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：**2018 级　　**上机实践成绩：**

指导教师：翁楚良　　**姓名：**郑佳辰

上机实践名称：I/O subsystem　　**学号：**10182100359　　**上机实践日期：**2020/5/26

上机实践编号：　　~6/13

一、 目的

熟悉类 UNIX 系统的 I/O 设备管理，熟悉 MINIX 块设备驱动，熟悉 MINIX RAM 盘.

二、 内容与设计思想

在 MINIX3 中安装一块几百 MB 大小的 RAM 盘，可以挂载并且进行存取文件操作。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，并用图表形式体现在实验报告中。

三、 使用环境

Vmware Workstation pro 15.5
Minix 3.3.0
MobaXterm Personal Edition v20.1
FileZilla 3.48.2.1
Dev-C++ 5.11
SourceInsight 4.0

四、 实验过程

1.增加 RAM 盘

MINIX 中已有 6 块用户可用 RAM 盘，7 块系统保留 RAM 盘。我们需要先增加一个 RAM 盘，取名 myram。

首先修改/usr/src/minix/drivers/storage/memory/memory.c，增加默认的用户 RAM 盘数。然后重新编译内核，输入 reboot 进行重启。

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

然后输入 mknod /dev/myram b 1 13 创建设备，并输入 ls/dev/ | grep ram 查看设备是否创建成功。

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

之后需要实现 buildmyram 初始化工具。参考 ramdisk.c 实现的 buildmyram.c 代码及运行结果如下。需要将 ramdisk.c 中的单位 KB 改为 MB。

```
#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 MFACTOR (1024*1024)
	size = atol(argv[1])*MFACTOR;

	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/MFACTOR);

	return 0;
}
```

```
# ./buildmyram 400 /dev/myram
size on /dev/myram set to 400MB
```

使用以下命令在盘上创建内存文件系统。

```
# mkfs.mfs /dev/myram
```

将 RAM 盘挂载到用户目录下。

```
# mount /dev/myram /root/myram
/dev/myram is mounted on /root/myram
```

输入 df 可以查看是否挂载成功。

```
# df
Filesystem    512-blocks      Used      Avail %Cap Mounted on
/dev/myram       1024000   1024000          0 100% /root/myram
/dev/c0d3p0s0     262144     80592     181552  30% /
none                   0         0          0 100% /proc
/dev/c0d3p0s2   13427696   4443272    8984424  33% /usr
/dev/c0d3p0s1    3078144    852728    2225416  27% /home
none                   0         0          0 100% /sys
```

需要注意，重启后用户自定义的 RAM 盘内容会丢失，需要重新设置大小，创建文件系统，并挂载。重复调用 buildmyram 程序之后的几步即可。

2.编写测试代码

2.1 变量和宏的定义及 get_time_left 函数

程序中使用的头文件，宏和全局变量代码如下。

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <stdbool.h>
#include <unistd.h>
#include <time.h>
#include <string.h>
#define Concurrency 20
#define Blocksize 65536
#define filesize (200 * 1024 * 1024)
#define maxline (2 * 1024 * 1024)

char examtext[maxline] = "abcdefghijklmnopqrstuvwxyz";
struct timeval starttime, endtime, spendtime;
char testtext[maxline];
char filepath[100];
char x[5];
```

其中，Concurrency 是设置的最大并发数。Blocksize 是最大的块大小。在我的测试代码中，采取了固定总字节数，改变块大小的方法计算吞吐量。

函数 get_time_left 用于计算读写所花费的时间，代码如下。

```
struct timeval get_time_left(struct timeval starttime, struct timeval endt
ime)
{
      struct timeval spendtime;
      spendtime.tv_sec=endtime.tv_sec-starttime.tv_sec;
      spendtime.tv_usec=endtime.tv_usec-starttime.tv_usec;
      if(spendtime.tv_usec<0){
            spendtime.tv_sec--;
            spendtime.tv_usec+=1000000;
      }
   return spendtime;
}
```

在 MINIX 系统中提供了两个用来计时的结构体。其中，timeval 结构体由两项组成，tv_sec 记录秒数，tv_usec 记录微秒数。而 timespec 结构体更为精确，tv_sec 记录秒数，tv_nsec 记录纳秒数。这里使用 timeval 来记录读写时间。本质上来说，get_time_left 函数就是 timeval 结构体的减法。

2.2 写文件函数 write_file

函数 write_file 是用于写文件的函数，它有 4 个参数。块大小由 blocksize 传入，filepath 是要写的文件的路径，fs 是要写的文件的字节数，israng 指示是否随机，值为真就是随机写，否则为顺序写。函数代码如下。

```
void write_file(int blocksize, bool israng, char *filepath, int fs)
{
```

```
	int fd;

	fd=open(filepath,O_WRONLY|O_CREAT|O_SYNC,0666);
	if(fd<0){
		perror("\nWrite Error: cannot open file! \n");
		return;
	}

    for(int i=1;i<=fs/blocksize;i++){
	int z=((rand()*RAND_MAX)+rand())%fs;
		if(israndl){
			lseek(fd, z, SEEK_SET);
		}

		int rsize=write(fd,examtext,blocksize);
		if(rsize<0){
			perror("\nWrite Error: return value of write < 0 ! \n");
			return;
		}
		else if(rsize==0){
			rsize=write(fd,examtext,blocksize);
			if(rsize!=blocksize){
				perror("\nWrite Error: write returns 0, cannot write enou
gh bytes! \n");
				return;
			}
		}
		else if(rsize<blocksize){
			rsize+=write(fd,examtext,blocksize-rsize);
			if(rsize!=blocksize){
				perror("\nWrite Error: cannot write enough bytes! \n");
				return;
			}
		}
	}
}
```

开始写文件时需要先用 open 系统调用打开文件，利用 O_SYNC 参数进行同步读写。循环中的 fs/blocksize 是需要读写的次数。循环体中，利用两遍 rand 函数使得随机读写时 z 可以覆盖到文件的各个位置。如果是随机读写，我们需要在每次读写之前调用 lseek 函数重新定位文件指针。

然后调用 write 函数将测试字符串写入文件中，并且检查该函数的返回值 rsize。如果 rsize 小于零，说明由于没有空间等原因，导致无法写入。如果 rsize==0，说明写入了零字节，这种情况一般不会出现。如果 0<rsize<blocesize，说明该没有写入足够的字节，此时可以再写一遍或打印错误。

2.3 读文件函数 read_file

读文件函数与写文件函数类似，代码如下。

```
void read_file(int blocksize, bool israndl, char *filepath, int fs)
{
	int fd;

	fd=open(filepath,O_RDONLY|O_SYNC,0666);
```

```
	if(fd<0){
		perror("\nRead Error: cannot open file! \n");
		return;
	}

	for(int i=1;i<=fs/blocksize;i++){
		int z=((rand()*RAND_MAX)+rand())%fs;
		if(israd){
			lseek(fd, z, SEEK_SET);
		}

		int rsize=read(fd,testtext,blocksize);
		if(rsize<0){
			perror("\nRead Error: return value of read < 0 ! \n");
			return;
		}
		else if(rsize==0){
			lseek(fd, 0, SEEK_SET);
			rsize=read(fd,examtext,blocksize);
			if(rsize!=blocksize){
				perror("\nRead Error: read returns 0, cannot read enough
bytes! \n");
				return;
			}
		}
		else if(rsize<blocksize){
			lseek(fd, 0, SEEK_SET);
			rsize+=read(fd,examtext,blocksize-rsize);
			if(rsize!=blocksize){
				perror("\nWrite Error: cannot read enough bytes! \n");
				return;
			}
		}
	}
}
```

对于读文件函数来说，与写文件时相同的部分不再赘述。函数 read 的返回值 rsize 与 write 时具有具有不同的含义。如果 rsize 小于零，说明由于各种原因无法读入。如果返回值 rsize 等于零，说明读入了 0 字节，很大概率是遇到了文件尾 EOF，此时需要将文件指针重新定位然后再读入。如果 0<rsize<blocesize，说明没有读入足够的字节，可能因为还未读够相应的字节，就遇到了文件尾 EOF，此时也需要将文件指针重新定位其然后再读入剩下的字节。

2.4 主函数 main

程序的主函数代码如下。

```
int main()
{
	printf("Please input path:\n");
	scanf("%s",filepath);
	for(int concurrency=Concurrency;concurrency<(Concurrency+1);concurrency++)
{
		printf("Blocksize|Random Write|Sequence Write|Random Read|Sequence R
ead(MB/s) concurrency=%d;\n",concurrency);
```

```
            for(int blocksize=Blocksize;blocksize>63;blocksize/=2){
                  printf("%dB\t",blocksize);
                  fflush(stdout);
                  for(int flag=1;flag<5;flag++){
                        //if(flag==1) continue;
                        //if(flag==2) continue;
                        //if(flag==3) continue;
                        //if(flag==4) continue;

                        gettimeofday(&starttime,NULL);
                        for(int i=0;i<concurrency;i++){
                              if(fork()==0){
                                    sprintf(x,"%d",i);
                                    strcat(filepath,x);
                                    switch(flag){
                                          case 1:
                                                //随机写
                                                write_file(blocksize, true,
                                                &filepath, filesize/concurrency);

                                                break;
                                          case 2:
                                                //顺序写
                                                write_file(blocksize, false,
                                                &filepath, filesize/concurrency);

                                                break;
                                          case 3:
                                                //随机读
                                                read_file(blocksize, true,
                                                &filepath, filesize/concurrency);

                                                break;
                                          case 4:
                                                //顺序读
                                                read_file(blocksize, false,
                                                &filepath, filesize/concurrency);

                                                break;
                                    };
                                    exit(0);
                              }
                        }
                        while(wait(NULL)!=-1);
                        gettimeofday(&endtime,NULL);
                        spendtime=get_time_left(starttime,endtime);

                        long us_ms = (1000000 * spendtime.tv_sec + spendtime.tv_u
sec);
                  double timeuse = (us_ms / 1000.0) / 1000.0;
                  double throughput=filesize/timeuse/1024/1024;
                  printf("%lf\t",throughput);
                  fflush(stdout);
```

```
                }
                printf("\n");
            }
        }
    return 0;
}
```

主函数内最外层的并发数的 concurrency 循环是在当需要以并发数为自变量时添加的循环。实验中仅需要观察块大小与吞吐量的关系时，则可使用固定的并发数，让此层循环只执行一次。中层的块大小 blocksize 循环由将块大小由 64KB 逐渐改变到 64B，每次块大小减一半。使用由大到小的顺序是由于块大小较大时吞吐量大，便于短时间内观察到更多数据。

最内层的 flag 循环是用来指示进行哪一类读写。1 代表随机写，2 代表顺序写，3 代表随机读，4 代表顺序读。中间注释掉的四条语句是调试时使用的，可以只执行任意几类读写。采用 fork 函数利用多个进程进行并发读写，每个进程会读写不同的文件。读写完毕后，我们利用 wait 回收所有的子进程，然后计算出花费的时间 spendtime。

由于我们的时间结构体是 timeval 类型，所以计算时间使用了 gettimeofday 函数。如果我们采用的是 timespec 结构体，那么就应使用 clock_gettime 函数。然后我们使用总字节数除以总时间即可得到本次读写的吞吐量。

3.性能测试

3.1 实验结果数据

将编写的测试程序传入虚拟机进行编译并运行。测试 RAM 的性能时，将路径设置为./myram，测试 DISK 盘的读写能力时将路径设置为/home。测试程序运行的结果如下图所示。

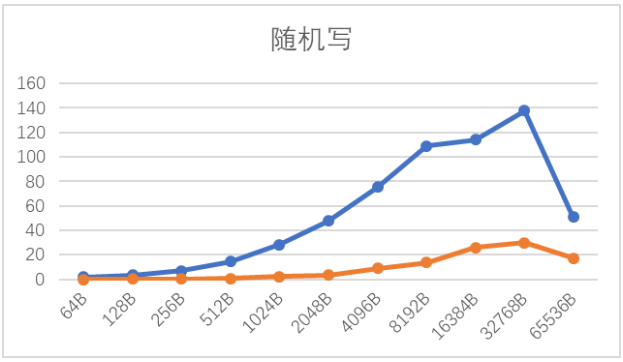
```
# ./test
Please input path:
./myram/wt
Blocksize|Random Write|Sequence Write|Random Read|Sequence Read(MB/s) concurrency=20;
65536B  51.063838       77.419365       272.727025      244.898159
32768B  137.931034      239.999808      244.898159      279.069638
16384B  114.285714      206.896480      187.500117      255.318932
8192B   109.090929      184.615441      136.363605      218.181739
4096B   75.471698       148.148148      96.000015       173.913043
2048B   47.619048       110.091723      63.492063       117.647059
1024B   28.368794       70.175439       33.994336       71.428571
512B    14.652015       38.095238       17.069701       43.010753
256B    6.940428        21.818181       9.202454        22.304832
128B    3.510825        11.288806       4.606526        10.723861
64B     1.960144        6.057547        2.330550        5.091218
# ./test
Please input path:
/home/wt
Blocksize|Random Write|Sequence Write|Random Read|Sequence Read(MB/s) concurrency=20;
65536B  17.341041       17.937220       173.913043      218.181739
32768B  29.999999       89.552252       184.615441      255.318932
16384B  25.917929       80.536891       166.666667      214.285791
8192B   14.002334       84.507030       126.315816      196.721247
4096B   9.174312        41.237113       64.864872       164.383517
2048B   3.508772        43.165471       44.609659       77.419365
1024B   2.491177        36.253774       26.490066       68.181826
512B    0.682749        7.726980        12.958964       34.682079
256B    0.588206        15.894040       7.994670        19.323671
128B    0.261472        2.498959        3.396547        9.748172
64B     0.139370        5.452067        2.337359        6.106870
```

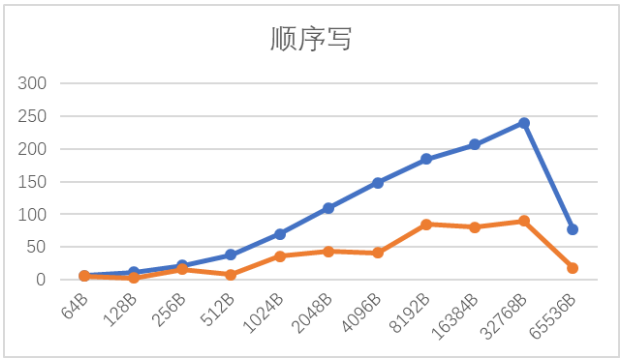
将实验结果的数据导入 Excel 表格，如下表所示。

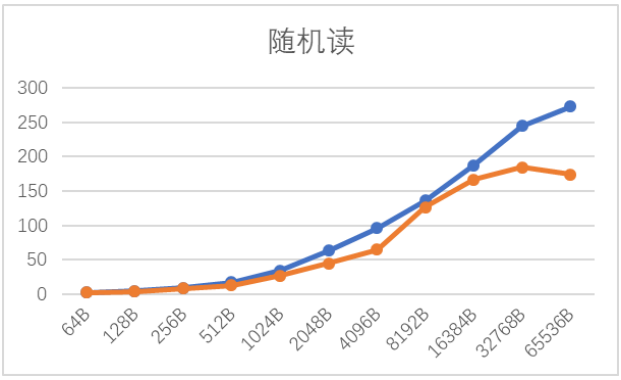
	RAM随机写	RAM顺序写	RAM随机读	RAM顺序读	DISK随机写	DISK顺序写	DISK随机读	DISK顺序读
64B	1.9601	6.0575	2.3306	5.0912	0.1394	5.4521	2.3374	6.1069
128B	3.5108	11.2888	4.6065	10.7239	0.2615	2.4990	3.3965	9.7482
256B	6.9404	21.8182	9.2025	22.3048	0.5882	15.8940	7.9947	19.3237
512B	14.6520	38.0952	17.0697	43.0108	0.6827	7.7270	12.9590	34.6821
1024B	28.3688	70.1754	33.9943	71.4286	2.4912	36.2538	26.4901	68.1818
2048B	47.6190	110.0917	63.4921	117.6471	3.5088	43.1655	44.6097	77.4194
4096B	75.4717	148.1481	96.0000	173.9130	9.1743	41.2371	64.8649	164.3835
8192B	109.0909	184.6154	136.3636	218.1817	14.0023	84.5070	126.3158	196.7212
16384B	114.2857	206.8965	187.5001	255.3189	25.9179	80.5369	166.6667	214.2858
32768B	137.9310	239.9998	244.8982	279.0696	30.0000	89.5523	184.6154	255.3189
65536B	51.0638	77.4194	272.7270	244.8982	17.3410	17.9372	173.9130	218.1817

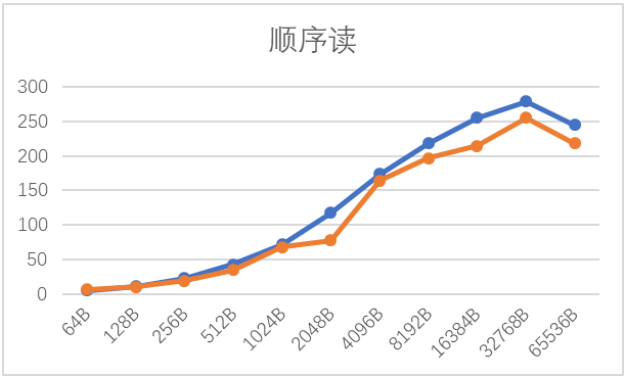
3.2 RAM 盘和 DISK 盘的比较

将 RAM 盘和 DISK 盘的吞吐量制成图表，如下。其中蓝色的折线代表 RAM 盘的吞吐量，橙色的折线代表 DISK 盘的吞吐量。









从理论上讲，实验的预计结果是 RAM 盘的性能高于 DISK 盘，尤其是在随机读写方面的表现。从上面的图表中可以看出，对于任意一种测试方式，从吞吐量方面来看，RAM 盘的性能总是好于 DISK 盘的性能，和预计实验结果相符。

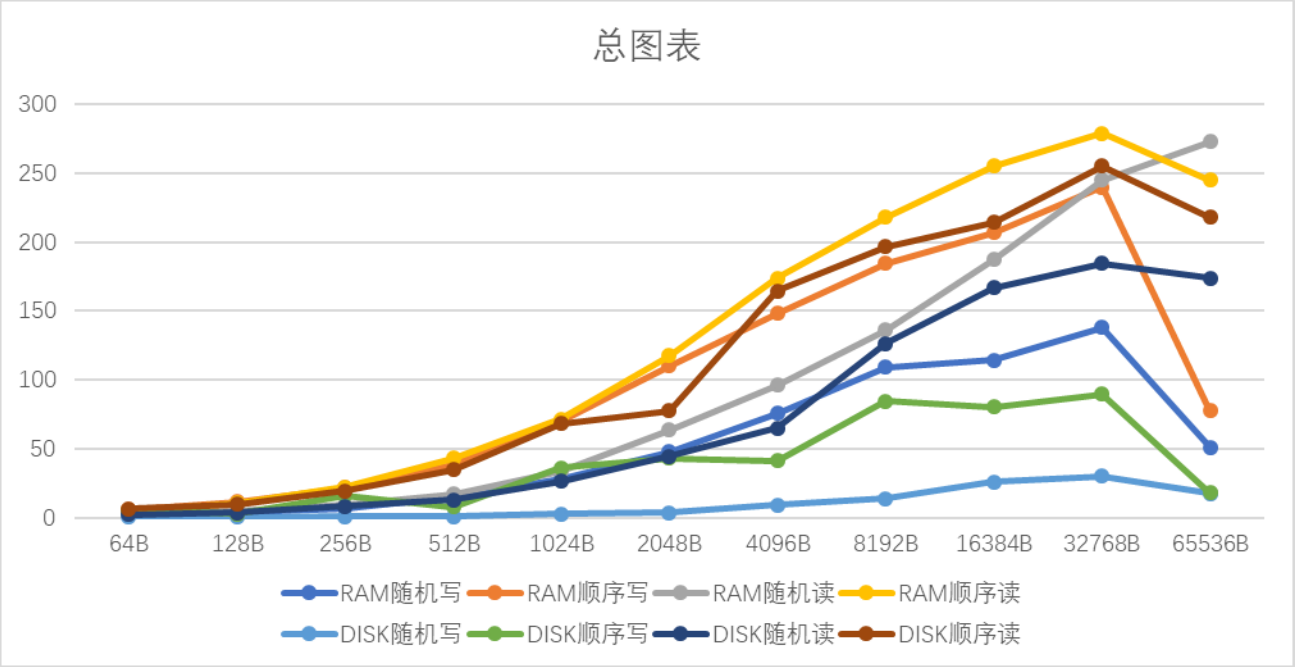
从我的实验结果来看，对于随机写来说，RAM 盘的吞吐量是 DISK 盘的好几倍，性能远好于 DISK 盘。对于随机读的性能，当块的大小较大时，差距也很明显。对于顺序读来说，两者差距不大。

这种差异主要是因为 RAM 盘是通过将一部分内存模拟为硬盘来使用。它并不提供永久存储，但是一旦文件被复制到这一区域，就能够以极快的速度进行访问。而磁盘来进行读写时有寻道时间，旋转延迟和实际的数据传输时间的时延。

其中寻道时间是将磁头臂移动到相应柱面所需的时间，相比其他两段时间要大得多。这些差异造成了我们在图表中看到的性能的差异。

3.3 不同块大小的比较

将所有数据绘制到同一张图表上，如下表所示。



性能测试中需要采用多进程并发的同步读写，并发数要增加到设备接近饱和的状态。即吞吐量难以继续提升，但是 I/O 延时恶化。在出现饱和前，总吞吐量随着并发数线性增长。通常情况下，7~15 个进程设备就会达到饱和。在我的程序中选取的并发数是固定的，为 20。

由图表可以看出，一般情况下，无论块扫描方式和存储器种类，随着块大小的增加，吞吐量会显著上升。这种现象主要是由于机械硬盘一般来说每次读写的块大小为 4KB。块大小比较小的话，磁头会频繁移动，增加了寻道时间和旋转延迟。随着块大小增加，一定范围内没有增加写入的块的数量。磁头移动减少，寻道时间变短，吞吐量增加。

对于固态硬盘来说，存在写放大的问题，一次写操作需要写一整个块。但是固态硬盘的随机读写性能远好于机械硬盘。如果 MINIX 虚拟机建在 SSD 下，会导致随机读写和顺序读写的差距减小。最好把虚拟机放在机械硬盘上，实验效果比较明显。

3.4 实验中未解决的部分问题

在本次实验中出现了许多难以解决的问题。首先，当块大小比较大，一般来说高于 16KB 时，会出现随机读快于顺序读的情况。从理论上讲，这种情况不可能出现。一开始的猜想是写入的字节数不足，但在整个程序在运行过程中，除了空间不足引发的错误均没有报错，这种情况随即被排除。

其次，当块大小从 32 KB 增加到 64 KB，即最大的块大小时，几乎所有情况下的吞吐量都发生了下降。从图表中可以看出，除 RAM 盘顺序读能勉强维持上升趋势外，其他情况下的吞吐量均有不同程度的下降，其中以 RAM 盘的顺序写最为明显。这可能是上一题中出现随机读效率高于顺序读的原因。初始猜想是由于第一个执行的就是块大小为 64 KB 的情况，所以可能是将建立文件所需的时间也算进了程序读写的时间里，但这个结论随即就被推翻，因为当文件已建立时仍

旧出现这种情况。

改变最大块大小使其从 32KB 开始后，仍可观察到类似的现象，如下图所示。但最后一个块的下降变得不明显。根据我的猜想，首先被执行的块大小吞吐量在测试时会得到较低的结果。但是当块大小按照递增顺序执行时，也会在 64KB 的块大小处出现吞吐量的下降。所以哪个块大小最先执行可能和这种现象有关，但并不是主要因素。

```
# ./testnew
Please input path:
./myram/wt
Blocksize|Random Write|Sequence Write|Random Read|Sequence Read(MB/s)
32768B  43.636357       40.000000       40.816327       31.662272
16384B  34.682079       40.677963       34.285716       31.413611
8192B   27.777778       39.087953       27.777778       30.690536
4096B   19.480519       37.267085       19.543973       28.915665
2048B   12.232416       32.432431       12.513035       27.906975
1024B   ^C
# ./testnew
Please input path:
/home/wt
Blocksize|Random Write|Sequence Write|Random Read|Sequence Read(MB/s)
32768B  10.899183       11.257035       13.714286       8.385744
16384B  5.681818        12.435233       11.183598       13.620886
8192B   4.948454        14.475271       8.379889        12.000000
4096B   2.500000        15.768726       5.444646        13.201320
2048B   1.529442        4.979253        3.077712        ^C
```

此外，在同一台虚拟机中先后进行的测试吞吐量差别很大。同样在上图中可以看出，往往在第一次测试中最高吞吐量可达 250MB 以上，但再次进行实验时最大吞吐量迅速下降至不到 20MB，此时最小吞吐量只有 0.13MB 左右。在清理过读写文件后再进行实验，吞吐量回升现象并不明显。这也是一个难以解决的问题，可能和机器本身有关。

纵然本次实验有很大的随机性，但这些现象带来的结果已经超出了误差能够涵盖的范围。这些问题有待进一步研究和解决。

五、 总结

这次实验的主要工作是在 MINIX3 系统中添加一个 RAM 盘并比较它和普通硬盘的性能。RAM 盘是在内存中保留一部分存储区域，使其象普通磁盘一样使用。它不提供永久存储，但可以快速访问。RAM 盘的驱动程序中，设备无关代码存放在 driver.c 中，设备相关代码存放在 memory.c 中。实验中我们也修改了 RAM 盘的设备相关代码。硬盘的驱动程序中，设备无关代码存放在 driver.c 中，设备相关代码存放在 at_wini.c 中。

测试 RAM 盘和 DISK 盘的性能时，需要采用多进程并发的同步读写，以达到设备的最大吞吐量。性能测试的两个变量分别为块大小和块扫描方式。分别测试在 RAM 盘和 DISK 盘下的随机读，顺序读，随机写，顺序写速度，并对比在不同的块大小下的性能，然后通过 Excel 把数据制作成图表。

通过这次实验，我对 RAM 盘和硬盘有了更深入的理解，明确了这两种存储设备性能上的差异，了解了局部性带来的块扫描方式对于吞吐量的差异，总体上来说，我对于输入输出系统有了更直观的印象，同时借助了 MINIX3 更好地理解了输入输出设备的原理。