基于 RISC-V 架构与Rust语言的16KiB页面操作系统——操作系统 Project 报告

1 项目简介

1.1 项目协作

姓名	学号	负责内容
叶璨铭	12011404	阅读RISCV手册，进行QEMU与地址空间设计。用Rust实现并测试物理页分配、进程切换算法。实验测定16KiB方案的优劣。
邓值仁	12012029	掌握Rust语言。深入理解并文档化rCore，深度优化 rCore 设计，使得每个部分都比较稳定可靠、简单清晰、符合rust规范。

1.2 git 仓库管理

本文档的完整最新pdf版JumbuckNucleus/Project Report.pdf at ch5虚拟内存+进程+ 稳定 · 2catycm/JumbuckNucleus (github.com)（时间太紧张了，写文档的时候还要复习AI，写一般git命令搞错了用分支把文档覆盖为旧版，如果你看到后面有残缺的部分比较感兴趣，请看这个链接的版本）。

仓库地址：2catycm/JumbuckNucleus at ch5虚拟内存+进程+ 稳定 (github.com)

持续集成测试：feat(os): 最新版本 · 2catycm/JumbuckNucleus@896c3a7 (github.com)

参考仓库： rCore-Tutorial-Book-v3 3.6.0-alpha.1 文档 (rcore-os.cn)

1.3 为什么页面大小选择为16KiB，而不是传统甚至是默认的4KiB?

内存的容量近年来不断发展，容量较大，然而分页方案的页面大小，仍然是上个世纪1960年代就使用的4KiB大小。[1]那么，基于16KiB页面的OS有什么优势呢？在做这次Project的之前和过程中，我们需要理解16KiB操作系统的具体优势，这样才能达到Project的效果与目的；否则，就只是做了任务要求而已。

1.3.1 从缓存（按需分页）的角度来看

从按需分页(Demand paging)的角度来看，内存是磁盘的缓存。所谓分页，就是缓存中的分 block ， 也就是一个传输单位。同一个block或者page的地址(访问需求), 由于访问的内容处于硬件上的同一个传输单位，因此只需第一次访问遍将内容缓存，加速大量的后续访问。

那么，是不是block越大，miss rate就越低呢？其实并不尽然。在计算机组成原理的经典教材《计算机组成与设计——硬件/软件接口》中，有这样一幅图。在图中，比较高的曲线的cache容量(按需分页中的内存大小)低，整体失效率比低的曲线要大；横向的变化趋势表示相同内存大小下选择不同的block 大小，失效率的变化。**可以看出，如果相对于cache本身的大小，一个block的大小过大，实际上失效率会逐渐上升。**[2]

对于16KiB和4KiB的页面问题来说，这个视角给我们带来了两个启发

- 4KiB页面的 Miss Penalty 较小。即 Page Replacement 完成所需时间较少。
- 16KiB页面虽然Miss Penalty大四倍。但是由于现代计算机的内存容量远远大于4KiB和16KiB，应当**适用于图中比较低的曲线**，所以16KiB应当显著降低Miss Rate。

1.3.2 从虚拟化的角度来看

上面我们考虑了“物理内存不够，需要磁盘来补充”的早期分页想法之一，然而如今物理内存未必小于虚拟内存。另一个分页的想法来自于 虚拟化为不同进程带来的保护。那么，我们应当注意到，不同进程的地址空间不同。

首先考虑进程之间地址空间的隔离

- 假设进程A, B, C, D分别连续访问了4KiB个字节
  - 由于四个进程的地址空间不同，他们分别申请了四个不同的16KiB页面而无法互相利用，产生了内碎片。
- 假设进程A连续访问了16KiB个字节，
  - 我们使用16KiB的分页方案，只有一次compulsory miss。
  - 这一次就非常合理地利用了内存空间。

可见，使用多大的页面，和各种进程平均需要访问的内容多少有关系。

另一方面，考虑进程间的通信，进程的内存共享是最简单的方式之一。

- 而内存共享的单位是页。
- 如果页的大小足够，那么通信之间发生的争端(resolution)就会少很多。[1]

1.3.3 从分段的角度来看

过大的block会导致内存碎片化的加剧，但是另一方面也减少了内存碎片化。

从外部碎片化的角度来看，

- 分页机制本身作为一种**定长的**分段方法（以前的分段方法是根据程序的code\stack\data\bss这样分的），本身就是为了避免外部碎片化。
  - 定长的页面大小让连续内存分配的算法更加容易设计，因此有更好的性能保证。
- 另一方面，更大的页让内存的可申请尺度变大，
  - 换句话说，碎片的最小大小是一个页，所以不存在比一个页还要小的外部碎片，也就不会出现环节了外部碎片化。

从内部碎片化的角度来看，

- 大的页面导致了内部碎片化。用户可能只申请了一两个字节，就获得了整个物理页面。
- 这种情况叫做系统颠簸(Trashing)。[1]

然而，Weisberg 和 Wiseman 认为，**随着内存容量的急剧增大，trashing的问题并不是什么问题**。如果用户需要很大的内存，那么大的页面很合适，如果用户所需内存很小，最多也就浪费一个页面的大小。相比于16GiB的内存，一个进程至多浪费4KiB显得根本不是问题，所以改成16KiB也不会造成什么影响。[1]

1.3.4 从硬件接口的角度来看

传统的硬盘设备的一个 Sector 往往就是4KiB，也就是说磁盘一次只能支持 4KiB 大小的传输，这就限制了操作系统的设计也将页面大小设计为4KiB。

然而，近年来，硬盘设备技术发展，16KiB的传输大小并不是什么难事。

1.3.5 从内存管理开销(overhead)的角度来看

- 大的页面，减少了相同大小内存下页面的数量，也就降低了PTE在页表中的数量，进而**减少单个页表的大小**[1]也减少了多级页表的页表总数量
- 然而实际上，由于MMU硬件的限制，比如RV64的Sv39方案，
  - MMU硬件只根据PPN，而不是PA来寻找页表。
  - 换句话说，一个页表必须占据一个页的大小。
  - 因此无法有效发挥这一优势。

- 虽然MMU有限制，第二个优点：降低了多级页表的总数量，还是有所体现的。
  -

63 54	53 28	27 19	18 10	9 8	7	6	5	4	3	2	1	0
Reserved	PPN[2]	PPN[1]	PPN[0]	RSW	D	A	G	U	X	W	R	V
10	26	9	9	2	1	1	1	1	1	1	1	1

  - 如图为Sv39分页方案，我们称PPN[2:0]对应的页表为二级、一级、零级页表。
  - 一个巨页为2MiB, 吉页为1GiB。
  - 若**连续申请**I个字节，至多需要$1+\lceil I/2MiB\rceil+\lceil I/1GiB\rceil$个页表（二级页表、一级页表、零级页表）[3]
  - 如果改为我们的16KiB页面方案（9,9,9,14）
    - 一个巨页为8MiB, 吉页为4GiB。
    - 那么至多需要$1+\lceil I/8MiB\rceil+\lceil I/4GiB\rceil$个页表，**页表的数量少了大约4倍**。

1.3.6 从TLB的角度来看

Weisberg 和 Wiseman 认为， 以上因素都有道理，但是之所以16KiB应当是最优选择，**最重要的因素是比起4KiB，16KiB可以显著提高TLB的命中率。**[1]

不妨设内存总大小一定、TLB能存的PTE大小一定、页表能存储的页表项一样多。

假设 TLB能存储256项PTE，Page Size是4KiB。

- 那么TLB能管辖的内存范围是1MiB，如果有连续1MiB的多次访问，TLB在访问过一次后就能对这1MiB范围的所有虚拟地址命中。
- 而如果增大Page Size为16KiB，管辖范围就能提高4倍。
- 对于相同的地址访问来说，TLB 的 miss rate就可以大幅降低。
- 当然，使用巨页也可以增加管辖范围，不过仍然有4倍的关系，而且16KiB优势更为显著。

近年来，硬件设备的速度提升、内存的急剧增大，使得仍为4KiB的页面大小降低了TLB相对于整个内存的覆盖率，因此，提高Page的大小是合理而有必要的。[1]

根据 Weisberg 和 Wiseman 的实验结果，16KiB的分页方案比起4KiB降低了1/3的TLB miss数，增加了2/3的内存空间使用，对于如今大的内存硬件而言，是值得的。[1]

1.4 为什么选择用Rust语言做Project?

本次 Project 我们没有使用Lab中的C语言代码，而是根据rCore的指导重新构建出一个操作系统。Rust是一门赋予每个人构建可靠且高效软件能力的语言，具有**高性能、可靠性、生产力**这三大优势。[4]本次我们使用Rust开发操作系统Project的过程中，充分运用了这三大优势。

1.4.1 生产力。

1. 可拔插的语言特性。

   比起C语言和C++写操作系统没有任何标准库，Rust语言在设计时便考虑了裸机环境、嵌入式设备下没有标准库的问题。使得我们在编写操作系统的时候，仍然可以使用core核心库。

   此外，只要在具体平台上编写相应的语言特性，便可以逐步恢复标准库的功能。这一点我们将在后文 “Rust执行环境的实现”一节描述。

2. 优质的包管理器。

   C/C++的包管理一直以来比较复杂、不能跨平台、不够统一。而Rust作为新兴语言，设计之初就给出了官方的包管理器`cargo`，统一了调库的接口。

例如，本次Project中我们使用riscv库和log库，只需要在项目配置文件`cargo.toml`中写：

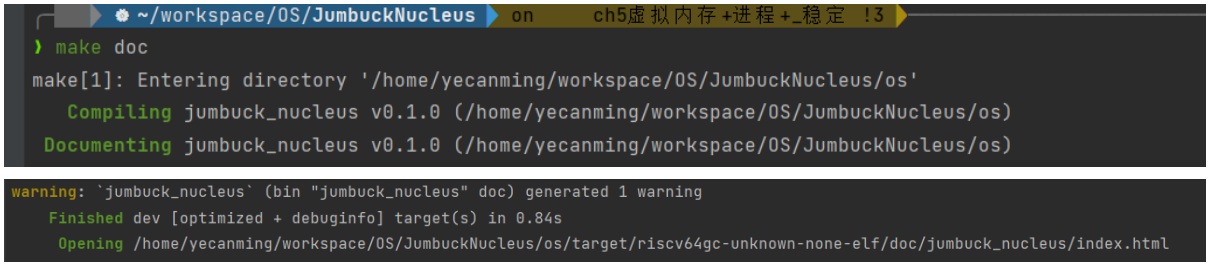
```
[dependencies]
riscv = { git = "https://github.com/rcore-os/riscv", features = ["inline-asm"] }
log = "0.4.8"
```

在使用`cargo build`构建的时候，便会自动下载对应版本的库(crate)，进行链接.

3. 出色的文档

在Rust中，可以使用"///"的语法为函数、模块等进行文档注释，支持`markdown`语法。

接下来，使用`cargo doc`可以生成文档，并且在浏览器中查看效果。



可以看到，除了支持markdown语法，rust文档还

- 支持链接别的模块、函数等的文档；
- 形成的文档也是非常美观而统一的，支持简单的搜索和查看源代码的功能
- 支持文档中给出代码使用示例。（图片未体现）

我们查看所调用的库时也可以看它的文档，快速理解用法。

4. 清晰的错误提示

```
warning: function cannot return without recursing
 --> src/bin/stack_overflow.rs:7:1
  |
7 | fn f(d: usize) {
  | ^^^^^^^^^^^^^^ cannot return without recursing
8 |     println!("d = {}", d);
9 |     f(d + 1);
  |     -------- recursive call site
  |
  = note: `#[warn(unconditional_recursion)]` on by default
  = help: a `loop` may express intention better if this is on purpose
```

这是我们为了测试操作系统的地址空间管理是否可以阻止用户态程序Stack Overflow并且杀死该进程而编写的一个递归调用不结束程序（一直增加函数栈的大小，超出地址空间合法范围）。

Rust编译器针对我们代码给出的一个递归的警告。可以看出，Rust编译器给的信息非常完善，**包括错误的位置、错误的原因、提醒错误原因的笔记和解决错误的帮助**。

而如果我们使用C++来写操作系统，可能使用模板的时候由于编译器历史遗留原因，产生大量不可以理解的奇怪错误。

1.4.2 可靠性。

Rust 丰富的类型系统和所有权模型保证了内存安全和线程安全，让您在编译期就能够消除各种各样的错误。[4]

Rust的管理方法与C++有异曲同工之妙，Rust默认使用C++中称为移动构造的技术，而复制构造则需要实现`Copy`Trait, 这样移动的语义（称为所有权模型）减少了潜在的复制降低性能的问题。另一方面，Rust和C++都支持RAII的技术，即栈上的资源管理变量（比如Vector,String）拥有指针，在函数控制流比较复杂、甚至包括异常处理的情况下，仍然能够保证资源管理变量能够在合适的时机调用析构函数，自动释放资源。这大量地减少C语言编程中出现的内存泄露、二次释放等问题，而且比Java等GC语言垃圾回收的方案高效很多。

在本次Project中，我们再管理物理页帧的时候使用了RAII这一技术。物理页帧是实现虚拟内存机制的重要部分，实现了物理内存的分页机制后我们才能进一步通过页表管理虚拟内存。操作系统是一种硬件资源管理器。[5]，作为资源管理器，操作系统需要讲物理页帧分配给进程，在进程变为僵尸进程，然后最终被回收的时候，将物理页帧回收。这个过程如果使用C语言编写，可能需要注意不要忘记回收、不要重复回收。如果使用Rust或者C++，我们可以这样写：

```
//mod.rs in frame_allocator. designed by rCore Team and improved by 叶璨铭 to
support continuous allocation.
pub struct FrameBlockTracker {
    pub(super)ppn: PhysPageNum,
    pub(super)count: usize,
}
impl FrameBlockTracker {
    ///创建空的 `FrameTracker`
    pub fn new(ppn: PhysPageNum, count: usize) -> Self {
        // 清理页的内容，全部变为0.
        for i in 0..count{
            PhysPageNum::from(usize::from(ppn)+i).clear();
```

```
            }
            Self { ppn,  count}
        }
    }
    impl Drop for FrameBlockTracker { //实现析构函数
        fn drop(&mut self) {
            dealloc_frames(self.ppn, self.count);
        }
    }
    /// 分配一系列连续页面。
    /// 接口暴露：本模块外的接口允许调用alloc方法，但是不允许调用dealloc。因为使用
    [`FrameBlockTracker`]来管理生命周期。
    #[allow(unused)]
    pub fn alloc_frames(count:usize)->Option<FrameBlockTracker>{
        FRAME_ALLOCATOR
            .exclusive_access()
            .alloc(count)
            .map(|ppn| FrameBlockTracker::new(ppn, count))
    }
    fn dealloc_frames(ppn: PhysPageNum, count:usize){
        FRAME_ALLOCATOR
            .exclusive_access().dealloc(ppn, count);
    }
```

首先我们实现了一个`FRAME_ALLOCATOR`， 实现了`best fit` `worse fit` `buddy system allocator` `first fit`等连续资源分配算法。但是`FRAME_ALLOCATOR`是private的，我们仅对外暴露`alloc_frames`这一接口，外部方法可以通过它获得一个`FrameBlockTracker`(如果获取失败，则为None)。操作系统其他部分的代码可以通过所有权的转移，延长`FrameBlockTracker`的生命周期，而最终析构时，自动调用这里编写的`drop`方法，准确而高效地释放相应的内存。

1.4.3 高性能。

Rust 和C++语言一样，提供了“零开销”的抽象。我们用Rust写操作系统，不仅不会比C语言慢，还可能因为更加清晰的类型系统和所有权机制使得导致变慢的操作在编译时就消除，从而获得更好的内存利用率。

2 应用程序与基本执行环境

2.1 16KiB下的QEMU

我们修改了QEMU中关于分页的部分, 为了简单起见，使用了9-9-9-14方案

```
// cpu_bits.h
/* Leaf page shift amount */
// #define PGSHIFT             12
#define PGSHIFT             (10+4) //16KiB
// cpu-param.hs
// #define TARGET_PAGE_BITS 12 /* 4 KiB Pages */
#define TARGET_PAGE_BITS (10+4) /* 4 KiB Pages */
```

我们QEMU编译的效果放在了[2catycm/qemu-bin (github.com)](https://github.com/2catycm/qemu-bin)

```
❯ qemu-as-7.0-16Ki
正在设置Qemu为 7.0-16Ki
当前脚本位置/home/yecanming/workspace/OS/JumbuckNucleus/qemu-bin
设置完成
```

将这个函数集成进makefile中，我们就可以比较方便的切换两种OS。

2.2 Rust 执行环境的实现

2.2.1 实现 致命错误处理器，蓝屏机制

前面提到rust可拔插的语言特性。只要我们实现了`panic_handler`（致命错误，或者在其他语言中说的不可以处理的异常）属性，那么标准库中各种算法、库报错的时候，都会自动转接到这个函数，我们实现蓝屏信息的打印提示用户，并且可以告知用户错误的位置，最后关机。

```
/// 打印 panic 信息并 [`sbi::shutdown`].
///
/// ### '#[panic_handler]' 属性
/// 声明此函数是 panic 的回调。
#[panic_handler]
fn panic_handler(info: &core::panic::PanicInfo) -> ! {
    console::open_blue_print();
    let repeated_space = "
                                                                ";
    println!(":(\n你的电脑遇到问题，需要关机。");
    println!("我们只为您收集关键信息，然后为您关机。");
    println!("\t如果您需要了解更多信息，请查看此错误:");
    if let Some(location) = info.location() {
        println!(
            "\t{}:{} \n\t\t'{:?}'",
            location.file(),
            location.line(),
            info.message().unwrap()
        );
    } else {
        println!("{}", info.message().unwrap());
```

```
    }
    console::close_console_effects();
    sbi::shutdown()
```

实现了这个功能之后，我们的OS获得了处理致命错误的能力。

例如，我们试图不断地申请堆内存

```
[jumbuck_nucleus::memory::heap_allocator]
WARN:   正在执行崩溃测试，接下来操作系统应当崩溃并且关机，而不是死机。
[jumbuck_nucleus::memory::heap_allocator]
INFO:   正在访问物理地址 0x80ba0150
[jumbuck_nucleus::memory::heap_allocator]
INFO:   正在访问物理地址 0x80b63980
[jumbuck_nucleus::memory::heap_allocator]
INFO:   正在访问物理地址 0x80aa6e80
[jumbuck_nucleus::memory::heap_allocator]
INFO:   正在访问物理地址 0x8086a380
[jumbuck_nucleus::memory::heap_allocator]
INFO:   正在访问物理地址 0x8092d880
[jumbuck_nucleus::memory::heap_allocator]
INFO:   正在访问物理地址 0x809f0d80
[jumbuck_nucleus::memory::heap_allocator]
INFO:   正在访问物理地址 0x804b4280
[jumbuck_nucleus::memory::heap_allocator]
INFO:   正在访问物理地址 0x80577780
[jumbuck_nucleus::memory::heap_allocator]
INFO:   正在访问物理地址 0x8063ac80
```

```
[jumbuck_nucleus::memory::heap_allocator]
INFO:   正在访问物理地址 0x8063ac80
20%
30%
40%
        如果您需要了解更多信息，请查看此错误：
        src/memory/heap_allocator.rs:12
                '绵羊核心堆内存申请异常，可能存在内存泄露！ Layout { size_: 4, align_: 4 (1 << 2) }'
50%
60%
70%
80%
90%
100%
```

2.2.2 实现log输出

- 方便debug调试，我们需要一个强大的log输出机制。

  这也可以通过拔插rust的语言特性实现，通过实现Log接口，就可以使用log宏了

  JumbuckNucleus/sheep_logger.rs at ch5虚拟内存+进程+ 稳定 · 2catycm/JumbuckNucleus (github.com)

2.2.3 实现条件编译（Makefile+rust feature）

- 允许快速选择4KiB还是16KiB

  我们可以是rust cargo的

2.2.4 Qemu远程gdb

Qemu支持

3 16KiB下内存布局与地址空间的设计

3.1 地址转换模式

在lab代码中，我们可以注意到内核转换存在一个过渡期, 通过entry.asm中写入页表的方式，把一个魔法地址"0xFF..F80200000"吉页转换为物理地址，从而开启虚拟内存。

在rCore中，并不是这样做的。首先，rCore没有在汇编中写页表，而是通过地址空间的合理rust抽象，在内核代码中写入页表；其次rCore中避免了魔法地址，直接使用恒等映射。

我们可以看到这样的代码：

```
// KERNEL SPACE 这个 memory set的初始化函数 的一部分
println!("mapping .bss section");
        memory_set.push(
            MapArea::new(
                (sbss_with_stack as usize).into(),
                (ebss as usize).into(),
                MapType::Identical,
                MapPermission::R | MapPermission::W,
            ),
            None,
        );
```

这样做的好处可以参考notebook/doc.md at main · YdrMaster/notebook (github.com)

3.2 物理内存分配算法（堆与物理页帧）

在rCore和xv6中，均认为内核中分配物理页面就是一页一页的分配，没有连续的需求，所以没有写物理页帧连续页面分配的算法。**因此，我们需要用rust自己实现这些算法。**

在Project要求中，"实现两种物理内存分配算法并完成相关测试(best fit, first fit, worst fit，next fit或者其他分配算 法)。""比如实现比较复杂的Buddy System分配算法"。 实际上他们都属于连续资源的分配算法，解决的是碎片问题。在我们Project中，有三个地方需要用到连续资源的分配

- 操作系统的启动堆内存管理，以便操作系统使用平衡二叉树、向量等动态数据结构。
- 操作系统管理物理页帧，以分配给用户进程
- 用户进程在虚拟的地址空间连续分配内存，以使用动态数据结构。

为了屏蔽掉这些使用场景的差异，我们可以设计一个rust接口：

```
trait ContinuousStorageAllocationAlgorithm {
    fn new() -> Self;
```

```
    ///
    ///
    /// # 参数
    ///
    /// * `start`: 注意是Inclusive的
    /// * `end`: 注意是Exclusive的. 左闭右开。
    ///
    /// returns: ()
    fn init(&mut self, start: usize, end: usize);
    fn alloc(&mut self, count: usize) -> Option<usize>;
    fn dealloc(&mut self, frame: usize, count: usize);
    fn get_remain_frame_cnt(&mut self) -> usize;
}
```

具体对于物理页帧的分配，则在写一个接口和struct与之对应

```
/// 物理页帧的分配器接口。
trait FrameAllocator {
    fn new() -> Self;
    fn init(&mut self, l: PhysPageNum, r: PhysPageNum);
    fn alloc(&mut self, count: usize) -> Option<PhysPageNum>;
    fn dealloc(&mut self, ppn: PhysPageNum, count: usize);
    fn get_remain_frame_cnt(&mut self) -> usize;
}
```

3.2.1 best\first\worst

这几个算法实现比较简单，我们以best_fit为例

```
use super::ContinuousStorageAllocationAlgorithm;
// use alloc::collections::linked_list::LinkedList;
use alloc::collections::vec_deque::VecDeque;

pub struct BestFitAllocator {
    captains: VecDeque<(usize, usize)>,
    // 一些统计数据
    allocated: usize,
    total: usize,
}

impl ContinuousStorageAllocationAlgorithm for BestFitAllocator {
    fn new() -> Self {
        Self {
            captains: VecDeque::new(),
            allocated: 0,
            total: 0,
```

```
        }
    }

    fn init(&mut self, start: usize, end: usize) {
        assert!(start <= end);
        self.total = end - start;
        self.captains = VecDeque::with_capacity(self.total);
        self.captains.push_back((start, self.total));
        log::info!("最佳匹配分配器启动成功！当前空闲物理页帧的数量为{}",
self.get_remain_frame_cnt())
    }

    fn alloc(&mut self, count: usize) -> Option<usize> {
        let mut min_diff = usize::MAX;
        let mut arg_min:Option<usize> = None;
        for i in 0..self.captains.len(){
            let (allocated_frame, troop_size) = self.captains[i];
            if troop_size > count{
                log::debug!("{:?} 队长是申请{}空间的一个选择", self.captains[i],
count);
                let diff = troop_size-count;
                if diff<min_diff{
                    min_diff = diff;
                    arg_min = Some(i);
                }
            }else if troop_size==count{
                //提前结束，这就是最好的。
                log::debug!("{:?} 队长接受了 {} 的请求", self.captains[i], count);
                self.captains.remove(i);
                log::debug!("队伍消失");
                return Some(allocated_frame);
            }
        }
        if let Some(arg_min) = arg_min{
            log::debug!("{:?} 队长是申请{}空间的 best fit 选择",
self.captains[arg_min], count);
            let allocated_frame = self.captains[arg_min].0;
            self.captains[arg_min].0 += count;
            self.captains[arg_min].1 -= count;
            log::debug!("队伍状态变更为{:?}", self.captains[arg_min]);
            return Some(allocated_frame);
        }else {
            log::warn!("无法找到合适的连续空间！");
            None
        }
    }
```

```
    fn dealloc(&mut self, frame: usize, count: usize) {

        for i in 0..self.captains.len(){
            let (start, troop_size) = self.captains[i];
            assert_ne!(start, frame);
            if start>frame{ //前面一直都是比frame小的。现在比它大，所以插在前面。
                log::debug!("{:?} 队长的左边可以释放({}, {})", self.captains[i],
frame, count);
                assert!(frame+count<=start); //不应当overlap
                //试图合并
                //可以不合并，但是算法就不完备。
                if frame+count==start{
                    self.captains[i].0 = frame;
                    self.captains[i].1 +=count;
                    log::debug!("可以合并， 队伍状态变更为{:?}", self.captains[i]);
                }else {
                    self.captains.insert(i, (frame, count)); //新的队长。
                    log::debug!("右边不可以合并， 已经插入{:?}", self.captains[i]);
                }
                //顺便要看看左边能不能合并，因为这个情况之前没有考虑。
                if i!=0 {
                    let (start, troop_size) = self.captains[i-1];
                    if start+troop_size==frame {
                        self.captains[i].0 = start;
                        self.captains[i].1 += troop_size;
                        self.captains.remove(i-1);
                        log::debug!("左边可以合并，删除左边并且队伍状态变更为{:?}",
self.captains[i-1]);
                    }
                }
                return;
            }else{
                assert!(start+troop_size<=frame); //不应当overlap
            }
        }
        //都比frame小，插入到最后。
        log::debug!("在所有队长之后，释放并产生新的队长({}, {})", frame, count);
        self.captains.push_back((frame, count));
    }

    fn get_remain_frame_cnt(&mut self) -> usize {
        self.total - self.allocated
    }
}
```

3.2.2 buddy system

buddy system我们参考了buddy_system_allocator - Rust (docs.rs)的实现，并没有做太多修改，只是让这个实现符合我们的`ContinuousStorageAllocationAlgorithm`接口。

```
use super::ContinuousStorageAllocationAlgorithm;
use alloc::collections::btree_set::BTreeSet;
use core::cmp::min;
use core::mem::size_of;

/// # 兄弟齐心系统分配器
/// 一个使用伙伴系统(buddy system)策略的动态连续存储资源分配器(dynamic continuous storage resource allocator)。
/// 常用于操作系统(作为硬件资源的管理器)管理启动堆内存、物理内存、虚拟内存的连续存储分配。
pub struct BuddySystemAllocator {
    // 32个平衡二叉树有序集。保存的是32种不同大小的页面的32棵树表示的空闲列表。
    free_list: [BTreeSet<usize>; 32],
    // 一些统计数据
    allocated: usize,
    total: usize,
}

impl ContinuousStorageAllocationAlgorithm for BuddySystemAllocator {
    /// 使用默认构造函数初始化数组。
    fn new() -> Self {
        Self {
            free_list: Default::default(),
            allocated: 0,
            total: 0,
        }
    }

    fn init(&mut self, start: usize, end: usize) {
        assert!(start<=end);
        let mut total = 0; //一共成功获得了多少个页面。
        let mut current_start = start;
        while current_start < end {
            let low_bit = if current_start > 0 {
                current_start & (!current_start + 1) //树状数组中应当管辖的数量。就是取得了自己第一个low_bit 的大小。比如 low_bit(8) = 1000 low_bit(6) = 10
            } else {
                32
            };
            let size = min(low_bit, prev_power_of_two(end - current_start));
            total += size;
            // trailing_zeros()是结尾有多少个0.
            self.free_list[size.trailing_zeros() as usize].insert(current_start);
            current_start += size;
```

```
        }
        self.total += total;
        log::info!("兄弟齐心系统分配器启动成功！当前空闲物理页帧的数量为{}",
self.get_remain_frame_cnt())
    }

    fn alloc(&mut self, count: usize) -> Option<usize> {
        let size = count.next_power_of_two();       // 比size大的第一个2的幂
        let class = size.trailing_zeros() as usize; // 比如申请的 count 是3， 那么
class是2
        for i in class..self.free_list.len() {
            // Find the first non-empty size class, 找到之后，外循环不会继续找。 比如申
请3， 内存一共有8，一开始找到了8， i=3
            if !self.free_list[i].is_empty() {
                // Split buffers  从高到低，按照 buddy 进行分裂。 比如从 j = 3 到 2+1
                for j in (class + 1..i + 1).rev() {
                    if let Some(block_ref) = self.free_list[j].iter().next() { //
因为非空，基本都是这里
                        let block = *block_ref;
                        self.free_list[j - 1].insert(block + (1 << (j - 1))); //
中间的大小
                        self.free_list[j - 1].insert(block);
                        self.free_list[j].remove(&block);
                    } else {
                        return None;
                    }
                }

                let result = self.free_list[class].iter().next().clone();
                if let Some(result_ref) = result {
                    let result = *result_ref;
                    self.free_list[class].remove(&result);
                    self.allocated += size;
                    return Some(result);
                } else {
                    return None;
                }
            }
        }
        None
    }

    fn dealloc(&mut self, frame: usize, count: usize) {
        let size = count.next_power_of_two();
        let class = size.trailing_zeros() as usize;

        // Merge free buddy lists
```

```
            let mut current_ptr = frame;
            let mut current_class = class;
            while current_class < self.free_list.len() {
                let buddy = current_ptr ^ (1 << current_class);
                if self.free_list[current_class].remove(&buddy) == true {
                    // Free buddy found
                    current_ptr = min(current_ptr, buddy);
                    current_class += 1;
                } else {
                    self.free_list[current_class].insert(current_ptr);
                    break;
                }
            }

            self.allocated -= size;
        }

        fn get_remain_frame_cnt(&mut self) -> usize {
            self.total-self.allocated
        }
    }

    ///
    fn prev_power_of_two(num: usize) -> usize {
        // leading_zeros是说这个数字前面有多少个0.
        1 << (8 * (size_of::<usize>()) - num.leading_zeros() as usize - 1)
    }
```

4 16KiB 相比 4KiB 性能测试

下面我们实际通过实验测试16KiB操作系统是否实际比4KiB的性能更好。

首先编写一个简单的用户态程序。

```
//! 用户程序 bench_pgfault
//! 测试16KiB分页方案是否有效
#![no_std]
#![no_main]

#[macro_use]
extern crate user_lib;
extern crate alloc;
use alloc::vec::Vec;
use core::mem;
use core::f32;
use core::f64;
use user_lib::getpid;
use user_lib::get_time;
```

```rust
#[no_mangle]
pub fn main() -> i32 {
    println!("分页性能测试");
    let start = get_time();
    let mut v: Vec<i32> = Vec::new();
    let l = 16 * 1024 / mem::size_of::<i32>() / 2;
    // 在一个16KiB中反复读写。
    for i in 0..100 {
        for j in 0..l {
            // v.push(((((j as f32)/2.5 + 1.2)*(-1.25)+3.0) / 3.4) as i32) ;
            v.push((10 * i + j - (i * j)) as i32) ;
        }
        v.sort();
    }
    println!("测试结束，用时{}ms", get_time() - start);
    0
}
```

4.1 16KiB

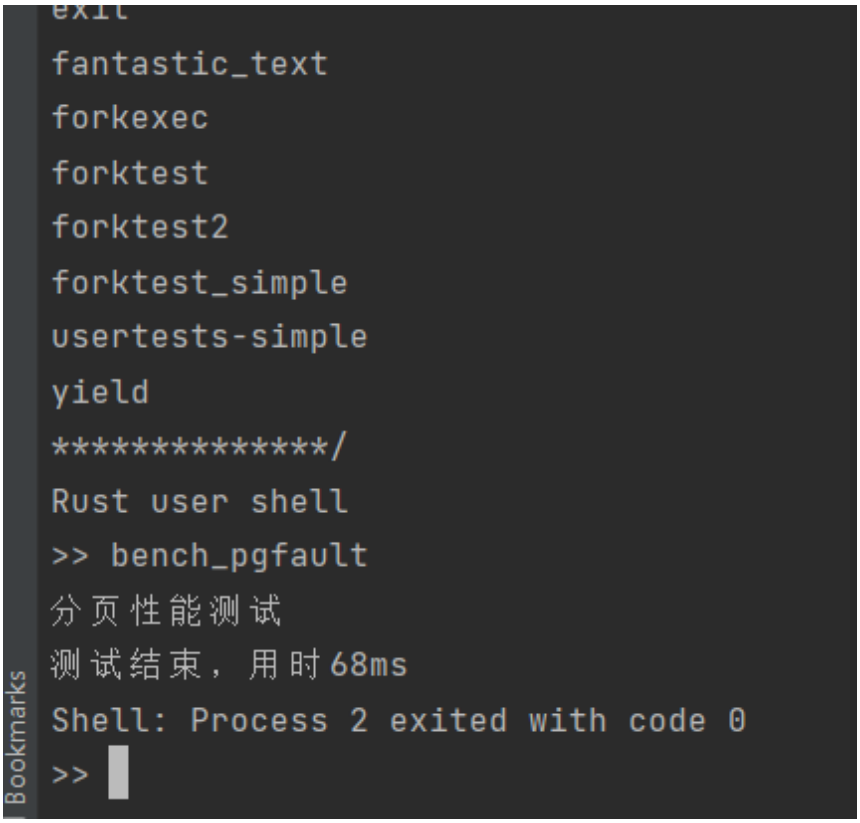
首先运行一次，让缓存加载起来。

接下来连续运行三次，我们认为这三次运行时间的平均数是16KiB系统运行该程序的期望时间。

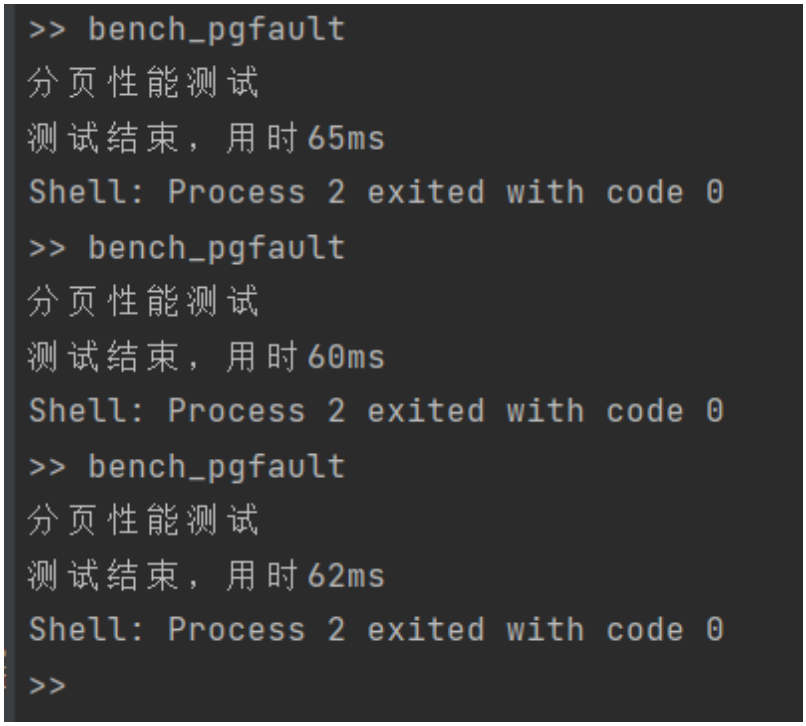
```
[jumbuck_nucleus]
INFO:   App 加载完成。开始调度运行。
Jumbuck Nucleus OS 绵羊核心操作系统 [Version 0.1.0]
>> bench_pgfault
分页性能测试
测试结束，用时58ms
user_shell: 进程 2 以退出码 0 结束。
>> bench_pgfault
分页性能测试
测试结束，用时55ms
user_shell: 进程 2 以退出码 0 结束。
>> bench_pgfault
分页性能测试
测试结束，用时53ms
user_shell: 进程 2 以退出码 0 结束。
>>
```

4.2 4KiB

我们使用未修改的4KiB rCore进行测试(仍然需要做一些堆大小的修改)，加入同样的bench_pgfault程序



似乎初始化更快。接下来运行三次



可以看出，16KiB的操作系统比4KiB的快10ms左右，提速约13%。

5 总结与展望

通过本次Project，我们

- 深入学习了rCore关于虚拟内存和进程管理的实现，受益匪浅
  - 比较了不同版本rCore和uCore实现的不同。比如内核过渡虚拟地址的不同。
  - 理解了操作系统的运行过程。
- 了解到rust语言，初步学习其使用。
  - 了解了rust语言的基本语法和内存管理机制
  - 使用rust语言复现了Lab和Assignment中shell, first fit, rr等算法。
- 练习了makefile的使用

这都加深了我们对操作系统理论课中学习到的知识的理解。然而，由于时间不够以及实现细节的复杂，还有一下几点我们做得不足

- qemu修改比较简单，没有设计一个专门的MODE，而是沿用Sv39。可以使用更加合理的11-11-11-14设计。
- rCore代码进程管理部分阅读比较粗浅，虚拟内存阅读地比较多，不过仍有部分细节和设计没有完全理解。
- rCore中在ch5分支没有实现页面置换、LRU等内容，这就让我们无法对16KiB系统的page fault数量进行统计，无法进一步测试16KiB系统的性能优势。

参考文献

1. P. Weisberg and Y. Wiseman, “Virtual Memory Systems Should use Larger Pages,” Aug. 2015, pp. 1 – 4. doi: 10.14257/astl.2015.106.01. ↩ ↩ ↩ ↩ ↩ ↩ ↩ ↩

2. Patterson and Hennessy, *计算机组成与设计——硬件/软件接口(原书第五版)（RISC-V版）*. ↩

3. “rCore-Tutorial-Book-v3 3.6.0-alpha.1 文档.” http://rcore-os.cn/rCore-Tutorial-Book-v3/index.html (accessed Jul. 27, 2022). ↩

4. Rust 程序设计语言.” https://www.rust-lang.org/zh-CN/ (accessed Jul. 29, 2022). ↩ ↩

5. R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating systems: Three easy pieces*, 1.00. Arpaci-Dusseau Books, 2018. ↩