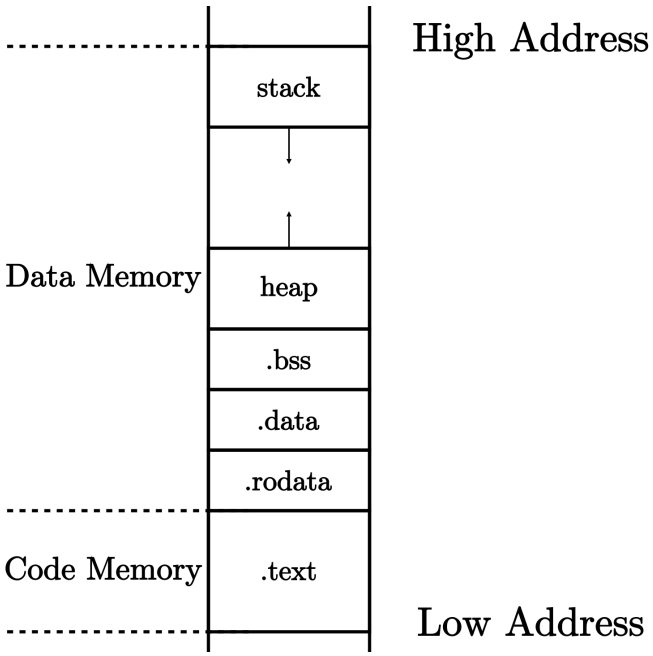
Lab-2

1. 原理：.bss 字段是什么含义？为什么我们要将动态分配的内存（堆）空间放在 .bss 字段？

   BSS（Block Started by Symbol）通常是指用来存放程序中未初始化的全局变量和静态变量的一块内存区域。特点是:可读写的，在程序执行之前BSS段会被操作系统清0。所以，未初始的全局变量在程序执行之前已经成0了。

   对于bss我曾经就踩过坑, 当时要实现一个Logger, 于是把 Logger 的init函数放在了main函数中执行, 不巧放在了clear_bss()函数前面, 因此clear_bss就把全局变量Logger给清除了，导致后面使用log的宏打日志的时候，控制台没有输出，这便是对bss段的认识跟实践有割裂造成的。

   在典型的内存布局中

   bss段与Heap是相邻的，所以我想将堆空间放在 .bss 字段是一种简易的实现方案，也可以专门为堆空间分配一个段

2. 分析：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？

   禁止套娃，本以为会有某种检测机制，然后程序直接报错，没想到会循环。

3. 实验

1. 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？

   算法实现如下

   ```
   pub struct StackedAllocator {
       list: Vec<(usize, usize)>,
   }
   impl Allocator for StackedAllocator {
       fn new(capacity: usize) -> Self {
           Self {
               list: vec![(0, capacity)],
           }
       }

       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
           }
       }

       fn dealloc(&mut self, index: usize) {
           self.list.push((index, index + 1));
       }
   }
   ```

   可以看出alloc和dealloc函数的时间复杂度都是 O(1)，StackedAllocator只有一个属性list， 空间复杂度是 O(n)

2. 二选一：实现基于线段树的物理页面分配算法（不需要考虑合并分配）；或尝试修改 `FrameAllocator`，令其使用未被分配的页面空间（而不是全局变量）来存放页面使用状态。

   我做的是基于线段树的物理页面分配算法，首先要解决的就是数据解构的问题，由于已经实现了堆上内存分配，所以这时候就可以使用Vec了, 我使用Vec来存储线段树节点

   ```
   pub struct SegmentTreeAllocator {
       tree: Vec<Option<Segment>>,
       len: usize
   }
   ```

   然后写出线断树的定义

```
pub struct Segment {
    left: usize,
    right: usize,
    mid: usize,
    remains: usize
}
```

之后就是构建二叉树了，由于这里的线段树是一个完全二叉树，且节点的度要么为0要么为2，所以我们可以先算出树的节点个数 len = 2*n-1，然后递归构建即可，递归边界条件便是当前构建的数组下标小于len

然后是编写查找页面函数，递归边界条件同样是数组下标小于len，之后如果当前线段已经没有剩余页面，就直接返回None，否则就分配一个页面 remains -= 1。再判断当前线段是不是元线段，如果是就直接return。不是的话就先去递归查找left_child是否有剩余页面，再去递归查找right_child。

最后是编写归还页面函数，递归边界条件同样是数组下标小于len，之后如果当前线段已满，就直接返回false，否则就归还一个页面 remains += 1。再判断当前线段是不是元线段且是不是要找的线段，如果是就直接return true。不是的话就二分法，判断要找的页面是否小于该线段的中点，然后在左半边或是右半边递归查找。

4. 挑战实验（选做）

    1. 在 `memory/heap2.rs` 中，提供了一个手动实现堆的方法。它使用 `algorithm::VectorAllocator` 作为其根本分配算法，而我们目前提供了一个非常简单的 bitmap 算法（而且只开了很小的空间）。请在 `algorithm` crate 中利用伙伴算法实现 `VectorAllocator` trait。

    2. 前面说到，堆的实现本身不能完全使用动态内存分配。但有没有可能让堆能够利用动态分配的空间，这样做会带来什么好处？

        我们以一个朴素的分配器算法为例：将每一次内存分配记录用链表存起来。分配器最初必须具有一个节点的静态空间。而每当它仅剩一个节点空间时，都可以用它来为自己分配一块更大的空间。如此，就实现了分配器动态分配自己。
        再考虑到，每次分配 1KB 或 1MB 都需要额外保存一份元信息。如果只用静态分配，就必须按最坏情况（每次都只分配最小单元）来预先留好空间。使用动态分配就可以减少空间浪费。