PARKING
3. 如果任务没有被唤醒，则进入调度器
4. 调度器可能将任务状态变为 PARKED，表示任务已被挂起
5. 当任务被 unpark 唤醒时，状态返回 RUNNING，任务重新可被调度

任务执行通过 run 方法进行，当执行完成后返回 `ExecuteStatus::Finished` 状态。



任务管理——Task与传统的Thread

Thread 代表POSIX线程语义和资源，负责实现线程的API和状态管理

Task 代表实际的调度单元，负责任务的执行和调度



任务管理——异步支持与阻塞操作

Task实现了block_on方法，可以在当前任务上阻塞执行一个Future对象

当任务需要等待某些事件时，
可以调用Task::park方法暂停自己

```
pub fn block_on<F>(future: F) -> F::Output
where
    F: Future,
{
    let waker = Waker::from(Task::current().clone());
    let mut context = Context::from_waker(&waker);
    let mut future = pin!(future);

    loop {
        if let Poll::Ready(output) = future.as_mut().poll(&mut context) {
            break output;
        }
        Task::park();
    }
}
```

进程资源单位——Process

共享一个地址空间

任务管理相关：

父进程

所属进程组、会话

WaitList：用于wait等

```
// ...  
2 implementations  
pub struct Process {  
    /// Process id  
    ///  
    /// This should never change during the life of the process.  
    pub pid: u32,  
  
    pub wait_list: WaitList,  
    pub mm_list: MMList,  
  
    /// Parent process  
    ///  
    /// `parent` must be valid during the whole life of the process.  
    /// The only case where it may be `None` is when it is the init process  
    /// or the process is kernel thread.  
    pub(super) parent: RCUPointer<Process>,  
  
    /// Process group  
    ///  
    /// `pgroup` must be valid during the whole life of the process.  
    /// The only case where it may be `None` is when the process is kernel thread.  
    pub(super) pgroup: RCUPointer<ProcessGroup>,  
  
    /// Session  
    ///  
    /// `session` must be valid during the whole life of the process.  
    /// The only case where it may be `None` is when the process is kernel thread.  
    pub(super) session: RCUPointer<Session>,  
  
    /// All things related to the process list.  
    pub(super) inner: Locked<ProcessInner, ProcessList>,  
}
```

任务管理重要部分——Session

较为重要的任务管理单位
POSIX中概念

控制终端、前台进程组

```
// [unimplemented]
2 implementations
v pub struct Session {
    pub sid: u32,
    pub leader: Weak<Process>,
    job_control: RwSemaphore<SessionJobControl>,

    groups: Locked<BTreeMap<u32, Weak<ProcessGroup>>, ProcessList>,
}
```

并发控制——Lock

所有种类锁的外部接口

9 implementations

```
pub struct Lock<Value: ?Sized, Strategy: LockStrategy> {  
    strategy_data: Strategy::StrategyData,  
    value: UnsafeCell<Value>,  
}
```

不同Strategy提供不同的锁种类：
SpinLock、Mutex、Semaphore

锁住对象用Guard来管理其上锁及释放周期



内核同步控制——CondVar

管理内核中的条件睡眠、唤醒

2 implementations

```
✓ pub struct CondVar<const INTERRUPTIBLE: bool> {  
    |     waiters: Spin<VecDeque<Arc<Thread>>>,  
    |  
    }  
}
```

wait: 释放锁并等待

notify_{one,all}: 唤醒一个或一批任务





4

设备管理

设备抽象

块设备——BlockRequestQueue

块设备：按块访问，可缓存

请求队列
(设备驱动)

1 implementation

```
✓ pub trait BlockRequestQueue: Send + Sync {  
  ✓ /// Maximum number of sectors that can be read in one request  
  ///  
    fn max_request_pages(&self) -> u64;  
  
    fn submit(&self, req: BlockDeviceRequest) -> KResult<()>;  
}
```

单个请求

0 implementations

```
pub struct BlockDeviceRequest<'lt> {  
  pub sector: u64, // Sector to read from, in 512-byte blocks  
  pub count: u64, // Number of sectors to read  
  pub buffer: &'lt [Page],  
}
```



块设备——BlockDevice

实际使用的接口

提供便于使用的接口

```
///  
pub fn read_raw(&self, mut req: BlockDeviceRequest) -> KResult<()> {  
|    // TODO: check disk size limit  
|    ///  
pub fn read_some(&self, offset: usize, buffer: &mut dyn Buffer) -> KResult<FillResult> {  
|    let mut sector_start: u64 = offset as u64 / 512;
```

6 implementations

```
✓ pub struct BlockDevice {  
    devid: DevId,  
    size: u64,  
    max_pages: u64,  
  
    dev_type: BlockDeviceType,
```

使用的人可以自己建立请求

或直接让接口帮你创建请求并完成数据分割、拷贝



块设备——BlockDevice

分为Disk和Partition

0 implementations

```
enum BlockDeviceType {  
    Disk(BlockDeviceDisk),  
    Partition(BlockDevicePartition),  
}
```

统一的接口

Partition帮你在请求处加上偏移
转发请求

0 implementations

```
struct BlockDeviceDisk {  
    queue: Arc<dyn BlockRequestQueue>,  
}
```

0 implementations

```
struct BlockDevicePartition {  
    disk_dev: DevId,  
    offset: u64,  
  
    queue: Arc<dyn BlockRequestQueue>,  
}
```

字符设备——CharDevice

2 implementations

```
pub struct CharDevice {  
    name: Arc<str>,  
    device: CharDeviceType,  
}
```

分为普通设备及终端

同样统一的接口：read、write

```
pub enum CharDeviceType {  
    Terminal(Arc<Terminal>),  
    Virtual(Box<dyn VirtualCharDevice>),  
}
```



字符设备——CharDevice

/dev/null

1 implementation

```
struct NullDevice;  
impl VirtualCharDevice for NullDevice {  
    fn read(&self, _buffer: &mut dyn Buffer) -> KResult<usize> {  
        Ok(0)  
    }  
  
    fn write(&self, _data: &[u8]) -> KResult<usize> {  
        Ok(_data.len())  
    }  
}
```



字符设备——Terminal

POSIX规范的终端设备

2 implementations

```
pub struct Terminal {  
    /// Lock with IRQ disabled. We might use this in IRQ context.  
    inner: Spin<TerminalInner>,  
    device: Arc<dyn TerminalDevice>,  
    cv: CondVar,  
}
```

cooked模式、raw模式、echo

任务控制：产生信号，与进程控制联动





5

多架构支持

多架构支持——硬件抽象层 (HAL)

所有架构相关的代码统一置于 crates/eonix_hal 硬件抽象层中

- 封装不同架构的底层硬件操作
- 提供统一的接口给内核调用
- 处理架构特定的内存管理、中断处理、上下文切换等功能
- 通过 cfg-if 宏条件编译

```
cfg_if::cfg_if! {  
    if #[cfg(target_arch = "x86_64")] {  
        mod x86_64;  
        pub use x86_64::*;  
    } else if #[cfg(target_arch = "riscv64")] {  
        pub mod riscv64;  
        pub use riscv64::*;  
    } else if #[cfg(target_arch = "loongarch64")] {  
        pub mod loongarch64;  
        pub use loongarch64::*;  
    } else {  
        compile_error!("Unsupported architecture");  
    }  
}
```

多架构支持——进程上下文抽象

所有架构的 TaskContext 都提供统一接口

- 基于trait, 利用 Rust 的静态分发特性
- 确保零运行时开销

```
pub trait RawTaskContext: Sized {
    fn new() → Self;
    fn set_program_counter(&mut self, pc: usize);
    fn set_stack_pointer(&mut self, sp: usize);
    fn is_interrupt_enabled(&self) → bool;
    fn set_interrupt_enabled(&mut self, is_enabled: bool);
    fn call(&mut self, func: unsafe extern "C" fn(usize) → !, arg: usize);
    unsafe extern "C" fn switch(from: &mut Self, to: &mut Self);
    unsafe extern "C" fn switch_to_noreturn(to: &mut Self) → ! {
        let mut from_ctx = Self::new();
        unsafe {
            Self::switch(&mut from_ctx, to);
        }
        unreachable!("We should never return from `switch_to_noreturn()`");
    }
}
```

多架构支持——Trap 上下文抽象

- 通过 RawTrapContext trait 提供多架构抽象
- Trap 类型分类包括系统调用、异常、外部中断、定时器中断。

```
pub trait RawTrapContext: Copy {  
    type FIrq: FnOnce(fn(irqno: usize));  
    type FTimer: FnOnce(fn());  
    fn new() → Self;  
    fn trap_type(&self) → TrapType<Self::FIrq, Self::FTimer>;  
    fn get_program_counter(&self) → usize;  
    fn get_stack_pointer(&self) → usize;  
    fn set_program_counter(&mut self, pc: usize);  
    fn set_stack_pointer(&mut self, sp: usize);  
    fn is_interrupt_enabled(&self) → bool;  
    fn set_interrupt_enabled(&mut self, enabled: bool);  
    fn is_user_mode(&self) → bool;  
    fn set_user_mode(&mut self, user: bool);  
    fn set_user_return_value(&mut self, retval: usize);  
    fn set_user_call_frame<E>(  
        &mut self,  
        pc: usize,  
        sp: Option<usize>,  
        ra: Option<usize>,  
        args: &[usize],  
        write_memory: impl Fn(VAddr, &[u8]) → Result<(), E>,  
    ) → Result<(), E>;  
}
```

多架构支持——处理器抽象

PercpuArea:
每CPU的相关状态及资源
percpu的区域

x86: GDT、TSS、APIC

```
const MAX_CPUS: usize = 256;

greatbridf, 3周前 | 1 author (greatbridf)
#[repr(align(16))]
0 implementations
pub struct PercpuData();

greatbridf, 3周前 | 1 author (greatbridf) | 1 implementation
pub struct PercpuArea {
    data: NonNull<PercpuData>,
}

greatbridf, 3周前 • feat(hal): smp initialization
static PERCPU_POINTERS: [AtomicPtr<PercpuData>; MAX_CPUS] =
    [const { AtomicPtr::new(null_mut()) }; MAX_CPUS];
```



6 未来的改进



未来改进

RCU的完善

针对测试用例的进一步优化

进一步完善异步任务管理

调度器的优化





谢谢