任务0

Benchmark 1: cat test.txt Time (mean ± σ): 426.9 ms ± 10.8 ms [User: 1.6 ms, System: 90.0 ms] Range (min … max): 413.6 ms … 448.2 ms 10 runs

任务1

跑了50分钟没跑出来；；；

后面尝试随机生成一个2MB的来跑，8s跑出来，说明代码没问题，应该是太大了

任务2

Benchmark 1: ./target/mycat2 test.txt Time (mean ± σ): 2.483 s ± 0.038 s [User: 0.045 s, System: 0.865 s] Range (min … max): 2.449 s … 2.572 s 10 runs

任务3缓冲区对齐的cat

实验结果：

Benchmark 1: ./target/mycat3 test.txt
  Time (mean ± σ): 2.411 s ± 0.018 s [User: 0.030 s, System: 0.506 s]
  Range (min … max): 2.387 s … 2.446 s 10 runs

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？

页面对齐的缓冲区匹配内存页边界（通常 4KB），可提高 CPU 缓存命中率，减少缓存未命中。优化 TLB 效率，降低页面转换开销。

实验结果mycat3 (2.411 秒) 比 mycat2 (2.483 秒) 快 0.072 秒。

支持猜想：页面对齐略微提升性能。但是提升有限。4KB 缓冲区较小，对齐收益不显著。

2.**为什么直接使用 malloc 函数分配的内存不能对齐到内存页，即使分配的内存大小是内存页大小的整数倍?**

malloc保证内存对齐到 8 或 16 字节（适合通用类型），但不保证页面对齐（例如 4096 字节）。即使分配 4096 字节，地址可能不是 4096 的倍数。

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 posix_memalign 分配内存，返回的指针直接存储并传递给 free。

posix_memalign 是一个 POSIX 标准的 C 库函数，用于分配动态内存，并确保返回的内存地址对齐到指定的边界。它特别适用于需要高性能 I/O 或硬件优化的场景。（学习自AI）

任务4

实验结果：
Benchmark 1: ./target/mycat4 test.txt
  Time (mean ± σ): 2.439 s ± 0.021 s [User: 0.023 s, System: 0.370 s]
  Range (min … max): 2.412 s … 2.482 s 10 runs

为什么在设置缓冲区大小的时候需要考虑文件系统块大小的问题?

- 文件系统以块为单位进行读写操作。如果缓冲区大小不是块大小的倍数，可能导致碎片化 I/O，读取不完整的块。

对于上面提到的两个注意事项你是怎么解决的?

不一致

- 动态获取输入文件（test.txt）的 st_blksize，确保缓冲区大小针对具体文件。
- 如果 stat 失败，回退到页面大小4096。

虚假块大小

验证 st_blksize是否合理：

- 范围限制在 512-65536 字节（常见块大小）。
- 检查是否为 2 的幂：(blksize & (blksize - 1)) == 0。
- 如果块大小无效（非 2 的幂或超出范围），回退到 4096，并打印警告。

任务5

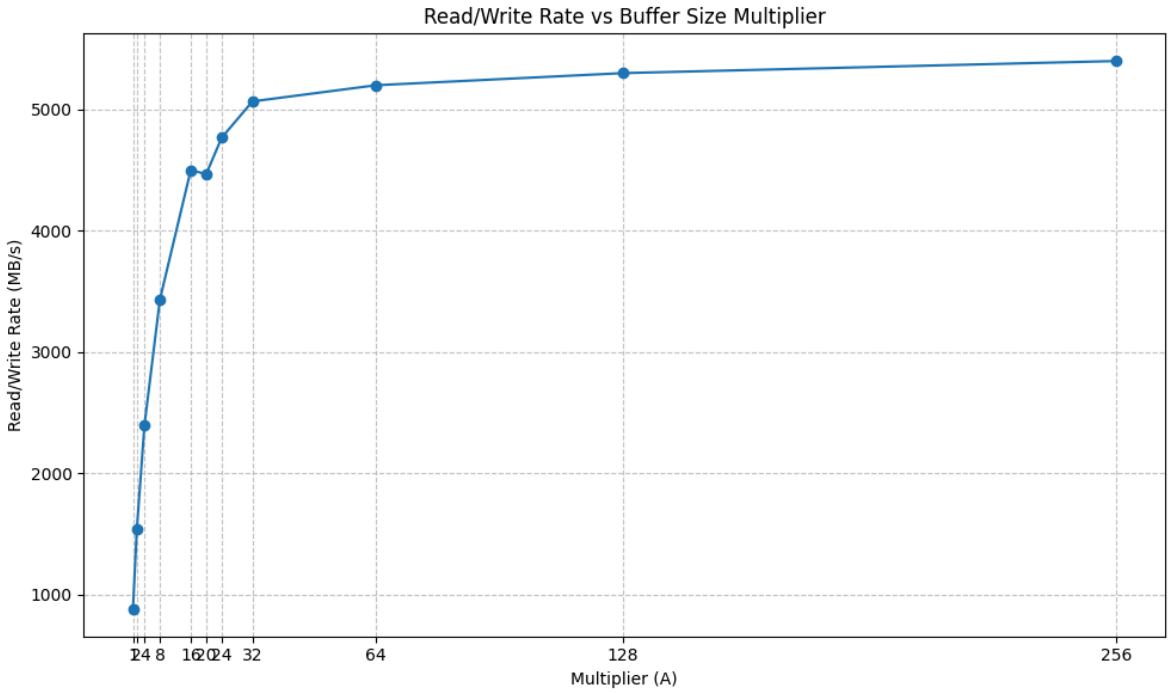
实验结果：
Benchmark 1: ./target/mycat5 test.txt
  Time (mean ± σ):   512.3 ms ± 10.6 ms   [User: 0.0 ms, System: 69.8 ms]
  Range (min … max):   501.6 ms … 533.2 ms   10 runs

1.测试脚本test_buffer.sh:
设计的思路主要是倍增和二分。先测试A=1,2,4,8,16——256等2的倍数

Multiplier	BufferSize	Rate_MBps
1	4096	879
2	8192	1533.33
4	16384	2400
8	32768	3433.33
16	65536	4500
20	81920	4466.66
24	98304	4766.66
32	131072	5066.66
64	262144	5200
128	524288	5300
256	1048576	5400

发现

A=2——128时rate都在增加，256反而减小。从32开始增加得非常不明显，于是在16和32之间二分，最终确定为16比较合适。



任务6

结果：

Benchmark 1: ./target/mycat6 test.txt
  Time (mean ± σ):   612.3 ms ± 17.9 ms   [User: 7.8 ms, System: 240.3 ms]
  Range (min … max):   588.2 ms … 645.6 ms   10 runs

1. 你是如何设置 fadvise 的参数的?

   使用 posix_fadvise(fd, 0, 0, POSIX_FADV_SEQUENTIAL)。

   fd：test.txt 的文件描述符。

   offset=0, len=0：应用于整个文件（0 表示全文件）。

   POSIX_FADV_SEQUENTIAL：提示顺序读取。

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

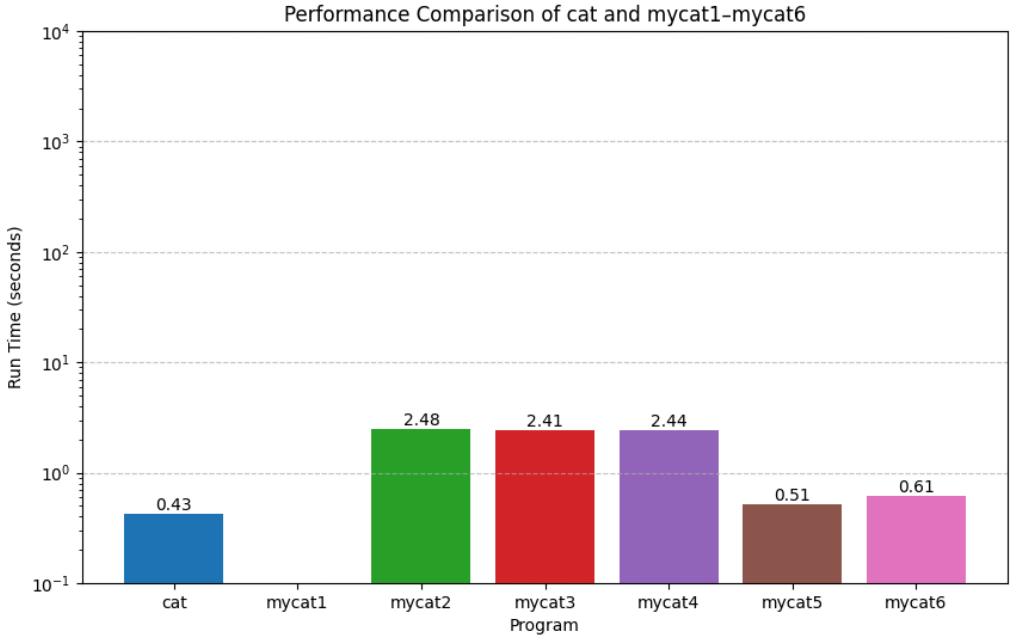
   顺序读写

   文件系统增大 readahead 。

   随机读写

   使用 POSIX_FADV_RANDOM，减小或禁用 readahead。

任务7



这里因为cat1没跑出来，所以就让mycat1的结果置为0。

实验结果分析

结果是否符合预期?

预期：

cat 应最快。

mycat1 最慢。

mycat2–mycat6 逐步优化，时间递减。

实际结果：**总体符合**：从 mycat1 到 mycat6，性能逐步提升，反映了缓冲区大小、页面对齐、块大小和预读的优化效果。

- **cat**：0.4269 s，最快，符合预期。
- **mycat1**：跑很久没跑出来，最慢。
- **mycat2**：2.483 s，4KB 缓冲区。
- **mycat3**：2.411 s，页面对齐提升0.072 s，幅度小。
- **mycat4**：2.439 s，略慢于 mycat3，不符合预期。
- **mycat5**：0.5123 s，比 mycat2 快 ~4.8 倍，符合任务 5 实验（A=16，4500 MB/s）的预期。
- **mycat6**：0.6123 s，比 mycat5 慢0.1 s，略不符合预期，但仍接近 cat，反映 posix_fadvise 的预读优化。

异常点：

mycat4 未优于 mycat3：文件系统块大小（512 字节）回退到 4096，未显著改变缓冲区行为。

mycat6 略慢于 mycat5可能原因：

- 系统负载（笔记本可能内存有限）。
- 测试波动（mycat6 的 17.9 ms 误差大于 mycat5 的 10.6 ms）。

实验结果的启示

- 缓冲区大小是 I/O 性能核心
- 页面对齐和块大小优化有限
- 预读优化显著
- 用户态 vs 内核态

  cat（0.4269 s）优于 mycat6（0.6123 s），反映内核级 I/O的高效。

  用户态程序需综合缓冲区大小、内存对齐和预读优化才能接近内核性能。
- 系统性编程

  健壮性（如错误处理、posix_memalign、posix_fadvise 的警告）确保程序可靠。

  动态适配（页面大小、块大小）提高代码通用性。