华东师范大学数据学院上机实践报告

课程名称：操作系统　　**年级**：2019 级　　**上机实践成绩**：

指导教师：翁楚良　　**姓名**：龚敬洋

上机实践名称：I/O 子系统　　**学号**：10195501436　　**上机实践日期**：2021/6/1

上机实践编号：

一、 目的

在 Minix3 中创建一块可用的 RAM 盘，并比较其与 DISK 盘在各类存取方式下的速度。

二、 内容与设计思想

1. 在 Minix3 中安装一块 X MB 大小的 RAM 盘(Minix 中已有 6 块用户可用 RAM 盘，7 块系统保留 RAM 盘)，可以挂载并且存取文件操作。
2. 测试 RAM 盘和 DISK 盘的文件读写速度，分析其读写速度差异原因。

三、 使用环境

开发环境：Clion 2020.3

宿主机系统环境：Mac OS Big Sur 11.2.3

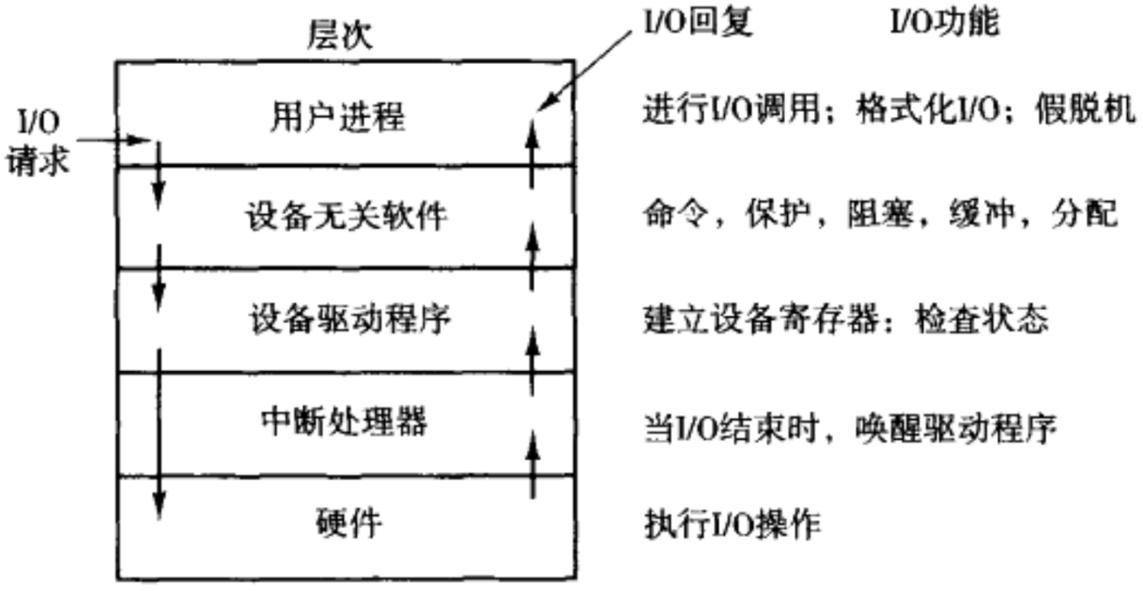
虚拟机应用：VirtualBox 6.1.18

虚拟机环境：Minix 3.3.0

四、 实验过程

1. Minix3 的存储管理策略

与其整体系统构架类似，Minix3 的 I/O 构架分为 5 层：用户进程层、资源调度层、设备驱动层、内核中断层及硬件层。



对于磁盘来说，其通常以块为单位进行存储。当一个用户程序要从一个文件读一个块时，操作系统首先在高速缓存中查找有关的块。如果需要的块不在其中，则调用设备驱动程

序，向硬件发出一个请求，从磁盘读取该块，然后将进程阻塞。当磁盘操作完成时，硬件产生一个中断，中断处理器随即从设备读取状态并唤醒休眠的的用户进程使其能够继续运行。

2. RAM 盘申请

RAM 盘是将主存中的部分空间当作普通磁盘来使用的一种存储模型。在许多场景下，这种使用方式是高效且重要的（尤其是在由外部设备引导的系统下）。Minix3 系统中共有 6 块固有的 RAM 盘，其设备控制程序分别被挂载在/dev/ram，/dev/kmem，/dev/boot，/dev/mem，/dev/null 和/dev/zero 下。

为了增加一块 RAM 盘，我们首先修改这一 RAM 盘常量：

（minix/drivers/storage/memory）

```
/* ramdisks (/dev/ram*) */
#define RAMDISKS     7
```

Minix 本身提供了一个用于创建 RAM 盘的 **ramdisk** 指令，但其单位为 KB。为了方便起见，我们实现一个单位为 MB 的 **buildmyram** 指令用于创建较大容量的 RAM 盘：

（minix/commands/ramdisk）

```
#include <minix/paths.h>
#include <sys/ioc_memory.h>
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int fd;
    signed long size;
    char *d;

    if(argc < 2 || argc > 3) {
        fprintf(stderr, "usage: %s <size in MB> [device]\n",
                argv[0]);
        return 1;
    }

    d = argc == 2 ? _PATH_RAMDISK : argv[2];
    if((fd=open(d, O_RDONLY)) < 0) {
        perror(d);
        return 1;
    }

#define KFACTOR 1024
    size = atol(argv[1])*KFACTOR*1024;

    if(size < 0) {
        fprintf(stderr, "size should be non-negative.\n");
        return 1;
    }

    if(ioctl(fd, MIOCRAMSIZE, &size) < 0) {
        perror("MIOCRAMSIZE");
        return 1;
    }

    fprintf(stderr, "size on %s set to %ldMB\n", d, size/KFACTOR/1024);

    return 0;
}
```

修改完必要的内核代码后，我们重新编译系统并重启进入。现在，我们就可以来实际在系统中申请 RAM 盘了。

与系统固有盘类似，我们首先使用 mknod 指令创建一个新申请 RAM 盘的设备控制节点：

```
# mknod /dev/myram b 1 13
# ls /dev/ | grep ram
myram
ram
ram0
ram1
ram2
ram3
ram4
ram5
```

随后，我们使用新实现的 buildmyram 指令申请一块大小为 500MB 的 RAM 盘：

```
# buildmyram 500 /dev/myram
size on /dev/myram set to 500MB
```

最后，我们在新申请的 RAM 盘上创建相应的文件系统，并将其挂载到/root/myram 目录下即可：

```
# mkfs.mfs /dev/myram
# mount /dev/myram /root/myram
/dev/myram is mounted on /root/myram
# df
Filesystem    512-blocks       Used      Avail %Cap Mounted on
/dev/myram       1024000      16088    1007912   1% /root/myram
/dev/c0d0p0s0     262144      72192     189952  27% /
none                   0          0          0 100% /proc
/dev/c0d0p0s2   33558432    4581504   28976928  13% /usr
/dev/c0d0p0s1    8112128      84944    8027184   1% /home
none                   0          0          0 100% /sys
```

通过 df 指令可以看到，RAM 盘已被成功创建。

3. 读写性能测试

接下来，我们需要编写一组用于测试和比较 DISK 盘和 RAM 盘读写性能的程序。由于 DISK 盘和 RAM 盘使用了同样的抽象模型，我们可以使用相同的逻辑来对其进行测试。

一块磁盘在使用过程中主要会遇到以下四种读写模式：**顺序读取、随机读取、顺序写入、随机写入**。对于读取操作，我们首先使用 open 系统调用打开相应的文件，随后使用 read 系统调用将文件中固定大小的内容读入缓存中。若为随机读取，则在读取完成后我还需要使用 lseek 和 rand 函数将文件指针重新指到一个随机的位置。此外，为了产生较为显著的运行时间以方便比较，我们在一次操作中重复读取 1000 轮：

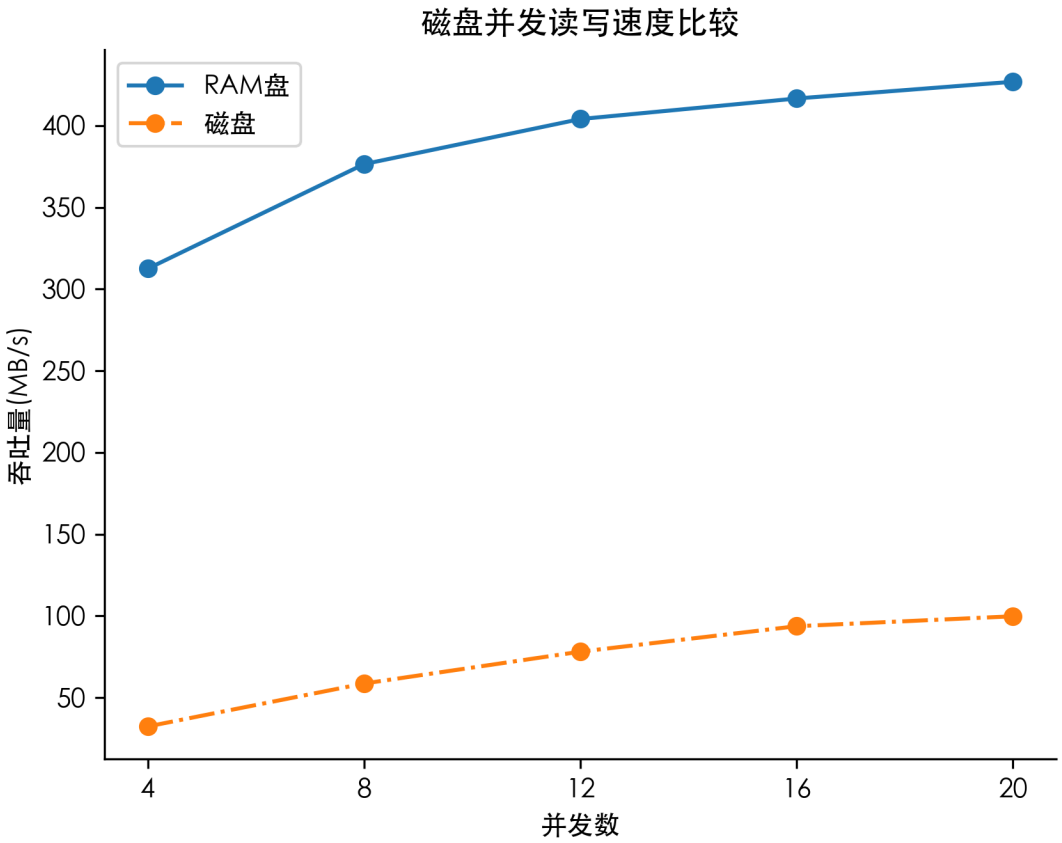
```
1.  #define ROUND 1000
2.
3.  void read_file(int blocksize, bool isrand, char *filepath){
4.      int fd = 0;
5.      fd = open(filepath, O_CREAT | O_RDWR | O_SYNC, S_IRWXU);
6.      if(fd < 0){
7.          fprintf(stdout, "Error occurred when opening file!");
8.          return;
```

```
    }
    char *buf_ext = (char *)malloc(sizeof(char) * blocksize);
    for(int i = 0; i < ROUND; i++){
        read(fd, buf_ext, blocksize);
        if(isrand){
            lseek(fd, rand() % ((blocksize - 1) * ROUND), SEEK_SET);
        }
    }
    free(buf_ext);
    lseek(fd, 0, SEEK_SET);
    close(fd);
}
```

写入操作与读取操作类似。我们首先构造一个 64Bytes 的字符串作为写入的最小单位，随后使用 strcat 函数将重复拼接到指定的写入大小，并通过 write 系统调用将其写入文件系统即可：

```
#define BUFSIZE (64)

char buffer[BUFSIZE] = "This is a 6KB block!";

void write_file(int blocksize, bool isrand, char *filepath){
    int fd = 0;
    fd = open(filepath, O_CREAT | O_RDWR | O_SYNC, S_IRWXU);
    if(fd < 0){
        fprintf(stdout, "Error occurred when opening file!");
        return;
    }
    char *buf_ext = (char *)malloc(sizeof(char) * blocksize);
    for(int i = 0; i < blocksize / BUFSIZE; i++){
        strcat(buf_ext, buffer);
    }
    for(int i = 0; i < ROUND; i++){
        write(fd, buf_ext, blocksize);
        if(isrand){
            lseek(fd, rand() % ((blocksize - 1) * ROUND), SEEK_SET);
        }
    }
    lseek(fd, 0, SEEK_SET);
    close(fd);
}
```

由于现代存储媒介大多已经可以应付较高的读写请求，为了最大程度测试 DISK 盘和 RAM 盘的性能，我们使用多线程并发读写的方式来尽可能地使磁盘吞吐达到饱和。经实测，在写入块大小为 4KB 时， RAM 和 DISK 盘的吞吐在并发数为 16～20 左右时基本达到了饱和：



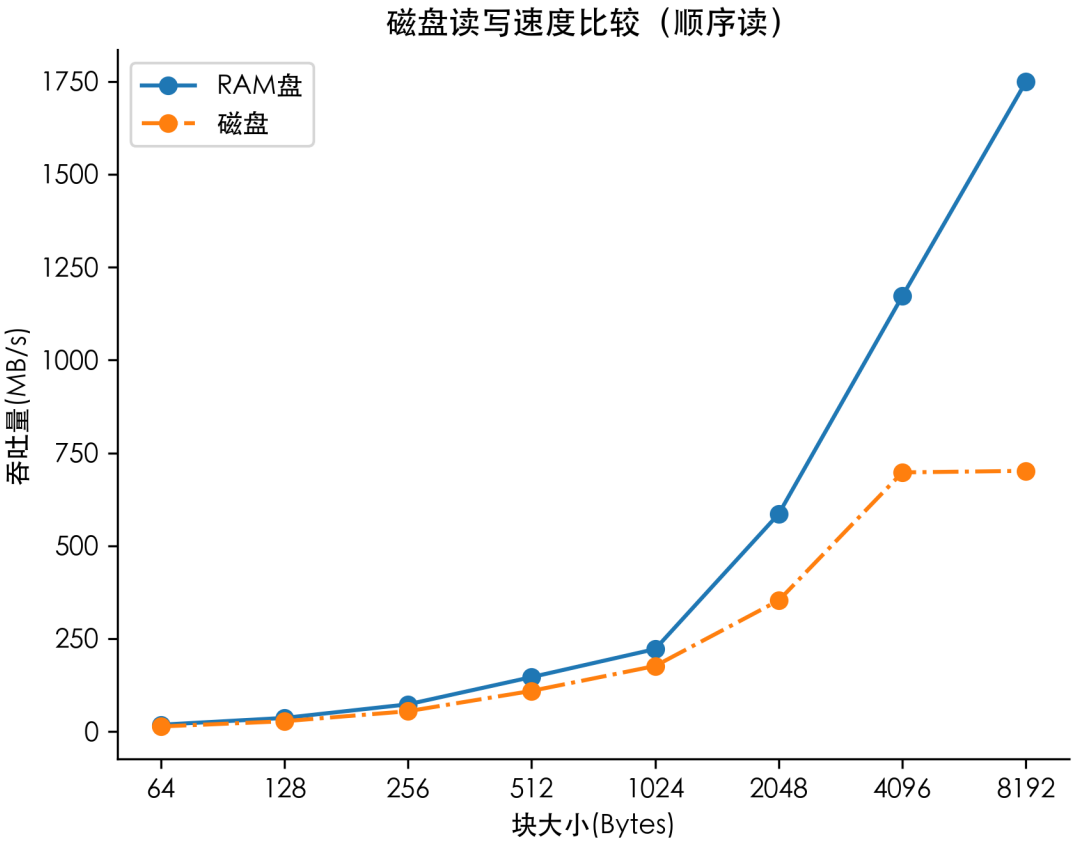
考虑到 SSD 磁盘的读写硬件特性，我们将并发数设置为 15。对于读写块大小，我们以 2 倍为步长，以测试从 64Bytes 到 8KB 时的情况：

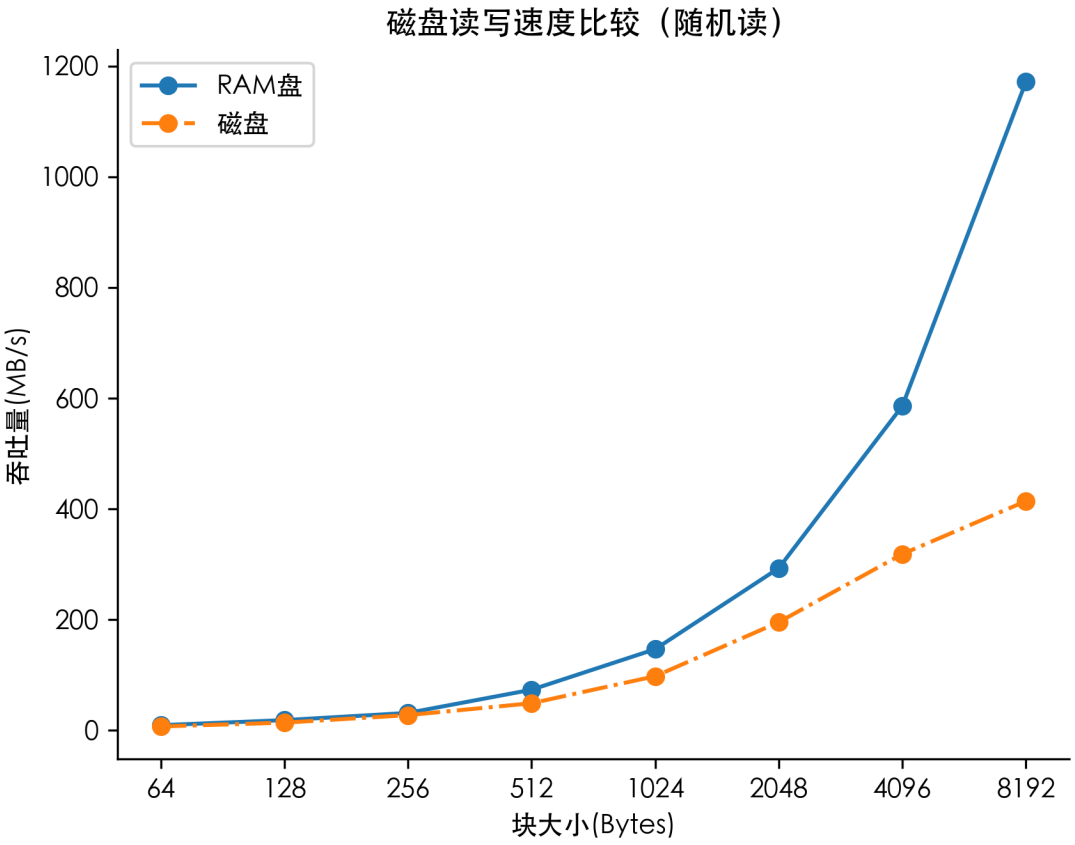
```
#define MAXSTR 100
#define CONCURRENCY 15

char *path_format[2] = {"/root/myram/disk_%d.txt", "/usr/disk_%d.txt"};

//Irrelevant code

double get_time_left(struct timeval starttime, struct timeval endtime){
    return ((endtime.tv_sec * 1000 + endtime.tv_usec / 1000) - (starttime.tv_sec * 1000 +
                                                                starttime.tv_usec / 1000))
 / 1000.0;
}

int main() {
    srand(time(0));
    for (int j = 0; j < 2; j++) {
        if (j == 0) printf("RAM:\n");
        else printf("Disk:\n");
        int block_size = 64;
        for (int k = 0; k < 8; k++) {
            struct timeval start_time, end_time;
            gettimeofday(&start_time, NULL);
            for (int i = 0; i < CONCURRENCY; i++) {
                char *filepath = (char *) malloc(sizeof(char) * MAXSTR);
                sprintf(filepath, path_format[j], i);
                if (fork() == 0) {
                    /* 顺序读取 */
                    read_file(block_size, false, filepath);
```

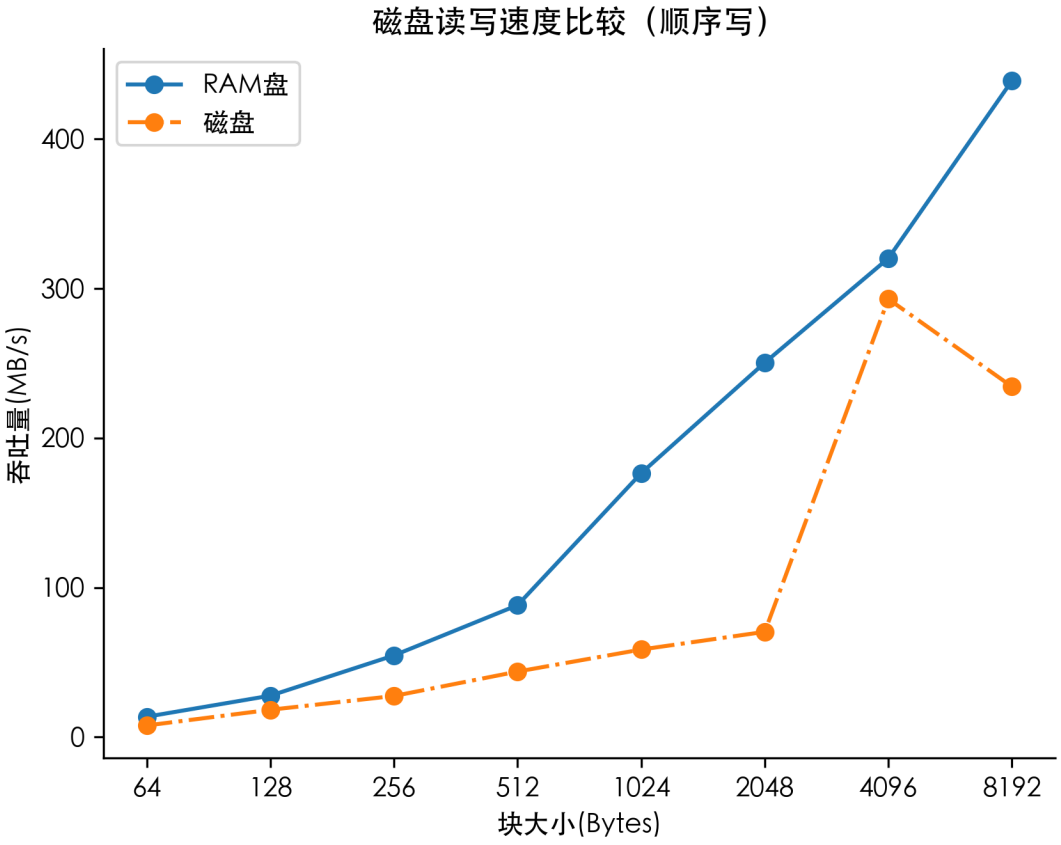
```
                    /* 随机读取 */
                    //read_file(block_size, true, filepath);
                    /* 顺序写入 */
                    //write_file(block_size, false, filepath);
                    /* 随机写入 */
                    //write_file(block_size, true, filepath);
                    exit(1);
                }
            }
            for (int i = 0; i < CONCURRENCY; i++) {
                wait(NULL);
            }
            gettimeofday(&end_time, NULL);
            double time_cost = get_time_left(start_time, end_time);
            double write_size = block_size * ROUND * CONCURRENCY / 1024.0 / 1024;
            printf("Blocksize: %d Bytes, Writesize: %f MB, Time: %f s\n", block_size, writ
    e_size, time_cost);
            block_size *= 2;
        }
    }
    return 0;
}
```

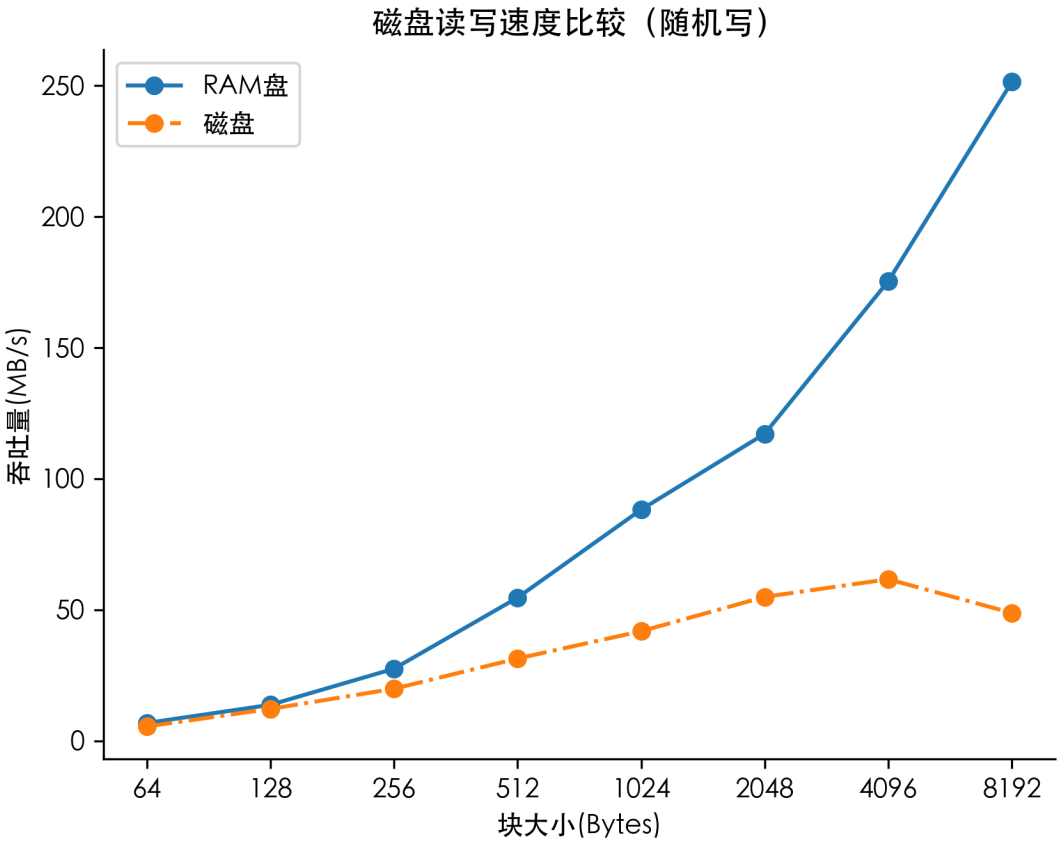
4. 测试结果分析

将测试程序编译并多次运行后，我们得到了一组 RAM 盘和 DISK 盘在不同块大小下各种读写情况时的运行时间。通过数据大小/运行时间，我们就可大致得到磁盘的平均读写速度。结果如下：

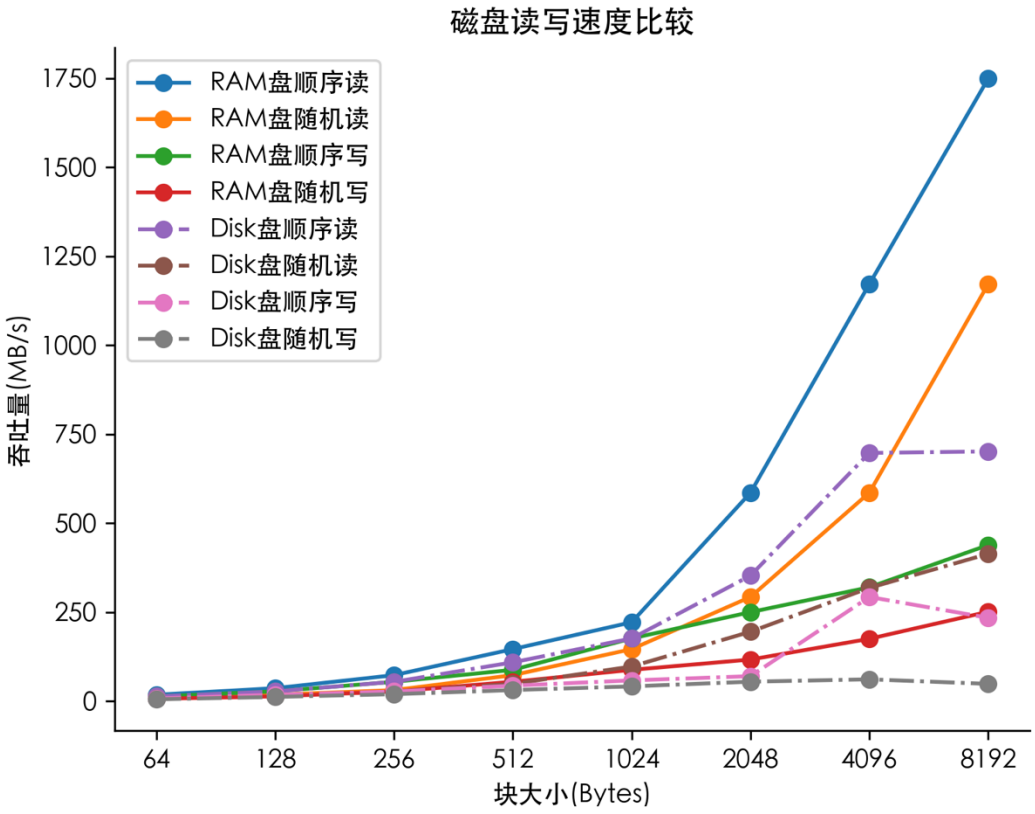



磁盘读写速度比较（随机读）
RAM盘
磁盘
吞吐量(MB/s)
0
200
400
600
800
1000
1200
64
128
256
512
1024
2048
4096
8192
块大小(Bytes)

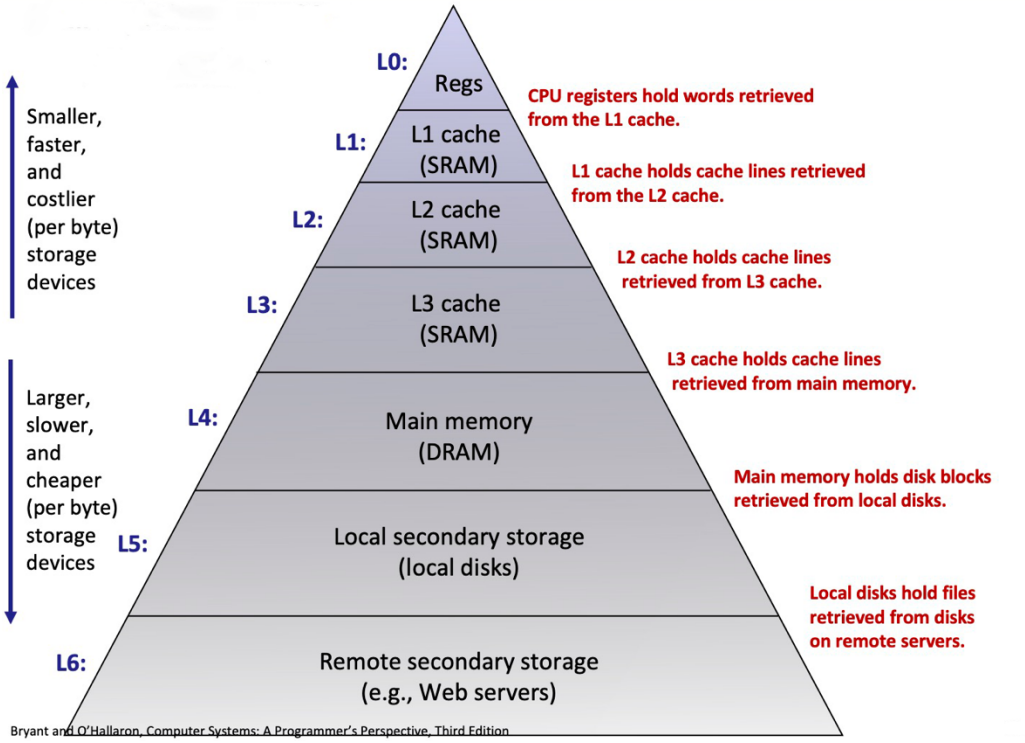

磁盘读写速度比较（顺序写）
RAM盘
磁盘
吞吐量(MB/s)
0
100
200
300
400
64
128
256
512
1024
2048
4096
8192
块大小(Bytes)



进一步地，我们把各种情况下的结果放在一起进行比较：



可以发现，随着操作块大小的增加，RAM 盘和 DISK 盘的吞吐量也逐渐增加。在各种读写场景下，RAM 盘的读写速度显著高于 Disk 盘，这与其实现原理及在计算机体系结构中的层级位置相一致：



此外，由于我们使用了 SSD（PCI-Express 协议）作为磁盘存储媒介，可以看到当块大小为 4KB 时，DISK 盘的吞吐量激增。这是由于在使用 SSD 磁盘时，系统通常会对其进行 4K 对齐优化以延长磁盘使用寿命，而 4KB 的读写块大小正好为一个磁盘块大小，因此磁盘控制器可以快速响应所需的请求。

五、 总结

在本实验中，我们在 Minix3 系统下分别划分了一块 DISK 盘空间与 RAM 盘空间，并通过一系列不同读写方式的组合测试了 DISK 盘和 RAM 盘的读写速度与特性，更加直观的认识了系统对于 RAM 和 DISK 存储媒介的不同管理方式及其在系统构架中的巨大传输速度差异，从而感受到了现代计算机系统构架的合理性。