



# 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

<b>1</b>	<b>实验内容</b>	<b>1</b>
<b>2</b>	<b>程序设计与实现</b>	<b>1</b>
2.1	程序设计 . . . . .	1
2.2	Makefile . . . . .	1
2.3	程序实现 . . . . .	2
<b>3</b>	<b>实验设计和方法</b>	<b>4</b>
3.1	实验设计 . . . . .	4
3.2	实验方法 . . . . .	4
<b>4</b>	<b>实验结论</b>	<b>5</b>
4.1	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 . . . . .	5
4.2	FreeBSD上的ZFS文件系统确实有优秀的性能表现 . . . . .	5
4.3	新的内核版本对减少系统调用开销有重大意义 . . . . .	5
4.3.1	Linux内核系统调用的实现 . . . . .	6
4.3.2	实验验证 . . . . .	7
<b>5</b>	<b>实验体会</b>	<b>7</b>
5.1	实验缺陷 . . . . .	7
	<b>附录：实验详细数据</b>	<b>8</b>
5.2	WSL上的Ubuntu . . . . .	8
5.3	WSL上的Ubuntu（访问Windows文件系统） . . . . .	8
5.4	VMWare上的Ubuntu . . . . .	8
5.5	VMWare上的FreeBSD . . . . .	8
5.6	服务器上的Linux 2.6 . . . . .	8
	<b>参考文献</b>	<b>9</b>

## 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：

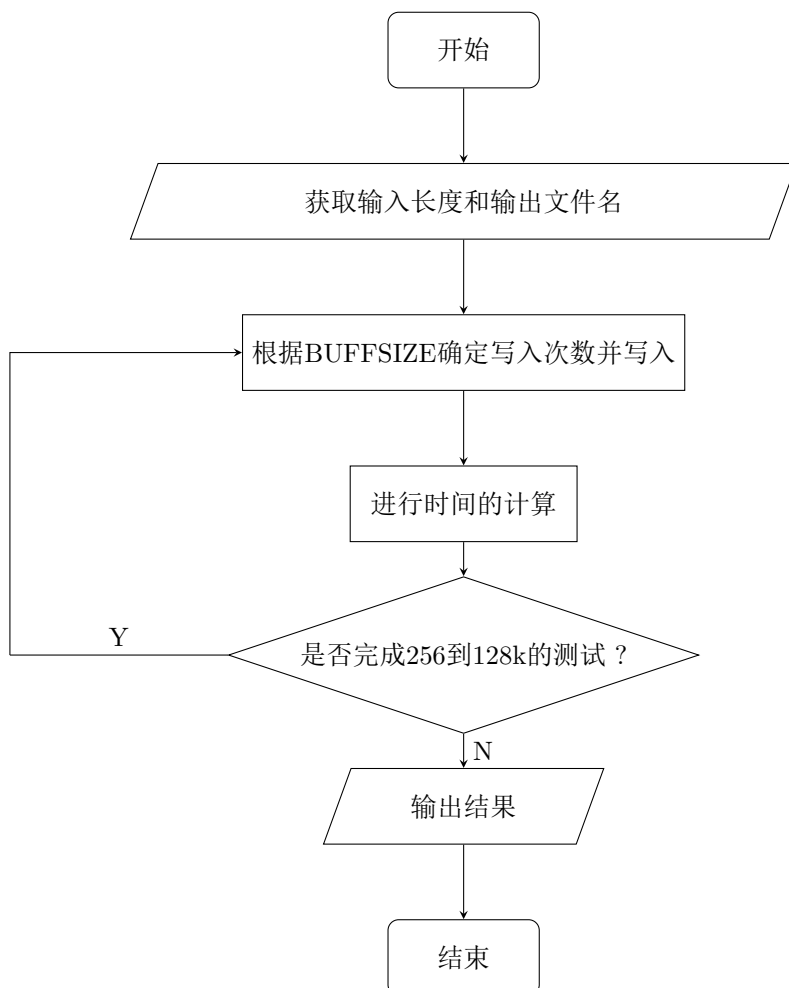
1. `timewrite <f1 f2` 表示输出文件`f2`不用`O_SYNC` 打开。
2. `timewrite f1 sync <f2` 表示输出文件`f1`用`O_SYNC` 打开。

## 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

```
1 OUT = build
2 SRC = .
3 CC = gcc
4 LD = ld
5
6 all : timewrite
7
8 ${OUT}:
9     mkdir -p ${OUT}
10
11 ${OUT}/timewrite.o: ${OUT} ${SRC}/timewrite.c
12     $(CC) -std=c99 -c ${SRC}/timewrite.c -o ${OUT}/timewrite.o
13
14 timewrite: ${OUT}/timewrite.o
15     $(CC) ${OUT}/timewrite.o -o timewrite
16
17 clean:
18     rm -rf ${OUT} timewrite
19
20 .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

```
1 #include <sys/times.h>
2 #include <unistd.h>
3 #include <fcntl.h>
4
5 #include <stdio.h>
6 #include <assert.h>
7 #include <string.h>
8 #include <stdlib.h>
9
10 #define FILE_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
11
12
13 int main(int argc, char** argv)
14 {
15     assert(argc == 2 || argc == 3);
16
17     if (argc == 3 && strcmp(argv[2], "sync") != 0)
18     {
19         printf("usage: %s <pathname> [sync]", argv[0]);
20         return 1;
21     }
22
23     const int mode = (argc == 3) ? O_RDWR | O_CREAT | O_TRUNC | O_SYNC : O_RDWR | O_CREAT | O_TRUNC;
24
```

```
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%.4lf\t%.4lf\t%.4lf\t%d\n", bufsize, user_time, sys_time,real_time, loop);
91
92 }
93
94 return 0;
95 }
```

### 3 实验设计和方法

#### 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

```
[cs204622@mc0re 1]$ ./timewrite output sync<input
256      0.0000  0.0000 10.5000 512
512      0.0000  0.0000  1.0200 256
1024     0.0000  0.0000  0.5100 128
2048     0.0000  0.0000  0.2500  64
4096     0.0000  0.0000  0.1300  32
8192     0.0000  0.0000  0.0800  16
16384    0.0000  0.0000  0.0500   8
32768    0.0000  0.0000  0.0400   4
65536    0.0000  0.0000  0.0300   2
131072   0.0000  0.0000  0.0400   1
[cs204622@mc0re 1]$
```

不同平台的实验结果详细数据见附录：实验详细数据

## 4 实验结论

### 4.1 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)

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

结果如图 6所示。

图 3: 异步写入加速比较



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。

### 4.2 FreeBSD上的ZFS文件系统确实有优秀的性能表现

BSD上的ZFS [2]被证实具有良好的性能表现，不仅仅和ZFS on Linux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 5。

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。

### 4.3 新的内核版本对减少系统调用开销有重大意义

观察图 ??，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。那么这种异常的高开销的原因是什么呢？

图 4: 不同操作系统上同步写入的时钟时间

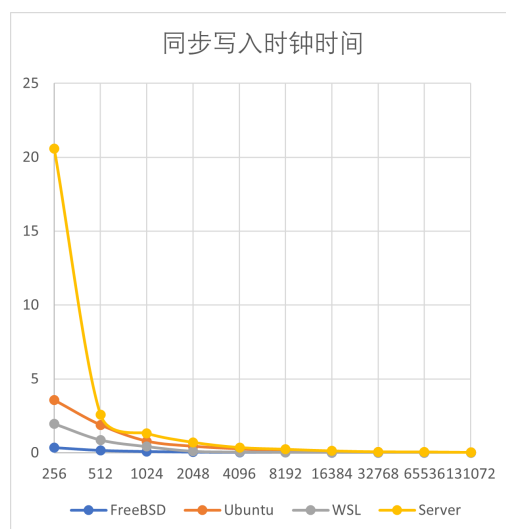
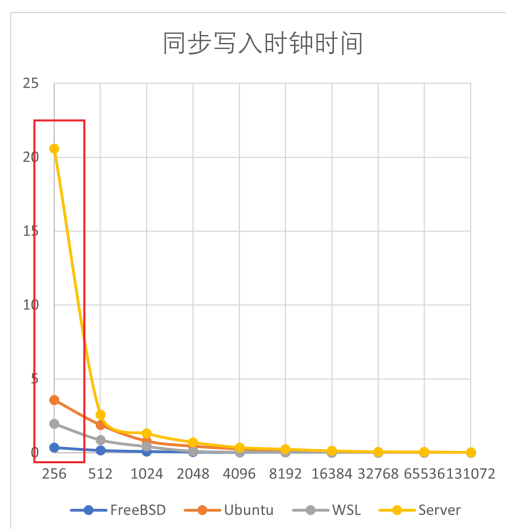


图 5: 服务器上的Linux 2.6具有异常高的写入开销



当BUFSIZE为256时，需要调用512次write()才能写入全部数据，而每次调用write()都要进行系统调用。而从内核态切换到用户态是有开销的。特别是当系统调用的实现方式比较传统时，该开销是相当可观的。

#### 4.3.1 Linux内核系统调用的实现

自CPU引入保护模式以来，应用程序开始只能工作在具有诸多限制的用户态。为了允许应用程序实现那些需要更高特权级的指令具有的功能，需要一种机制来允许应用程序进入内核态。这种机制在UNIX/类UNIX内核下就是系统调用。Linux内核的系统调用实现大致分为两个阶段。

- **INT 80** 早期的Linux内核和当时绝大多数UNIX内核的系统调用都采用了软中断的形式。中断指令会自动将当前的状态保存在栈中，然后将控制权移交给运行在内核态的中断处理程序。在Linux内核中，使用的中断是**INT 0x80**。并且系统调用的调用者会把系统调用编号存储在EAX寄存器中。这样的机制的优点是系统调用的处理和中断的处理可以使用同一套机制，保存状态等过程实现简单。但是这种机制的缺点是并非所有的寄存器都需要保存，因此这个读写内存的过程大大降低了处理系统调用的效率。
- **sysenter/syexit和syscall/sysret** sysenter/syexit和syscall/sysret分别由Intel [4]和AMD [5]引入，用于快速地实现内核态和用户态的切换。它们没有寄存器状态入栈的内存操作，并且使用MSR寄存器来传



递所需的信息。这样的机制有更高的性能。Linux内核在版本2.6中开始引入基于这些指令实现的系统调用机制。

在Linux内核上进行系统调用依赖GLibC库。考虑到Linux 2.6刚刚引入新的快速系统调用机制，我作出假设 4.1:

**Assumption 4.1.** 配套的GLibC库可能没有恰当地更新，以至于旧的系统调用机制仍然在使用，这意味着较高的开销，再加上`BUFSIZE`为256时较多的调用次数，使得总体开销显得过高。

### 4.3.2 实验验证

为了验证这个猜想，我查阅了相应GLibC版本的源码。首先通过`ldd--version`命令获取了服务器上的GLibC版本是2.5。因此查找相应的分支的代码。GLibC进行系统调用的源码位置位于`sysdeps/unix/sysv/linux/i386/sysdep.h`。该文件为i386架构的可执行文件提供系统调用的实现。内容如图 6a。其中`ENTER_KERNEL`的内容如图 6b。

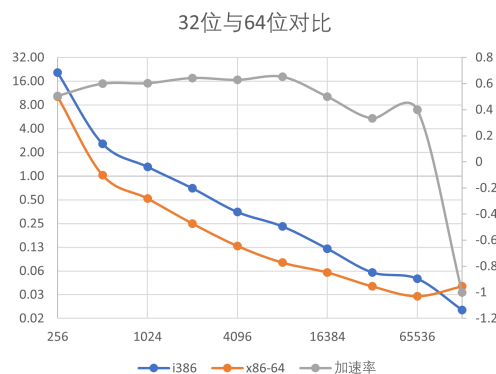
图 6: GLibC的系统调用实现



可见32位系统调用实现由编译参数定义的宏的不同，可能使用旧的INT80机制的，这可能带来较大的开销。而对应的64位系统调用直接使用`syscall/sysret`指令组合，一定会使用新的系统调用机制。

因此，为了验证猜想 4.1，我重新编译了64位版本的timewrite，运行结果如图 7。可以看出64位版本的timewrite的写入比32位版本快约50%以上。从常识出发，这超出了字长的变化可能对性能带来的影响。因此可以认为节省的时间是由于系统调用实现机制的不同而导致的效率区别，进而可以认为猜想 4.1是正确的。

图 7: 64位版本与32位版本结果比较



## 5 实验体会

### 5.1 实验缺陷

## 附录：实验详细数据

### 5.2 WSL上的Ubuntu

### 5.3 WSL上的Ubuntu（访问Windows文件系统）

### 5.4 VMWare上的Ubuntu

### 5.5 VMWare上的FreeBSD

### 5.6 服务器上的Linux 2.6

## 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].
- [4] Part Guide. Intel® 64 and ia-32 architectures software developer’s manual. *Volume 3B: System programming Guide, Part, 2*(11), 2011.
- [5] ARM AMD. Amd64 architecture programmers manual volume 2: System programming, 2018.