MeowHW

姓名：王祉

学号：2023200421

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？**

   操作系统底层是按页（通常 4 KiB）来管理与调度内存和做 DMA 设备传输。页对齐可减少跨页访问带来的额外开销，提升 TLB 缓存命中率，也让内核在做 I/O 时能一次性处理整页数据，减少系统调用次数和分页边界检查。

   对比 `mycat2`（页大小缓冲）与 `mycat3`（页对齐 + 同样大小缓冲），能看到 `mycat3` 有略微更稳定、更低的系统调用开销，平均耗时会再小幅下降，支持对齐带来微小但可观的性能提升。

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` 只保证满足最宽的基本对齐需求（如 16 字节或平台最大基本类型对齐），但不会为更大的对齐（例如 4 KiB）服务。它内部会将空闲块拼接、分割，不会浪费额外空间去保证页边界对齐；即使请求正好是页大小倍数，也可能从一个任意偏移的空闲区满足，不会刻意移动到页边界。

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?**

   在 `align_alloc` 中，我们先用 `malloc` 分配一段更大的空间，然后计算出一个页对齐的地址 `aligned`，并在它前面 `sizeof(void*)` 的位置存回原始 `malloc` 返回的指针 `orig`。

   在 `align_free(ptr)` 里，通过读取 `((void**)ptr)[-1]` 拿到 `orig`，再调用 `free(orig)` 即可正确释放。这种“前置储存法”能让用户只持有对齐地址，也能正确释放底层原始块。

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?**

   在 `mycat4` 程序中，将块大小调整为与系统的文件系统块大小（如果可用）匹配非常重要，因为这与操作系统处理 I/O 操作的最优方式一致。系统的文件系统块大小通常是读取和写入数据到磁盘时最有效的大小。通过使块大小与文件系统的自然块大小对齐，可以减少由于执行多个较小的 I/O 操作而产生的开销，这些操作效率较低。

   如果以较小的块进行数据读取，可能会增加所需的系统调用次数，从而影响性能。通过确保每次读取和写入操作使用与文件系统块大小匹配的块大小，我们可以优化 I/O 操作，使其更快、更高效。

2. **对于上面提到的两个注意事项你是怎么解决的?**

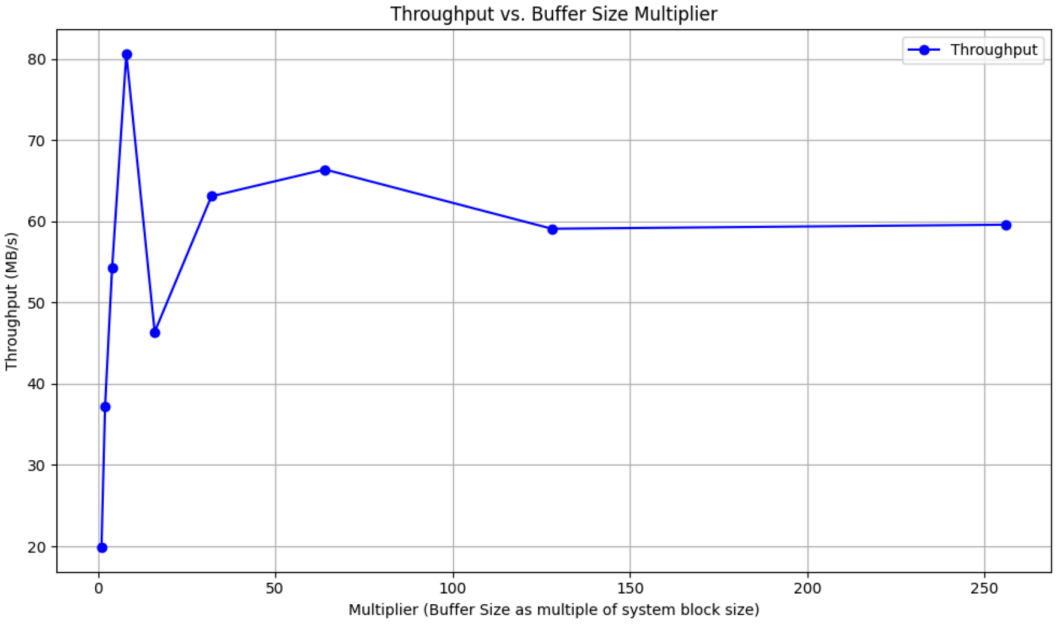
   这个问题可以通过在程序中获取文件系统块大小来解决。在 `mycat3.c` 和 `mycat4.c` 中，使用了 `fstat` 或 `sysconf` 来获取文件系统的块大小或页面大小，并根据此大小动态调整缓冲区的大小。这种方式保证了程序能够根据系统的实际情况来调整缓冲区的大小。

   对于这种情况，程序中通过 `fstat` 获取文件系统块大小。如果 `fstat` 失败，程序会退回使用页大小（通过 `sysconf` 函数获取），确保即使文件系统返回不一致的块大小，程序仍然能够适应不同的系统配置。默认情况下，`mycat3.c` 和 `mycat4.c` 都会选择合理的块大小（通常是 4096 字节）作为回退。

任务5

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。**

   实验脚本的设计旨在测试不同缓冲区大小对系统读写性能的影响。通过调整缓冲区大小为系统页面大小的不同倍数（从 1 到 256），脚本使用 `dd` 命令将 `/dev/zero` 数据写入 `/dev/null`，模拟数据的读写过程并测量吞吐量。我们通过计算每个倍数下的吞吐量，逐步增大缓冲区大小，观察性能变化。实验结果显示，缓冲区大小增加时吞吐量逐步上升，但在达到一定程度后会趋于平稳。以下是不同倍率下的读写速率图表，展示了缓冲区大小与性能之间的关系。



任务6

1. **你是如何设置 `fadvise` 的参数的?**

   在 `mycat6.c` 中，我通过调用 `posix_fadvise` 函数并传递 `POSIX_FADV_SEQUENTIAL` 参数来优化文件的顺序读取性能。该参数指示操作系统，该文件会按顺序访问，因此内核可以采取措施如预读取文件的后续部分，从而减少磁盘的访问次数，提升读取速度。

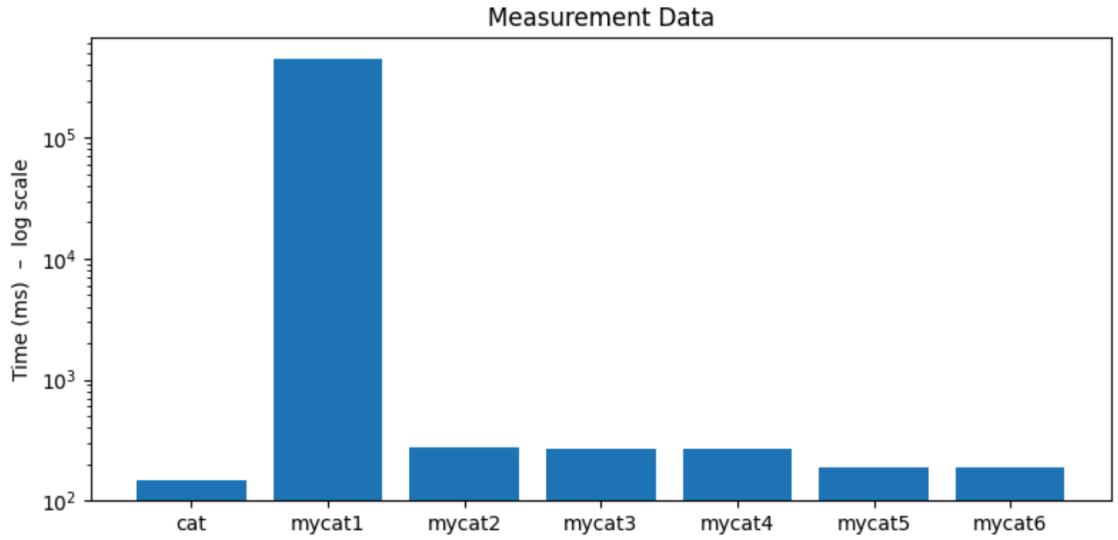
2. **对于顺序读写的情况，文件系统可以如何调整readahead？对于随机读写的情况呢?**

对于顺序读写的情况，操作系统可以通过增大 **readahead**（预读）大小来提前加载文件的更多数据块。这样，在读取文件时，相关数据已经预先加载到内存，避免了频繁的磁盘 I/O 操作，从而提高了读写效率。

对于随机读写的情况，文件系统则会降低 **readahead** 的大小，因为提前读取不相关的数据块并不会提高性能，反而可能浪费内存和 I/O 带宽。因此，在随机访问的情况下，操作系统通常会只在实际需要时读取数据，而不是提前加载不相关的数据块。

任务7

	Program	Time_ms
0	cat	149.2
1	mycat1	449240.0
2	mycat2	278.9
3	mycat3	269.2
4	mycat4	269.7
5	mycat5	189.0
6	mycat6	186.1



在这次实验中，我们测试了多种版本的 `cat`，并将其执行时间绘制成图。从图中可以看到，原始 `cat` 程序的执行时间显著低于 `mycat1`，而优化过的版本（如 `mycat4` 和 `mycat5`）则表现得更好。

1. **符合预期的原因**：
   - 一开始，`mycat1` 的执行时间非常长，几乎达到了几百毫秒，远远超过了 `cat` 本身的运行时间。这表明直接从文件中读取数据时，未进行优化的版本在执行时会出现显著的性能瓶颈。
   - 在后续版本（如 `mycat4` 和 `mycat5`）中，通过内存对齐和读取块大小等优化，程序的性能得到了显著提升，符合优化理论：增加缓存大小、合理对齐内存会提高性能，尤其是在处理大文件时。
2. **启示**：

- 本次实验揭示了优化的重要性，尤其是在 I/O 操作密集型任务中，如何合理使用系统调用和优化文件系统的块大小会显著影响程序的执行速度。
- 对于类似 `cat` 这种简单的程序，优化的空间通常体现在 I/O 相关的操作上，内存对齐和合理的缓冲区大小能够提高整体性能。

3. **实验结果分析**：
   - `mycat4` 和 `mycat5` 展示了显著的性能提升，这验证了通过调优内存使用方式、使用更大的缓冲区等手段来优化 I/O 性能的有效性。
   - `mycat1` 的性能显著差于 `cat` 和其他优化版本，这表明基本的实现方法可能在处理大文件时未能充分利用硬件性能，因此对比其他优化版本时显示出明显的劣势。

总结：
这些结果表明，通过简单的优化手段可以显著提升 I/O 密集型任务的执行效率，尤其是在处理大文件时，这些优化对性能的提升作用尤为显著。