

一、伙伴系统原理

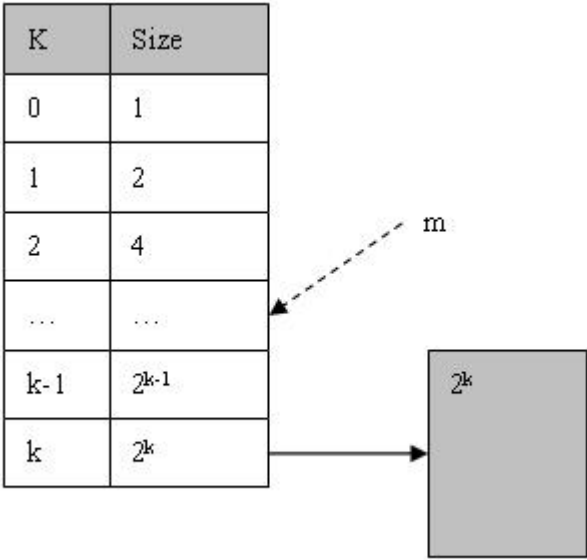
1.什么是伙伴？

定义：由一个母实体分成的两个**各方面属性一致**的两个子实体，这两个子实体就处于伙伴关系。在操作系统分配内存的过程中，一个内存块常常被分成两个大小相等的内存块，这**两个大小相等的内存块就处于伙伴关系**。它满足 3 个条件：

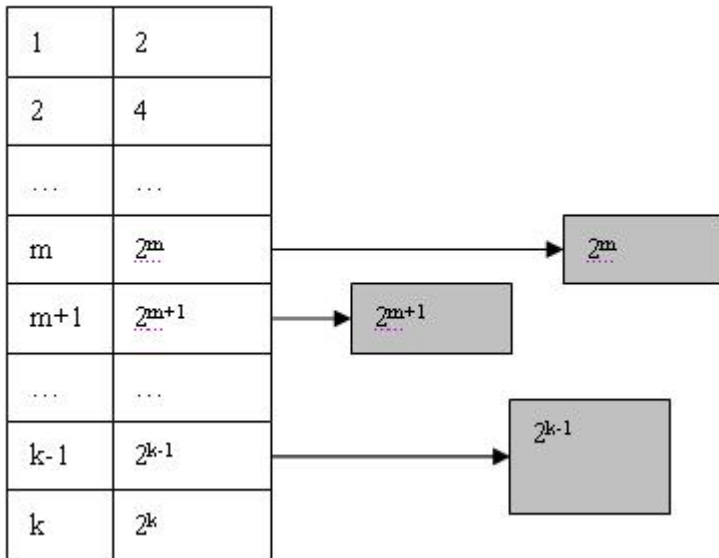
- 两个块具有相同大小，记为 2^K
- 它们的物理地址是连续的
- 从同一个大块中拆分出来

2.伙伴分配算法原理

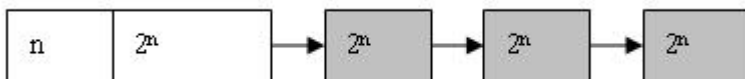
Buddy System把系统中的可用存储空间划分为存储块(Block)来进行管理，每个存储块的大小必须是2的n次幂($Pow(2, n)$)，即1, 2, 4, 8, 16, 32, 64, 128... 假设系统全部可用空间为 $Pow(2, k)$ 字节，那么在Buddy System初始化时将生成一个长度为 $k + 1$ 的可用空间表List, 并将全部可用空间作为一个大小为 $Pow(2, k)$ 的块挂接在List的最后一个节点上，如下图：



当用户申请size个字节的存储空间时，Buddy System分配的Block大小为 $Pow(2, m)$ 个字节大小($Pow(2, m-1) < size \leq Pow(2, m)$)。此时Buddy System将在List中的m位置寻找可用的Block。显然List中这个位置为空，于是Buddy System就开始寻找向上查找 $m+1, m+2$, 直达到k为止。找到k后，便得到可用Block(k)，此时Block(k)将分裂成两个大小为 $Pow(k-1)$ 的块，并将其中一个插入到List中k-1的位置，同时对另外一个继续进行分裂。如此以往直到得到两个大小为 $Pow(2, m)$ 的块为止，如下图所示：



如果系统在运行一段时间之后，List中某个位置n可能会出现多个块，则将其其他块依次链接到可用块链表的末尾：



当Buddy System要在n位置取可用块时，直接从链表头取一个就行了。

当一个存储块不再使用时，用户应该将该块归还给Buddy System。此时系统将根据Block的大小计算出其在List中的位置，然后插入到可用块链表的末尾。在**这一步完成后，系统立即开始合并操作**。该操作是将伙伴合并到一起，并放到List的下一个位置中，并继续对更大的块进行合并，直到无法合并为止。

因此，何谓“伙伴”？如前所述，在分配存储块时经常会将一个大的存储块分裂成两个大小相等的小块，那么这两个小块就称为“伙伴”。在Buddy System进行**合并时，只会将互为伙伴的两个存储块合并成大块**，也就是说假如有两个存储块大小相同，地址也相邻，但是不是由同一个大块分裂出来的，也不会被合并起来。正常情况下，起始地址为p, 大小为 $\text{Pow}(2, k)$ 的存储块，其伙伴块的起始地址为: $p + \text{Pow}(2, k)$ 或 $p - \text{Pow}(2, k)$ 。即：

起始地址为 p , 大小为 2^k 的内存块, 其伙伴块的起始地址为：

$$\text{buddy}(p, k) = \begin{cases} p + 2^k & (\text{若 } p \bmod 2^{k+1} = 0) \\ p - 2^k & (\text{若 } p \bmod 2^{k+1} = 2^k) \end{cases}$$

理解如下：

当P是大小为 $\text{Pow}(2, K+1)$ 的内存块的起始地址时， $P \bmod \text{Pow}(2, K+1) = 0$ ，显然，P指向的大小为 $\text{Pow}(2, K)$ 的内存块的伙伴块的起始地址为 $P + \text{Pow}(2, K)$ 。

当P是大小为 $\text{Pow}(2, K+1)$ 的内存块的中间地址时，意味着大小为 $\text{Pow}(2, K+1)$ 的内存块从它这里被分成两半， $P \bmod \text{Pow}(2, K+1) = \text{Pow}(2, K)$ ，那么，P指向的内存块的伙伴块的起始地址为 $P - \text{Pow}(2, K)$ 。

3.示例

Step	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K
1	2 ⁴															
2.1	2 ³								2 ³							
2.2	2 ²				2 ²				2 ³							
2.3	2 ¹		2 ¹		2 ²				2 ³							
2.4	2 ⁰	2 ⁰	2 ¹		2 ²				2 ³							
2.5	A: 2 ⁰	2 ⁰	2 ¹		2 ²				2 ³							
3	A: 2 ⁰	2 ⁰	B: 2 ¹		2 ²				2 ³							
4	A: 2 ⁰	C: 2 ⁰	B: 2 ¹		2 ²				2 ³							
5.1	A: 2 ⁰	C: 2 ⁰	B: 2 ¹		2 ¹		2 ¹		2 ³							
5.2	A: 2 ⁰	C: 2 ⁰	B: 2 ¹		D: 2 ¹		2 ¹		2 ³							
6	A: 2 ⁰	C: 2 ⁰	2 ¹		D: 2 ¹		2 ¹		2 ³							
7.1	A: 2 ⁰	C: 2 ⁰	2 ¹		2 ¹		2 ¹		2 ³							
7.2	A: 2 ⁰	C: 2 ⁰	2 ¹		2 ²				2 ³							
8	2 ⁰	C: 2 ⁰	2 ¹		2 ²				2 ³							
9.1	2 ⁰	2 ⁰	2 ¹		2 ²				2 ³							
9.2	2 ¹		2 ¹		2 ²				2 ³							
9.3	2 ²				2 ²				2 ³							
9.4	2 ³								2 ³							
9.5	2 ⁴															

Step1 初始化时, 系统拥有1M的连续内存, 允许的最小的内存块为64K, 图中白色的部分为空闲的内存块, 着色的代表分配出去的内存块。

Step2 程序A申请一块大小为34K的内存, 对应的order为0, 即 $2^0=1$ 个最小内存块

Step2.1 系统中不存在order 0(64K)的内存块, 因此order 4(1M)的内存块分裂成两个order 3的内存块(512K)

Step2.2 仍然没有order 0的内存块, 因此order 3的内存块分裂成两个order 2的内存块(256K)

Step2.3 仍然没有order 0的内存块, 因此order 2的内存块分裂成两个order 1的内存块(128K)

Step2.4 仍然没有order 0的内存块, 因此order 1的内存块分裂成两个order 0的内存块(64K)

Step2.5 找到了order 0的内存块, 将其中的一个分配给程序A, 现在伙伴系统的内存为一个order 0的内存块, 一个order 1的内存块, 一个order 2的内存块以及一个order 3的内存块

Step3 程序B申请一块大小为66K的内存, 对应的order为1, 即 $2^1=2$ 个最小内存块, 由于系统中正好存在一个order 1的内存块, 所以直接用来分配

Step4 程序C申请一块大小为35K的内存, 对应的order为0, 同样由于系统中正好存在一个order 0的内存块, 直接用来分配

Step5 程序D申请一块大小为67K的内存, 对应的order为1

Step5.1 系统中不存在order 1的内存块, 于是将order 2的内存块分裂成两块order 1的内存块

Step5.2 找到order 1的内存块, 进行分配

Step6 程序B释放了它申请的内存, 即一个order 1的内存块

Step7 程序D释放了它申请的内存

Step7.1 一个order 1的内存块回收到内存当中

Step7.2 由于该内存块的伙伴也是空闲的，因此两个order 1的内存块合并成一个order 2的内存块

Step8 程序A释放了它申请的内存，即一个order 0的内存块

Step9 程序C释放了它申请的内存

Step9.1 一个order 0的内存块被释放

Step9.2 两个order 0伙伴块都是空闲的，进行合并，生成一个order 1的内存块

Step9.3 两个order 1伙伴块都是空闲的，进行合并，生成一个order 2的内存块

Step9.4 两个order 2伙伴块都是空闲的，进行合并，生成一个order 3的内存块

Step9.5 两个order 3伙伴块都是空闲的，进行合并，生成一个order 4的内存块

二、rCore团队对buddy_system_allocator的实现和使用

rCore系列教程中，buddy_system_allocator的版本为0.6.0。

```
// os/src/mm/heap_allocator.rs
use buddy_system_allocator::LockedHeap;
use crate::config::KERNEL_HEAP_SIZE;

#[global_allocator]
static HEAP_ALLOCATOR: LockedHeap = LockedHeap::empty();

#[alloc_error_handler]
pub fn handle_alloc_error(layout: core::alloc::Layout) -> ! {
    panic!("Heap allocation error, layout = {:?}", layout);
}

/// 堆空间：目前放在内核的bss段上
/// 全局变量的初始值是0，编译器将该全局变量放在bss段上
static mut HEAP_SPACE: [u8; KERNEL_HEAP_SIZE] = [0; KERNEL_HEAP_SIZE];

pub fn init_heap() {
    unsafe {
        HEAP_ALLOCATOR.lock().init(HEAP_SPACE.as_ptr() as usize,
        KERNEL_HEAP_SIZE);
    }
}
```

1.堆数据结构Heap

在堆分配器眼中，它面对的堆的数据结构是怎样的呢？

```
pub struct Heap {
    // buddy system with max order of 32
    free_list: [linked_list::LinkedList; 32],

    // statistics
    user: usize,
    allocated: usize,
    total: usize,
}
```

free_list其实就是32个链表头的裸指针。它还有一些重要的方法，后面再分析。

2.谁是堆分配器？

我们必须告诉Rust编译器它应该使用哪个堆分配器，这就是#[global_allocator]属性发挥作用的地方。#[global_allocator]属性告诉Rust编译器应该使用哪个分配器实例作为全局堆分配器。该属性仅适用于**实现GlobalAlloc trait的静态对象**。**全局分配器适用于项目中的所有crate。**

GlobalAlloc trait是怎样的？因为我们需要为所有的分配器类型实现该属性，因此有必要仔细研究其声明：

```
pub unsafe trait GlobalAlloc {
    // required methods
    unsafe fn alloc(&self, layout: Layout) -> *mut u8;
    unsafe fn dealloc(&self, ptr: *mut u8, layout: Layout);
    // provided methods
    unsafe fn alloc_zeroed(&self, layout: Layout) -> *mut u8 { ... }
    unsafe fn realloc(
        &self,
        ptr: *mut u8,
        layout: Layout,
        new_size: usize
    ) -> *mut u8 { ... }
}
```

它定义了两个必需的方法**alloc**和**dealloc**：

alloc方法接收一个**Layout**参数，该实例描述分配的内存应具有的大小和对齐方式。如下：

```
pub struct Layout {
    // 请求的内存块的大小，以字节为单位。
    size_: usize,
    // 请求的内存块的对齐方式，以字节为单位。
    // 我们确保始终是2的幂值且不为0.....
    align_: NonZeroUsize,
}
```

它返回**原始指针**，指向分配的内存块的第一个字节。alloc方法如果发生错误，不返回显式的错误值，而是**返回一个空指针来表示分配错误**。这有点不习惯，但它的优点是包装现有的系统分配器很容易，因为它们使用相同的约定。

dealloc方法负责再次释放内存块。它接收两个参数，一个是alloc返回的指针，另一个是用于分配的布局。

该trait还定义了两个方法**alloc_zeroed**和**realloc**，它们都有默认的实现：

alloc_zeroed方法等效于调用alloc，然后将分配的内存块设置为零，这正是提供的默认实现所执行的。如果可能的话，分配器实现可以使用更有效的自定义实现来覆盖默认实现。

realloc方法允许增加或减少分配的大小，默认实现分配一个具有所需大小的新内存块，并复制先前分配的所有内容。同样，分配器实现可能可以提供此方法的更有效实现，例如，如果可能的话，在适当的位置增加/减少分配。

要注意的是，trait本身和所有trait方法都被声明为不安全的：将该trait声明为不安全的原因是程序员必须保证分配器类型的trait实现是正确的。例如，alloc方法绝不能返回已在其他地方使用的内存块，因为这将导致未定义的行为。类似地，方法不安全的原因是调用者在调用方法时必须确保各种变量没有错误，例如，传递给alloc的布局指定一个非零大小。这在实践中并不真正相关，因为编译器通常直接调用这些方法，以确保满足需求。

正如我们在讨论GlobalAlloc特征时所了解的，alloc函数可以通过返回空指针来发出分配错误的信号。问题是：Rust运行时应该如何应对这样的分配失败？这就是**#[alloc_error_handler]属性的作用：它指定在分配错误发生时调用的函数**，类似于在发生紧急情况时调用紧急处理程序的方式。可能是因为这个属性还不稳定，我们需要在main.rs中加上这个条件：

```
#![feature(alloc_error_handler)]
```

3.互斥锁简介

互斥锁：对共享数据进行锁定，保证同一时刻只能有一个线程去操作。

注意: 互斥锁是**多个线程一起去抢**，抢到锁的线程先执行，没有抢到锁的线程需要等待，等互斥锁使用完释放后，其它等待的线程再去抢这个锁。

Mutex是Rust标准库中提供的互斥锁的实现。使用的示例代码如下所示：

```
use std::sync::Mutex;
fn main() {
    let m = Mutex::new(5);
    {
        let mut num = m.lock().unwrap();
        *num = 6;
    }
    println!("m = {:?}", m);
}
```

共享变量是存储于Mutex对象的内部的，**如果要访问共享的数据，需要先用lock()方法先获取锁，然后才能访问内部的数据。**

4.堆分配器的类型:LockedHeap

为什么叫LockedHeap? 因为它使用spin::Mutex进行同步, 以避免多个线程同时访问。若线程加锁失败, CPU释放该线程, 执行其他线程。

```
#[global_allocator]
static HEAP_ALLOCATOR: LockedHeap = LockedHeap::empty();
pub fn init_heap() {
    unsafe {
        HEAP_ALLOCATOR.lock().init(HEAP_SPACE.as_ptr() as usize,
        KERNEL_HEAP_SIZE);
    }
}
```

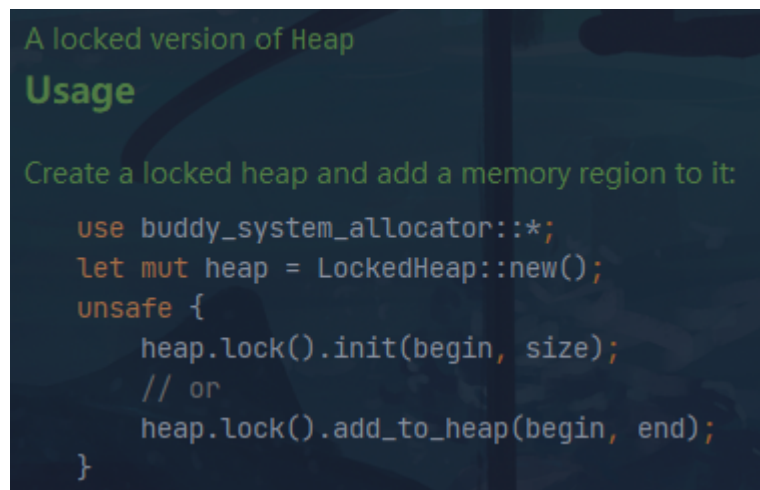
在对堆分配器进行任何操作之前都要先获取锁 (使用.lock()方法) 以避免其他线程同时对它进行操作导致数据竞争。

而LockedHeap是怎样的类型呢?

```
#[cfg(feature = "use_spin")]
pub struct LockedHeap(Mutex<Heap>);
```

其实就是对Mutex<Heap>的包装。

文档中也很清晰的给出了使用方法:



A locked version of Heap

Usage

Create a locked heap and add a memory region to it:

```
use buddy_system_allocator::*;
let mut heap = LockedHeap::new();
unsafe {
    heap.lock().init(begin, size);
    // or
    heap.lock().add_to_heap(begin, end);
}
```

```
// .../lib.rs
pub struct LockedHeap(Mutex<Heap>);
// .../heap_allocator.rs
pub fn init_heap() {
    unsafe {
        HEAP_ALLOCATOR.lock().init(HEAP_SPACE.as_ptr() as usize,
        KERNEL_HEAP_SIZE);
    }
}
```

```
}
}
```

刚开始看上述代码时很好奇，为什么HEAP_ALLOCATOR:LockedHeap能直接使用lock()方法？难道不应该把Mutex<Heap>声明为public然后使用HEAP_ALLOCATOR.0.lock()吗？这是因为实现了Deref trait，**假设 T 实现了 Deref Trait，隐式解引用转化可以把对 T 的引用转化为 T 经过 Deref 操作后生成的引用**，使得HEAP_ALLOCATOR能自动转化为&Mutex<Heap>，因此HEAP_ALLOCATOR.lock().init()几乎等同于Heap::init()

```
impl Deref for LockedHeap {
    type Target = Mutex<Heap>;

    fn deref(&self) -> &Mutex<Heap> {
        &self.0
    }
}
```

于是，分析可得堆分配器的初始化工作应该集中在Heap::init中进行。它主要的作用是让堆分配器有一段可用来分配的内存。

5.堆分配器的第一项工作：对堆内存初始化

```
pub unsafe fn init(&mut self, start: usize, size: usize) {
    self.add_to_heap(start, start + size);
}
```

首先明确一点，堆初始化时，还没开启分页，所有地址均为物理地址。

init函数要求传入堆内存的起始地址和堆的大小，这些都好办。目前堆放在内核的bss段上面，堆大小定为KERNEL_HEAP_SIZE: usize = 0x30_0000。因此如果要问动态分配的内存地址在哪个范围里？答案是在内核的bss段上，相当于内核代码的一部分。

随后将堆的起止地址传给add_to_heap。其具体实现细节暂时不分析，只知道它能够将传入的一段起止内存添加到堆就行了。

6.GlobalAlloc trait的实现

LockedHeap是全局堆分配器的类型。那么必须为其实现GlobalAlloc trait，它是如何分配堆内存的呢？

```
unsafe impl GlobalAlloc for LockedHeap {
    unsafe fn alloc(&self, layout: Layout) -> *mut u8 {
        self.0
            .lock()
            .alloc(layout)
            .ok()
            .map_or(0 as *mut u8, |allocation| allocation.as_ptr())
    }
}
```



```

    unsafe fn dealloc(&self, ptr: *mut u8, layout: Layout) {
        self.0.lock().dealloc(NonNull::new_unchecked(ptr), layout)
    }
}

```

可以看到，堆分配器要分配一块堆内存时使用LockedHeap::alloc函数，传入Layout，返回u8指针。在其内部，Heap::alloc返回的是Result<u8指针, ()>，ok将其转化为Option<T>，即Some(u8指针)或None，map_or进一步对Option包裹的T进行处理，返回u8裸指针或默认值0（表示分配失败）。

所以实质就是调用Heap::alloc。其中使用了伙伴分配算法，暂不分析细节。

堆分配器要释放一块内存使用LockedHeap::dealloc函数，传入Layout和内存的起始地址。其实质也就是调用Heap::dealloc，其中使用了伙伴回收的算法，如回收相邻的伙伴块，暂不分析其细节。

7.Heap::alloc与dealloc的实现细节

主要考虑了对齐的问题。例如，当申请pow(2, 14)大小的内存时，首先到free_list[14]中查找有无空闲块，若无则尝试找更大的空闲块进行拆分。

```

/// Alloc a range of memory from the heap satifying `layout` requirements
pub fn alloc(&mut self, layout: Layout) -> Result<NonNull<u8>, ()> {
    // next_power_of_two() returns the smallest pow(2, m) >= layout.size()
    // let size = 要分配的大小，最小要满足4/8字节、对齐方式
    let size = max(
        layout.size().next_power_of_two(),
        max(layout.align(), size_of::<usize>()),
    );
    // trailing_zeros()返回二进制表示中尾随零的数量
    // let n = Wrapping(0b0101000i64);
    // assert_eq!(n.trailing_zeros(), 3);
    let class = size.trailing_zeros() as usize;
    for i in class..self.free_list.len() {
        // Find the first non-empty size class
        if !self.free_list[i].is_empty() {
            // Split buffers
            for j in (class + 1..i + 1).rev() {
                if let Some(block) = self.free_list[j].pop() {
                    unsafe {
                        // 把freelist[j]拆成两个freelist[j - 1]，地址靠后的先入
                        // j = 15, 1 << (j - 1) = 1 << 14 = pow(2, 14)
                        self.free_list[j - 1]
                            .push((block as usize + (1 << (j - 1))) as *mut
                                usize);
                        self.free_list[j - 1].push(block);
                    }
                } else {
                    return Err(());
                }
            }
        }
    }
}

```

```

        let result = NonNull::new(
            self.free_list[class]
                .pop()
                .expect("current block should have free space now")
                as *mut u8,
        );
        return if let Some(result) = result {
            self.user += layout.size();
            self.allocated += size;
            Ok(result)
        } else {
            Err(())
        }
    }
}
Err(())
}

```

Heap::dealloc

```

/// Dealloc a range of memory from the heap
pub fn dealloc(&mut self, ptr: NonNull<u8>, layout: Layout) {
    // 找到dealloc的大小
    let size = max(
        layout.size().next_power_of_two(),
        max(layout.align(), size_of::<usize>()),
    );
    let class = size.trailing_zeros() as usize;
    unsafe {
        // Put back into free list, 把要回收的block放回链表尾
        self.free_list[class].push(ptr.as_ptr() as *mut usize);

        // Merge free buddy lists
        let mut current_ptr = ptr.as_ptr() as usize;
        let mut current_class = class;
        while current_class < self.free_list.len() {
            // 找到伙伴块的地址
            let buddy = current_ptr ^ (1 << current_class);
            let mut flag = false;
            for block in self.free_list[current_class].iter_mut() {
                if block.value() as usize == buddy {
                    // 拿出伙伴块
                    block.pop();
                    flag = true;
                    break;
                }
            }

            // Free buddy found
            if flag {
                self.free_list[current_class].pop();
            }
        }
    }
}

```

```

        current_ptr = min(current_ptr, buddy);
        current_class += 1;
        self.free_list[current_class].push(current_ptr as *mut usize);
    } else {
        break;
    }
}

self.user -= layout.size();
self.allocated -= size;
}

```

```

pub(crate) fn prev_power_of_two(num: usize) -> usize {
    // leading_zeros()返回二进制表示中前导零的数量
    // let n = Wrapping(u64::MAX) >> 2;
    // assert_eq!(n.leading_zeros(), 2);
    1 << (8 * (size_of::<usize>()) - num.leading_zeros() as usize - 1)
}

```

三、问题与改进

先前设计的时候，堆的大小是固定的（0x8000），经过测试，发现主要有两个问题：

一是初始堆内存过大，对于一般程序而言，不需要这么大的初始内存，调整为0x2000左右即可；

二是有时候会出现堆内存不够，但实际上还有可用内存的情况。

因此改为堆初始化时分配0x2000大小的内存，当程序的堆不够时，动态扩张堆的大小。具体实现上，在0.6.0的版本基础上做出如下改变：

```

pub struct LockedHeap {
    inner: Mutex<Heap>,
    rescue: fn(&mut Heap, Layout),
}

```

rescue用于在对内存不够时进行动态扩张，默认的rescue函数实现如下，主要是移动brk：

```

pub const fn empty() -> LockedHeap {
    LockedHeap {
        inner: Mutex::new(Heap::new()),
        rescue: |heap: &mut Heap, layout: Layout| unsafe {
            // get current brk
            let cur_brk = brk(None).unwrap();
            // get request
            let request_size = layout.size();

```

```

        let allocated_size = max(
            layout.size().next_power_of_two(),
            max(layout.align(), size_of::()),
        );
        let class = allocated_size.trailing_zeros() as usize;
        if cur_brk & !(0xFFFFFFFFFFFFFFFF << class) == 0 {
            // 对齐
            let new_brk = brk(Some(cur_brk + allocated_size)).unwrap();
            heap.add_to_heap_rescue(cur_brk, new_brk);
        } else {
            // 没对齐
            let aligned_new_brk = (cur_brk & (0xFFFFFFFFFFFFFFFF <<
class)) + (1usize << class);
            let new_brk = brk(Some(aligned_new_brk +
allocated_size)).unwrap();
            heap.add_to_heap(cur_brk, aligned_new_brk);
            heap.add_to_heap_rescue(aligned_new_brk, new_brk);
        }
    },
}
}
}

```

随后将多出来的空间添加到堆管理器即可:

```

pub unsafe fn add_to_heap(&mut self, mut start: usize, mut end: usize) {
    // avoid unaligned access on some platforms
    start = (start + size_of::() - 1) & (!size_of::() + 1);
    end = end & (!size_of::() + 1);
    assert!(start <= end);
    let mut total = 0;
    let mut current_start = start;
    while current_start + size_of::() <= end {
        let lowbit = current_start & (!current_start + 1);
        let size = min(lowbit, prev_power_of_two(end - current_start));
        total += size;

        self.free_list[size.trailing_zeros() as usize].push(current_start as
*mut usize);
        current_start += size;
    }
    self.total += total;
}

```

四、使用方法

添加为用户库user_lib的一部分：可以添加到user/src/buddy_allocator文件夹下并修改Cargo.toml，也可以直接将代码合并到用户库中。

参考

[伙伴系统-Buddy System](#)

[linux内核内存管理之物理内存管理](#)

[伙伴系统的概述](#)

[动态内存之堆的分配](#)

[Rust 标准库研究系列: alloc](#)

[RUST——互斥锁的使用](#)