对ArceOS中Mimalloc内存算法实现无锁多线程支持的尝试

Mimalloc指出：传统的allocator根据内存块大小分类来管理free list，其中一类大小的堆块会以同个单链表连接起来。优点在于访问确实是O(1)，缺点却是堆块会排布在整个堆上。

为改善局部性，Mimalloc采用了一种叫free list sharding的设计。它先将堆分为一系列固定大小的页，之后每页都通过一个free list管理。

在mimalloc中page归属于线程堆, 线程仅从本地堆中分配内存, 但其它线程同样可以释放本线程的堆. 为避免引入锁, mimalloc再为每页增加一个thread free list用于记录由其它线程释放(本线程申请)的内存, 当非本地释放发生时使用atomic_push(&page->thread_free, p)将其放入thread free list中.

mimalloc中内存块的最小单位是mi_block_t, 区别于ptmalloc中malloc_chunk复杂的结构, mi_block_t只有一个指向(同样大小的)下一空闲内存块的指针.
这是因为在mimalloc中所有内存块都是size classed page中分配的, 不需要对空闲内存块做migrate, 因此不用保存本块大小, (物理连续的块的)状态及大小等信息.

当前的Mimalloc算法是一个单线程版本。（即，多线程支持是通过Mutex互斥锁来实现的）

这里做了一些对MiAllocator的数据结构的改造尝试。

```rust
/// mimalloc的heap结构
pub struct MiHeap {
    // page链表
    pub pages: [PagePointer; TOT_QUEUE_NUM],
    // thread id
    pub thread_id: usize,
    // thread delayed free
    pub thread_delayed_free: AtomicBlockPtr,
}
```

```rust
/// mimalloc的一个page控制头
#[derive(Clone)]
pub struct Page {
    // 块大小
    pub block_size: usize,
    // free链表
    pub free_list: BlockPointer,
    // local free list
    pub local_free_list: BlockPointer,
    // thread free list
    pub thread_free_list: AtomicBlockPtr,
    // page开始地址
    pub begin_addr: usize,
    // page结束地址
    pub end_addr: usize,
    // 尚未分配过的地址起点
    pub capacity: usize,
    // page链表中的上一项
    pub prev_page: PagePointer,
    // page链表中的下一项
    pub next_page: PagePointer,
    // 剩余块数
    pub free_blocks_num: usize,
}
```

```rust
pub struct Segment {
    // 把mi_heap藏在第一个段的开头
    pub mi_heap: MiHeap,
    // page种类
    pub page_kind: PageKind,
    // 段的大小
    pub size: usize,
    // 包含多少个page
    pub num_pages: usize,
    // 每个page的头结构
    pub pages: [Page; MAX_PAGE_PER_SEGMEGT],
    /// thread_id
    pub thread_id: usize,
    // padding，使空间对齐到8192
    pub padding: [usize; 431],
    // 接下来就是每个page的实际空间，注意第一个page会小一些
}
```

由于Segment添加了新的字段，所以会导致align_test不通过。解决方法是修改segment的padding。

如果要实现无锁多线程支持，我们需要先仿照rust中Mutex与RefCell的实现，来构造一个新的数据结构。

```
error[E0596]: cannot borrow `self.mi_alloc` as mutable, as it is behind a `&` reference
   --> crates/allocator/tests/global_allocator.rs:164:34
    |
164 |                 if let Ok(ptr) = self.mi_alloc.inner_mut().alloc(size, align_pow2) {
    |                                  ^^^^^^^^^^^^^^ `self` is a `&` reference, so the data it refers to cannot be borrowed as mutable
    |
help: consider changing this to be a mutable reference
    |
119 |     pub unsafe fn alloc(&mut self, layout: Layout) -> AllocResult<usize> {
    |                         ~~~~~~~~~

error[E0596]: cannot borrow `self.mi_alloc` as mutable, as it is behind a `&` reference
   --> crates/allocator/tests/global_allocator.rs:213:17
    |
213 |                 self.mi_alloc.inner_mut().dealloc(pos, size, align_pow2);
    |                 ^^^^^^^^^^^^^^ `self` is a `&` reference, so the data it refers to cannot be borrowed as mutable
    |
help: consider changing this to be a mutable reference
    |
174 |     pub unsafe fn dealloc(&mut self, pos: usize, layout: Layout) {
    |                           ~~~~~~~~~

For more information about this error, try `rustc --explain E0596`.
error: could not compile `allocator` (test "test_main") due to 2 previous errors
```

以上问题的解决，可以使用RefCell，但是问题是一定会造成线程访问资源的冲突（BorrowMutError）。

```rust
type BorrowFlag = AtomicIsize;
/// A byte-granularity memory allocator based on the [mimalloc_allocator] written by rust
pub struct MiAllocator {
    //Atomic...
    borrow: BorrowFlag,
    data: UnsafeCell<MiAllocatorInner>,
}

pub struct MiAllocatorInner {
    inner: Option<Heap>,
}

impl MiAllocator {
    /// Creates a new empty `TLSFAllocator`.
    pub const fn new() -> Self {
        Self {
            borrow: AtomicIsize::new(0),
            data: UnsafeCell::new(MiAllocatorInner::new())
        }
    }

    pub fn inner_mut(&mut self) -> &mut MiAllocatorInner {
        self.data.get_mut()
    }

    pub fn inner(&self) -> &MiAllocatorInner {
            let ptr = self.data.get();
            let reference: &MiAllocatorInner = unsafe {
                &*ptr
            };
            reference
    }
}
```

这是到最后，数据结构的一个初步实现。

总结

如果要继续这项工作，首先要对rust中支持多线程处理的数据结构进行细致地剖析，以便重新构造一个内部为Miallocator本体，而其他字段控制多线程并发操作安全与无锁的功能。