



中山大學
SUN YAT-SEN UNIVERSITY

操作系统实验报告

实验七：更好的内存管理

姓 名： 刘家祥
学 号： 23336152
教学班号： 计科二班
专 业： 计算机科学与技术
院 系： 计算机学院

2024~2025 学年第二学期

更好的内存管理

一. 合并代码

1. 我注意到新增代码中的 `pkg/kernel/src/proc/vm/mod.rs` 相比于原来的代码, 除了“添加了堆内存、ELF 文件映射的初始化和清理函数”之外还有不少删改, 我首先基于新代码尝试完成实验。如果难以进行, 考虑在原代码基础上加入新增部分重新实验。
2. 在 `pkg/lib` 中引入了 `chrono`: 一个非常流行的、用来处理日期和时间的库, 同时引入了 `linked_list_allocator`: 一个基于链表实现的堆内存分配器。
3. 新增 `allocator` 文件夹: 将不同种类的内存分配器实现进行模块化管理。
4. 关于 `processVm` 的角色:
 - 4.1. `ProcessVm` 就像一个进程的内存管家, 记录这个进程“认为”自己拥有的进程是什么样的, 包括内存布局和页表。
 - 4.2. 在操作系统中, 有很多事件 (即下级函数) 需要修改进程的内存, 比如:

- `mmap` 系统调用：映射文件到内存。
- `brk / sbrk` 系统调用：扩大或缩小内存空间。
- 处理缺页异常：当程序访问一个还未分配物理内存的页面时。

这些不同的操作（即下级函数）都需要文档提到的两个核心工具：

- `mapper`：一个能修改页表的对象。
- `allocator`：一个物理页帧分配器。

4.3. 如果让每个下级函数都自己去获取这两个工具，就会很混乱。每个函数都需要重复写类似的代码来加锁、获取资源，很容易出错，也难以维护。

5. 解决方案：统一调用时机

意思不是说在同一时刻调用，而是“在同一个环节、用同一种方式去调用”。`ProcessVm` 规定了一个流程：任何下级函数（比如处理 `mmap` 的函数）在开始工作前，不能自己去拿工具，必须先通过 `ProcessVm` 这个管家，由 `ProcessVm` 统一负责把 `mapper` 和 `frame_allocator` 这两个工具准备好，然后再把工具传递下级函数使用。

二. 帧分配器的内存回收

1. 首先简单地使用一个 `Vec<PhysFrame>` 作为已经回收的帧的集合

在 `init` 函数里面初始化 `recycled` 字段为一个空的 `Vec`

```
pub unsafe fn init(memory_map: &MemoryMap, size: usize) → Self {
    BootInfoFrameAllocator {
        size,
        frames: create_frame_iter(memory_map),
        used: 0,
        recycled: Vec::new(),
    }
}
```

2. 实现 `allocator_frame` 函数，让它先尝试从 `recycled` 里分配，如果为空再从 `frame` 迭代器里分配。

```

unsafe impl FrameAllocator<Size4KiB> for BootInfoFrameAllocator {
    fn allocate_frame(&mut self) → Option<PhysFrame> {
        if let Some(frame) = self.recycled.pop() {
            return Some(frame);
        }
        match self.frames.next() {
            Some(frame) ⇒ {
                self.used += 1;
                Some(frame)
            }
            None ⇒ None,
        }
    }
}

```

3. 实现 `deallocate_frame` 函数，让它把回收的帧放入 `recycled` 中

```

impl FrameDeallocator<Size4KiB> for BootInfoFrameAllocator {
    unsafe fn deallocate_frame(&mut self, frame: PhysFrame) {
        self.recycled.push(frame)
    }
}

```

三. 用户程序的内存统计

1. 在了解完 Linux 的进程内存布局、管理和相关信息查看方式之后，我们着手实现用户程序的内存统计。首先是重构 `elf` 模块的 `load_elf` 函数和 `load_segment` 函数，代码在指导中均已给出，值得注意的是，放弃了 `file_buf` 的使用，改为使用 `elf.input.as_ptr()`：

```

- let data = unsafe { file_buf.add(file_offset as usize) };
+ let data = unsafe { elf.input.as_ptr().add(file_offset as usize) };

```

2. 获取用户程序 `ELF` 文件映射的内存占用的最好方法是在加载 `ELF` 文件时记录内存占用，这需要对 `elf` 模块中的 `load_elf` 函数进行修改，这部分代码指导书中已经给出进行简单增补替换即可。
3. 接下来我们完善一下 `load_elf_code` 函数，实现在加载 `elf` 文件时记录内存占用。

```
fn load_elf_code(&mut self, elf: &ElfFile, mapper: MapperRef, alloc:
FrameAllocatorRef) {
    // FIXME: make the `load_elf` function return the code pages
    self.code =
        elf::load_elf(elf, *PHYSICAL_OFFSET.get().unwrap(), mapper, alloc,
true).unwrap();

    // FIXME: calculate code usage
    self.code_usage = self
        .code
        .iter()
        .map(|range| range.count() as u64 * Page::<Size4KiB>::SIZE)
        .sum();
}
```

4. 到这里我们已经实现了内存占用的记录与统计，接下来我们只需要添加打印内存占用的功能即可。这部分代码也已经给出，进行简单增补替换即可。

注意： 由于目前尚未实现 `clean_up` 函数和 `unmap_range` 函数，编译时会产生报错，所以暂时不测试演示阶段性成果

四. 用户程序的内存释放

1. 首先为 `PageTableContext` 添加一个 `using_count` 方法，用于获取当前页表被引用的次数，这部分其实已经实现了。
2. 接下来为 `stack` 实现 `clean_up` 函数，根据实验指导：由于栈是一块连续的内存区域，且进程间不共享栈区，因此在进程退出时直接释放栈区的页面即可。

```
pub fn clean_up(
    &mut self,
    mapper: MapperRef,
    dealloc: FrameAllocatorRef,
) → Result<(), UnmapError> {
    if self.usage == 0 {
        warn!("Stack is empty, no need to clean up.");
        return Ok(());
    }

    for page in self.range.clone() {
        let (frame, flusher) = mapper.unmap(page)?;
        unsafe { dealloc.deallocate_frame(frame) };
        flusher.flush();
    }

    self.usage = 0;
    self.range = Page::range(STACK_INIT_TOP_PAGE, STACK_INIT_TOP_PAGE);

    Ok(())
}
```

3. 接下来我们关注 `ProcessVm` 的相关实现，这里不知道为什么在新增代码中已经含有 `clean_up` 函数的情况下，指导书中又给出了一个 `clean_up` 函数，我们进行合并：

```
pub(super) fn clean_up(&mut self) → Result<(), UnmapError> {

    let mapper = &mut self.page_table.mapper();
    let dealloc = &mut *get_frame_alloc_for_sure();
```

```

// statistics for logging and debugging
// NOTE: you may need to implement `frames_recycled` by yourself
let start_count = dealloc.frames_recycled();

// 1. 释放栈区: 调用 Stack 的 clean_up 函数
self.stack.clean_up(mapper, dealloc)?;

// 2. 如果当前页表被引用次数为 1, 则进行共享内存的释放, 否则跳过至第 7 步
if self.page_table.using_count() == 1 {
    // 3. 释放堆区: 调用 Heap 的 clean_up 函数
    self.heap.clean_up(mapper, dealloc)?;

    // 4. 释放 ELF 文件映射的内存区域: 根据记录的 code 页面范围数组, 依次调用
elf::unmap_range 函数
    for page_range in self.code.iter() {
        elf::unmap_range(*page_range, mapper, dealloc, true)?;
    }

    // 5. 清理页表: 调用 mapper 的 clean_up 函数, 这将清空全部无页面映射的一至三级
    // 6. 清理四级页表: 直接回收 PageTableContext 的 reg.addr 所指向的页面
unsafe {
    // free P1-P3
    mapper.clean_up(dealloc);

    // free P4
    dealloc.deallocate_frame(self.page_table.reg.addr);
}
}

// 7. 统计内存回收情况, 并打印调试信息
let end_count = dealloc.frames_recycled();

debug!(
    "Recycled {}({:.3} MiB) frames, {}({:.3} MiB) frames in total.",
    end_count - start_count,
    ((end_count - start_count) * 4) as f32 / 1024.0,
    end_count,
    (end_count * 4) as f32 / 1024.0
);

Ok(())
}

```

在实现完 Drop 之后不要忘记更新 `process` 中的 `kill` 函数, 使用 `take` 来释放进程的内存。

```
// consume the Option<ProcessVm> and drop it
self.proc_vm.take();
```

注意： Drop trait 是 Rust 中用于定义当一个值离开其作用域时（即被销毁时）需要执行的代码。这通常用于释放资源，例如内存、文件句柄、网络连接等。

4. 阶段性成果

到这里我们完成了用户程序的内存统计与释放，以 `fact` 阶乘递归程序为例：

```
欢迎使用YSOS Shell!
输入 help 获取帮助信息
ysos> ps
当前运行的进程列表:
  PID | PPID | Process Name | Ticks | Memory | Status
  # 1 | # 0 | kernel       | 23526 | 2.0 MiB | Ready
  # 2 | # 1 | shell        | 23458 | 40.0 KiB | Running
Memory : 1.08 MiB / 44.37 MiB ( 2.43%)
Queue  : [1]
CPUs    : [0: 2]
ysos> run APP/FACTOR~1
正在运行程序: APP/FACTOR~1
[INFO ] Loading ELF from path 'APP/FACTOR~1' as process 'FACTOR~1', entry point: 0x1111000013b0
[INFO ] ELF header bytes: [7f, 45, 4c, 46, 02, 01, 01, 00, 00, 00, 00, 00, 00, 00]
[INFO ] Loading & mapping segment: virtual_addr=0x111100000000, file_size=0x6fc, mem_size=0x6fc, flags=Flags(4)
[INFO ] Loading & mapping segment: virtual_addr=0x111100001000, file_size=0x2d3f, mem_size=0x2d3f, flags=Flags(5)
[INFO ] Loading & mapping segment: virtual_addr=0x111100004000, file_size=0x1020, mem_size=0x1020, flags=Flags(6)
[INFO ] Loading & mapping segment: virtual_addr=0x111100006000, file_size=0x0, mem_size=0x1, flags=Flags(6)
进程ID: 3
Input n: 999999
Before calculating factorial:
  PID | PPID | Process Name | Ticks | Memory | Status
  # 1 | # 0 | kernel       | 36002 | 2.0 MiB | Ready
  # 2 | # 1 | shell        | 35934 | 40.0 KiB | Ready
  # 3 | # 2 | factor~1     | 2804 | 32.0 KiB | Running
Memory : 1.14 MiB / 44.37 MiB ( 2.56%)
Queue  : [2, 1]
CPUs    : [0: 3]
```

```
After calculating factorial:
  PID | PPID | Process Name | Ticks | Memory | Status
  # 1 | # 0 | kernel       | 36134 | 2.0 MiB | Ready
  # 2 | # 1 | shell        | 36066 | 40.0 KiB | Ready
  # 3 | # 2 | factor~1     | 2936 | 15.3 MiB | Running
Memory : 16.42 MiB / 44.37 MiB (37.01%)
Queue  : [2, 1]
CPUs    : [0: 3]
The factorial of 999999 under modulo 1000000007 is 128233642.
程序 'APP/FACTOR~1' 已退出, 返回值: 0
ysos> PS
未知命令: 'PS'
输入 help 获取可用命令列表
ysos> ps
当前运行的进程列表:
  PID | PPID | Process Name | Ticks | Memory | Status
  # 1 | # 0 | kernel       | 43079 | 2.0 MiB | Ready
  # 2 | # 1 | shell        | 43011 | 40.0 KiB | Running
Memory : 1.08 MiB / 44.37 MiB ( 2.43%)
Queue  : [1]
CPUs    : [0: 2]
ysos>
```

可以看到 `fact` 程序的内存占用依次为 0、15.3MB、0。用户程序的内存统计和内存释放都已成功实现。即：我的页面被成功被成功回收了！

五. 内核的内存统计

1. 首先在 `pkg/boot/src/lib.rs` 中，定义一个 `KernelPages` 类型，用于传递内核的内存占用信息，并将其添加到 `BootInfo` 结构体的定义中：

```
pub type KernelPages = ArrayVec<PageRangeInclusive, 8>;

pub struct BootInfo {
    // ...

    // Kernel pages
    pub kernel_pages: KernelPages,
}
```

2. 在 `pkg/boot/src/main.rs` 中，将 `load_elf` 函数返回的内存占用信息传递至 `BootInfo` 结构体中：

```
// FIXME: load and map the kernel elf file
let kernel_pages_vec = load_elf(
    &elf,
    config.physical_memory_offset,
    &mut page_table,
    &mut UEFIFrameAllocator,
    false,
).expect("Failed to load kernel ELF");

// Convert Vec<PageRangeInclusive> to KernelPages (ArrayVec)
let kernel_pages: KernelPages = kernel_pages_vec.into_iter().collect();
```

3. 成功加载映射信息后，将其作为 `ProcessManager` 的初始化参数，用于构建 `kernel` 进程

```
- let proc_vm = ProcessVm::new(PageTableContext::new()).init_kernel_vm();
+ let proc_vm =
    ProcessVm::new(PageTableContext::new()).init_kernel_vm(&boot_info.kernel_pages);
```

4. 为 `ProcessVm` 添加 `init_kernel_vm` 函数，用于初始化内核的内存布局

```
pub fn init_kernel_vm(mut self, pages: &KernelPages) → Self {
    // FIXME: record kernel code usage
    self.code = pages.iter().cloned().collect();
    self.code_usage = pages
        .iter()
        .map(|range| range.count() as u64 * Page::<Size4KiB>::SIZE)
        .sum();

    self.stack = Stack::kstack();

    // ignore heap for kernel process as we don't manage it

    self
}
```

5. 不要忘记 `manager.rs` 中也调用了旧的 `init_kernel_vm` 函数，为那个调用提供一个空的 `KernelPages`，因为那是在创建新的内核进程时使用的

```
- let vm = ProcessVm::new(PageTableContext::new()).init_kernel_vm();
+ let empty_kernel_pages = boot::KernelPages::new();
+ let vm =
ProcessVm::new(PageTableContext::new()).init_kernel_vm(&empty_kernel_pages);
```

6. 阶段性成果

```
欢迎使用YSOS Shell!
输入 help 获取帮助信息
ysos> ps
当前运行的进程列表:
  PID | PPID | Process Name | Ticks | Memory | Status
  # 1 | # 0 | kernel      | 6386 | 10.4 MiB | Ready
  # 2 | # 1 | shell       | 6268 | 40.0 KiB | Running
Memory : 10.43 MiB / 44.36 MiB (23.50%)
Queue  : [1]
CPUs    : [0: 2]
ysos> █
```

- 可以看到内核的内存占用 `10.4MB` 已经成功统计并打印了出来！与之前显示的 `2.0MB` 大不相同。

- 之前的不完整实现只统计了内核栈：之前的 `init_kernel_vm()` 函数只初始化了内核栈（`Stack::kstack()`），所以显示的 2.0 MiB 主要是内核栈的大小
- 缺少内核代码段统计：没有统计内核 ELF 文件加载时占用的代码段、数据段等内存
- 缺少准确的页面信息：没有从 `bootloader` 传递实际的内核页面使用信息
- 现在我们实现了完整的内核内存统计：现在显示的 10.4 MiB 包括：
 - 内核代码段（从 ELF 文件加载的所有段）
 - 内核数据段
 - 内核栈
 - 其他内核占用的页面

六. 内核栈的自动增长

1. 在 `pkg/boot/src/main.rs` 中加入判断逻辑，来确认了启动加载器会根据 `kernel_stack_auto_grow` 配置项来决定初始化时分配的栈大小

```
// FIXME: map kernel stack

let (stack_start, stack_size) = if config.kernel_stack_auto_grow > 0 {
    let init_size = config.kernel_stack_auto_grow;
    let bottom_offset = (config.kernel_stack_size - init_size) * 0x1000;
    let init_bottom = config.kernel_stack_address + bottom_offset;
    (init_bottom, init_size)
} else {
    (config.kernel_stack_address, config.kernel_stack_size)
};

let _stack_pages = map_range(
    stack_start,
    stack_size,
    &mut page_table,
    &mut UEFIFrameAllocator,
).expect("Failed to map kernel stack");
```

2. 更新内核栈的常量定义，以支持自动增长

```
- pub const KSTACK_DEF_PAGE: u64 = 512; // 设置为512页 (2MB)
+ pub const KSTACK_DEF_PAGE: u64 = 8;
```

3. 修改配置文件, 开启内核栈自动增长, 并设置合适的初始值和最大值

```
kernel_stack_size = 1048576
kernel_stack_auto_grow = 8
```

4. 在缺页中断处理函数中添加了日志, 以便在内核发生缺页时进行观察。这是实现“自动增长”的核心。

我们在缺页中断的总处理函数 `handle_page_fault` 中, 加入了一个判断: `if current.pid() == KERNEL_PID`。当内核执行的程序 (比如我们的测试函数) 用完了最初的 8 页栈空间, 试图访问第 9 页时, CPU 会发现这个地址是“空”的 (未映射), 于是立即触发一个缺页中断。我们的新代码会捕获这个中断, 判断出是内核自己出了问题, 然后打印出我们看到的日志 `Page fault on kernel at ...`。随后, 它会调用底层的内存管理函数, 为这个出错的地址分配一个新的物理内存页, 并建立映射。当中断返回后, CPU 再次尝试访问刚才出错的地址, 这次就成功了! 这就实现了栈的“自动增长”。

5. 在 `pkg/kernel/src/lib.rs` 中添加测试代码, 添加了 `grow_stack` 函数, 它试图在栈上创建一个远大于初始 8 页 (32KB) 的数组。这个函数的作用就是故意让内核栈溢出, 主动去触发上面说的缺页中断。当看到日志中先打印 `Test stack grow.`, 然后打印 `Page fault on kernel...`, 最后还能成功打印 `Stack grow test done.` 时, 就证明我们的自动增长机制成功地处理了这次“危机”, 并且没有让内核崩溃。

```

pub fn init(boot_info: &'static BootInfo) {
    // ...

    info!("Test stack grow.");

    grow_stack();

    info!("Stack grow test done.");
}

#[inline(never)]
#[unsafe(no_mangle)]
pub fn grow_stack() {
    const STACK_SIZE: usize = 1024 * 4;
    const STEP: usize = 64;

    let mut array = [0u64; STACK_SIZE];
    info!("Stack: {:?}", array.as_ptr());

    // test write
    for i in (0..STACK_SIZE).step_by(STEP) {
        array[i] = i as u64;
    }

    // test read
    for i in (0..STACK_SIZE).step_by(STEP) {
        assert_eq!(array[i], i as u64);
    }
}

```

6. 阶段性成果:从下图中的初始化日志中我们可以看
到 `Test stack grow.` 和 `Stack grow test done` , 另外, 在
`shell` 中使用 `ps` 指令时我们看到 `kernel` 占用的 `Memory` 不
再是 `10.4MB` 而是 `8.4MB` , 因为我们之前是为内核栈分配了
`2.0MB` 的栈空间, 现在改造成一个初始很小、按需自动增长的
动态空间, 从而节约内存。这样就说明我们的内核栈自动增长
机制成功实现! 同时, 经过对 `kernel_stack_auto_grow` 的值进
行修改测试, 发现 ≥ 8 的时候内核才能正常启动, 最小值为8。

```
PROBLEMS 9 OUTPUT DEBUG CONSOLE TERMINAL PORTS 9 AUGMENT NEXT EDIT python3 - 0x07 + - [ ] [ ] ...

[INFO ] CHS End: C1023/H15/S63
[INFO ] Opening disk device...
[INFO ] Identifying drive 0
[INFO ] Drive QEMU HARDDISK QM00001 (504 MiB) opened
[INFO ] Mounting filesystem...
[INFO ] Initialized Filesystem.
[INFO ] Test stack grow.
[INFO ] Page fault on kernel at 0xffffffff01ffff7b80
[INFO ] Stack: 0xffffffff01ffff7c28
[INFO ] Stack grow test done.
[INFO ] YatsenUS initialized.
[+] App list: factorial, hello, shell, counter, dinner, mq, fork_test
[INFO ] Loading ELF from bootloader app 'shell', entry point: 0x111100002680
[INFO ] ELF header bytes: [7f, 45, 4c, 46, 02, 01, 01, 00, 00, 00, 00, 00, 00, 00, 00, 00]
[INFO ] Loading & mapping segment: virtual_addr=0x111100000000, file_size=0xd4c, mem_size=0xd4c, flags=Flags(4)
[INFO ] Loading & mapping segment: virtual_addr=0x111100001000, file_size=0x4345, mem_size=0x4345, flags=Flags(5)
[INFO ] Loading & mapping segment: virtual_addr=0x111100006000, file_size=0x1038, mem_size=0x1038, flags=Flags(6)
[INFO ] Loading & mapping segment: virtual_addr=0x111100008000, file_size=0x0, mem_size=0x1, flags=Flags(6)
欢迎使用YSOS Shell!
输入 help 获取帮助信息
ysos> ps
当前运行的进程列表:
  PID | PPID | Process Name | Ticks | Memory | Status
  # 1 | # 0 | kernel       | 69429 | 8.4 MiB | Ready
  # 2 | # 1 | shell        | 69348 | 40.0 KiB | Running
Memory : 8.46 MiB / 46.32 MiB (18.27%)
Queue : [1]
CPUs : [0: 2]
ysos> ps
当前运行的进程列表:
  PID | PPID | Process Name | Ticks | Memory | Status
  # 1 | # 0 | kernel       | 112680 | 8.4 MiB | Ready
  # 2 | # 1 | shell        | 112599 | 40.0 KiB | Running
Memory : 8.46 MiB / 46.32 MiB (18.27%)
Queue : [1]
CPUs : [0: 2]
ysos> run APP/FACTOR~1
正在运行程序: APP/FACTOR~1
[INFO ] Loading ELF from path 'APP/FACTOR~1' as process 'FACTOR~1', entry point: 0x1111000013b0
```

七. 用户态堆

1. 实现 `brk` 系统调用，首先在 `pkg/syscall/src/lib.rs` 中添加 `Brk = 12` 到 `Syscall` 枚举中。
2. 将给出的用户态接口实现代码添加到 `pkg/lib/src/syscall.rs` 中。
3. 在系统调用分发器中添加 `brk` 的处理函数

```
Syscall::Brk => {
    context.set_rax(sys_brk(&args));
},
```

4. 在 `pkg/kernel/src/proc/vm/heap.rs` 中为 `Heap` 结构体实现 `brk` 函数

```
pub fn brk(
    &self,
    new_end: Option<VirtAddr>,
    mapper: MapperRef,
    alloc: FrameAllocatorRef,
) → Option<VirtAddr> {
    use x86_64::structures::paging::{PageTableFlags, Page, Size4KiB};
    use core::sync::atomic::Ordering;

    // 如果参数为 None, 返回当前的堆区结束地址
    if new_end.is_none() {
        return Some(VirtAddr::new(self.end.load(Ordering::SeqCst)));
    }

    let target_addr = new_end.unwrap();

    // 检查目标地址是否合法, 即是否在 [HEAP_START, HEAP_END] 区间内
    if target_addr.as_u64() < HEAP_START || target_addr.as_u64() > HEAP_END {
        return None;
    }

    let current_end = self.end.load(Ordering::SeqCst);
    let target_end = target_addr.as_u64();

    // 将目标地址向上对齐到页边界
    let target_end_aligned = (target_end + crate::memory::PAGE_SIZE - 1) & !
        (crate::memory::PAGE_SIZE - 1);

    // 计算当前结束地址和目标地址的差异
    let current_end_aligned = (current_end + crate::memory::PAGE_SIZE - 1)
        & !(crate::memory::PAGE_SIZE - 1);

    // 打印堆区差异用于调试
    log::debug!("brk: current_end={:#x}, target_end={:#x},
        current_aligned={:#x}, target_aligned={:#x}",
        current_end, target_end, current_end_aligned,
        target_end_aligned);

    // 设置页面标志: 存在、可写、用户可访问
    let flags = PageTableFlags::PRESENT | PageTableFlags::WRITABLE |
        PageTableFlags::USER_ACCESSIBLE;
```

```

    if target_end == self.base.as_u64() {
        // 用户希望释放整个堆区：目标地址为 base，释放所有页面，end 重置为 base
        if current_end > self.base.as_u64() {
            let start_page = Page::containing_address(self.base);
            let end_page =
                Page::containing_address(VirtAddr::new(current_end_aligned - 1));

            for page in Page::range_inclusive(start_page, end_page) {
                if let Ok((frame, flusher)) = mapper.unmap(page) {
                    unsafe {
                        alloc.deallocate_frame(frame);
                    }
                    flusher.flush();
                }
            }
        }

        // 重置 end 为 base
        self.end.store(self.base.as_u64(), Ordering::SeqCst);
        return Some(self.base);
    } else if target_end_aligned < current_end_aligned {
        // 用户希望缩小堆区：目标地址比当前 end 小，释放多余的页面
        let start_page =
            Page::containing_address(VirtAddr::new(target_end_aligned));
        let end_page =
            Page::containing_address(VirtAddr::new(current_end_aligned - 1));

        for page in Page::range_inclusive(start_page, end_page) {
            if let Ok((frame, flusher)) = mapper.unmap(page) {
                unsafe {
                    alloc.deallocate_frame(frame);
                }
                flusher.flush();
            }
        }
    } else if target_end_aligned > current_end_aligned {
        // 用户希望扩大堆区：目标地址比当前 end 大，分配新的页面
        let start_page =
            Page::containing_address(VirtAddr::new(current_end_aligned));
        let end_page =
            Page::containing_address(VirtAddr::new(target_end_aligned - 1));
    }

```



```

    for page in Page::range_inclusive(start_page, end_page) {
        let frame = match alloc.allocate_frame() {
            Some(frame) => frame,
            None => return None, // 分配失败
        };

        unsafe {
            match mapper.map_to(page, frame, flags, alloc) {
                Ok(flusher) => flusher.flush(),
                Err(_) => {
                    // 映射失败, 释放已分配的帧
                    alloc.deallocate_frame(frame);
                    return None;
                }
            }
        }
    }
}

// 更新 end 地址
self.end.store(target_end, Ordering::SeqCst);
Some(VirtAddr::new(target_end))
}

```

5. 第一次测试:

```

ysos> run APP/BRK_TEST
正在运行程序: APP/BRK_TEST
[INFO ] Loading ELF from path 'APP/BRK_TEST' as process 'BRK_TEST', entry point: 0x1111000017f0
[INFO ] ELF header bytes: [7f, 45, 4c, 46, 02, 01, 01, 00, 00, 00, 00, 00, 00, 00]
[INFO ] Loading & mapping segment: virtual_addr=0x111100000000, file_size=0x8ac, mem_size=0x8ac, flags=Flags(4)
[INFO ] Loading & mapping segment: virtual_addr=0x111100001000, file_size=0x2f7f, mem_size=0x2f7f, flags=Flags(5)
[INFO ] Loading & mapping segment: virtual_addr=0x111100004000, file_size=0x1030, mem_size=0x1030, flags=Flags(6)
[INFO ] Loading & mapping segment: virtual_addr=0x111100006000, file_size=0x0, mem_size=0x1, flags=Flags(6)
进程ID: 3
🚀 开始测试 brk 系统调用...
🔴 步骤 1: 获取当前堆起始地址
✅ 堆起始地址: 0x200000000000
🔴 步骤 2: 计算目标堆结束地址: 0x200000001000 (大小: 4096 字节)
🔴 步骤 3: 扩展堆到目标地址
✅ brk 返回地址: 0x200000001000
✅ 堆扩展成功!
🔴 步骤 4: 测试堆内存的写入和读取
📁 写入测试数据...
📁 读取并验证数据...
✅ 数据写入和读取测试通过!
🔴 步骤 5: 测试堆缩小
✅ 堆缩小成功! 新结束地址: 0x200000000800
🔴 步骤 6: 验证最终堆状态
✅ 当前堆结束地址: 0x200000000800
🎉 所有测试通过! brk 系统调用工作正常!
程序 'APP/BRK_TEST' 已退出, 返回值: 0
ysos>

```

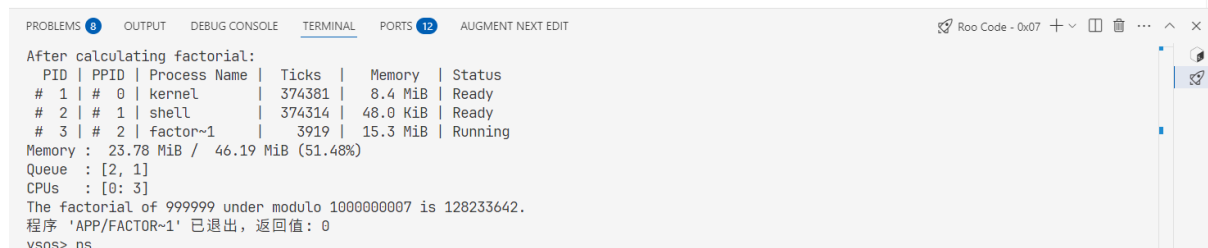
可以看出对于 brk 系统调用的测试完全成功

6. 接下来我们在 `pkg/lib/Cargo.toml` 中将用户程序的默认内存分配器改成 `brk_allocate`。

```
[features]
- default = ["kernel_alloc"]
+ default = ["brk_alloc"]
```

重新编译运行内核、加载并运行用户程序，用户程序均可以正常运行：

```
pkg > lib > [T] Cargo.toml > {} features > [ ] kernel_alloc
1  [package]
2  name = "yslib"
3  version.workspace = true
4  edition.workspace = true
5
6  [dependencies]
7  syscall_def = { workspace = true }
8  chrono = { workspace = true }
9  linked_list_allocator = { workspace = true, optional = true }
10
11 [features]
12 default = ["brk_alloc"]
13 kernel_alloc = []
14 brk_alloc = ["dep:linked_list_allocator"]
15
```



```
After calculating factorial:
PID | PPID | Process Name | Ticks | Memory | Status
# 1 | # 0 | kernel | 374381 | 8.4 MiB | Ready
# 2 | # 1 | shell | 374314 | 48.0 KiB | Ready
# 3 | # 2 | factor~1 | 3919 | 15.3 MiB | Running
Memory : 23.78 MiB / 46.19 MiB (51.48%)
Queue : [2, 1]
CPUs : [0: 3]
The factorial of 999999 under modulo 1000000007 is 128233642.
程序 'APP/FACTOR~1' 已退出，返回值: 0
ysos> ps
```

现在我们可以使用新的内存分配器 `brk_alloc` 来让之前的每个用户程序正常执行了。

八. 思考题

1. 当在 `Linux` 中运行程序的时候删除程序在文件系统中对应的文件，会发生什么？程序能否继续运行？遇到未被映射的内存会发生什么？

- 程序可以继续运行，`Linux` 的文件由文件名和 `inode` 组成，`inode` 存储了文件的所有元数据，而文件名只是一个指向对应 `node` 的标签或指针，在 `Linux` 中，当我们执行 `rm` 指令时，只是删除了文件名和 `inode` 之间的链接，但是当我们执行一个程序时，内核会加载这

个程序，在这个过程中，Linux 内核会打开这个文件，把它的代码段、数据段等映射到虚拟内存空间中，这个“打开并映射”的动作，也相当于进行了一次“链接”，这就导致虽然此时程序的文件名链接数为 0，但总链接数不为 0，而内核会一直保存 inode 和它的代码块，直到指向它的最后一个链接被释放掉，因此程序能够继续运行，当程序退出时内核才会认为这个文件不再被需要，释放 inode 并将它占用的磁盘块标记为可用，这些空间才能被新文件覆盖。

- 遇到未被映射的内存，需要分为两种情况，一种是访问非法内存地址（如空指针），会触发段错误导致程序终止。另外如果访问到的是合法但未加载的内存地址，会触发 Page Fault，内核会从磁盘加载数据，程序继续运行。

2. 为什么要通过 `Arc::strong_count` 来获取 `Arc` 的引用计数？查看它的定义，它和一般使用 `&self` 的方法有什么不同？出于什么考虑不能直接通过 `&self` 来进行这一操作？

- 语法上，这是标准库 `Arc` 的唯一调用接口。
- 设计上，这个 API 的设计本身也是在提醒我们，获取引用计数是一个有副作用和局限性的操作，不应滥用。
- 正确性，在 `ysos` 的 `Debug` 中，这种用法仅用于提供调试信息，这样的操作是安全且恰当的。

3. bootloader 加载内核并为其分配初始栈区时，至少需要多少页内存才能保证内核正常运行？

前面的测试结果显示当 `kernel_stack_auto_grow` 的初始值大于等于 8 时，即至少 8 页内存才能保证内核正常运行，当修改为小于 8 的值，分为下面 3 种情况：

- 0：此时触发了设定好的 `panic!` 信息，表示内核的物理帧分配器已经没有可用的内存页了。

```

tri+Shift+M) - Total 8 Problems | c/config.rs@054: parse kernel_stack_size = 1048576
[ INFO]: pkg/boot/src/config.rs@054: parse physical_memory_offset = 0xFFFF800000000000
[ INFO]: pkg/boot/src/config.rs@054: parse kernel_path = \KERNEL.ELF
[ INFO]: pkg/boot/src/config.rs@054: parse kernel_stack_auto_grow = 0
[ INFO]: pkg/boot/src/config.rs@054: parse log_level = info
[ INFO]: pkg/boot/src/config.rs@054: parse load_apps = 1
[ INFO]: pkg/boot/src/main.rs@042: Config: Config {
[ INFO]: pkg/boot/src/main.rs@042:     kernel_stack_address: 0xffffffff00000000,
[ INFO]: pkg/boot/src/main.rs@042:     kernel_stack_auto_grow: 0x0,
[ INFO]: pkg/boot/src/main.rs@042:     kernel_stack_size: 0x100000,
[ INFO]: pkg/boot/src/main.rs@042:     physical_memory_offset: 0xffff800000000000,
[ INFO]: pkg/boot/src/main.rs@042:     kernel_path: "\\KERNEL.ELF",
[ INFO]: pkg/boot/src/main.rs@042:     cmdline: "",
[ INFO]: pkg/boot/src/main.rs@042:     load_apps: true,
[ INFO]: pkg/boot/src/main.rs@042:     log_level: "info",
[ INFO]: pkg/boot/src/main.rs@042: }
[ INFO]: pkg/boot/src/fs.rs@052: Load file "KERNEL.ELF" to memory, size = 478088
[ INFO]: pkg/boot/src/main.rs@056: Loading apps...
[ INFO]: pkg/boot/src/fs.rs@052: Load file "brk_alloc_test" to memory, size = 51328
[ INFO]: pkg/boot/src/fs.rs@052: Load file "factorial" to memory, size = 46184
[ INFO]: pkg/boot/src/fs.rs@052: Load file "hello" to memory, size = 45712
[ INFO]: pkg/boot/src/fs.rs@052: Load file "shell" to memory, size = 50952
[ INFO]: pkg/boot/src/fs.rs@052: Load file "counter" to memory, size = 47768
[ INFO]: pkg/boot/src/fs.rs@052: Load file "dinner" to memory, size = 55008
[ INFO]: pkg/boot/src/fs.rs@052: Load file "brk_test" to memory, size = 45840
[ INFO]: pkg/boot/src/fs.rs@052: Load file "mq" to memory, size = 52104
[ INFO]: pkg/boot/src/fs.rs@052: Load file "fork_test" to memory, size = 46472
[ INFO]: pkg/boot/src/main.rs@139: Loaded 9 apps
[ INFO]: /home/camellia/ysos/0x07/pkg/elf/src/lib.rs@146: Loading & mapping segment: virtual_addr=0xffffffff00000000, file_size=0xec04, mem_size=0xec04, flags=Flags(4)
[ INFO]: /home/camellia/ysos/0x07/pkg/elf/src/lib.rs@146: Loading & mapping segment: virtual_addr=0xffffffff000000f000, file_size=0x40345, mem_size=0x40345, flags=Flags(5)
[ INFO]: /home/camellia/ysos/0x07/pkg/elf/src/lib.rs@146: Loading & mapping segment: virtual_addr=0xffffffff0000050000, file_size=0x6128, mem_size=0x6128, flags=Flags(6)
[ INFO]: /home/camellia/ysos/0x07/pkg/elf/src/lib.rs@146: Loading & mapping segment: virtual_addr=0xffffffff0000057000, file_size=0x0, mem_size=0x80d528, flags=Flags(6)
[PANIC]: panicked at pkg/boot/src/allocator.rs:10:14:
Failed to allocate frame: Error { status: OUT_OF_RESOURCES, data: () }
camellia@LAPTOP-Camellia:~/ysos/0x07$

```

- 1~3：触发 Triple Fault，无法完成初始化,内核初始化进行到一半（打印出 Kernel Heap Size 后），整个系统就重置了，然后又从头开始初始化，形成一个无限重启的循环。

```

PROBLEMS 8 OUTPUT DEBUG CONSOLE TERMINAL PORTS 17 AUGMENT NEXT EDIT

v0.4.0

[+] Serial Initialized.
[INFO] Logger Initialized with level: info
[INFO] Physical Offset : 0xffff800000000000
[INFO] Privilege Stack : 0xffffffff00008582a8-0xffffffff000085c2a8
[INFO] Double Fault IST : 0xffffffff000085c2a8-0xffffffff00008602a8
[INFO] Page Fault IST : 0xffffffff00008602a8-0xffffffff00008602a8
[INFO] Timer IST : 0xffffffff00008602a8-0xffffffff00008642a8
[INFO] Syscall IST : 0xffffffff00008642a8-0xffffffff00008642a8
[INFO] Total IST size : 64 KiB
[INFO] Kernel Heap Size :

```

- 4~7：内核栈深度不够出现栈溢出导致陷入无限循环状态。系统就这样陷入了“页缺失 -> 双重故障 -> 处理 -> 返回 -> 再次页缺失”的死循环中。它没有崩溃重启，但也无法向前执行任何一步。

```
[INFO ] Page fault on kernel at 0xffffffff01ffff8000
[INFO ] Page fault on kernel at 0xffffffff01ffff8000
[INFO ] Page fault on kernel at 0xffffffff01ffff8000
[INFO ] Page fault on kernel at 0xffffffff01ffff8000
[INFO ] Page fault on kernel at 0xffffffff01ffff8000
[INFO ] Page fault on kernel at 0xffffffff01ffff8000
[INFO ] Page fault on kernel at 0xffffffff01ffff8000
[INFO ] Page fault on kernel at 0xffffffff01ffff8000
[INFO ] Page fault on kernel at 0xffffffff01ffff8000
[INFO ] Page fault on kernel at 0xffffffff01ffff8000
[INFO ] Page fault on kernel at 0xffffffff01ffff8000
[INFO ] Page fault on kernel at 0xffffffff01ffff8000
[INFO ] Page fault on kernel at 0xffffffff01ffff8000
[INFO ] Page fault on kernel at 0xffffffff01ffff8000
[INFO ] Page fault on kernel at 0xffffffff01ffff8000
[INFO ] Page fault on kernel at 0xffffffff01ffff8000
```

注意：依赖的子系统有中断/异常处理系统、内存管理子系统、栈管理子系统，上面的错误与栈管理子系统、物理帧分配器和中断处理配置都有关

4. 尝试查找资料，了解 mmap、munmap 和 mprotect 系统调用的功能和用法，回答下列问题：

4.1. mmap 的主要功能是什么？它可以实现哪些常见的内存管理操作？

`mmap` (memory map) 的主要功能是将一个文件或者其它对象映射到调用进程的地址空间。它通过在虚拟内存中创建一个映射，直接将文件内容与一段内存区域关联起来，从而让程序可以像访问普通内存数组一样读写文件，而无需调用 `read()` 和 `write()` 系统调用。

它可以实现以下常见的内存管理操作：

- **文件映射 (File-backed mapping)：**这是 `mmap` 最常见的用途。将文件内容映射到内存，实现高效的文件 I/O。对映射内存的修改最终会写回文件。
- **匿名映射 (Anonymous mapping)：**不与任何文件关联，创建一块“纯粹”的、内容初始化为零的内存区域。它常被用于实现程序的数据段 (BSS)、堆 (`malloc` 底层实现) 和栈。
- **共享内存 (Shared memory)：**多个进程可以映射同一个文件或创建共享的匿名映射，从而实现高效的进程间通信 (IPC)。一个进程对共享内存的修改可以立即被其他进程看到。
- **动态库加载：**操作系统的动态链接器 (如 `ld.so`) 使用 `mmap` 将共享库 (`.so` 文件) 的代码段和数据段加载到进程的地址空间中。

4.2. munmap 的主要功能是什么？什么时候需要使用 munmap？

`munmap` 的主要功能是解除由 `mmap` 创建的内存映射。当调用 `munmap` 后，指定的内存区域将不再有效，访问它会导致段错误 (Segmentation Fault)。

需要使用 `munmap` 的主要时机是：

- **资源释放：**当程序不再需要某个内存映射区域时，应该调用 `munmap` 来释放它。这会将虚拟地址空间返还给操作系统，并可能释放相关的物理内存。

- **防止内存泄漏**：每一次成功的 `mmap` 调用都应该有一个对应的 `munmap` 调用。忘记解除映射是常见的内存泄漏原因，会导致进程持续占用不必要的内存资源，直到进程终止。

4.3. mprotect 的主要功能是什么？使用 mprotect 可以实现哪些内存保护操作？

`mprotect` 的主要功能是改变一个已存在的内存映射区域的访问权限。例如，可以将一个只读的内存区域变为可读可写。

使用 `mprotect` 可以实现以下内存保护操作，权限标志可以组合使用：

- `PROT_READ`：页面可被读取。
- `PROT_WRITE`：页面可被写入。
- `PROT_EXEC`：页面可被执行（包含机器指令）。
- `PROT_NONE`：页面不能被访问。

常见的应用场景包括：

- **写时复制 (Copy-on-Write, COW)**：在 `fork()` 系统调用后，父子进程共享的页面可以被标记为只读。当任何一方尝试写入时，会触发一个页错误，内核此时才会为写入方复制一个新的、可写的页面副本。
- **即时编译 (Just-In-Time, JIT)**：JIT 编译器可以在运行时生成机器码。它可以先将一块内存映射为可读可写（`PROT_READ | PROT_WRITE`），将生成的机器码写入，然后使用 `mprotect` 将其权限改为可读可执行（`PROT_READ | PROT_EXEC`），以防止代码被意外修改并允许 CPU 执行。
- **调试和安全**：可以设置“哨兵页面”（Guard Page），即一块 `PROT_NONE` 的内存区域放在一个数据结构（如栈）的末尾。如果发生溢出并访问到哨兵页面，会立即触发段错误，帮助开发者快速定位问题。

4.4. 编写 C 程序，使用 mmap 将一个文件映射到内存中，并读写该文件的内容。

下面是一个简单的 C 程序示例。它创建一个名为 `test.txt` 的文件，使用 `mmap` 将其映射到内存，读取并修改其内容，最后解除映射。

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/stat.h>

int main() {
    const char *filepath = "test.txt";
    const char *text = "Hello, mmap world!";
    off_t filesize = strlen(text);

    // 1. 创建并写入一个文件
    int fd = open(filepath, O_RDWR | O_CREAT | O_TRUNC, (mode_t)0644);
    if (fd == -1) {
        perror("open");
        exit(EXIT_FAILURE);
    }
    if (write(fd, text, filesize) != filesize) {
        perror("write");
        exit(EXIT_FAILURE);
    }

    // 2. 将文件映射到内存
    // MAP_SHARED 意味着修改会写回文件
    char *map = mmap(NULL, filesize, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (map == MAP_FAILED) {
        perror("mmap");
        exit(EXIT_FAILURE);
    }

    // 3. 读取并打印映射的内存
    printf("Original content: %s\n", map);

    // 4. 修改映射的内存
    // 这会间接修改文件内容
    memcpy(map, "Hi, ", 4);
    printf("Modified content: %s\n", map);

    // 5. (可选) 强制将修改同步到磁盘文件
    if (msync(map, filesize, MS_SYNC) == -1) {
        perror("msync");
    }
    printf("msync called to flush changes to disk.\n");
}

```

```
// 6. 解除内存映射
if (munmap(map, filesize) == -1) {
    perror("munmap");
}

// 7. 关闭文件描述符
close(fd);

return 0;
}
```

4.5. 思考：文件内容什么时候会被写入到磁盘？

对于通过 `mmap` 以 `MAP_SHARED` 方式映射的文件，程序对内存的修改不会立即写入磁盘。操作系统为了性能，会将修改缓存起来（这些被修改过的、但还未写入磁盘的内存页被称为“脏页”）。真正的写盘时机由操作系统内核决定，通常在以下几种情况发生：

1. **内核刷新时机**：操作系统有一个后台进程（如 `pdflush` 或 `kworker`）会周期性地脏页回写到磁盘，以释放内存并保证数据持久性。
2. **内存压力**：当系统可用物理内存不足时，内核会选择将一些脏页写回磁盘，以便回收这些物理页帧用于其它目的。
3. **显式调用同步**：程序可以调用 `msync()` 函数，强制要求内核将指定内存区域的修改立即或异步地写回磁盘。
4. **解除映射时**：当调用 `munmap()` 解除文件映射时，内核通常会确保所有相关的脏页都被写回磁盘。
5. **正常关机**：系统正常关闭或重启时，会执行 `sync` 操作，确保所有文件系统缓存都被清空并写入持久化存储。

4.6. 综合考虑有关内存、文件、I/O 等方面的知识，讨论为什么 `mmap` 系统调用在现代操作系统中越来越受欢迎，它具有哪些优势？

`mmap` 系统调用之所以在现代操作系统中备受欢迎，是因为它优雅地统一了文件 I/O 和内存管理，带来了显著的性能和便利性优势：

1. **减少内存拷贝**：传统的 `read()` / `write()` 系统调用至少涉及两次数据拷贝：一次是从磁盘到内核的页缓存（Page Cache），第二次是从页缓存到用户空间的缓冲区。而 `mmap` 通过让用户空间直接访问内核的页缓存，**完全消除了第二次拷贝**。数据只需从磁盘拷贝到页缓存一次，CPU 就可以直接处理它。这对于大数据量的读写操作能极大地提升效率。
2. **简化编程模型**：程序员可以像操作内存数组一样直接读写文件，无需管理缓冲区、文件偏移量（`lseek`）和读写循环。代码变得更简洁、直观，且不易出错。

3. **内核级别的页管理**：文件的读写变成了对内存的访问。当程序访问一个尚未加载到内存的映射区域时，会触发一个缺页异常（Page Fault）。内核会自动处理这个异常，从磁盘加载对应的文件页到物理内存中。这种按需加载（Lazy Loading）的机制避免了一次性将整个文件读入内存的开销，启动速度更快，内存使用更高效。
4. **高效的进程间通信**：多个进程映射同一个文件（`MAP_SHARED`），可以得到一块共享内存。这块内存由内核维护，一个进程的写入对其它进程立即可见，是最高效的 IPC 方式之一，避免了管道、套接字等需要数据拷贝的通信方式。
5. **统一的内存和文件缓存**：由于 `mmap` 直接使用了内核的页缓存，操作系统可以统一管理文件数据和其它内存。当内存紧张时，内核可以智能地将干净的（未修改的）文件页丢弃，需要时再从磁盘读回，从而更灵活地调度物理内存资源。

综上所述，`mmap` 通过减少数据拷贝、简化编程、利用内核的惰性加载和缓存机制，提供了无与伦比的 I/O 性能和便利性，使其成为数据库、Web 服务器、动态链接器、编译器等高性能应用的首选技术。

九. 加分项

1. 😊 尝试借助 `brk` 为用户态堆实现自动扩容：

- `LockedHeap` 支持 `extend` 方法，可以在堆区不足时扩容大小，但是需要用户程序分配好所需的空间；
- 自定义数据结构 `BrkAllocator`，并为其实现 `GlobalAlloc` trait：

测试结果，说明我们的新的全局分配器起到了作用，并且成功实现了自动扩容。

```
ysos> run APP/BRK_AL~1
正在运行程序：APP/BRK_AL~1
[INFO ] Loading ELF from path 'APP/BRK_AL~1' as process 'BRK_AL~1', entry point: 0x1111000022d0
[INFO ] ELF header bytes: [7f, 45, 4c, 46, 02, 01, 01, 00, 00, 00, 00, 00, 00, 00, 00]
[INFO ] Loading & mapping segment: virtual_addr=0x111100000000, file_size=0xca4, mem_size=0xca4, flags=Flags(4)
[INFO ] Loading & mapping segment: virtual_addr=0x111100001000, file_size=0x4a0a, mem_size=0x4a0a, flags=Flags(5)
[INFO ] Loading & mapping segment: virtual_addr=0x111100006000, file_size=0x1028, mem_size=0x1028, flags=Flags(6)
[INFO ] Loading & mapping segment: virtual_addr=0x111100008000, file_size=0x0, mem_size=0x48, flags=Flags(6)
进程 ID: 4
🚀 开始测试 brk_alloc 内存分配器...
🔴 测试 1: Vec 动态数组分配
✅ Vec 测试通过！总和：4950
🔴 测试 2: String 字符串分配
✅ String 测试通过！内容：'Hello, brk_alloc allocator!'
🔴 测试 3: Box 堆分配
✅ Box 测试通过！256个元素都是42
🔴 测试 4: 大量小分配测试
✅ 大量小分配测试通过！50个Box分配成功
🔴 测试 5: 混合分配测试
✅ 混合分配测试通过！20个字符串分配成功
🔴 测试 6: 内存释放测试
✅ 内存释放测试通过！平方和：328350
🔴 所有 brk_alloc 分配器测试通过！
📊 测试统计：
- Vec 动态数组：✅
- String 字符串：✅
- Box 堆分配：✅
- 大量小分配：✅
- 混合分配：✅
- 内存释放：✅
程序 'APP/BRK_AL~1' 已退出，返回值：0
ysos>
```

```

use linked_list_allocator::LockedHeap;
use core::alloc::{GlobalAlloc, Layout};
use core::sync::atomic::{AtomicUsize, Ordering};

use crate::*;

const INITIAL_HEAP_SIZE: usize = 8 * 1024; // 8 KiB 初始大小
const MAX_HEAP_SIZE: usize = 8 * 1024 * 1024; // 8 MiB 最大大小
const EXTEND_SIZE: usize = 4 * 1024; // 每次扩容 4 KiB

#[global_allocator]
static ALLOCATOR: BrkAllocator = BrkAllocator::empty();

struct BrkAllocator {
    allocator: LockedHeap,
    current_size: AtomicUsize,
}

pub fn init() {
    ALLOCATOR.init();
}

impl BrkAllocator {
    pub const fn empty() → Self {
        Self {
            allocator: LockedHeap::empty(),
            current_size: AtomicUsize::new(0),
        }
    }

    pub fn init(&self) {
        // 获取当前堆的起始地址
        let heap_start = sys_brk(None).unwrap();

        // 设置初始堆大小
        let initial_end = heap_start + INITIAL_HEAP_SIZE;
        let ret = sys_brk(Some(initial_end)).expect("Failed to allocate initial heap");

        assert!(ret == initial_end, "Failed to allocate initial heap");

        // 初始化 LockedHeap
        unsafe {
            self.allocator.lock().init(heap_start as *mut u8, INITIAL_HEAP_SIZE);
        }
    }
}

```

```

        // 记录当前堆大小
        self.current_size.store(INITIAL_HEAP_SIZE, Ordering::SeqCst);
    }

    pub unsafe fn extend(&self) → bool {
        let current_size = self.current_size.load(Ordering::SeqCst);

        // 检查是否已达到最大大小
        if current_size ≥ MAX_HEAP_SIZE {
            return false;
        }

        // 计算新的堆大小, 确保不超过最大限制
        let new_size = core::cmp::min(current_size + EXTEND_SIZE, MAX_HEAP_SIZE);

        if new_size == current_size {
            return false;
        }

        // 获取当前堆的起始地址
        let heap_start = match sys_brk(None) {
            Some(start) ⇒ start,
            None ⇒ return false,
        };

        // 计算新的堆结束地址
        let new_end = heap_start + new_size;

        // 尝试扩展堆
        match sys_brk(Some(new_end)) {
            Some(actual_end) if actual_end == new_end ⇒ {
                // 扩展成功, 重新初始化分配器以包含新的堆大小
                unsafe {
                    self allocator.lock().init(heap_start as *mut u8, new_size);
                }

                // 更新当前堆大小
                self.current_size.store(new_size, Ordering::SeqCst);
                true
            }
            _ ⇒ false,
        }
    }
}

```

```

unsafe impl GlobalAlloc for BrkAllocator {
    unsafe fn alloc(&self, layout: Layout) → *mut u8 {
        // 首先尝试分配
        let mut ptr = unsafe { self.allocator.alloc(layout) };

        // 如果分配失败，尝试扩展堆然后再次分配
        if ptr.is_null() {
            if unsafe { self.extend() } {
                ptr = unsafe { self.allocator.alloc(layout) };
            }
        }

        ptr
    }

    unsafe fn dealloc(&self, ptr: *mut u8, layout: Layout) {
        unsafe { self.allocator.dealloc(ptr, layout) }
    }
}

#[cfg(not(test))]
#[alloc_error_handler]
fn alloc_error_handler(layout: alloc::alloc::Layout) → ! {
    panic!("Allocation error: {:?}", layout)
}

```