



# 课程实验报告

## 0x07 更好的内存管理

课程名称	操作系统原理实验
专业名称	计算机科学与技术
学生姓名	陈政宇
学生学号	23336003
实验地点	东校园-实验中心大楼 B201
实验成绩	
实验日期	2025 年 6 月 24 日

# 目录

<b>1 实验目的 .....</b>	<b>3</b>
<b>2 代码实现 .....</b>	<b>3</b>
2.1 帧分配器的内存回收 .....	3
2.2 用户程序的内存统计 .....	4
2.3 用户程序的内存释放 .....	9
2.4 内核的内存统计 .....	11
2.5 内核栈的自动增长 .....	12
2.6 用户态堆 .....	13
<b>3 思考题 .....</b>	<b>15</b>
3.1 在 Linux 中删除运行中的程序文件 .....	15
3.1.1 会发生什么？程序能否继续运行？ .....	15
3.1.2 遇到未被映射的内存会发生什么？ .....	15
3.2 Arc::strong_count 的设计 .....	16
3.2.1 为什么要通过 Arc::strong_count 获取引用计数？ .....	16
3.2.2 它和一般使用 &self 的方法有什么不同？ .....	16
3.2.3 出于什么考虑不能直接通过 &self 来进行这一操作？ .....	16
3.3 内核初始栈区大小 .....	17
3.3.1 至少需要多少页内存？ .....	17
3.3.2 增大内核栈区大小的观察与分析 .....	17
3.4 mmap, munmap, mprotect 系统调用 .....	18
3.4.1 mmap .....	18
3.4.2 munmap .....	18
3.4.3 mprotect .....	18
3.4.4 C 程序示例：使用 mmap 读写文件 .....	19
3.5 mmap 受欢迎的原因及其优势 .....	20

## 1 实验目的

1. 实现帧分配器的内存回收，操作系统的内存统计。
2. 实现操作系统栈的自动增长。
3. 尝试实现 brk 系统调用，实现用户态程序的动态内存分配。

## 2 代码实现

### 2.1 帧分配器的内存回收

#### ☰ 内存回收

在 Lab 4 的加分项中，提到了尝试实现帧分配器的内存回收。在本次实验中将进一步完善这一功能。

在进行帧分配器初始化的过程中，内核从 bootloader 获取到了一个 MemoryMap 数组，其中包含了所有可用的物理内存区域，并且内核使用 into\_iter() 将这一数据结构的所有权交给了一个迭代器，你可以在 pkg/kernel/src/memory/frames.rs 中了解到相关类型和实现。

迭代器是懒惰的，只有在需要时才会进行计算，因此在进行逐帧分配时，并没有额外的内存开销。但是，当需要进行内存回收时，就需要额外的数据结构来记录已经分配的帧，以便进行再次分配。

相对于真实的操作系统，本实验中的内存回收是很激进的：即能回收时就回收，不考虑回收对性能的影响。在实际的操作系统中，内存回收是一个复杂的问题，需要考虑到内存的碎片化、内存的使用情况、页面的大小等细节；进而使用标记清除、分段等策略来减少内存回收的频率和碎片化。

因此对于本实验的帧分配器来说，内存回收的操作是非常简单的，只需要将已经分配的帧重新加入到可用帧的集合中即可。

为了减少内存占用，这一操作通常使用位图来实现，即使用一个位图来记录每一帧的分配情况。由于 Rust 的标准库中并没有提供位图的实现，因此你可以简单地使用一个 Vec<PhysFrame> 作为已经回收的帧的集合。

```
1 unsafe impl FrameAllocator<Size4KiB> for BootInfoFrameAllocator {  Rust
2     fn allocate_frame(&mut self) -> Option<PhysFrame> {
```

```

3      // DONE: allocate a frame from the bootloader's memory map
4      // 尝试从 recycled 中获取已回收的帧
5      if let Some(frame) = self.recycled.pop() {
6          return Some(frame);
7      }
8
9      // 如果没有可回收的帧, 则从 frames 中获取新的帧
10     if let Some(frame) = self.frames.next() {
11         self.used += 1;
12         Some(frame)
13     } else {
14         None
15     }
16 }
17 }
18
19 impl FrameDeallocator<Size4KiB> for BootInfoFrameAllocator {
20     unsafe fn deallocate_frame(&mut self, _frame: PhysFrame) {
21         // DONE: deallocate frame (not for lab 2)
22         self.recycled.push(_frame);
23     }
24 }

```

以上代码中, `BootInfoFrameAllocator` 结构体中新增了一个 `recycled` 字段, 用于存储已回收的帧。在 `allocate_frame` 方法中, 首先尝试从 `recycled` 中获取已回收的帧, 如果没有可回收的帧, 则从 `frames` 中获取新的帧。在 `deallocate_frame` 方法中, 将传入的帧加入到 `recycled` 中。

## 2.2 用户程序的内存统计

### ☞ 内存统计

在目前的实现 (Lab 3) 中, 用户程序在进程结构体中记录的内存区域只有栈区, 堆区由内核进行代劳, 同时 ELF 文件映射的内存区域也从来没有被释放过, 无法被其他程序复用。

而相较于 Linux, 本实验并没有将内存管理抽象为具有上述复杂功能的结构: 用户程序的内存占用严格等同于其虚拟内存大小, 并且所有页面都会被加载到物理内存中, 不存在文件映射等概念, 只有堆内存和栈内存是可变的。

因此，其内存统计并没有那么多的细节，只需要统计用户程序的栈区和堆区的大小即可。在 Stack 和 Heap 中，已经实现了 `memory_usage` 函数来获取栈区和堆区的内存占用字节数。

那么根据上述讨论，对本实验的内存占用而言，只剩下了 ELF 文件映射的内存区域和页表的内存占用，为实现简单，本部分忽略页表的内存占用，只统计 ELF 文件映射的内存占用。

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：

```

1  /// Load & Map ELF file
2  ///
3  /// for each segment, load code to new frame and set page table
4  pub fn load_elf(
5      elf: &ElfFile,
6      physical_offset: u64,
7      page_table: &mut impl Mapper<Size4KiB>,
8      frame_allocator: &mut impl FrameAllocator<Size4KiB>,
9      user_access: bool,
10 ) -> Result<(Vec<PageRangeInclusive>, u64), MapToError<Size4KiB>> {
11     trace!("Loading ELF file...{:?}", elf.input.as_ptr());
12     let mut loaded_page_ranges: Vec<PageRangeInclusive> = Vec::new();
13     let mut total_usage_bytes: u64 = 0;
14
15     for segment in elf.program_iter() {
16         if segment.get_type().unwrap() != program::Type::Load {
17             continue;
18         }
19
20         if segment.mem_size() == 0 {
21             continue;
22         }
23
24         match load_segment(
25             elf,
26             physical_offset,
27             &segment,
28             page_table,
29             frame_allocator,
30             user_access,

```



```

31     ) {
32         Ok((page_range, usage)) => {
33             if page_range.count() > 0 {
34                 loaded_page_ranges.push(page_range);
35             }
36             total_usage_bytes += usage;
37         }
38         Err(e) => return Err(e),
39     }
40 }
41 Ok((loaded_page_ranges, total_usage_bytes))
42 }

```

在 `load_elf` 函数中，新增了一个 `total_usage_bytes` 变量，用于记录加载的 ELF 文件的内存占用字节数。在加载每个段时，调用 `load_segment` 函数，并将返回的内存占用字节数累加到 `total_usage_bytes` 中。

### ☞ 打印内存占用


之后，在 `pkg/kernel/src/proc/vm` 中完善 `ProcessVm` 的 `load_elf_code` 函数，在加载 ELF 文件时记录内存占用。

为了便于测试和观察，在 `pkg/kernel/src/proc/manager.rs` 的 `print_process_list` 和 `Process` 的 `fmt` 实现中，添加打印内存占用的功能。

```

1  fn load_elf_code(&mut self, elf: &ElfFile, mapper: MapperRef,
2      alloc: FrameAllocatorRef) {
3      // DONE: make the `load_elf` function return the code pages
4      // DONE: calculate code usage
5      let result = elf::load_elf(elf, *PHYSICAL_OFFSET.get().unwrap(),
6          mapper, alloc, true);
7      match result {
8          Ok((loaded_code_pages, total_code_usage_bytes)) => {
9              self.code = loaded_code_pages;
10             self.code_usage = total_code_usage_bytes;
11         }
12         Err(e) => {
13             error!("Failed to load ELF for process: {:?}", e);
14             panic!("ELF loading failed: {:?}", e);
15         }
16     }
17 }

```

 Rust

在 `load_elf_code` 函数中，调用 `load_elf` 函数，并将返回的内存占用字节数赋值给 `self.code_usage`。

```

1  fn format_system_memory_usage(name: &str, used_bytes: u64,
2      total_bytes: u64) -> String {
3      if total_bytes == 0 {
4          return format!(
5              "{:<6} : {:>6} {:>3} / {:>6} {:>3} ( N/A %)\n", // 中文冒号
6              name,
7              humanized_size(used_bytes as u64).0,
8              humanized_size(used_bytes as u64).1,
9              humanized_size(total_bytes as u64).0,
10             humanized_size(total_bytes as u64).1
11         );
12     }
13     let (used_float, used_unit) = humanized_size(used_bytes as u64);
14     let (total_float, total_unit) = humanized_size(total_bytes as u64);
15     format!(
16         "{:<6} : {:>6.*} {:>3} / {:>6.*} {:>3} ({:>5.2}%)\n", // 中文冒号
17         name,
18         2, // 小数点后两位
19         used_float,
20         used_unit,
21         2, // 小数点后两位
22         total_float,
23         total_unit,
24         used_bytes as f32 / total_bytes as f32 * 100.0
25     )
26 }
27
28 pub fn print_process_list(&self) {
29     let mut output = String::from(" PID | PPID | ProceName      |
30     MemoryUsage | Ticks   | Status\n"); // 修改表头为中文，并调整列名
31
32     self.processes
33         .read()
34         .values()
35         .filter(|p| p.read().status() != ProgramStatus::Dead)
36         .for_each(|p| output += &format!("{}", p));
37
38     if let Some(alloc_mutex) = FRAME_ALLOCATOR.get() {

```

```

38     let alloc = alloc_mutex.lock();
39     let frames_used = alloc.frames_used();
40     let frames_recycled = alloc.frames_recycled_count();
41     let frames_total = alloc.frames_total();
42
43     // (已用帧 - 已回收帧) * 页大小
44     let active_system_frames =
45         frames_used.saturating_sub(frames_recycled);
46     let used_mem_bytes = active_system_frames as u64 * PAGE_SIZE;
47     let total_mem_bytes = frames_total as u64 * PAGE_SIZE;
48
49     output += &format_system_memory_usage("SystemMemory",
50         used_mem_bytes, total_mem_bytes);
51 } else {
52     output += "SystemMemory: frame allocator do not init\n";
53 }
54
55 output += &format!("ready_queue : {:?}\n",
56     self.ready_queue.lock()); // 中文标签
57
58 print!("{}", output); // 最终打印
59
60 impl core::fmt::Display for Process {
61     fn fmt(&self, f: &mut core::fmt::Formatter) -> core::fmt::Result {
62         let inner = self.inner.read();
63         let (size, unit) = inner.proc_vm.as_ref().map_or((0.0, "B"), |vm|
64             {
65                 let usage = vm.memory_usage();
66                 humanized_size(usage)
67             });
68         write!(
69             f,
70             // 在 Process Name 和 Ticks 之间添加了内存占用的列
71             " #{:<3} | #{:<3} | {:<12} | {:>5.1} {:<2} | {:<7} | {:?}" ,
72             self.pid.0, // PID
73             inner.parent().map(|p| p.pid.0).unwrap_or(0), // PPID
74             inner.name, // 进程名
75             size, // 内存大小

```



```

76         unit,                                // 内存单位
77         inner.ticks_passed,                    // Ticks
78         inner.status                           // 状态
79     )
80 }
81 }

```

在 `print_process_list` 函数中，调用 `format_system_memory_usage` 函数来格式化系统内存占用信息，并将其添加到输出中。在 `Process` 的 `fmt` 实现中，添加了内存占用的列。

## 2.3 用户程序的内存释放

### 内存释放的实现

出于模块化设计，先为 `Stack` 实现 `clean_up` 函数，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。

接下来重点关注 `ProcessVm` 的相关实现，位于 `pkg/kernel/src/proc/vm/mod.rs` 中，首先为它添加 `clean_up` 函数：

```

1  pub(super) fn clean_up(
2      &mut self,
3      mapper: MapperRef,
4      dealloc: FrameAllocatorRef,
5  ) -> Result<(), UnmapError> {
6      if self.usage == 0 {
7          return Ok(());
8      }
9      // elf::unmap_pages 期望起始地址和页数
10     // self.range 是 PageRange<Size4KiB>
11     // PageRange 包含 start 和 end Page
12     // PageRangeInclusive 包含 start 和 end Page
13     let start_addr = self.range.start.start_address().as_u64();
14     let page_count = self.range.count(); // PageRange::count() 返回 usize
15
16     if page_count > 0 {
17         elf::unmap_pages(start_addr, page_count as u64, mapper, dealloc,
18             true)?;
19     }

```

```

19
20     self.usage = 0;
21     Ok(())
22 }

```

在 Stack 中添加了 `clean_up` 函数，用于释放虚拟内存区域的页面。该函数首先检查 `usage` 是否为 0，如果是，则直接返回；否则，调用 `elf::unmap_pages` 函数来释放页面，并将 `usage` 设置为 0。

```

1  pub(super) fn clean_up(&mut self) -> Result<(), UnmapError> { 🦀 Rust
2      let mapper = &mut self.page_table.mapper();
3      let dealloc = &mut *get_frame_alloc_for_sure();
4
5      // FIXME: implement the `clean_up` function for `Stack`
6      self.stack.clean_up(mapper, dealloc)?;
7
8      if self.page_table.using_count() == 1 {
9          // free heap
10         // FIXME: implement the `clean_up` function for `Heap`
11         self.heap.clean_up(mapper, dealloc)?;
12
13         // free code
14         for page_range in self.code.iter() {
15             elf::unmap_range(*page_range, mapper, dealloc, true)?;
16         }
17
18         unsafe {
19             // free P1-P3
20             mapper.clean_up(dealloc);
21
22             // free P4
23             dealloc.deallocate_frame(self.page_table.reg.addr);
24         }
25     }
26
27     // NOTE: maybe print how many frames are recycled
28     //      **you may need to add some functions to
29     //      `BootInfoFrameAllocator`**
30     let recycled_count_after_cleanup = dealloc.frames_recycled_count();
31     info!("Frames recycled after this cleanup: {} (this is a cumulative
32     count from allocator)", recycled_count_after_cleanup);

```

```
32     Ok(())
33 }
```

在 `ProcessVm` 中添加了 `clean_up` 函数，用于释放虚拟内存区域的页面。该函数首先获取页表的映射器和帧分配器，然后调用 `Stack` 的 `clean_up` 函数来释放栈区的页面。如果页表的使用计数为 1，则表示这是最后一个使用该页表的进程，因此需要释放堆区和代码区的页面。

## 2.4 内核的内存统计

### ☰ 内核的内存统计

至此，用户程序的内存管理已经得到了较好的实现，但是内核占用了多少内存呢？

类似于用户进程的加载过程，可以通过在内核加载时记录内存占用来实现内核的初步内存统计，即在 `bootloader` 中实现这一功能。

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：

```
1 pub type KernelPages = ArrayVec<PageRangeInclusive, 8>;
2
3 pub struct BootInfo {
4     // ...
5
6     // Kernel pages
7     pub kernel_pages: KernelPages,
8 }
```

```
1 pub fn init_kernel_vm(mut self, pages: &KernelPages) -> Self {
2     // DONE: load `self.code` and `self.code_usage` from `pages`
3     self.code = pages.iter().cloned().collect();
4
5     let mut total_usage: u64 = 0;
6     for range in self.code.iter() {
7         total_usage += range.count() as u64 * Size4KiB::SIZE;
8     }
9     self.code_usage = total_usage;
10    info!(
11        "Kernel code usage: {} bytes, {} sections",
12        self.code_usage,
```

```

13     self.code.len()
14 );
15 // DONE: init kernel stack (impl the const `kstack` function)
16 //     `pub const fn kstack() -> Self`
17 //     use consts to init stack, same with kernel config
18 self.stack = Stack::kstack();
19 self
20 }

```

在 `init_kernel_vm` 函数中，加载内核的代码区和内存占用信息，并初始化内核栈。

## 2.5 内核栈的自动增长

### ☞ 内核栈的自动增长

在 Lab 3 中简单实现了用户进程的栈区自动增长，但是内核的栈区并没有进行相应的处理，这将导致内核栈溢出时无法进行自动增长，从而导致内核崩溃。

为了在之前的实验中避免这种情况，实验通过 `bootloader` 直接为内核分配了  $512 * 4 \text{ KiB} = 2 \text{ MiB}$  的栈区来避免可能的栈溢出问题。但这明显是不合理的，因为内核的栈区并不需要这么大的空间。

与其分配一个固定大小的栈区，不如在缺页中断的基础上实现一个简单的栈区自动增长机制，当栈区溢出时，自动为其分配新的页面。

```

1  fn grow_stack(
2      &mut self,
3      addr: VirtAddr,
4      mapper: MapperRef,
5      alloc: FrameAllocatorRef,
6  ) -> Result<(), MapToError<Size4KiB>> {
7      debug_assert!(self.is_on_stack(addr), "Address is not on stack.");
8
9      let new_start_page = Page::containing_address(addr);
10     let page_count = self.range.start - new_start_page;
11
12     trace!(
13         "Fill missing pages...[{:#x} -> {:#x}) ({} pages)",
14         new_start_page.start_address().as_u64(),
15         self.range.start.start_address().as_u64(),
16         page_count

```

 Rust

```

17     );
18
19     let user_access = processor::get_pid() != KERNEL_PID;
20
21     elf::map_pages(
22         new_start_page.start_address().as_u64(),
23         page_count,
24         mapper,
25         alloc,
26         user_access,
27     )?;
28
29     self.range = Page::range(new_start_page, self.range.end);
30     self.usage = self.range.count() as u64;
31
32     Ok(())
33 }

```

在 Stack 中添加了 grow\_stack 函数，用于在栈区溢出时自动增长栈区。该函数首先检查地址是否在栈区内，如果不在，则 panic。然后计算新的起始页面，并调用 elf::map\_pages 函数来分配新的页面。

## 2.6 用户态堆

### ≡ brk 系统调用

最后，为了提供给用户程序更多的内存管理能力，还需要实现一个系统调用：sys\_brk，用于调整用户程序的堆区大小。

```

1  pub fn brk(
2      &self,
3      new_end: Option<VirtAddr>,
4      mapper: MapperRef,
5      alloc: FrameAllocatorRef,
6  ) -> Option<VirtAddr> {
7      if new_end.is_none() {
8          return Some(VirtAddr::new(self.end.load(Ordering::Relaxed)));
9      }
10
11     let new_end = new_end.unwrap();

```

 Rust

```

12
13     if new_end > self.base + HEAP_SIZE || new_end < self.base {
14         error!("Heap brk: new_end is out of heap range");
15         return None;
16     }
17
18     let cur_end = self.end.load(Ordering::Acquire);
19     // heap: [base, cur_end, cur_end + 1) or [base, base)
20     let mut cur_end_page =
21     Page::containing_address(VirtAddr::new(cur_end));
22     if cur_end != self.base.as_u64() {
23         // the heap is already initialized, add 1 to exclude cur_end
24         cur_end_page += 1;
25     }
26     // heap: [base, new_end, new_end + 1) or [base, base)
27     let mut new_end_page = Page::containing_address(new_end);
28     if new_end != self.base {
29         // the new_end is not the base, add 1 to include new_end
30         new_end_page += 1;
31     }
32     debug!("Heap end addr: {:#x} -> {:#x}", cur_end, new_end.as_u64());
33     debug!(
34         "Heap end page: {:#x} -> {:#x}",
35         cur_end_page.start_address().as_u64(),
36         new_end_page.start_address().as_u64()
37     );
38
39     match new_end_page.cmp(&cur_end_page) {
40         core::cmp::Ordering::Greater => {
41             // heap: [base, cur_end, new_end) -> map [cur_end, new_end -
42             // 1]
43             let range = Page::range_inclusive(cur_end_page, new_end_page
44             - 1);
45             elf::map_range(range, mapper, alloc, true).ok()?;
46         }
47         core::cmp::Ordering::Less => {
48             // heap: [base, new_end, cur_end) -> unmap [new_end, cur_end
49             // - 1]
50             let range = Page::range_inclusive(new_end_page, cur_end_page
51             - 1);
52             elf::unmap_range(range, mapper, alloc, true).ok()?;

```

```
49     }
50     core::cmp::Ordering::Equal => {}
51 }
52
53 self.end.store(new_end.as_u64(), Ordering::Release);
54 Some(new_end)
55 }
```

在 Heap 中添加了 brk 函数，用于调整堆区的结束地址。该函数首先检查新的结束地址是否在堆区范围内，如果不在，则返回 None。然后计算当前结束地址和新的结束地址对应的页面，并根据它们的关系进行页面的映射或取消映射。

## 3 思考题

### 3.1 在 Linux 中删除运行中的程序文件

#### 3.1.1 会发生什么？程序能否继续运行？

当在 Linux 中删除一个正在运行的程序对应的可执行文件时：

- 会发生什么：rm 命令只是从目录中移除了文件名到 inode 的链接。文件系统会减少该 inode 的“链接数”。
- 能否继续运行：能。一个正在运行的进程是该文件 inode 的另一个“使用者”（通过打开的文件描述符）。只要进程还在运行，操作系统就会保持该 inode 和其占用的数据块不被释放。因此，程序可以继续执行内存中已经加载的代码，并按需从磁盘（现在已无文件名）加载新的页面。只有当程序的最后一个使用者（即进程退出）关闭后，并且链接数也为零时，文件系统才会真正回收该文件占用的磁盘空间。

#### 3.1.2 遇到未被映射的内存会发生什么？

这与删除程序文件没有直接关系，而是一个通用的内存管理问题。当程序试图访问一个在其虚拟地址空间中未被映射的地址时：

1. CPU 无法在页表中找到该虚拟地址对应的物理地址，触发一个页错误（**Page Fault**）异常。
2. CPU 将控制权转移给操作系统内核的页错误处理程序。

3. 内核检查该地址是否合法。对于一个未被映射的地址，它不属于任何合法的内存区域（如代码段、数据段、堆、栈）。
4. 内核判定这是一次非法的内存访问，会向该进程发送一个 SIGSEGV（Segmentation Fault）信号。
5. 如果进程没有自定义的信号处理程序来捕获 SIGSEGV，其默认行为是终止程序，并通常会生成一个核心转储（core dump）文件用于调试。

## 3.2 Arc::strong\_count 的设计

### 3.2.1 为什么要通过 Arc::strong\_count 获取引用计数？

Arc::strong\_count 是 Arc 提供的标准、安全的公共 API，用于获取其强引用计数。

### 3.2.2 它和一般使用 &self 的方法有什么不同？

Arc::strong\_count 的函数签名是 `pub fn strong_count(this: &Self) -> usize`，它确实接收一个 `&self`（即 `&Arc<T>`）作为参数。它与普通 `&self` 方法的关键不同在于其内部实现而非签名：

- Arc<T> 本身只是一个指向堆上 ArcInner<T> 结构的指针。这个 ArcInner 结构包含了元数据（强引用计数和弱引用计数）和数据 T。
- 所有克隆的 Arc 实例都指向同一个 ArcInner。
- 在多线程环境中，多个线程可能同时尝试读取或修改引用计数。为了防止数据竞争，对计数的访问必须是原子的。strong\_count 内部使用原子指令（如 `load`）并配合正确的内存排序（Memory Ordering）来安全地读取计数值，确保即使在其他线程正在修改计数值时，也能读到一个有效的值。

### 3.2.3 出于什么考虑不能直接通过 &self 来进行这一操作？

这里的“直接操作”应理解为“非原子操作”。如果 Arc 只是简单地允许通过常规的内存读取来访问计数值，将会导致灾难性的后果：

- 数据竞争：在一个线程读取计数值的同时，另一个线程可能正在增加或减少它。非原子读写可能导致读取到一个被部分修改的、无意义的中间值。
- 编译器优化问题：没有原子指令和内存栅栏（memory fence），编译器可能会自由地重排指令，导致程序逻辑错误。例如，编译器可能认为计数值没有改变而缓存了旧值。



因此，虽然方法签名是 `&self`，但其实现必须通过原子操作来保证线程安全，这是 Arc 设计的核心。

### 3.3 内核初始栈区大小

#### 3.3.1 至少需要多少页内存？

理论上，保证内核最基本的启动流程，至少需要 1 页内存（在 x86 架构上通常是 4KB）。

这个初始栈区由 bootloader 分配，用于内核执行其最开始的初始化代码（例如，设置基本的页表、初始化中断描述符表等），直到内核自己的内存管理器和任务调度器接管。

然而，1 页的栈空间非常小，很容易在稍微复杂的初始化函数调用中耗尽，导致栈溢出。因此，在实践中，为了安全起见，通常会分配一个稍大一些的栈区，例如 4 到 16 页（16KB 到 64KB），以确保在内核完全控制内存之前有足够的栈空间可用。

#### 3.3.2 增大内核栈区大小的观察与分析

- 观察：当逐渐增大内核栈区大小时，在某个临界点，内核可能无法正常启动，系统可能会直接重启（触发三重错误）或卡在启动阶段。
- 原因分析：最可能的原因是内存覆盖。Bootloader 在加载内核时，通常会将内核的各个段（如 `.text`, `.data`）和初始栈区连续地放置在物理内存中。

...   Kernel .text   Kernel .data   Initial Stack   ...
---

如果分配的栈区过大，它的末端地址就可能与 bootloader 加载的其他关键数据或内核代码段的起始地址重叠。当内核开始使用这个栈（向下增长）时，压栈操作就会破坏紧邻的内核代码或数据。当程序执行到被破坏的代码，或访问到被破坏的数据（例如页表、GDT/IDT），就会导致不可预测的行为，最常见的就是页错误或保护性异常。如果连异常处理程序本身都被破坏了，就会导致双重错误乃至三重错误，使系统重启。

- 提示分析：
  - 缺页中断依赖的子系统：中断处理机制（IDT）、页表管理器、物理帧分配器。
  - 报错：可能是 Page Fault，如果页错误处理程序被破坏，则会升级为 Double Fault 或 Triple Fault。

- 导致问题的子系统：最可能是 **bootloader** 的内存布局逻辑。它在分配过大的栈时，没有正确处理与其他内存区域的边界，导致了覆盖。

## 3.4 mmap, munmap, mprotect 系统调用

### 3.4.1 mmap

- 主要功能：mmap 用于在调用进程的虚拟地址空间中创建一个新的内存映射。这个映射可以关联到一个文件，也可以是匿名的（不关联任何文件）。
- 常见内存管理操作：
  1. 文件映射：将一个文件的全部或一部分内容映射到内存。之后对这块内存的读写就相当于对文件的读写。这是实现高性能文件 I/O 和进程间共享内存（IPC）的主要方式。
  2. 匿名映射：创建一个不与任何文件关联的、内容初始化为零的内存区域。这常被用作 malloc 的底层实现，用于分配大块内存。
  3. 设备映射：将硬件设备的内存（如显存）映射到进程空间，允许直接读写硬件。

### 3.4.2 munmap

- 主要功能：munmap 用于解除先前由 mmap 创建的内存映射。
- 使用时机：当进程不再需要某个内存映射区域时，应调用 munmap 来释放它。这会将对应的虚拟地址空间标记为可用，并减少相关资源的引用计数。对于文件映射，解除映射是确保修改被写回磁盘（如果需要）并释放文件句柄引用的重要步骤。忘记调用 munmap 会导致资源泄漏。

### 3.4.3 mprotect

- 主要功能：mprotect 用于修改一个已存在的内存映射区域的访问权限。
- 可实现的内存保护操作：
  - 权限设置：可以设置为只读 (PROT\_READ)、可写 (PROT\_WRITE)、可执行 (PROT\_EXEC)、不可访问 (PROT\_NONE) 或它们的任意组合。
  - JIT 编译器：即时编译器（JIT）可以先创建一块可读写的内存区域，将动态生成的机器码写入其中，然后调用 mprotect 将其权限修改为只读、可执行，以防止代码被意外修改并允许 CPU 执行。
  - 写时复制 (Copy-on-Write)：fork() 系统调用后，父子进程共享的页面可以被标记为只读。任何一方尝试写入时，会触发保护性页错误，内核此时才为写入方复制一个新的、可写的页面副本。

- 调试与安全：可以创建“哨兵页面”（Guard Page），将其权限设为 `PROT_NONE`。任何对该页面的访问都会立即触发段错误，可用于检测缓冲区溢出或栈溢出。

### 3.4.4 C 程序示例：使用 mmap 读写文件

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/mman.h>
4  #include <sys/stat.h>
5  #include <fcntl.h>
6  #include <unistd.h>
7  #include <string.h>
8
9  int main() {
10     const char *filepath = "mmap_test.txt";
11     const char *text = "Hello, mmap world!";
12     int fd;
13     struct stat sb;
14     char *mapped_mem;
15
16     // 1. 创建并写入一个测试文件
17     fd = open(filepath, O_RDWR | O_CREAT | O_TRUNC, (mode_t)0600);
18     if (fd == -1) {
19         perror("open");
20         exit(EXIT_FAILURE);
21     }
22     write(fd, text, strlen(text));
23
24     // 2. 获取文件大小
25     if (fstat(fd, &sb) == -1) {
26         perror("fstat");
27         exit(EXIT_FAILURE);
28     }
29
30     // 3. 将文件映射到内存
31     // PROT_READ | PROT_WRITE: 内存可读可写
32     // MAP_SHARED: 修改会写回文件，并对其他映射该文件的进程可见
33     mapped_mem = mmap(NULL, sb.st_size, PROT_READ | PROT_WRITE,
34                       MAP_SHARED, fd, 0);
35     if (mapped_mem == MAP_FAILED) {
36         perror("mmap");
37         exit(EXIT_FAILURE);
38     }
```

```
37     }
38
39     // 不再需要文件描述符，可以关闭
40     close(fd);
41
42     // 4. 读写映射的内存
43     printf("Original content: %s\n", mapped_mem);
44
45     // 修改内存内容
46     memcpy(mapped_mem, "Greetings", 9);
47     printf("Modified content: %s\n", mapped_mem);
48
49     // 5. (可选) 强制将修改同步到磁盘文件
50     if (msync(mapped_mem, sb.st_size, MS_SYNC) == -1) {
51         perror("msync");
52     }
53     printf("Content synced to disk.\n");
54
55     // 6. 解除映射
56     if (munmap(mapped_mem, sb.st_size) == -1) {
57         perror("munmap");
58         exit(EXIT_FAILURE);
59     }
60
61     return 0;
62 }
```

• 思考：文件内容什么时候会被写入到磁盘？

- 当调用 `msync()` 时，会强制将内存中的修改同步到磁盘。
- 当调用 `munmap()` 解除映射时，操作系统会负责将脏页（被修改过的页面）写回磁盘。
- 由操作系统内核的后台机制决定。Linux 内核会周期性地将脏页写回磁盘，但这个时机是不确定的。不能假设对内存的修改会立即反映在磁盘上。

### 3.5 mmap 受欢迎的原因及其优势

`mmap` 系统调用在现代操作系统中越来越受欢迎，因为它在内存、文件和 I/O 方面提供了显著的优势：

#### 1. 性能与效率：

- **消除内存拷贝：**传统 `read()/write()` 调用涉及数据在内核空间缓冲区和用户空间缓冲区之间的至少一次拷贝。mmap 通过让进程直接访问内核的页缓存 (Page Cache)，完全消除了这次拷贝，极大地降低了 CPU 开销和内存带宽消耗。
- **减少系统调用：**一旦文件被映射，后续的读写操作都变成了简单的内存访问，无需再陷入内核态，减少了上下文切换的开销。

## 2. 懒加载 (Lazy Loading):

- mmap 并不会在调用时立即将整个文件读入内存。相反，它只建立虚拟地址到文件块的映射关系。物理内存页只有在第一次被访问时，通过页错误机制，才会被真正从磁盘加载。这使得映射大文件时的启动速度极快，且内存使用更高效。

## 3. 统一的访问模型:

- mmap 将文件 I/O 抽象为内存访问，简化了编程模型。程序员可以使用指针运算、`memcpy` 等熟悉的内存操作函数来处理文件，比使用 `lseek`, `read`, `write` 管理文件偏移和缓冲区更方便。

## 4. 高效的进程间通信 (IPC):

- 多个进程可以映射同一个文件到各自的地址空间 (使用 `MAP_SHARED` 标志)，从而获得一块共享内存。这是最高效的 IPC 方式之一，因为一个进程写入的数据可以被其他进程立即看到，无需任何数据拷贝。