预备

1. Fork ArceOS的工程，clone到本地。工程链接如下

   ```
   git@github.com:arceos-org/arceos.git
   ```

2. 在main分支下，创建并切换到新的分支week1，执行

   ```
   git checkout -b week1
   ```

   后面的实验都在该分支下进行。

练习3

为内存分配器实现新的内存算法*early*，禁用其它算法。early分配算法以apps/memtest为测试应用。

首先，为memtest增加feature = "paging"，便于查看对Page的分配情况。修改apps/memtest/Cargo.toml，对比变动如下

```
-axstd = { path = "../../ulib/axstd", features = ["alloc"], optional = true }
+axstd = { path = "../../ulib/axstd", features = ["alloc", "paging"], optional =
true }
```

然后，修改modules/axruntime/src/lib.rs增加下面几行，对比变动：

```
+    {
+        let ga = axalloc::global_allocator();
+        info!("Used pages {} / Used bytes {}", ga.used_pages(),
ga.used_bytes());
+    }
     unsafe { main() };
```

这几行的目的，在调用应用的main()之前，输出对页和字节的用量。

最后，用下面的内容替换modules/axalloc/src/lib.rs，禁用其它算法，只使用我们将要增加的early算法。

```
#![no_std]

#[macro_use]
extern crate log;
extern crate alloc;

mod page;

use allocator::{AllocResult, EarlyAllocator};
use core::alloc::{GlobalAlloc, Layout};
use core::ptr::NonNull;
use spinlock::SpinNoIrq;

const PAGE_SIZE: usize = 0x1000;
```

```rust
pub use page::GlobalPage;

/// The global allocator used by ArceOS.
pub struct GlobalAllocator {
    inner: SpinNoIrq<EarlyAllocator<PAGE_SIZE>>,
}

impl GlobalAllocator {
    /// Creates an empty [`GlobalAllocator`].
    pub const fn new() -> Self {
        Self {
            inner: SpinNoIrq::new(EarlyAllocator::new()),
        }
    }

    /// Returns the name of the allocator.
    pub const fn name(&self) -> &'static str {
        "early"
    }

    /// Initializes the allocator with the given region.
    pub fn init(&self, start_vaddr: usize, size: usize) {
        self.inner.lock().init(start_vaddr, size);
    }

    /// Add the given region to the allocator.
    pub fn add_memory(&self, _start_vaddr: usize, _size: usize) -> AllocResult {
        unimplemented!()
    }

    /// Allocate arbitrary number of bytes. Returns the left bound of the
    /// allocated region.
    pub fn alloc(&self, layout: Layout) -> AllocResult<NonNull<u8>> {
        self.inner.lock().alloc(layout)
    }

    /// Gives back the allocated region to the byte allocator.
    pub fn dealloc(&self, pos: NonNull<u8>, layout: Layout) {
        self.inner.lock().dealloc(pos, layout)
    }

    /// Allocates contiguous pages.
    pub fn alloc_pages(&self, num_pages: usize, align_pow2: usize) ->
AllocResult<usize> {
        self.inner.lock().alloc_pages(num_pages, align_pow2)
    }

    /// Gives back the allocated pages starts from `pos` to the page allocator.
    /// [`alloc_pages`]: GlobalAllocator::alloc_pages
    pub fn dealloc_pages(&self, pos: usize, num_pages: usize) {
        self.inner.lock().dealloc_pages(pos, num_pages)
    }

    /// Returns the number of allocated bytes in the byte allocator.
    pub fn used_bytes(&self) -> usize {
```

```
        self.inner.lock().used_bytes()
    }

    /// Returns the number of available bytes in the byte allocator.
    pub fn available_bytes(&self) -> usize {
        self.inner.lock().available_bytes()
    }

    /// Returns the number of allocated pages in the page allocator.
    pub fn used_pages(&self) -> usize {
        self.inner.lock().used_pages()
    }

    /// Returns the number of available pages in the page allocator.
    pub fn available_pages(&self) -> usize {
        self.inner.lock().available_pages()
    }
}

unsafe impl GlobalAlloc for GlobalAllocator {
    unsafe fn alloc(&self, layout: Layout) -> *mut u8 {
        if let Ok(ptr) = GlobalAllocator::alloc(self, layout) {
            ptr.as_ptr()
        } else {
            alloc::alloc::handle_alloc_error(layout)
        }
    }

    unsafe fn dealloc(&self, ptr: *mut u8, layout: Layout) {
        GlobalAllocator::dealloc(self, NonNull::new(ptr).expect("dealloc null
ptr"), layout)
    }
}

#[cfg_attr(all(target_os = "none", not(test)), global_allocator)]
static GLOBAL_ALLOCATOR: GlobalAllocator = GlobalAllocator::new();

/// Returns the reference to the global allocator.
pub fn global_allocator() -> &'static GlobalAllocator {
    &GLOBAL_ALLOCATOR
}

/// Initializes the global allocator with the given memory region.
pub fn global_init(start_vaddr: usize, size: usize) {
    debug!(
        "initialize global allocator at: [{:#x}, {:#x})",
        start_vaddr,
        start_vaddr + size
    );
    GLOBAL_ALLOCATOR.init(start_vaddr, size);
}

/// Add the given memory region to the global allocator.
pub fn global_add_memory(_start_vaddr: usize, _size: usize) -> AllocResult {
    unimplemented!()
}
```

上面已经给出了axalloc组件的主框架和对crates/allocator/early算法组件的调用，同时去除了其它算法，方便大家实验。

现在，如果执行`make ARCH=riscv64 A=apps/memtest LOG=debug run`，当然会出错。

我们的实验目标就是实现early算法，算法描述见第一周第2次课程的ppt。

要求:

按照课程ppt的算法，从前往后字节分配，从后往前页分配。

提示:

1. modules/axalloc组件要去调用实现在crates/allocator中的算法。现在其它算法都无效了，我们只要在crates/allocator/src/lib.rs中增加如下几行，引入early模块，代表算法的类型为EarlyAllocator。

   ```
   mod early;
   pub use early::EarlyAllocator;
   ```

   新建crates/allocator/src/early.rs，后面主要的逻辑实现都在这个模块中。

2. 可以参考禁用掉的buddy/slab等字节分配算法、bitmap页分配算法，来实现early算法。正常来说，EarlyAllocator需要实现像BaseAllocator、ByteAllocator、PageAllocator这三个trait来为axalloc提供调用服务，但为了实验方便，也可以不实现，直接为EarlyAllocator关联对应的方法即可。

预期输出:

执行`make ARCH=riscv64 A=apps/memtest LOG=debug run`，输出

```
[  0.090478 0 axruntime:211] Initialize global memory allocator...
[  0.092667 0 axruntime:212]   use early allocator.
[  0.094352 0 axalloc:113] initialize global allocator at: [0xffffffc080250000, 0xffffffc088000000)
[  0.098117 0 axruntime:145] Initialize kernel page table...
[  0.100399 0 axruntime:149] Initialize platform devices...
[  0.101792 0 axruntime:184] Primary CPU 0 init OK.
[  0.103955 0 axruntime:194] Used pages 6 / Used bytes 104
Running memory tests...
test_vec() OK!
test_hashmap_map() OK!
Memory tests run OK!
[  1.345623 0 axruntime:202] main task exited: exit_code=0
[  1.346734 0 axhal::platform::riscv64_qemu_virt::misc:3] Shutting down...
```

注意：这个已分配的页面数正常就是6，而显示的字节数可能与你执行时的情况不同，但不会是0。

练习4

解析dtb(FDT的二进制格式)，打印物理内存范围和所有的virtio_mmio范围。以apps/memtest为测试应用。

当ArceOS启动时，上一级SBI向我们传递了dtb的指针，一直会被传递到axruntime，我们就在axruntime中执行解析并打印。

首先，在/modules/axruntime/src/lib.rs的rust_main函数中，插入如下代码以显示获取到的信息，diff内容如下：

```
--- a/modules/axruntime/src/lib.rs
```

```
+++ b/modules/axruntime/src/lib.rs
@@ -140,6 +140,20 @@ pub extern "C" fn rust_main(cpu_id: usize, dtb: usize) -> !
{
     #[cfg(feature = "alloc")]
     init_allocator();

+    // Parse fdt for early memory info
+    let dtb_info = match parse_dtb(dtb.into()) {
+        Ok(info) => info,
+        Err(err) => panic!("Bad dtb {:?}", err),
+    };
+
+    info!("DTB info: ==================================");
+    info!("Memory: {:#x}, size: {:#x}", dtb_info.memory_addr,
dtb_info.memory_size);
+    info!("Virtio_mmio[{}]:", dtb_info.mmio_regions.len());
+    for r in dtb_info.mmio_regions {
+        info!("\t{:#x}, size: {:#x}", r.0, r.1);
+    }
+    info!("============================================");
+
     #[cfg(feature = "paging")]
     {
         info!("Initialize kernel page table...");
@@ -297,3 +311,74 @@ fn init_tls() {
     unsafe { axhal::arch::write_thread_pointer(main_tls.tls_ptr() as usize) };
     core::mem::forget(main_tls);
 }
```

关于parse_dtb和它的返回值类型，类似如下定义，你对它们的具体定义有所可以不同：

```
extern crate alloc;

use core::str;
use alloc::string::String;
use alloc::vec::Vec;
use axdtb::util::SliceRead;

// 参考类型定义
struct DtbInfo {
    memory_addr: usize,
    memory_size: usize,
    mmio_regions: Vec<(usize, usize)>,
}

// 参考函数原型
fn parse_dtb(dtb_pa: usize) -> Result<DtbInfo> {
    // 这里就是对axdtb组件的调用，传入dtb指针，解析后输出结果。这个函数和axdtb留给大家实现
}
```

然后就是本实验的中心任务：modules目录中增加一个axdtb的组件，它负责解析dtb，获取目标信息。

提示：

解析DTB(FDT的二进制格式)有点繁琐，但是网络上存在很多现成的crate实现，例如crate.io中。

但是注意：有可能这些crate不直接符合我们的需要，需要改造。主要是很多实现都是以文件路径为输入，加载文件到内存后，再解析。我们只需要进行解析的那部分。

预期输出：

执行`make ARCH=riscv64 A=apps/memtest LOG=info run`

```
[  0.104036 0 axruntime:225] Initialize global memory allocator...
[  0.107082 0 axruntime:226]   use early allocator.
[  0.113712 0 axruntime:149] DTB info: ==================================
[  0.115284 0 axruntime:150] Memory: 0x80000000, size: 0x8000000
[  0.117057 0 axruntime:151] Virtio_mmio[8]:
[  0.119386 0 axruntime:151]    0x10008000, size: 0x1000
[  0.121829 0 axruntime:151]    0x10007000, size: 0x1000
[  0.124366 0 axruntime:151]    0x10006000, size: 0x1000
[  0.126094 0 axruntime:151]    0x10005000, size: 0x1000
[  0.128793 0 axruntime:151]    0x10004000, size: 0x1000
[  0.131165 0 axruntime:151]    0x10003000, size: 0x1000
[  0.133212 0 axruntime:151]    0x10002000, size: 0x1000
[  0.136307 0 axruntime:151]    0x10001000, size: 0x1000
[  0.138569 0 axruntime:151] ==========================================
[  0.141656 0 axruntime:159] Initialize kernel page table...
[  0.145231 0 axruntime:163] Initialize platform devices...
[  0.147888 0 axruntime:199] Primary CPU 0 init OK.
[  0.150456 0 axruntime:208] Used pages 6 / Used bytes 104
```

可以看到输出了解析dtb后的物理内存信息，和virtio_mmio区域信息。

发送结果

能够完成练习3, 4的同学，请先把运行结果截图发项目1的群，然后把你完成工作的github工程链接发到下面的邮箱

```
sun_ye@masscloud.com
```