MEOWLAB报告

马亚辛 2023202293

任务1

代码主要逻辑如下

```
if(argc < 2){
    ....
    char buffer[1];
    ssize_t byte_read;
    while((byte_read = read(fd, buffer, 1)) > 0){
        printf("%s", buffer);
    }
```

通过while循环读入在输出

不过这里运行时间实在是太长没有完整运行完毕

任务2

任务二主要就是使用sysconf(_SC_PAGESIZE)函数获取内存页大小后，每次读入一个页面大小的buffer后写入到标准输入，主要核心代码如下

```
   long io_blocksize(){
    long page_size = sysconf(_SC_PAGESIZE);
    return page_size;
}
    ....

    long size = io_blocksize();
    char* buffer = malloc(size);
    ssize_t byte_read;
    while((byte_read = read(fd, buffer, size)) > 0){
        if (write(STDOUT_FILENO, buffer, byte_read) == -1) {
            perror("error writing to stdout");
            close(fd);
            return 1;
        }
    }
    free(buffer);
    close(fd);
```

得到输出结果自然是快了不少!

```
%%bash
hyperfine --warmup 3 './target/mycat2 test.txt'
```

[14] ✓ 6.3s

```
Benchmark 1: ./target/mycat2 test.txt
  Time (mean ± σ):     349.6 ms ±  16.6 ms    [User: 48.6 ms, System: 301.0 ms]
  Range (min … max):   338.3 ms … 395.4 ms    10 runs
```

任务3

实现两个函数char* align_alloc(size_t size)和void align_free(void* ptr)来代替malloc和free，具体代码如下：

```
char* origin_ptr;
char* align_alloc(size_t size) {
    size_t page_size = sysconf(_SC_PAGESIZE);  //获取页大小
    size_t total_size = size + page_size - 1;  //总共分配的大小
    char* ptr = (char*)malloc(total_size);
    origin_ptr = ptr;   //全局变量记录原本的指针
    if (ptr == NULL) return NULL;
    char* aligned_ptr = (char*)(((unsigned long long)(ptr + page_size - 1)) & ~
(page_size - 1));    //获取对齐后的指针
    return aligned_ptr;
}

void align_free(void* ptr) {
    free(ptr);
}
```

```
%%bash
hyperfine --warmup 3 './target/mycat3 test.txt'
```

[15] ✓ 4.1s

```
Benchmark 1: ./target/mycat3 test.txt
  Time (mean ± σ):     299.7 ms ±  10.7 ms    [User: 45.9 ms, System: 253.7 ms]
  Range (min … max):   284.8 ms … 317.9 ms    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？

   当缓冲区是页对齐的时候，可以降低不必要的内存拷贝。当不是对齐的时候，数据可能要先写入内核的页对齐缓冲区，之后再写入用户非对齐的缓冲区，增加的拷贝次数，同时，再write时也需要访问多个page。而如果是对齐的情况，数据可以直接写入对齐的缓存区，write时也可以只访问一个缓冲区。

   实验支持这个猜想。时间明显减少了。

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。

   这与malloc的实现有关。为了内存占用率和效率，malloc的实现仅仅满足字对齐而不是页对齐，即使我们分配的内存大小已经是内存页大小的整数倍。

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？

   我这里是直接使用了一个全局变量记录，因为在程序中实际只进行了一次malloc，所以这个方法不会失效。一个更通用的方法应该是将原本的指针记录在分配的空间中，比如aligned_ptr的前八个字节中。

任务4

任务四主要是在设置缓冲区大小时考虑块大小

```
long io_blocksize(char* filename){
    long size = 0;
    long page_size = sysconf(_SC_PAGESIZE);
    struct statfs buf;
    if (statfs(filename, &buf) == -1) {
        perror("Error getting filesystem info");
        return -1;
    }
    if(buf.f_bsize % 2 == 0){
        size = buf.f_bsize;
    }
    return size;
}
```

```
%%bash
hyperfine --warmup 3 './target/mycat4 test.txt'
```

✓ 3.9s

```
Benchmark 1: ./target/mycat4 test.txt
  Time (mean ± σ):     297.8 ms ±   4.4 ms    [User: 41.3 ms, System: 256.5 ms]
  Range (min … max):   291.3 ms … 305.8 ms    10 runs
```

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?

   因为我们在操作文件时是以文件系统的块大小为单位进行处理的，一次处理一个块。设置与文件系统块大小适应的缓冲区有利于减少不必要的块访问，提高io速度。

2. 对于上面提到的两个注意事项你是怎么解决的?

   文件系统的每个文件块大小不总是相同：只需要在处理前得到该文件的块大小即可，也就是不固定块大小。

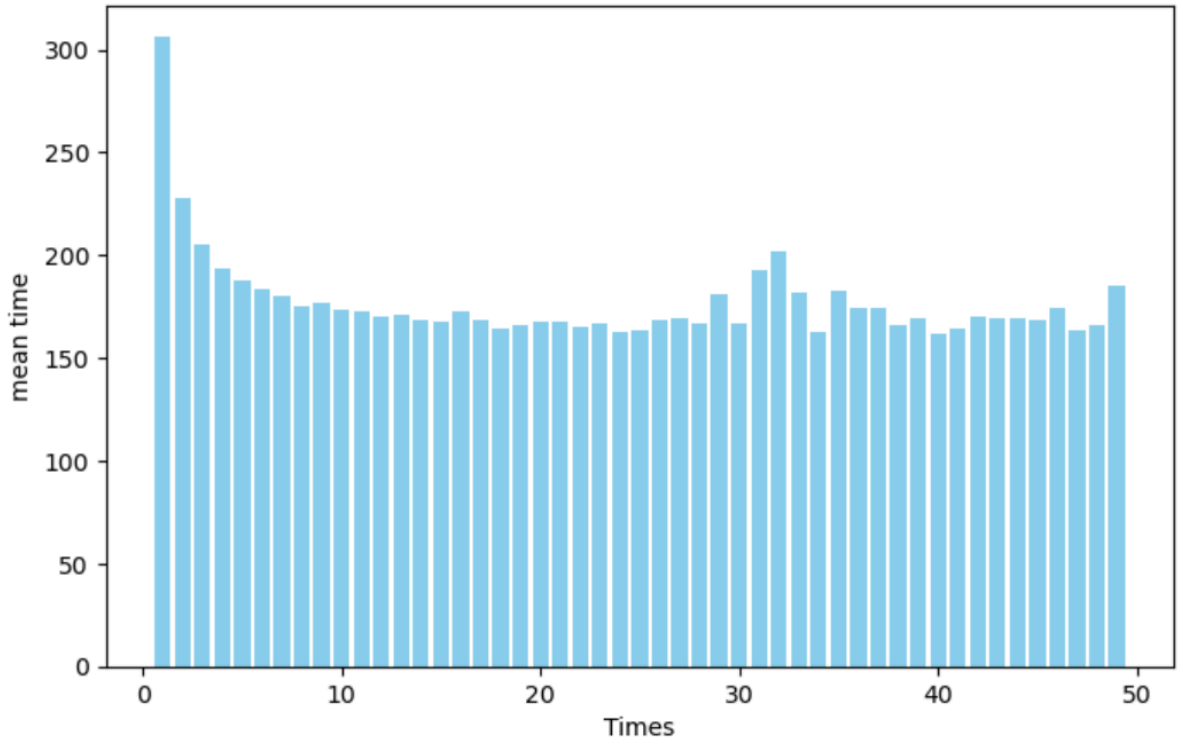
   有的文件系统可能会给出虚假的块大小，这种虚假的文件块大小可能根本不是2的整数次幂： 在使用这个块大小前检测其是否是2的整数倍，如果不是就不使用了。

任务5

脚本通过subprocess包依次处理了1-49倍率，通过正则表达式提取其平均时间，储存在result1 list中，之后再通过matplotlib绘制柱状图，脚本具体代码如下。

```
import subprocess
import re
import matplotlib.pyplot as plt
result1=[]
with open("output.txt", "w", encoding="utf-8") as f:
    for x in range(1, 50):
        print(f"正在处理 x = {x}")
        command_to_run = f"hyperfine --warmup 3 './target/mycat5 test.txt {x} >
/dev/null'"    #重定向使其输出到/dev/null
        result = subprocess.run(
```

```
            command_to_run,
            shell=True,
            capture_output=True,
            text=True,
            check=True
        )
        regex_pattern = re.compile(r"Time \(mean ± σ\):\s*([\d.]+)\s*ms.*")  #正则
表达式提取使平均时间
        match = regex_pattern.search(result.stdout)
        mean_time_str = match.group(1)
        mean_time_value = float(mean_time_str)
        result1.append(mean_time_value)
x = range(1,50) #绘图
plt.figure(figsize=(8, 5))
plt.bar(x, result1, color='skyblue')
plt.xlabel('Times')
plt.ylabel('mean time')
plt.title('')
plt.show()
```



可以看到，大概前十五次下降较快，之后只有一些波动。

任务六

主要就是加了一句 `posix_fadvise(fd, 0, 0, POSIX_FADV_SEQUENTIAL);`

1. 你是如何设置`fadvise`的参数的?

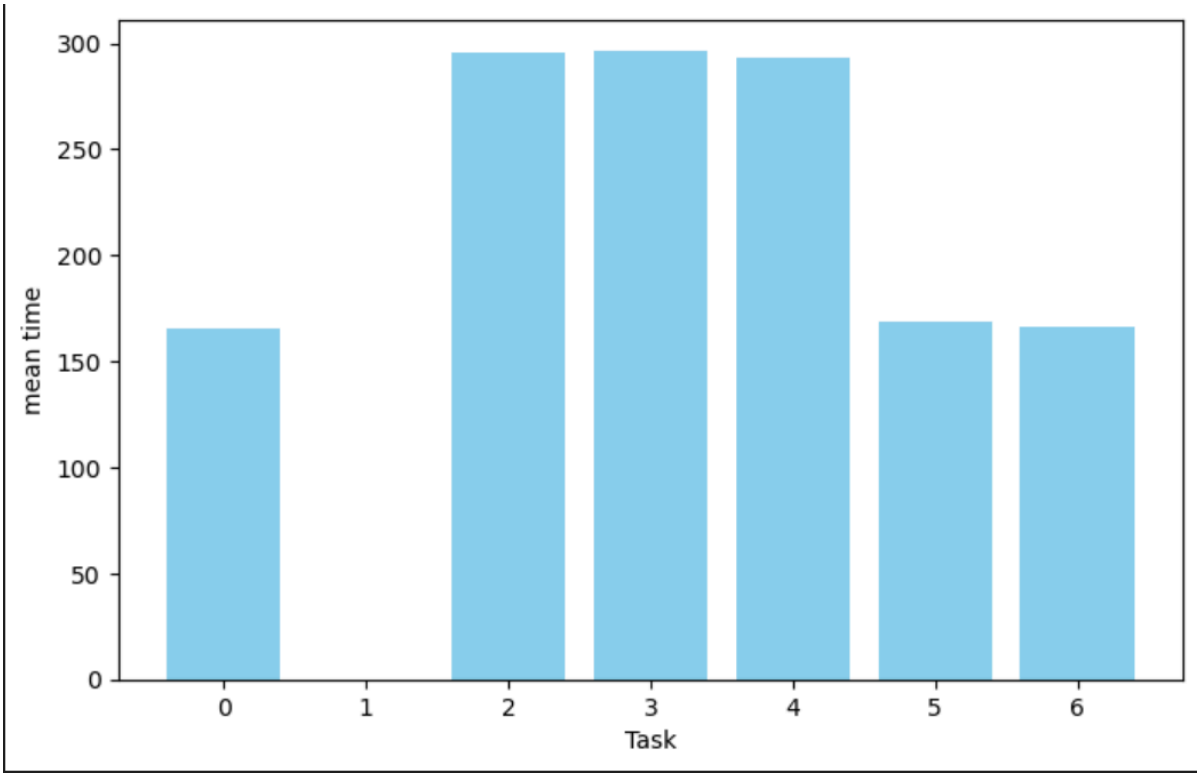
   因为是顺序读这个文件，所以设置为(fd, 0, 0, POSIX_FADV_SEQUENTIAL);

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?

   增加预读的量，把预读的数据放进cache里之后可以直接利用。同时也可以提前进行预读，再文件还没打开但是接受到建议后便进行预读，加快第一次访问。

   如果是随机读写应该取消预读，因为随机访问是不确定的，预读很可能没有用处。

任务七



从图中可以看到，当没有扩大缓冲区倍数时平均时间还是较高，趋于300ms左右，也就是系统调用的开销还是比较高，当扩大缓冲区倍数后并且使用fdavise后，可以明显有效的降低平均时间，与cat的具体实现已经差不多。