

# CS353 Linux 内核 Final Project

## 实验要求

本实验包括内核态部分和用户态部分：在内核态中需要编写一个内核模块，可以获取程序的实际运行时间和内存读写量；在用户态中编写一个程序，定时通过内核模块跟踪程序的 CPU 利用率和内存读写量，两者比较分析程序是计算密集的还是内存密集的。

内核模块一个可能的实现方式为：创建一个 proc 文件，向 proc 文件写入进程 PID 之后，每次读取 proc 文件，可以得到上次读取文件之后进程调度到 CPU 上的运行时间以及读写的内存页个数。

用户态程序则需要启动 benchmark 进程，将进程的 PID 写入 proc 文件，然后定时读取 proc 文件，计算得到进程的 CPU 使用率以及内存读写频率，并写入一个日志文件。

具体的实现是开放的，但是你的实现应该通过内核模块编程从内核数据结构获得数据，**而不是通过内核已经实现的 proc 文件获得。**

## 实验过程

### ● 内核态部分

#### 1、CPU 上的运行时间

进程实际运行时间的获取可以参考内核中对于 proc 文件 `/proc/[pid]/stat` 的实现。**该文件中有一个参数为 `utime`**，可以一定程度上表示该进程的实际运行时间。

查阅linux源码，参考 `/proc/[pid]/stat` 的实现，文件创建部分在 `/fs/proc/base.c` 中，可知 `stat` 中的值是由 `proc_tgid_stat` 函数所得到的。

```
ONE("stat",          S_IRUGO, proc_tgid_stat),
```

`do_task_stat` 由 `proc_tgid_stat` 封装起来，真正的实现在 `do_task_stat` 中

```
return do_task_stat(m, ns, pid, task, 1);
```

阅读 `do_task_stat` 关于 `utime` 部分。

```
thread_group_cputime_adjusted(task, &utime, &stime);
```

如下代码所示，`utime` 在 `thread_group_cputime` 中被赋值，第一轮读取时尝试无RCU锁(Read-Copy Update)读取，若读取失败则获得该锁，并用 `for_each_thread` 读取每个线程的 `utime`，逐步累加起来，得到整个进程的 `utime`，过程如下：

```
rcu_read_lock();
/* Attempt a lockless read on the first round. */
nextseq = 0; // 第一次尝试无锁读取
do {
    seq = nextseq;
    flags = read_seqbegin_or_lock_irqsave(&sig->stats_lock, &seq);
    times->utime = sig->utime;
    times->stime = sig->stime;
```

```

times->sum_exec_runtime = sig->sum_sched_runtime;
// 多线程累加
for_each_thread(tsk, t) {
    task_cputime(t, &utime, &stime);
    times->utime += utime;
    times->stime += stime;
    times->sum_exec_runtime += read_sum_exec_runtime(t);
}
/* If lockless access failed, take the lock. */
nextseq = 1;
} while (need_seqretry(&sig->stats_lock, seq));
done_seqretry_irqrestore(&sig->stats_lock, seq, flags);
rcu_read_unlock();

```

参考 `thread_group_cputime` 的实现，对函数进行简化，实现自己的版本。

utime先由signal中的utime初始化后，再由各个线程的utime累加起来，并且舍弃了锁的

```

static u64 get_utime(struct task_struct *taskp)
{
    u64 utime = taskp->signal->utime;
    struct task_struct* t;
    // 多线程累加
    for_each_thread(taskp, t) {
        utime += t->utime;
    }
    return utime;
}

```

## 2、读写的内存页个数

判断该页是否被进程读写：页目录项中有一个标志位 `young`，CPU 在每次访问该页时会将该标志位置为 1。可以通过先清空该标志位，一段时间后再读取的方式，判断进程在这段时间内是否读写该页。

内核中有函数 `ptep_test_and_clear_young` 函数可以在获取 `young` 标志位的同时清空它，可以达到其功能。`ptep_test_and_clear_young` 首先利用 `pte_young()` 宏来判断Linux版本的页表项中是否包含 `L_PTE_YOUNG` 比特位，如果没有设置该比特位，则返回0，表示映射PTE最近没有被访问引用过。如果 `L_PTE_YOUNG` 设置比特位，那么需要调用 `test_and_clear_bit` 来清这个比特位。

```

int ptep_test_and_clear_young(struct vm_area_struct *vma,
                             unsigned long addr, pte_t *ptep)
{
    int ret = 0;
    if (pte_young(*ptep))
        ret = test_and_clear_bit(_PAGE_BIT_ACCESSED,
                                (unsigned long *) &ptep->pte);
    return ret;
}

```

遍历该进程的每个内存页，判断该页是否被进程读写。

```

static int get_pagenum(struct task_struct *taskp)
{
    int cnt = 0;
    unsigned long vaddr;
    struct vm_area_struct *vma;

```

```

    for (vma = taskp->mm->mmap; vma; vma = vma->vm_next){
        for (vaddr = vma->vm_start; vaddr < vma->vm_end; vaddr += PAGE_SIZE) {
            cnt += _ptep_test_and_clear_young(taskp, vaddr);
        }
    }

    return cnt;
}

```

参考内核 `ptep_test_and_clear_young` 代码，实现自己的版本，根据 `task_struct` 和虚拟地址 `vaddr` 即页的首地址得到 `pte`，查询 `pte` 是否被访问过，若访问过则清空 `young` 标志位，一段时间后再读取，判断进程在这段时间内是否读写该页，若访问过则返回1，表示该页正被进程所读写。

```

static inline int _ptep_test_and_clear_young(struct task_struct *taskp,
unsigned long vaddr)
{
    int ret = 0;
    pgd_t *pgd;
    p4d_t *p4d;
    pud_t *pud;
    pmd_t *pmd;
    pte_t *pte;

    pgd = pgd_offset(taskp->mm, vaddr);
    if (pgd_none(*pgd))
        return 0;
    p4d = p4d_offset(pgd, vaddr);
    if (p4d_none(*p4d))
        return 0;
    pud = pud_offset(p4d, vaddr);
    if (pud_none(*pud))
        return 0;
    pmd = pmd_offset(pud, vaddr);
    if (pmd_none(*pmd))
        return 0;
    pte = pte_offset_kernel(pmd, vaddr);

    if(pte_young(*pte))
        ret = test_and_clear_bit(_PAGE_BIT_ACCESSED, (unsigned long *)&pte->pte);
    return ret;
}

```

在 `proc_read` 函数中调用上述函数，得到该进程的 `utime` 和内存页个数。再用 `sprintf` 将两个参数读入 `output`，在复制到缓存中。

```

static ssize_t proc_read(struct file *fp, char __user *ubuf, size_t len, loff_t
*pos)
{
    int count = 0; /* the number of characters to be copied */
    u64 utime=0;
    unsigned long pagenum = 0;

    if (*pos == 0) {
        /* a new read, update process' status */

```

```

        /* TODO */
        utime = get_utime(taskp);
        pagenum = get_pagenum(taskp);
        sprintf(output, "%11d, %1d\n", utime, pagenum);
        out_len = strlen(output);
    }

    if (out_len - *pos > len) {
        count = len;
    } else {
        count = out_len - *pos;
    }

    pr_info("Reading the proc file\n");
    if (copy_to_user(ubuf, output + *pos, count)) return -EFAULT;
    *pos += count;

    return count;
}

```

## ● 用户态部分

用户态程序需要启动 benchmark 进程，将进程的 PID 写入 proc 文件，然后定时读取 proc 文件，计算得到进程的 CPU 使用率以及内存读写频率，并写入一个日志文件。

本实验以sysbench为例进行测试：

### 1、CPU testbench

sysbench的cpu测试是在指定时间内，循环进行素数计算。sysbench将通过将数字除以2和该数字的平方根之间的所有数字来验证素数。如果任何一个数的余数为0，则计算下一个数。

`-cpu-max-prime`：素数生成数量的上限

`-threads`：线程数

`-time`：运行时长，单位秒

`-events`：event上限次数

运行sysbench测试程序，然后用pgrep根据程序名称返回相应进程标识符，定向输入到 `/proc/watch` 文件中。

```
sysbench cpu --cpu-max-prime=2000000 --threads=2 run
```

### 2、memory testbench

当在sysbench中使用内存测试时，基准测试应用程序将分配一个内存缓冲区，然后对其进行读写，每次都是一个指针的大小(因此是32位或64位)，每次执行，直到读取或写入总缓冲区大小。然后重复此操作，直到达到所提供的卷(--memory-total-size)为止。

```
sysbench --test=memory --num-threads=4 run
```

### 3、fileio testbench

在使用fileio时，将创建一组测试文件。

```
sysbench --test=fileio --file-test-mode=seqwr run
```

每隔0.01秒读取 /proc/watch 文件内容，保存在一个文本文件中；

```
while true;
do
cat /proc/watch >> test.txt;
sleep 0.01;
done;
```

## CPU 利用率的计算

参数	解释
utime	该任务在用户态运行的时间，单位为jiffies
stime	该任务在内核态运行的时间，单位为jiffies
cutime	所有已死线程在用户态运行的时间，单位为jiffies
cstime	所有已死在内核态运行的时间，单位为jiffies

· Jiffies 为 Linux 核心变数，是一个 unsigned long 类型的变量，它被用来记录系统自开机以来，已经过了多少 tick。每发生一次 timer interrupt，jiffies 变数会被加 1。

进程的总CPU时间：

$$processCpuTime = utime + stime + cutime + cstime$$

线程的CPU时间：

$$threadCpuTime = utime + stime$$

总的CPU时间：

$$totalCpuTime = user + nice + system + idle + iowait + irq + softirq + stealstolen + guest$$

某一进程CPU利用率的计算：

· 采样两个足够短的时间间隔的cpu快照与进程快照，

a) 每一个cpu快照均为(user、nice、system、idle、iowait、irq、softirq、stealstolen、guest)的9元组；

b) 每一个进程快照均为(utime、stime、cutime、cstime)的4元组；

· 分别计算出两个时刻的总的cpu时间与进程的cpu时间，分别记作：

$$totalCpuTime1、totalCpuTime2、processCpuTime1、processCpuTime2$$

· 计算该进程的cpu使用率

$$pcpu = 100 * (processCpuTime2 - processCpuTime1) / (totalCpuTime2 - totalCpuTime1)$$

(按100%计算，如果是多核情况下还需乘以cpu的个数)；

## 内存读写量的计算

linux内存页默认大小是4K，根据得到的读写内存页个数就可以得到该进程的读写的内存大小，（但是只能大致估计，不太准确，在思考题3中有阐释）。

## 实验步骤

### 1、编译

```
make
```

### 2、安装模块

```
sudo insmod watch.ko
```

### 3、查看文件是否创建

```
ls /proc/watch
```

### 4、指定pid

```
echo [pid] > /proc/watch
```

### 5、查看结果

```
cat /proc/watch
```

### 6、运行测试程序，得到结果（一个文本文件）

```
sh test.sh
```

### 7、对结果进行分析。

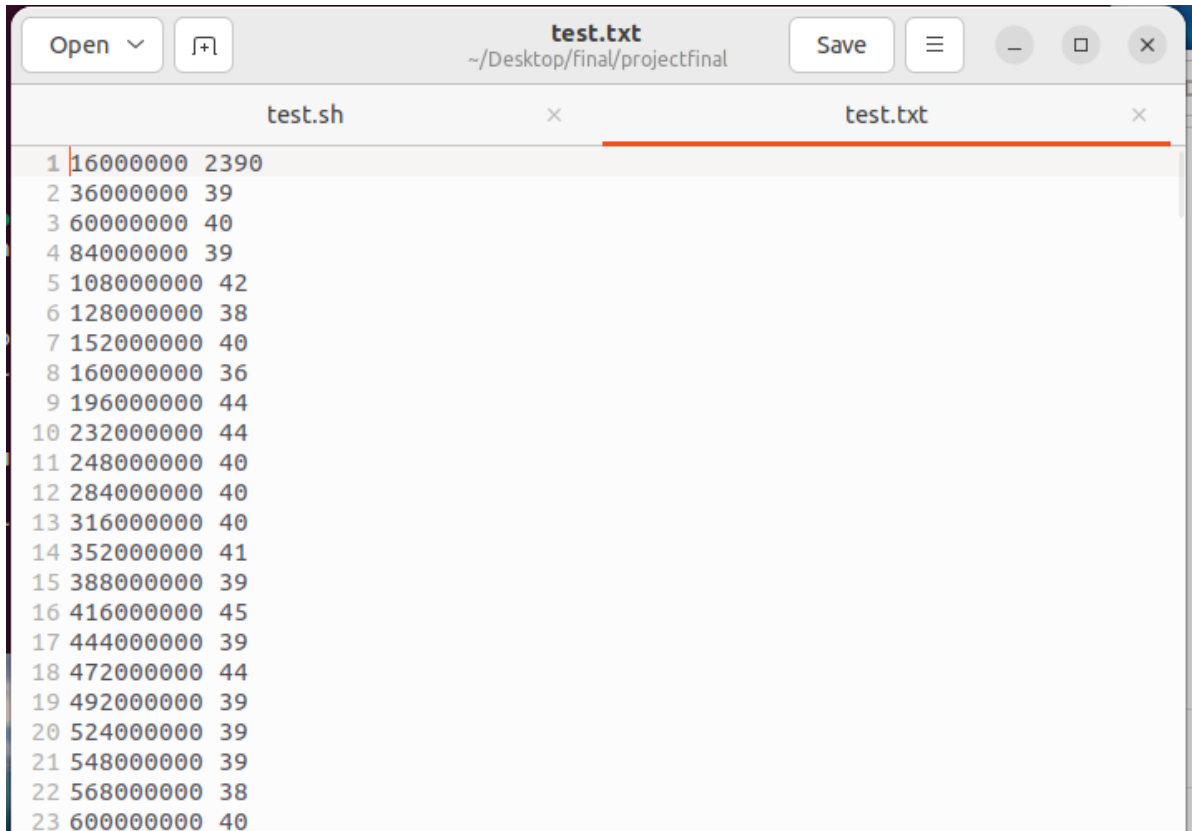
## 实验结果

### 1、模块安装与运行

```
pengjq@pengjq-virtual-machine:~/Desktop/final/projectfinal$ make
make -C /lib/modules/5.15.46/build M=/home/pengjq/Desktop/final/projectfinal modules
make[1]: Entering directory '/usr/src/linux-5.15.46'
  CC [M]  /home/pengjq/Desktop/final/projectfinal/watch.o
  MODPOST /home/pengjq/Desktop/final/projectfinal/Module.symvers
  CC [M]  /home/pengjq/Desktop/final/projectfinal/watch.mod.o
  LD [M]  /home/pengjq/Desktop/final/projectfinal/watch.ko
make[1]: Leaving directory '/usr/src/linux-5.15.46'
pengjq@pengjq-virtual-machine:~/Desktop/final/projectfinal$ sudo insmod watch.ko
pengjq@pengjq-virtual-machine:~/Desktop/final/projectfinal$ echo 1 > /proc/watch

pengjq@pengjq-virtual-machine:~/Desktop/final/projectfinal$ cat /proc/watch
152000000, 348
pengjq@pengjq-virtual-machine:~/Desktop/final/projectfinal$
```

## 2、运行脚本文件，得到以下结果



```
test.txt
~/Desktop/final/projectfinal
Save

test.sh × test.txt ×
1 160000000 2390
2 360000000 39
3 600000000 40
4 840000000 39
5 1080000000 42
6 1280000000 38
7 1520000000 40
8 1600000000 36
9 1960000000 44
10 2320000000 44
11 2480000000 40
12 2840000000 40
13 3160000000 40
14 3520000000 41
15 3880000000 39
16 4160000000 45
17 4440000000 39
18 4720000000 44
19 4920000000 39
20 5240000000 39
21 5480000000 39
22 5680000000 38
23 6000000000 40
```

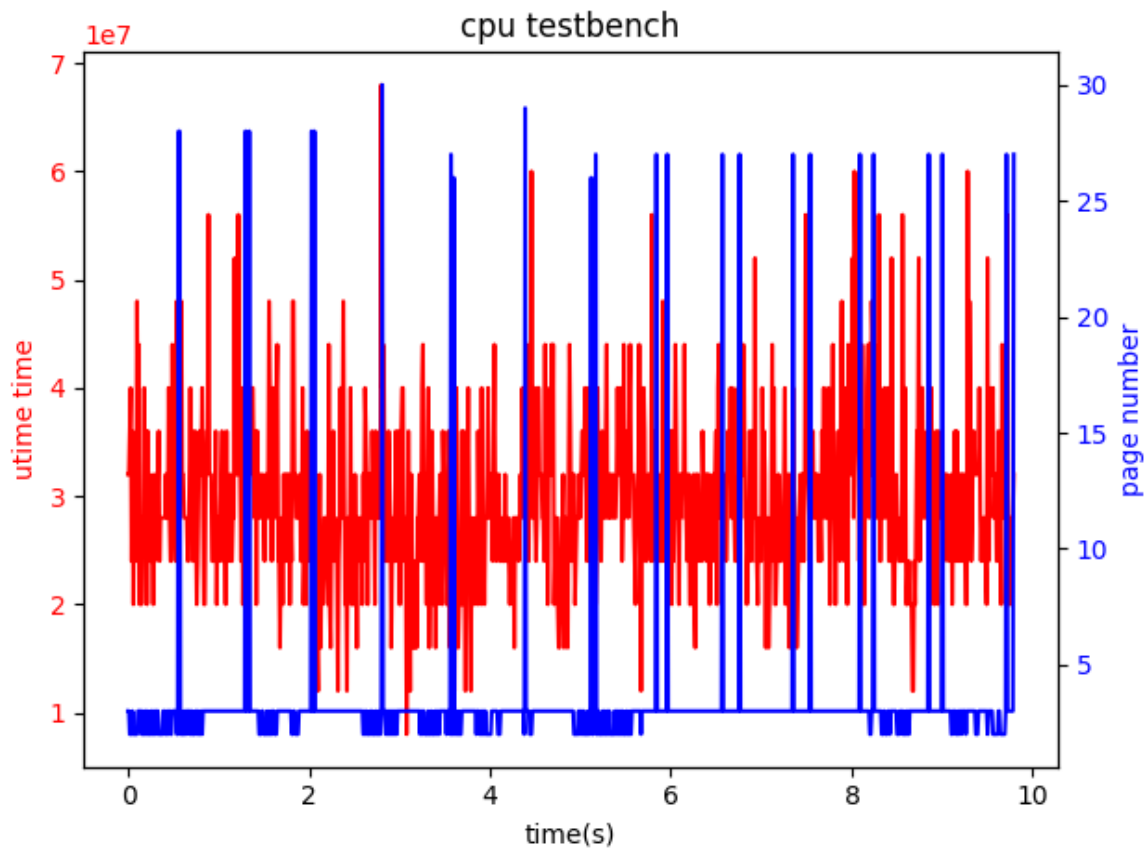
## 3、进程分析

选取不同的 benchmark 进程，绘制 CPU 使用率和内存读写频率随时间的变化图。

### ● cpu testbench

```
sysbench cpu --cpu-max-prime=2000000 --threads=2 run
```

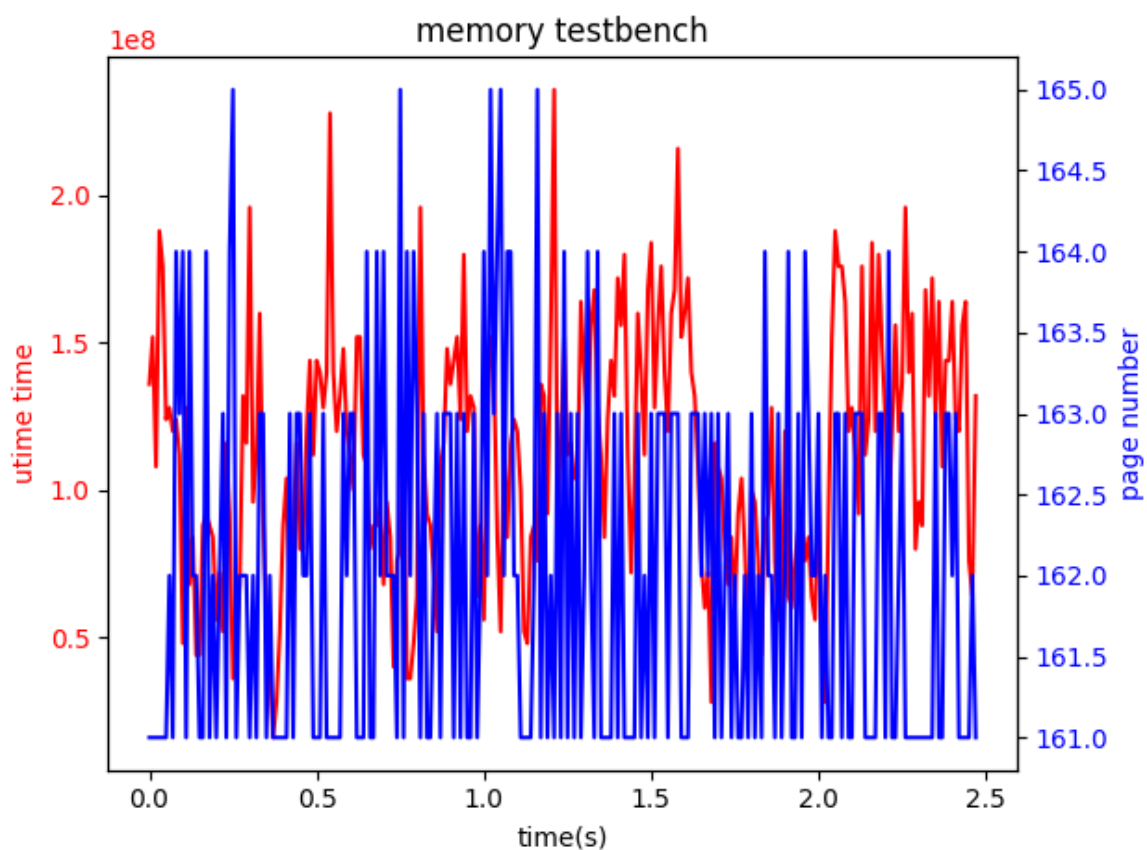
由下图可知，在测试CPU程序时，周期性读写内存，但是读写内存页数量较少，但是占用CPU时间多，属于计算密集型。



- memory testbench

```
sysbench --test=memory --num-threads=4 run
```

由下图可知，在测试memory 程序时，内存页读写数量大，utime也比较大，因为线程数相较上一次增大一倍。相较之下属于内存密集型。

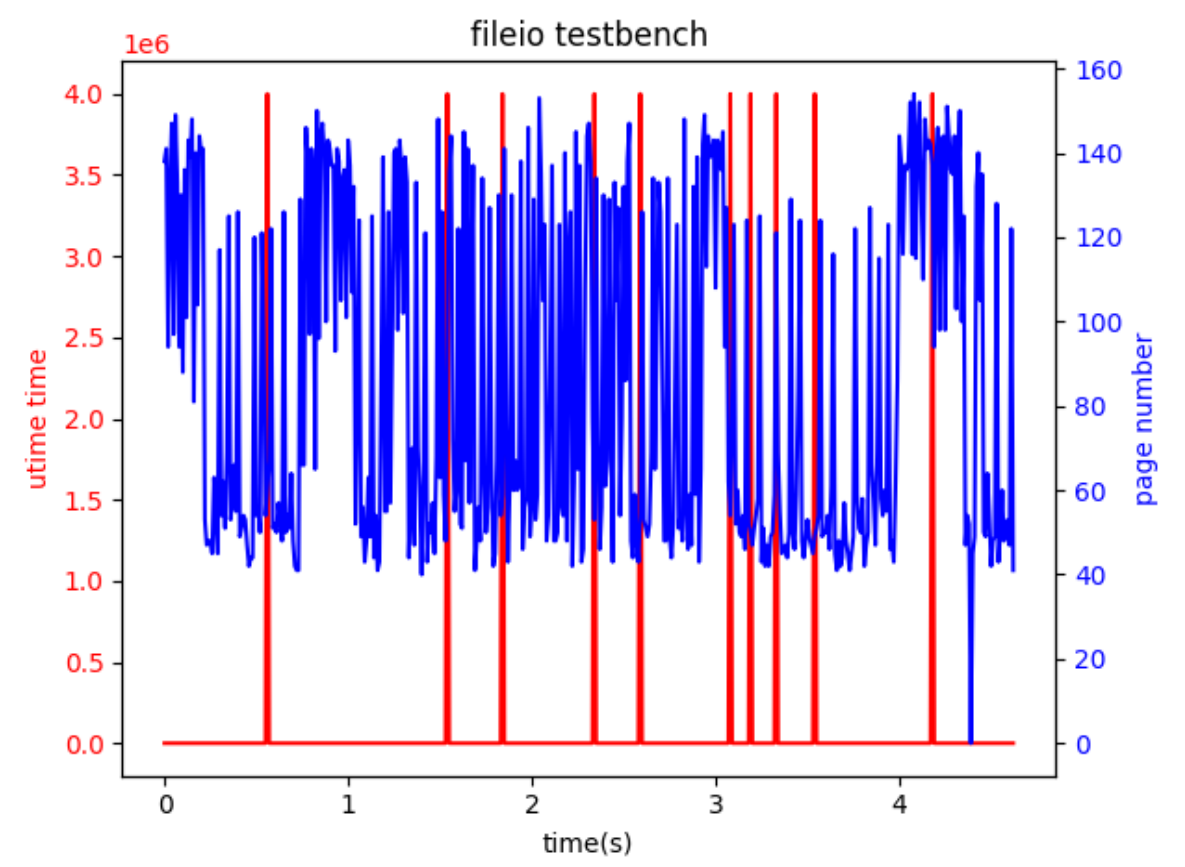




● fileio testbench

```
sysbench --test=fileio --file-test-mode=seqwr run
```

由下图可知，在测试fileio程序时，内存页读写数量大，一直处在比较高的状态，utime比较小，几乎不占CPU时间。相较之下属于内存密集型。



思考题

(注：以下问题可以在实验报告中回答，也可以实现在程序中，实现在程序中请注明)

1. 大多数 benchmark 是多线程或者是多进程的，你的程序中有考虑这种情况吗？若没有，应该怎么解决？

已考虑该情况，用 `for_each_thread` 对每个线程进行遍历。

2. `utime` 表示程序程序的什么时间？`/proc/[pid]/stat` 还有一个参数为 `stime`，它表示什么？你觉得在这个实验里面适合使用它吗？

参数	解释
utime	该任务在用户态运行的时间，单位为jiffies
stime	该任务在内核态运行的时间，单位为jiffies
cutime	所有已死线程在用户态运行的时间，单位为jiffies
cstime	所有已死在内核态运行的时间，单位为jiffies

· Jiffies 为 Linux 核心变数，是一个 unsigned long 类型的变量，它被用来记录系统自开机以来，已经过了多少 tick。每发生一次 timer interrupt，Jiffies 变数会被加 1。

`stime` 在这个实验里不合适。

3.你觉得当前实验中以页为单位统计进程的内存读写合适吗？如果合适，原因是什么？如果不合适，有没有更好的方法？

不合适，当前实验以一个完整的页大小来统计进程的内存读写量，但是进程读写了该页并不意味着使用了页的所有部分，还存在未读写的部分，所以得到的内存读写量与实际还是有些偏差（偏大）。

## 实验心得

---

这次实验综合这学期所学的知识，通过编写一个内核模块，获取程序的实际运行时间和内存读写量，设计的知识有内存管理，进程管理等，使我对这学期的知识更加有所巩固。这次实验令我对linux源码进一步熟悉起来，发现自己相比刚学时有些许进步，可以根据功能需求快速定位所有可能用得着的函数，对进程task\_struct、struct page 结构等的内涵更加熟悉起来。同时也积攒了不少小技巧，如虚拟地址到物理地址的映射等操作。通过这次实验我终于明白了通过ps或者top命令得到的参数如CPU利用率、内存占用等的计算方式，日后运行某个进程可以对其进行更细致的分析。

最后，谢谢助教和老师的指导，使我在这学期收获了许多，不仅收获了linux内核的相关知识，还锻炼了编写调试模块的能力。