Unix程序设计

实验（二）

姓　　名	熊恪峥
学　　号	22920202204622
日　　期	2022年10月7日
学　　院	信息学院
课程名称	Unix程序设计

实验（二）

目录

1 实验内容

编写程序**timewrite <outfile> [sync]**。不得变更程序的名字和使用方法。 sync参数为可选，若有，则输出文件用O_SYNC打开。例：

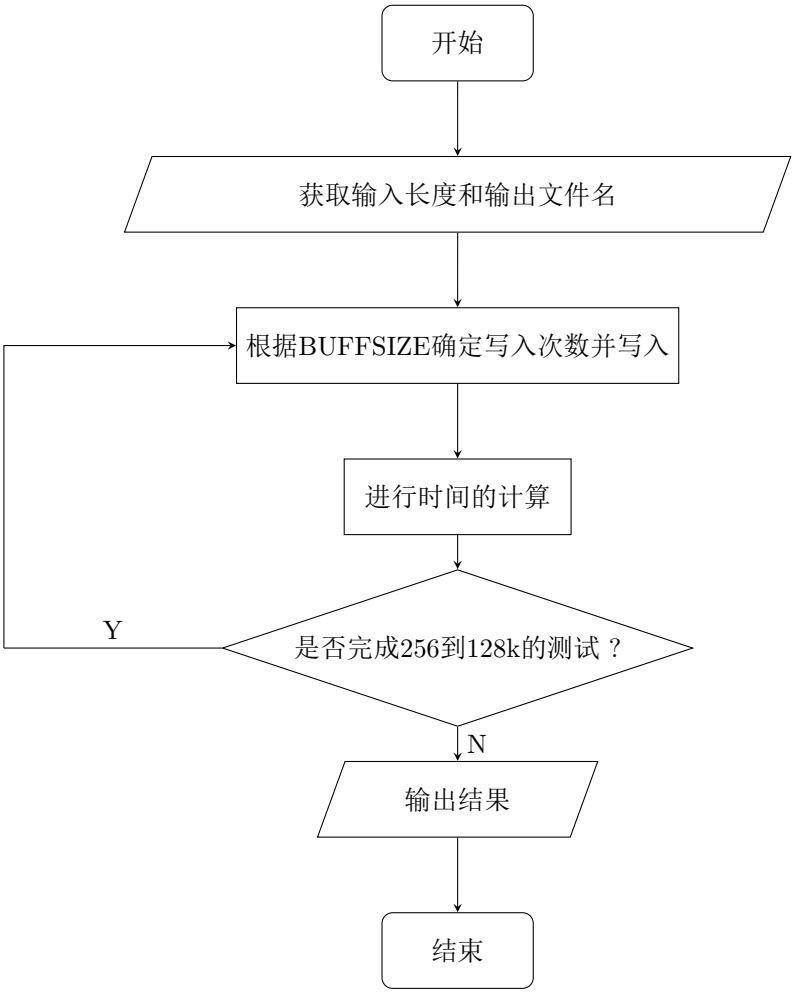
1. **timewrite <f1 f2** 表示输出文件f2不用O_SYNC 打开。
2. **timewrite f1 sync <f2** 表示输出文件f1用O_SYNC 打开。

2 程序设计与实现

2.1 程序设计

程序流程的设计如图1。该程序对算法的要求比较简单。可以简单地根据题目中的要求对程序进行实现。

图 1: 流程图



2.2 Makefile

首先使用Makefile作为构建系统。Makefile中的内容如代码 1。其中，为了使用较新的语言特性来编写程序，因此在编译时使用了**-std=c99**选项启用C99标准，这启用了C99标准中的特性，可以使程序更符合现在的最佳实践。

代码 1: Makefile

```
OUT = build
SRC = .
CC = gcc
LD = ld

all : timewrite

${OUT}:
	mkdir -p $(OUT)

${OUT}/timewrite.o: ${OUT} $(SRC)/timewrite.c
	$(CC) -std=c99 -c $(SRC)/timewrite.c -o $(OUT)/timewrite.o

timewrite : $(OUT)/timewrite.o
	$(CC) $(OUT)/timewrite.o -o timewrite

clean :
	rm -rf $(OUT) timewrite

.PHONY: all clean
```

2.3 程序实现

根据题目和流程图 1实现程序如代码 2。

代码 2: 程序实现

```
#include <sys/times.h>
#include <unistd.h>
#include <fcntl.h>

#include <stdio.h>
#include <assert.h>
#include <string.h>
#include <stdlib.h>

#define FILE_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)


int main(int argc, char** argv)
{
    assert(argc == 2 || argc == 3);

    if (argc == 3 && strcmp(argv[2], "sync") != 0)
    {
        printf("usage: %s <pathname> [sync]", argv[0]);
        return 1;
    }

    const int mode = (argc == 3) ? O_RDWR | O_CREAT | O_TRUNC | O_SYNC : O_RDWR | O_CREAT | O_TRUNC;

```

```
25      int out_file = open(argv[1], mode, FILE_MODE);
26      if (out_file < 0)
27      {
28          printf("open error");
29          return 1;
30      }
31
32      int length = lseek(STDIN_FILENO, 0, SEEK_END);
33      if (length < 0)
34      {
35          printf("lseek error");
36          return 1;
37      }
38
39      if (lseek(STDIN_FILENO, 0, SEEK_SET) < 0)
40      {
41          printf("lseek error");
42          return 1;
43      }
44
45      char* buff = malloc(length);
46      if (buff == NULL)
47      {
48          printf("malloc error");
49          return 1;
50      }
51
52      if (read(STDIN_FILENO, buff, length) != length)
53      {
54          printf("read error");
55          return 1;
56      }
57
58      const int lock_per_second = sysconf(_SC_CLK_TCK);
59
60      for (int bufsize = 256;bufsize <= 131072;bufsize <<= 1)
61      {
62          lseek(out_file, 0, SEEK_SET);
63
64          int g = length / bufsize, res = length % bufsize;
65          struct tms start, end;
66          clock_t start_clock, end_clock;
67          start_clock = times(&start);
68
69          for (int i = 0;i < g;i++)
70          {
71              if (write(out_file, buff + i * bufsize, bufsize) != bufsize)
72              {
73                  printf("write error");
74                  return 1;
75              }
76          }
77
78          if (res != 0 && write(out_file, buff + g * bufsize, res) != res)
79          {
```

```
80              printf("write error");
81              return 1;
82          }
83          end_clock = times(&end);
84
85          const int loop = g + (res != 0);
86          const double real_time = (double)(end_clock - start_clock) / lock_per_second;
87          const double user_time = (double)(end.tms_utime - start.tms_utime) / lock_per_second;
88          const double sys_time = (double)(end.tms_stime - start.tms_stime) / lock_per_second;
89
90          printf("%d\t%7.4lf\t%7.4lf\t%7.4lf\t%d\n", bufsize, user_time, sys_time,real_time, loop);
91
92      }
93
94      return 0;
95  }
```

3 实验结果

3.1 实验设计

为了全方位地测试同步写入和异步写入的效率，我在4种不同的UNIX/类UNIX操作系统环境下进行了测试，分别是Windows Subsystem for Linux（WSL）环境下的Ubuntu 22.04， WSL环境下的Ubuntu 22.04（访问Windows文件系统），VMWare虚拟机中的Ubuntu 22.04 、MWare虚拟机中的FreeBSD（GhostBSD发行版）和课程服务器上的Linux 2.6。这些不同的操作系统环境的主要特点如表 1。

表格 1: 测试平台

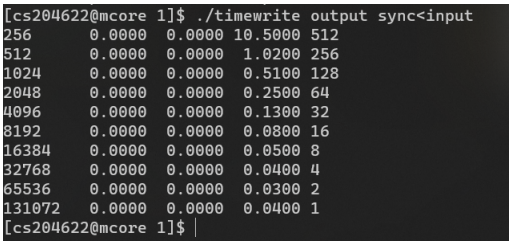
测试平台	编译器	文件系统	特点
WSL环境下的Ubuntu 22.04	gcc 11	ext4	无
WSL环境下的Ubuntu 22.04 （访问Windows文件系统）	gcc 11	NTFS	文件读写效率低 [1]
FreeBSD （GhostBSD发行版）	clang 13	ZFS	有较为先进的ZFS文件系统 [2]
VMWare虚拟机中的Ubuntu 22.04	gcc 11	ext4	无
课程服务器上的Linux 2.6	gcc 4.1.2	ext4	无

这些实验涵盖了2种不同的类UNIX操作系统，涵盖了3种常见的文件系统，互相形成对照，能够较好地形成相互对照，并良好地反映出同步写入和异步写入的效率差异。

3.2 实验方法

在不同的平台上编译并运行该程序，得到的输出示例如图 2所示。

图 2: 实验结果



不同平台的实验结果详细数据见附录：实验详细数据

3.3 实验结论

3.3.1 异步写在写入性能较差的介质和文件系统下有更强的加速效果

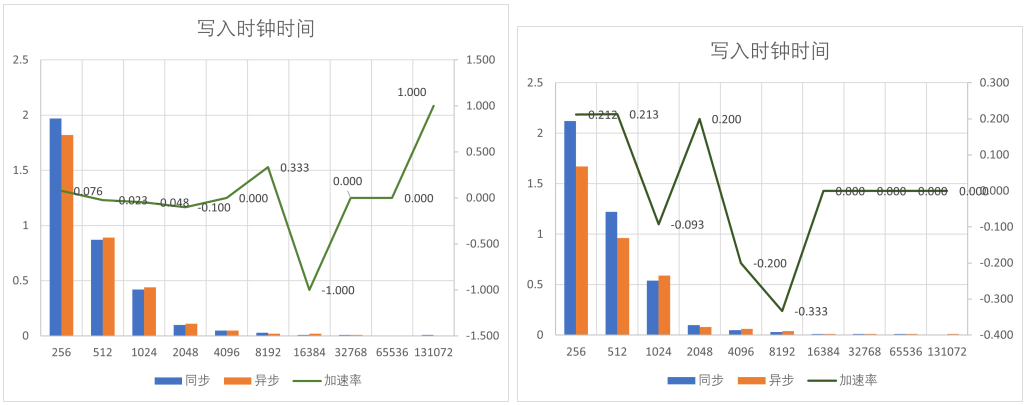
WSL2中的Linux在写入Windows操作系统的分区时会出现较大的性能开销 [1]。因此，借助WSL2中的Linux访问**WSL2映像中的文件**和访问**Windows操作系统分区中的文件**的对比，可以反映出异步写入对效率的提升与存储介质和文件系统性能的关系。

为了直观地比较同步写入和异步写入的效率，在统计运行时间之外，还统计异步写入加速比率，计算方法如(1)

$$\text{acc} = \frac{t_{sync} - t_{async}}{t_{sync}} \times 100\% \tag{1}$$

结果如图 3所示。

图 3: 异步写入加速比较



(a) WSL (b) WSL（访问Windows文件系统）

可以看出，在文件系统的读写效率较高、访问存储介质的性能能够充分发挥的情况下，异步写并不能带来明显的性能提升，如图 3a。然而当文件系统性能低下、访问存储介质的性能不佳时，异步写能够对写入操作进行明显的加速。如图 3b。

随着每次写入数据量（BUFFSIZE）的增大，异步写入的加速能力下降，并在某一处值处开始转变为0贡献和负贡献。这一阈值在文件系统的读写效率较高时来的更早（图 3a），这侧面印证了异步写入对性能本身较差的文件系统和存储介质的加速效果更好这一结论。

3.3.2 FreeBSD上的ZFS文件系统确实有优秀的的性能表现

BSD上的ZFS [2]被证实具有良好的性能表现，不仅仅和ZFS on Linus相比，更能和EXT4等传统的文件系统相比 [3]。本次实验也证明了一点。统计不同操作系统上同步写入的时钟时间，如图 4。

可见使用ZFS的FreeBSD有最好的性能表现，远好于其它对手。这一点也与 [3]的结论一致。因此，在维护和管理服务器时我们应当采用先进技术，这样可以有效地提升性能。现如今IO瓶颈早已成为制约计算机性能的重要因素，因此采用性能良好的文件系统意义重大。

3.3.3 新的内核版本对减少系统调用开销有重大意义

观察图 5，可以发现在256的BUFFSIZE下，在学校服务器上运行该程序，写入时间相对而言特别长，而随着BUFFSIZE增加，这种对比完全消失了。那么这背后的原因是什么呢？

当BUFFSIZE为256时，需要调用512次**write()**才能

图 4: 不同操作系统上同步写入的时钟时间

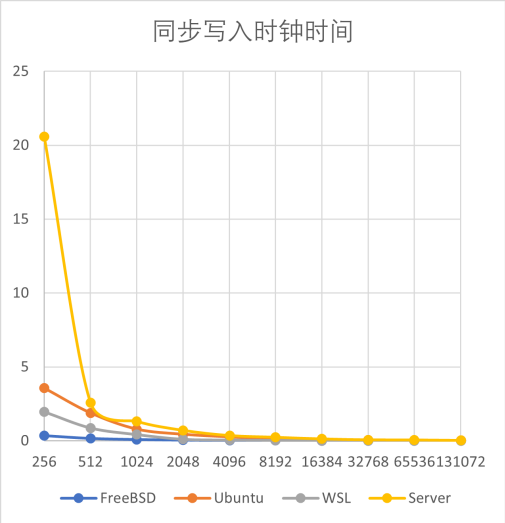
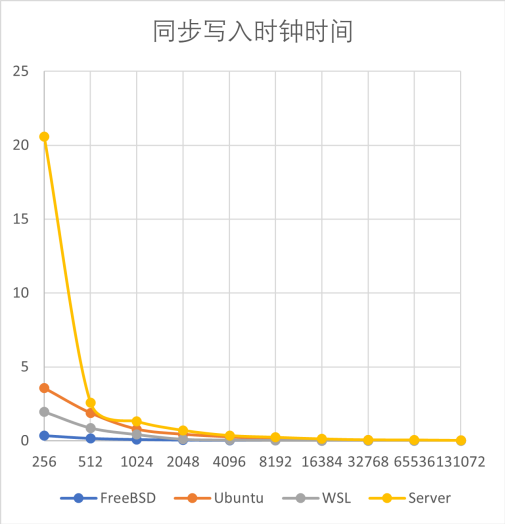


图 5: 不同操作系统上同步写入的时钟时间



4 实验体会

4.1 实验缺陷

附录：实验详细数据

参考文献

References

[1] ioweb gr. [wsl2] filesystem performance is much slower than wsl1 in /mnt. https://github.com/microsoft/WSL/issues/4197, 2019. [Accessed 07-Oct-2022].

[2] Ohad Rodeh and Avi Teperman. zfs-a scalable distributed file system using object disks. In *20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies, 2003.(MSST 2003). Proceedings.*, pages 207–218. IEEE, 2003.

[3] Michael Larabel. Freebsd zfs vs. zol performance, ubuntu zfs on linux reference. https://www.phoronix.com/review/freebsd-zol-april/2, 2019. [Accessed 07-Oct-2022].