任务三

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？
   主要原因：可以将缓冲区更高效地映射或操作，减少页面分割，防止产生额外的零碎空间。同理，可以提高I/O的效率
   实验结果不太支持这个猜想。
   mycat3 的最低运行时间只比 mycat2 略有下降，但幅度不大，且平均运行时间几乎无变化。
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函数为了防止内存溢出，包括metadata的存在，实际分配的内存会比请求的大一点，并且起始地址可能不与页面边界对齐。
   同时，存在内存碎片的情况。
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？
   `void align_free(void* ptr) {`
   `if (ptr == NULL) return;`
   `// 从对齐指针中获取原始指针`
   `void* raw_ptr = *((void**)((char*)ptr - sizeof(void*)));`
   `free(raw_ptr);`
   `}`
   在`align_alloc`函数中，在对齐时预存了原始指针。在`align_free`函数中，利用对齐后的指针减去空闲大小（sizeof(void*)），得到预存的原始指针。

任务四

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？
   因为要使缓冲区大小和块大小匹配。如果缓冲区大小调整为 4KB，与块大小匹配，一次 read 操作即可完成，可以显著减少开销。如果数据未对齐，可能导致部分块未充分利用，浪费内存带宽。
2. 对于上面提到的两个注意事项你是怎么解决的？

```
size_t io_blocksize(const char* filename) {
 long page_size = sysconf(_SC_PAGESIZE);
 if (page_size == -1) {
     fprintf(stderr, "获取页面大小出错: %s\n", strerror(errno));
     page_size = 4096; // 退回到 4KB（不常见情况）
 }

 // 获取文件系统的块大小
 struct stat st;
 if (stat(filename, &st) == -1) {
     fprintf(stderr, "获取文件 %s 状态出错: %s\n", filename, strerror(errno));
     return (size_t)page_size; // 退回到页面大小
 }
 long fs_block_size = st.st_blksize;
 if (fs_block_size <= 0 || (fs_block_size & (fs_block_size - 1)) != 0) {
     fprintf(stderr, "无效的文件系统块大小 %ld，使用页面大小 %ld 替代\n",
fs_block_size, page_size);
     return (size_t)page_size; // 如果块大小无效或非 2 的幂，退回到页面大小
 }

 // 取页面大小和文件系统块大小的最大值
```

```
return (size_t)(page_size > fs_block_size ? page_size : fs_block_size);
}
```

对于注意事项1，通过stat动态获取当前块大小，可以自动适配最优块大小。如果 stat 失败返回-1，则使用内存页大小代替，确保兼容性。

对于注意事项2，首先确保`fs_block_size > 0`，为有效值；其次，使用位运算检查块大小是否为2的米次。如果不是，程序会以内存页大小作为替代。

任务五

1. 解释一下你的实验脚本是怎么设计的。你应该尝试了多种倍率，请将它们的读写速率画成图表包含在文档中。

   脚本设计：

   首先，设置4KB（即页大小）为初始缓冲区大小。

   `buf_size=4096`

   定义数组，设置为4KB的倍数，用于测试不同缓冲区大小对读写速率的影响。

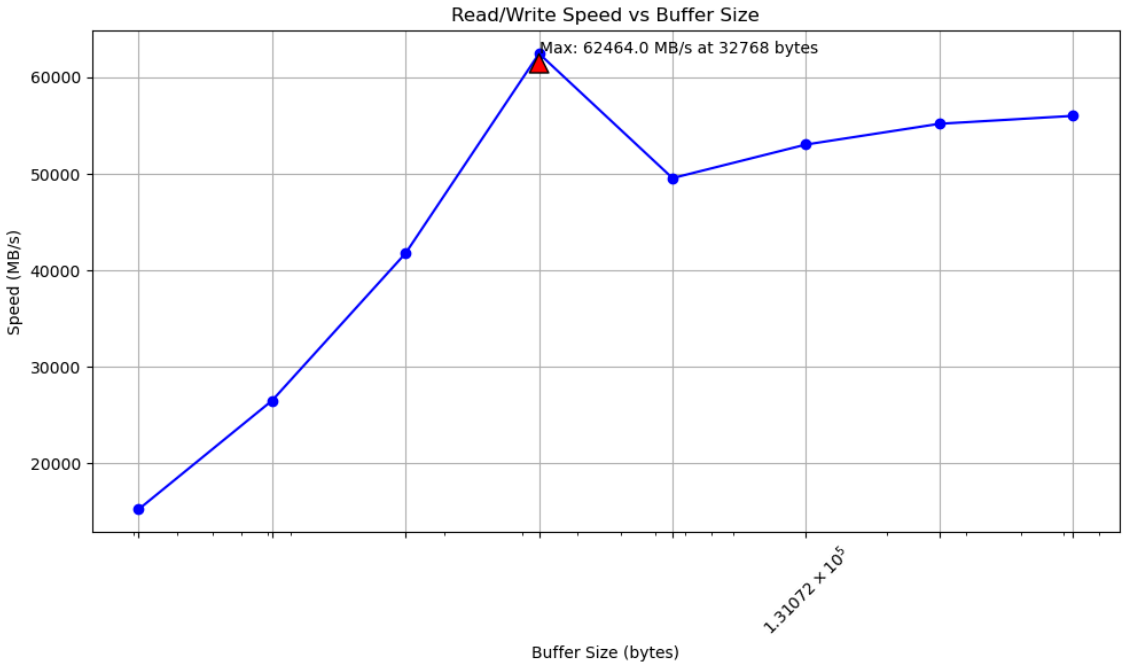
   `sizes=(1 2 4 8 16 32 64 128)`

   根据提示，使用 dd 命令模拟读写操作，将结果存储到 result 变量。

   利用`/dev/null`，`/dev/zero`隔离系统调用和内存复制开销，模拟 I/O 行为

   `result=$(dd if=/dev/zero of=/dev/null bs=$bs count=2000000 2>&1)`

2. 结果图像：



任务六

1. 你是如何设置`fadvise`的参数的?

   ```
   if (posix_fadvise(fd, 0, 0, POSIX_FADV_SEQUENTIAL) == -1) {
    fprintf(stderr, "设置 fadvise 出错: %s\n", strerror(errno));
    close(fd);
    exit(EXIT_FAILURE);
    }
   ```

   传入文件描述符fd，从文件开头开始，到文件末尾结束，按顺序读取文件。

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？

   文件系统检测到POSIX_FADV_SEQUENTIAL参数，会动态增加readahead，将后续数据块预加载到页面缓存。在随机读写时，文件系统可能会放弃 readahead，或者调整策略。

任务七

1. 你的全部实验结果的柱状图。
   由于mycat1运行时间过长，无法画出柱状图。
2. 你对上述实验结果的分析。
   在实验2中，实现了简单的带缓冲区的cat。其中，缓冲区的大小为内存页的大小。与实验1相比，显著提升了运行速度，从运行时间过长到37.97s。
   在实验3中，尝试将缓冲区对齐到系统的内存页，缓冲区的大小仍然为内存页的大小。在实现对齐后，运行的时间与改进前几乎相当。可以看出，malloc并不能如我们理想的那样实现完美的对齐。
   在实验4中，设置缓冲区大小为文件系统块大小的整数倍，缓冲区大小既考虑到内存页大小也考虑到文件系统的块大小。
   在实验5中，我尝试了不同大小的缓冲区，根据折线图找到最合适的32倍率。显著提高了cat的运行效率。该实验的假设是文件系统的大部分文件的块大小都一致，可知缓冲区大小为32倍页大小的时候效率最高。
   在实验6中，设置了`fadvise`的参数，开辟readahead预读窗口的优化策略。根据运行结果可知，该策略和实验5的优化结构几乎相同。