任务3

1.为什么将缓冲区对齐到系统的内存可能提高性能？你的实验结果支持这个猜想吗？为什么？

当内存访问跨越页面边界时，需要查找页表来更新TLB，如果缓冲区对齐且大小是页面大小的倍数，可以减少TLB未命中。

实验结果的提升并不大，只有10ms左右的提升，感觉是支持的，因为有提升但是提升比较小

2.为什么我们直接使用`malloc`函数分配的内存不能对齐到内存页，即使我们分配的内存大小已经是内存页大小的整数倍了？

malloc会在payoff地址前有inode信息，导致返回的指针不会与页边界对齐。

3.你是怎么在不知道原始的malloc返回的指针的情况下正确释放内存的？

使用了`mmap`来分配内存。

在返回指针前保存了原始指针`*((void **)(aligned_ptr - sizeof(void *))) = ptr;`

在align_free函数中`void *original_ptr = *((void **)((char *)ptr - sizeof(void *)));`

任务4

1. 为什么在设置缓冲区大小的时候需要考虑到文件系统块的大小的问题？

这样可以避免额外的IO操作 ，如果缓冲区跨越多个文件系统块，可能会导致多次独立的磁盘访问。而对齐后的缓冲区可以保证每次读取都尽可能多地包含有效数据。

2. 对于上面提到的两个注意事项你是怎么解决的

一：文件系统中的每个文件，块大小不总是相同的。

为每个打开的文件动态获取其所在文件系统的块大小：

- 在`io_blocksize`函数中传入文件名参数；
- 使用`statvfs(filename, &statbuf)`获取该文件所在文件系统的实际块大小；
- 根据`f_frsize`（根本块大小）优先原则，确保每个文件使用的缓冲区大小与其所在文件系统匹配。

这样就解决了不同文件可能位于不同文件系统、具有不同块大小的问题。

二：有的文件系统可能会给出虚假的块大小，这种虚假的文件块大小可能根本不是2的整数次幂。

合理性检查： 在获取到文件系统块大小后，加入了边界检查，防止出现异常值（例如小于512或大于合理上限的值）。

最小公倍数： 使用内存页大小和文件系统块大小的最小公倍数作为缓冲区大小，即使文件系统块大小不是2的幂，也能生成一个合适的缓冲区大小。

最大缓冲区大小限制： 设置了一个8MB的缓冲区大小上限，防止因为非2幂的块大小导致缓冲区异常大。

任务5

当缓冲区大小小于 `A × buf_size` 时，性能显著下降；

当缓冲区大小大于 `A × buf_size` 时，性能提升不明显。

最终将缓冲区设置为 `buf_size × A` 的固定值

变量参数：

`bs=BASE_SIZE × m`： `m` 是倍率数组 `[1, 2, 4, 8, 16, 32, 64, 128]`

`BASE_SIZE=4096`：初始缓冲区大小，通常是一个内存页大小

最后用output=($ddif = /dev/zeroof = /dev/nullbs$ =BUFFER_SIZE count=100000 2>&1)

模拟从 `/dev/zero` 读取数据并写入 `/dev/null` 的过程

输出对应速率

任务6

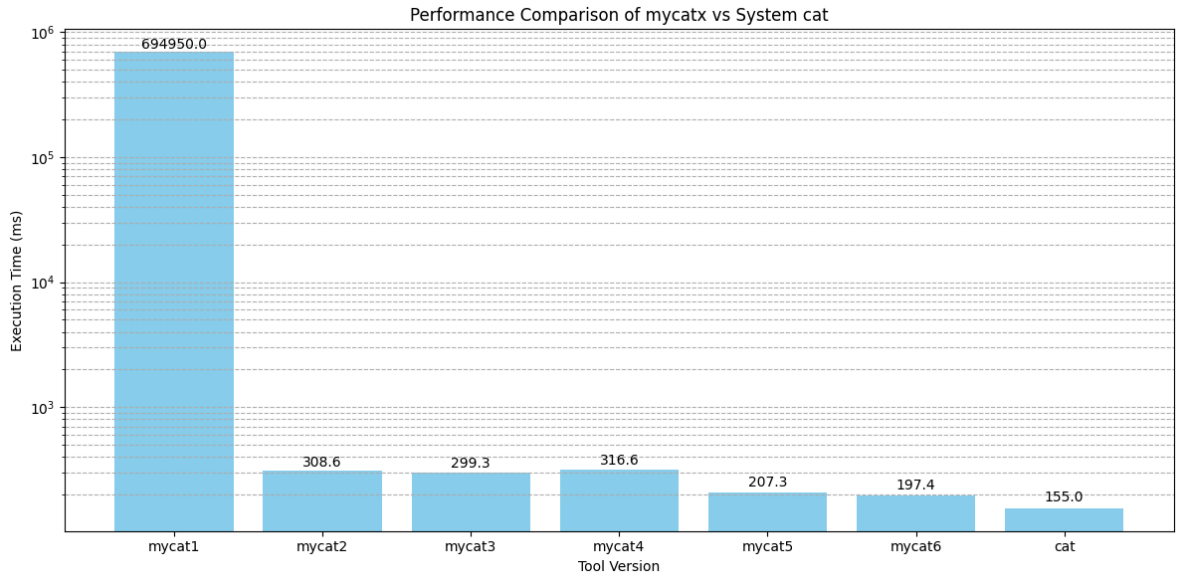
1. 如何设置 `fadvise` 的参数?

- `fd`: 文件描述符（通过 `open()` 获取）
- `offset`: `0`，表示从文件开头开始
- `len`: `0`，表示整个文件都适用该建议（传入 0 表示直到文件末尾）
- `advice`: `POSIX_FADV_SEQUENTIAL`，表示顺序读取文件

2. 对于顺序读写的情况，文件系统可以如何调整 readahead？对于随机读写的情况呢?

顺序读写，文件系统会增大 readahead 窗口大小，每次多读一些数据进内存缓存。

随机读写，减小或关闭 readahead



任务7

启示

1.缓冲对io性能的影响很大，有缓冲区

2.更加直观地感受到了从用户态切换到内核态的代价

通过逐步优化 `mycatx`，我们从最初比系统 `cat` 慢几十倍的版本，最终达到了接近系统命令的性能水平。这一过程不仅让我们理解了系统 IO 的工作原理，也展示了优化的重要性。