# 教程

#### 动态内存分配

这一节是为了之后的两个教程打基础的章节

**我们的目的:**

实现Trait GlobalAlloc

unsafe fn alloc(&self, layout: Layout) -> \*mut u8;

unsafe fn dealloc(&self, ptr: \*mut u8, layout: Layout);

为了实现这个trait 必须要实现上面两个函数，也就是分配-回收内存

Layout包括两个字段,size是分配的字节数,align是对齐方式,地址必须是align的倍数,align必须是2的整数幂

我们利用伙伴系统来做这件事

代码无需多说，这里主要对伙伴系统的代码进行一个初步的分析

#### 物理内存管理

**第三章可以把前两章的知识都概括了**

首先定义PhysicalAddress与PhysicalNumber的结构体，也就是页号和物理地址

它的形式如下：

pub struct PhysicalAddress(pub usize);

pub struct PhysicalNumber (pub usize);

然后定义一页的大小：

pub const PAGE\_SIZE: usize = 4096; 必须是2的幂次

定义可以访问的内存区域起始地址：

pub const MEMORY\_START\_ADDRESS: PhysicalAddress = PhysicalAddress(0x8000\_0000);

/// 可以访问的内存区域结束地址

pub const MEMORY\_END\_ADDRESS: PhysicalAddress = PhysicalAddress(0x8800\_0000);

正好是8M

实现一个分配器进行分配和回收

其中有代码

pub fn address(&self) -> PhysicalAddress {

self.0.into()

}

/// 帧的物理页号

pub fn page\_number(&self) -> PhysicalPageNumber {

self.0

}

所以PhysicalPageNumber必须实现into

但是Into有一个默认的实现,如果U实现了From<T>,T类型调用into就可以转换为U 然后就会调用析构函数

封装一个物理页分配器

pub trait Allocator {

/// 给定容量，创建分配器

fn new(capacity: usize) -> Self;

/// 分配一个元素，无法分配则返回 `None`

fn alloc(&mut self) -> Option<usize>;

/// 回收一个元素

fn dealloc(&mut self, index: usize);

}

FrameAllocator以Allocator为泛型

pub struct FrameAllocator<T: Allocator> {

/// 可用区间的起始

start\_ppn: PhysicalPageNumber,

/// 分配器

allocator: T,

}

为这个分配器trait实例化

impl<T: Allocator> FrameAllocator<T> {

/// 创建对象

pub fn new(range: impl Into<Range<PhysicalPageNumber>> + Copy) -> Self {

FrameAllocator {

start\_ppn: range.into().start,

allocator: T::new(range.into().len()),

}

}

Impl Trait这种用法就相当于 使用trait限定的泛型

这里又调用了Range的实现

#[derive(Clone, Copy, Debug, Eq, PartialEq)]

pub struct Range<T: From<usize> + Into<usize> + Copy> {

pub start: T,

pub end: T,

}//满足Into和From Trait以及Copy Trait

pub fn alloc(&mut self) -> MemoryResult<FrameTracker> {

self.allocator

.alloc()

.ok\_or("no available frame to allocate")

.map(|offset| FrameTracker(self.start\_ppn + offset))

}

//这里使用了MemoryResult<FrameTracker>

pub type MemoryResult<T> = Result<T, &'static str>;

成功则返回Ok(FrameTracker(self.start\_ppn + offset)) 失败Err

/// 这个函数会在 [`FrameTracker`] 被 drop 时自动调用，不应在其他地方调用

pub(super) fn dealloc(&mut self, frame: &FrameTracker) {

self.allocator.dealloc(frame.page\_number() - self.start\_ppn);

}

最后来看帧分配器

lazy\_static!{

pub static ref FRAME\_ALLOCATOR:Mutex<FrameAllocator<AllocatorImpl>> = Mutex::new(FrameAllocator::new(Range::from(PhysicalPageNumber::ceil(PhysicalAddress::from(\*KERNEL\_END\_ADDRESS))..PhysicalPageNumber::floor(MEMORY\_END\_ADDRESS),)));

}

lazy\_static是给静态变量延迟赋值的宏。

使用这个宏,所有 static类型的变量可在执行的代码在运行时被初始化。 这包括任何需要堆分配,如vector或hash map,以及任何非常量函数调用。需要ref才可以使用

这里利用了1..5与Mutex结构 Mutex是给它“上锁的” 同一时间只能有一个线程操作

**进行分解**

FrameAllocator::new(Range::from(PhysicalPageNumber::ceil(PhysicalAddress::from(\*KERNEL\_END\_ADDRESS))..PhysicalPageNumber::floor(MEMORY\_END\_ADDRESS),)) 是分配内存的命令

for Range<T> {

fn from(range: core::ops::Range<U>) -> Self {

Self {

start: range.start.into(),

end: range.end.into(),

}

}

}

也就是创建一个Range区间

继续分解：

PhysicalPageNumber::ceil(PhysicalAddress::from(\*KERNEL\_END\_ADDRESS)) ..PhysicalPageNumber::floor(MEMORY\_END\_ADDRESS)

pub static ref KERNEL\_END\_ADDRESS: PhysicalAddress = PhysicalAddress(kernel\_end as usize);

pub const MEMORY\_END\_ADDRESS: PhysicalAddress = PhysicalAddress(0x8800\_0000);

就是从kernel\_end -- MEMORY\_END\_ADDRESS

pub const fn floor(address: $address\_type) -> Self {

Self(address.0 / PAGE\_SIZE)

}

/// 将地址转换为页号，向上取整

pub const fn ceil(address: $address\_type) -> Self {

Self(address.0 / PAGE\_SIZE + (address.0 % PAGE\_SIZE != 0) as usize)

}

其中 floor 就是 0x8800\_0000/4096 其实也就是取整

Ceil就是kernel\_end/4096 根据kernel\_end的值 决定取得的起始页是否加一

也就是约定好了有多少页

**整个调用流程**

Mutex::new(FrameAllocator::new(Range::from(start..end,)));

let frame\_1 = match memory::frame::FRAME\_ALLOCATOR.lock().alloc() {

Result::Ok(frame\_tracker) => frame\_tracker,

Result::Err(err) => panic!("{}", err)

};

然后进行Range赋值 start…end 进行new操作 然后lock

pub fn new(range: impl Into<Range<PhysicalPageNumber>> + Copy) -> Self {

FrameAllocator {

start\_ppn: range.into().start,

allocator: T::new(range.into().len()),

}

}

start\_ppn 是range.into().start起始地址

allocator: range.into().len()就是所有可分配的页

pub fn alloc(&mut self) -> MemoryResult<FrameTracker> {

self.allocator

.alloc()

.ok\_or("no available frame to allocate")

.map(|offset| FrameTracker(self.start\_ppn + offset))

}

Start\_ppn不变 返回Option<usize> 解析得到 offset就是这个usize 然后FrameTracker里的链表改变

fn alloc(&mut self) -> Option<usize> {

if let Some((start, end)) = self.list.pop() {

if end - start > 1 {

self.list.push((start + 1, end));

}

Some(start)

} else {

None

}

}

具体的栈实现方法

List初始为[(0,len)]

经过一个alloc=>[(1,len)] 一直递归进行

每次返回start 也就是0、1、2..相当于offset

dealloc

fn dealloc(&mut self, index: usize) {

self.list.push((index, index + 1));

}

也就是将frame.page\_number()-self.start\_ppn push进去 证明这个空间已经空了 搞明白了！

# 实验

#### 回答：我们在动态内存分配中实现了一个堆，它允许我们在内核代码中使用动态分配的内存，例如 Vec Box 等。那么，如果我们在实现这个堆的过程中使用 Vec 而不是 [u8]，会出现什么结果

#### 回答：algorithm/src/allocator 下有一个 Allocator trait，我们之前用它实现了物理页面分配。这个算法的时间和空间复杂度是什么？

具体可以看代码

fn alloc(&mut self) -> Option<usize> {

if let Some((start, end)) = self.list.pop() {

if end - start > 1 {

self.list.push((start + 1, end));

}

Some(start)

} else {

None

}

}

只有一个pop和push 所以时间复杂度是O(1)

至于空间复杂度 我们考虑一个极端情况 n个空间已经被分配 现在同时析构

所以空间复杂度是O(n)

#### 实现基于线段树的物理页面分配算法

我们仿照第一种栈结构 定义一种分配器命名为 lineAlllocator

我们使用静态数组来构造线段树,也就是总容量的四倍

在这里参考论文<<基于线段树的高效内存管理算法及其空间优化>>

建立优化的线段树物理页面分配算法

**建立线段树节点struct**

struct node{

left\_address:usize, //起始地址

right\_address:usize, //结束地址

max\_free\_interval:usize, //连续最长空闲长度

left\_available:uize, //由起始地址向右最长连续空闲长度

right\_available:usize, //由结束地址向左

lazy:usize, //延迟更新标记

}

**建立线段树struct 这里利用了栈结构**

我们可以发现 这里存的是线段树的有序结构 就是 i的左子树是2i 右子树是2i+1

fn createTree(&mut self){

let Node{left\_address:start,right\_address:end,..}=self.list[0];

let mut stack1=vec![(start,end)];

let mut stack2=vec![];

loop{

if let Some((left,right)) = stack1.pop() {//pop 1 to 2

if(left!=right)

stack2.push((left,(left+right)/2));

stack2.push(((left+right)/2+1,right));

self.list.push(Node::new(left,(left+right)/2));

self.list.push(Node::new((left+right)/2+1,right));//build a sort

}

}else{

while let Some((left,right)) = stack2.pop(){

stack1.push((left,right));

}

if(stack1.len()==0){break;}

}

}

}

**接下来要建立内存分配，我们利用延迟更新，这样效率更高**

首先我们需要进行两个步骤 ：

1. 寻找适合分配内存的最小块
2. 进行update 改变节点的分配状态