CS353: Linux kernel Project final 实验报告

1 内核态部分

1.1 大致思路

本次实验要求编写一个内核模块，通过该模块可以获取进程的实际运行时间和内存的读写量。在实验指导书的启发下，我决定使用`/proc`文件系统来完成这个功能，实现用户和内核之间的信息传递。编写内核模块并创建`/proc/watch`文件，将需要统计的进程的`pid`写入`/proc/watch`文件中，内核模块会统计该进程运行的时间参数：`utime`，`stime`，`cutime`，`cstime`和内存读写的页数。每当用户读取`/proc/watch`文件时，内核模块将上述的5个参数传递到用户态中，最终实现对进程实际运行时间和内存读写量的统计。

1.2 实现步骤

在本节中，我将展示向`/proc/watch`写入目标进程的`pid`后，内核模块如何通过访问Linux kernel中相关的数据结构来获取目标进程的运行时间和内存读写量，并将其传递到用户态中。

1.2.1 进程运行时间统计

本次实验的目的是统计进程运行的时间，进程执行总时间的计算公式：
`pc=utime+stime+cutime+cstime`。

> `utime`指进程在用户态的运行时间；
>
> `stime`指进程在内核态的运行时间；
>
> `cutime`指所有层次子进程在用户态的运行时间总和；
>
> `cstime`指所有层次子进程在核心态的运行时间总和

在`proc`文件系统中，`/proc/pid/stat`：包含了进程的状态信息，具体如下图所示：

```
chn@chn-virtual-machine:~$ cat /proc/2309/stat
2309 (bash) S 2281 2309 2309 34816 2495 4194304 886 4526 3 106 1 1 4 22 20 0 1 0 10947 14159872 1277 18446744073709551615 94270793773056 94270794495701 140732739917152 0 0 0 65536 367
0020 1266777851 1 0 0 17 4 0 0 0 0 0 94270794726640 94270794774020 94270800961536 140732739922284 140732739922289 140732739922289 140732739923950 0
```

其中的第14、15、16、17位分别表示该进程的`utime`，`stime`，`cutime`和`cstime`。关于上述参数的详细解释请参阅Linux manual page。根据实验指导书的要求，不能借助现有的proc文件来获取进程信息，而需要编写内核模块来实现获取上述参数的功能。因此，通过阅读源码来了解内核是如何统计CPU运行时间的。下面简述一下源码中的函数功能和相互调用关系：

1. `fs/proc/base.c`

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

2. `fs/proc/array.c`

```
int proc_tgid_stat(struct seq_file *m, struct pid_namespace *ns,
                   struct pid *pid, struct task_struct *task)
{
        return do_task_stat(m, ns, pid, task, 1);
}
```

3. `do_task_stat()`

   根据上述两步的函数调用，确定了`/proc/pid/stat`文件对应到内核中的函数：`do_task_stat`。该函数中向文件中输入时间信息的部分如下：

```
seq_put_decimal_ull(m, " ", nsec_to_clock_t(utime));
seq_put_decimal_ull(m, " ", nsec_to_clock_t(stime));
seq_put_decimal_ll(m, " ", nsec_to_clock_t(cutime));
seq_put_decimal_ll(m, " ", nsec_to_clock_t(cstime));
```

可以看到在该函数在此处已经获取到了这四个时间信息，在该部分只对它们需要进行单位转换，将nsec转换为clock tick就可以直接输入到文件中了。而获取上述的四个时间信息的步骤如下：

```
    u64 cutime, cstime, utime, stime;//初始化
    unsigned long flags;
    ......
    cutime = cstime = utime = stime = 0;//初始化后首先四个时间信息变量皆是赋值为0
    ......
    if (lock_task_sighand(task, &flags)) {
        struct signal_struct *sig = task->signal;
        ......
        cutime = sig->cutime;//只有当lock_task_sighand()函数执行返回非0时
        cstime = sig->cstime;//赋值cutime、cstime，数据来源task->signal
        ......
        /* add up live thread stats at the group level */
        if (whole) {
            ......
            thread_group_cputime_adjusted(task, &utime, &stime);
        }//whole非0且lock_task_sighand()非0，调用
thread_group_cputime_adjusted()更新utime、stime
        ......
        unlock_task_sighand(task, &flags);
    }
    ......
    if (!whole) {//只需要whole为0便可调用task_cputime_adjusted()更新utime、stime
        .......
        task_cputime_adjusted(task, &utime, &stime);
        .......
    }
/*
注：函数lock_task_sighand()和unlock_task_sighand()的作用分别是对task_struct中的成员sighand上锁和解锁(sighand：指向进程的信号处理程序描述符)，也就是说在这段时间内task_struct中的成员sighand指向的信号处理程序描述符暂时不会处理来自其他地方的信号处理要求。
*/
```

大体步骤可以总结如下：首先初始化变量 `cutime`，`cstime`，`utime`，`stime`。在加锁成功后，获取该进程所对应的 `task_struct->signal->cutime` 和 `task_struct->signal->cstime` 并将其赋值给 `cutime` 和 `cstime`。至此，子进程相关的CPU时间参数统计完成。而 `utime` 和 `stime` 的计算，根据 `whole` 是否为0， `do_task_stat` 选择调用了两个不同的函数 `thread_group_cputime_adjusted()` 和 `task_cputime_adjusted()` 来更新 `utime` 和 `stime`。

```
int proc_tid_stat(struct seq_file *m, struct pid_namespace *ns,
                  struct pid *pid, struct task_struct *task)
{
        return do_task_stat(m, ns, pid, task, 0);
}

int proc_tgid_stat(struct seq_file *m, struct pid_namespace *ns,
                  struct pid *pid, struct task_struct *task)
{
        return do_task_stat(m, ns, pid, task, 1);
}
```

由上面 `do_task_stat()` 的参数定义，可知最后一个参数就是 `whole`。并且由上面的函数 `proc_tid_stat()` 和 `proc_tgid_stat()` 可以得出结论，当 `do_task_stat()` 要输出的进程的 `pid` 等于该组进程的 `tgid` 的时候，`whole` 为1；反之，`whole` 为0。

注：此处关于whole的取值以及相关函数的调用可以回答思考题1

当whole取值为1是，utime和stime统计的cpu时间会包含同一个线程组的所有线程，以此达到统计多线程的目的。

4. `thread_group_cputime_adjusted()`

   在 `whole` 为1的情况下，`do_task_stat()` 选择调用的函数是 `thread_group_cputime_stat()`

```
void thread_group_cputime_adjusted(struct task_struct *p, u64 *ut, u64 *st)
{
        struct task_cputime cputime;

        thread_group_cputime(p, &cputime);

        *ut = cputime.utime;
        *st = cputime.stime;
}
```

而 `thread_group_cputime_adjusted()` 又调用了 `thread_group_cputime` 来获得相应的utime和stime

```
void thread_group_cputime(struct task_struct *tsk, struct task_cputime
*times)
{
    struct signal_struct *sig = tsk->signal;
    u64 utime, stime;
    struct task_struct *t;
    unsigned int seq, nextseq;
    unsigned long flags;
    //如果当前要查询的进程正在运行，那么更新其sum_sched_runtime。
    if (same_thread_group(current, tsk))
        (void) task_sched_runtime(current);

    rcu_read_lock();
    /* Attempt a lockless read on the first round. */
    nextseq = 0;
    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;
```

```
        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();
}
```

在该函数中，首先获得pid=tgid的进程的`task_struct->signal->utime`和`task_struct->signal->stime`,之后，通过调用`for_each_thread`,遍历该进程组的所有task_struct结构体，获得utime和stime的总和，最终统计得到整个线程组的utime和stime。

至此，梳理了`/proc/pid/stat`文件是如何统计特定进程的相关时间信息。**参考内核的相关代码，我实现了自己的`/proc/watch`文件中关于CPU时间的信息统计。具体的代码实现请参见提交文件中`519021910480_src/watch.c/cpu_measure()`的内容。**

1.2.2 内存读写统计

本节的目的是完成内核模块统计相关进程的内存读写量的功能。

1. 遍历`struct vm_area_struct`结构体

   Linux 内核使用`vm_area_struct`结构来表示一个独立的**线性区**，由于每个不同地址的虚拟内存区域功能和内部机制都不同，因此 一个进程使用多个`vm_area_struct`结构来分别表示不同类型的虚拟内存区域，包括虚拟内存的起始和结束地址，以及内存的访问权限等 。各个`vm_area_struct`结构使用链表或者树形结构链接，方便进程快速访问。

   以下是《深入理解LINUX内核第3版》对于**线性区**的表述：

   进程的地址空间（*address space*）由允许进程使用的全部线性地址组成。每个进程所看到的线性地址集合是不同的，一个进程所使用的地址与另外一个进程所使用的地址之间没有什么关系。后面我们会看到，内核可以通过增加或删除某些线性地址区间来动态地修改进程的地址空间。

   内核通过所谓线性区的资源来表示线性地址区间，线性区是由起始线性地址、长度和一些访问权限来描述的。为了效率起见，起始地址和线性区的长度都必须是4096的倍数，以便每个线性区所识别的数据完全填满分配给它的页框。下面是进程获得新线性区的一

   我们注意到，每一个线性区(`vm_area_struct`)的长度都为PAGE_SIZE(4096)的整数倍，因此，我们可以通过遍历由`vm_area_struct`所组成的链表，而在每一个`vm_area_struct`中，又可以遍历其所包含的每一个页，最终实现对特定进程空间每一个页的遍历。具体实现如下：

```
for (vma = taskp->mm->mmap; vma; vma = vma->vm_next){          //遍历由
`vm_area_struct`所组成的链表
        int page_number = (vma->vm_end - vma->vm_start) / PAGE_SIZE;
        int i = 0;
        for (i = 0; i < page_number; i++)
        {
            unsigned long virt_addr = vma->vm_start + i * PAGE_SIZE;  //遍历
其所包含的每一个页
        }
}
```

2. `ptep_test_and_clear_young()`

   当一个程序访问内存时，在VA->PA转换过程中，硬件会自动将access位设置位1，表明最近被访问过。但是硬件不会自动将access位清0，需要软件将其清0，然后由软件在一定时间内查看access位是否置1，如果置1说明最近被访问过。在内核中`ptep_test_and_clear_young()`函数实现检查access位置1，如果被访问过将其清0，后续一段时间之后根据需要查看是否再次被置1。

   在本次实验中，需要自己实现和`ptep_test_and_clear_young()`函数类似的功能，在每次遍历进程的所有页的过程中，获得该页所对应的`pte`中的access 位。若access 位为1，则将计数器加1，表示进程访问过该页，之后手动将access 位置为0；若access 位为0，则表示进程没有访问该页。

3. `PTE Access位`

   查阅《深入Linux内核架构》中对于PTE的介绍，具体如下：

   > 虚拟地址分为几个部分，用作各个页表的索引，这是我们熟悉的方案。根据使用的体系结构字长不同，各个单独的部分长度小于32或64个比特位。从给出的内核源代码片段可以看出，内核（以及处理器）使用32或64位类型来表示页表项（不管页表的级数）。这意味着并非表项的所有比特位都存储了有用的数据，即下一级表的基地址。多余的比特位用于保存额外的信息。附录A详细描述了各种体系结构上页表的结构。

   3. 特定于PTE的信息

   最后一级页表中的项不仅包含了指向页的内存位置的指针，还在上述的多余比特位包含了与页有关的附加信息。尽管这些数据是特定于CPU的，它们至少提供了有关页访问控制的一些信息。下列位在Linux内核支持的大多数CPU中都可以找到。

   其中指出，在64位的机器中，PTE页表项为64位，其中多余的比特位用于表示附加信息，而本次实验所用到的Access位便是其中之一。书中的介绍如下：

   > CPU每次访问页时，会自动设置_PAGE_ACCESSED。内核会定期检查该比特位，以确认页使用 的活跃程度（不经常使用的页，比较适合于换出）。在读或写访问之后会设置该比特位

   而不同架构下的Access位并不相同，本次实验所采用的架构为`x86_64`架构。查阅源码`/arch/x86/include/asm/pgtable_types.h`中对于`_PAGE_BIT_ACCESSED`的定义,具体如下：

   ```
   #define _PAGE_BIT_ACCESSED    5        /* was accessed (raised by CPU) */
   ```

   得到，PTE的第5为做完Access位。因此，我在内核模块中实现了`ptep_test_and_clear_young()`

```
            //完成地址转换
            pgd_t *pgd;
            p4d_t *p4d;
            pud_t *pud;
            pmd_t *pmd;
            pte_t *pte;
            pgd = taskp->mm->pgd;
            pgd += pgd_index(virt_addr);
```

```
            p4d = p4d_offset(pgd, virt_addr);
            pud = pud_offset(p4d, virt_addr);
            pmd = pmd_offset(pud, virt_addr);
            pte = pte_offset_kernel(pmd, virt_addr)
            //获得pte的Access位，并将其清零
            int flag = 0;
            unsigned long MASK = 1 << 5;
            flag = !!((pte->pte) & MASK);
            pte->pte = pte->pte & ~MASK;
            if (flag)
            {
                page_count += 1; //计数器+1
            }
```

至此，介绍了内核模块如何以页为单位统计内存的读写量。**具体的代码实现请参见提交文件中`519021910480_src/watch.c/ram_measure()`的内容**

2 用户态部分

在完成了内核模块的编写之后，我们可以通过读取`/proc/watch`文件来获得指定进程的内存访问页数以及CPU占用的时间参数`utime，stime，cstime，cutime`。接下来，需要完成一个用户态程序，通过不断的访问`/proc/watch`文件来统计指定进程的CPU和内存使用情况，将它们的 CPU 使用率和内存读写频率随时间的变化画图表示，进一步判断该进程是CPU密集型还是内存密集型进程。

2.1 大致思路

我选择采用bash脚本+python这两种语言完成对用户态程序的编程。

- 其中bash脚本用于启动benchmark程序，向`/proc/watch`文件中写入benchmark程序对应的`pid`。之后每间隔固定的时间读取/proc/watch，并将其记录到csv文件中交由python程序处理
- python程序用于读取csv文件，处理原始数据，从时间参数`utime，stime，cstime，cutime`中计算出CPU占有率。并绘制CPU占有率和内存读写频率随时间的变化曲线。

2.2 实现步骤

2.2.1 CPU占有率计算

为方便计算某进程的CPU占有率，根据CPU占有率计算公式：`pc%=(totalCpuTime2-totalCpuTime1)/(cputime2-cputime1)`和进程执行总时间的计算公式：`cputime=utime+stime+cutime+cstime`。我们已经可以通过`/proc/watch`获得进程的运行总时间，接下来，我们需要获得`totalCpuTime2`、`totalCpuTime1`，它们分别指该进程在统计的结束时刻和统计的起始时刻下的执行总时间。

阅读Linux manual page，发现`/proc/stat`文件下的内容正好可以满足我们计算`totalCpuTime1`和`totalCpuTime2`的需求。该文件包含了所有CPU活动的信息，该文件中的所有值都是从系统启动开始累计到当前时刻。

```
user   (1) Time spent in user mode.
nice   (2) Time spent in user mode with low priority (nice).
system (3) Time spent in system mode.
idle   (4) Time spent in the idle task.  This value should be USER_HZ times
the second entry in the                 /proc/uptime pseudo-file.
iowait (5) Time waiting for I/O to complete.
irq    (6) Time servicing interrupts.
softirq(7) Time servicing softirqs.
```

因此，总的CPU时间`totalCpuTime = user + nice + system+ idle + iowait + irq + softirq`。所以，在bash脚本的运行过程中，在读取/proc/watch文件的同时，也应该同时读取/proc/stat文件。具体实现如下：

```
#!/bin/bash
echo 'accessed_page,utime,stime,cutime,cstime,total_time' > data.csv
sysbench --test=cpu --cpu-max-prime=200000 --num-threads=1 run &  #启动benchmark进程
cat_ret()
{
    ps -e |grep sysbench| awk '{print $1}'
}
ret=`cat_ret`
echo ${ret} > /proc/watch                                   #将pid写入/proc/watch文件
while sleep 0.01
do
   cat /proc/watch | awk '{ printf "%s,%s,%s,%s,%s,",$1,$2,$3,$4,$5 } '
>>data.csv #保存数据到csv文件
   cat /proc/stat  | awk '$1=="cpu1" {printf "%s\n", $2+$3+$4+$5+$6+$7+$8} '
>>data.csv
done
```

最后，运行python程序，读取`data.csv`文件，利用计算公式`pc%=(totalCpuTime2-totalCpuTime1)/(cputime2-cputime1)`算出目标进程的CPU占有率，并绘图。

2.2.2 单位统一

在计算CPU占有率的过程中，需要保证/proc/stat和/proc/watch所读出的时间数据具有相同的单位。

`/proc/stat`做为Linux kernel自带的`proc`文件，其在Linux manual page中指明了数据的单位为clock tick。

而在实现`/proc/watch`的过程中，从`task_struct->signal`中读出的时间单位为纳秒，因此，我在内核模块中重新实现了`nsec_to_clock_t()`函数，将纳秒转换为了clock tick，统一了单位。具体如下：

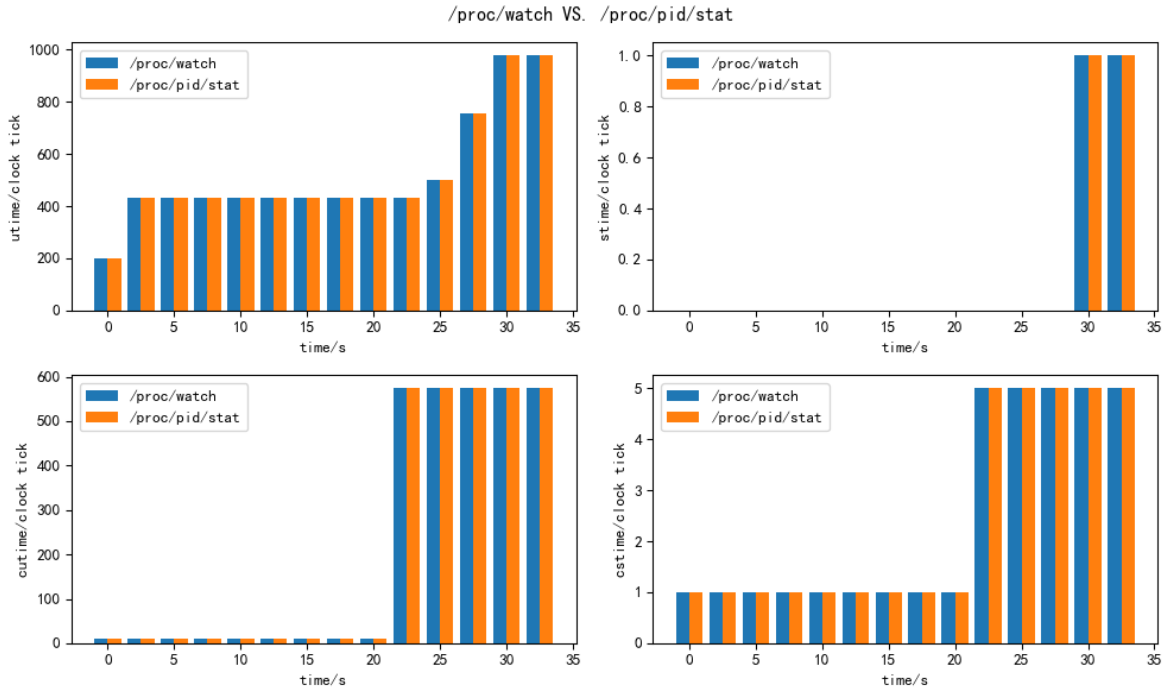
```
static u64 nsecc_to_clock_t(u64 x)
{
#if (NSEC_PER_SEC % USER_HZ) == 0
    return div_u64(x, NSEC_PER_SEC / USER_HZ);
#elif (USER_HZ % 512) == 0
    return div_u64(x * USER_HZ / 512, NSEC_PER_SEC / 512);
#else
    return div_u64(x * 9, (9ull * NSEC_PER_SEC + (USER_HZ / 2)) / USER_HZ);
#endif
}
```

注：时钟计时单元（clock tick），可以调用sysconf(*SC_CLK_TCK)来获得每秒包含的时钟计时单元数，在本次实验的机器上，sysconf*(SC_CLK_TCK)返回100。与此对应的内核常量是USER_HZ。这个值即为/proc/stat和/proc/pid/stat中记录时间的单位

3 实验结果

3.1 /proc/watch 与 /proc/pid/stat对比

根据实验设计，所实现的内核模块应该具有和Linux系统自带的`/proc/pid/stat`相同的进程时间信息。为验证内核模块的正确性，我编写了bash脚本，针对同一个进程，同时读取`/proc/watch`文件和`/proc/pid/stat`中的`utime、stime、cutime、cstime`，并作柱状图对比，具体如下：



我们可以观察到从两个不同的`proc`文件中读取出来四个时间参数基本一致，证明了内核模块`/proc/watch`在统计CPU利用率方面的正确性。

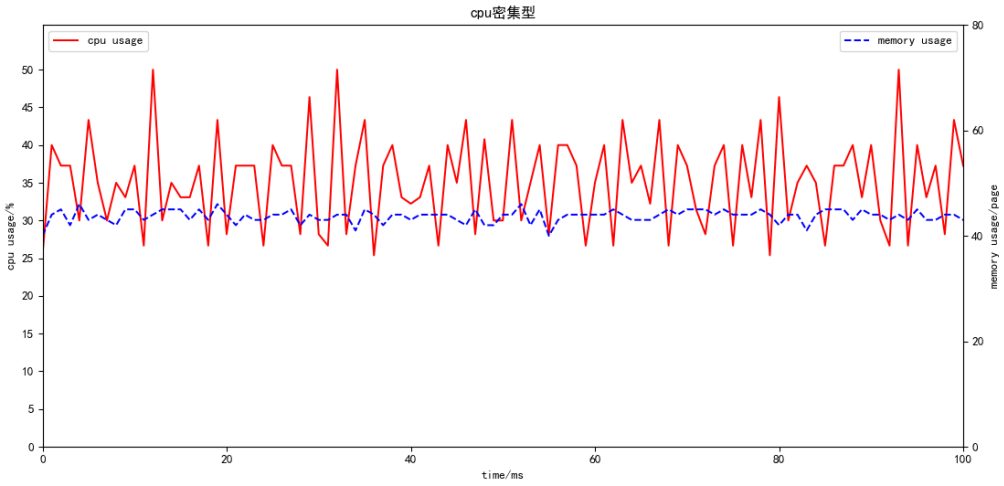
3.2 内存密集型和CPU密集型

在本次实验中，我选择sysbench做为benchmark程序进行运行时间和内存的读写量的统计。sysbench是一款开源的多线程性能测试工具，可以执行CPU/内存/线程/IO/数据库等方面的性能测试。在本次实验中，我主要使用了其中的CPU性能测试和内存性能测试。查阅sysbench官方文档得到CPU和内存的测试过程大致如下：

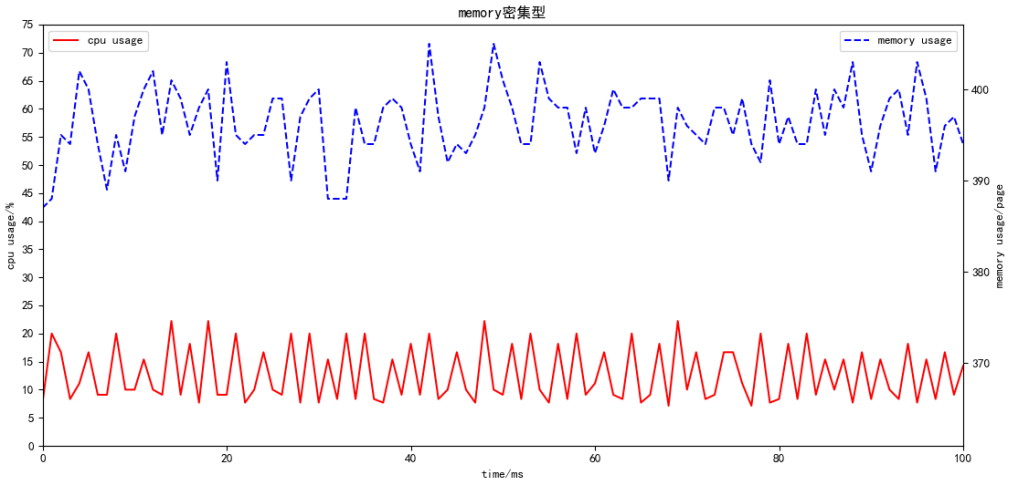
> CPU测试：When running with the CPU workload, sysbench will verify prime numbers by doing standard division of the number by all numbers between 2 and the square root of the number. If any number gives a remainder of 0, the next number is calculated. As you can imagine, this will put some stress on the CPU, but only on a very limited set of the CPUs features.
>
> 内存测试：When using the *memory* test in sysbench, the benchmark application will allocate a memory buffer and then read or write from it, each time for the size of a pointer (so 32bit or 64bit), and each execution until the total buffer size has been read from or written to

分别使用如下命令开启benchmark进程,并绘制CPU利用率和内存读写量曲线如下

```
sysbench --test=cpu --cpu-max-prime=20000 --num-threads=2 run
```




```
sysbench --test=memory --memory-block-size=1M --memory-total-size=1000G --num-threads=2 run
```



对比上述两图，我们可以发现：针对CPU的测试进程中CPU的利用率高于针对内存的测试进程，而内存读写量则低于针对内存的测试进程。这也反应出了两种不同的benchmark进程分别属于CPU密集型进程和内存密集型进程。这与sysbench官方文档对于这两种进程的描述相符合。

3.3 内存读写量对比

为进一步验证不同benchmark程序下内存读写量的对比，我选择更改sysbench中内存测试的`--memory-block-size`参数来调整内存读写量。

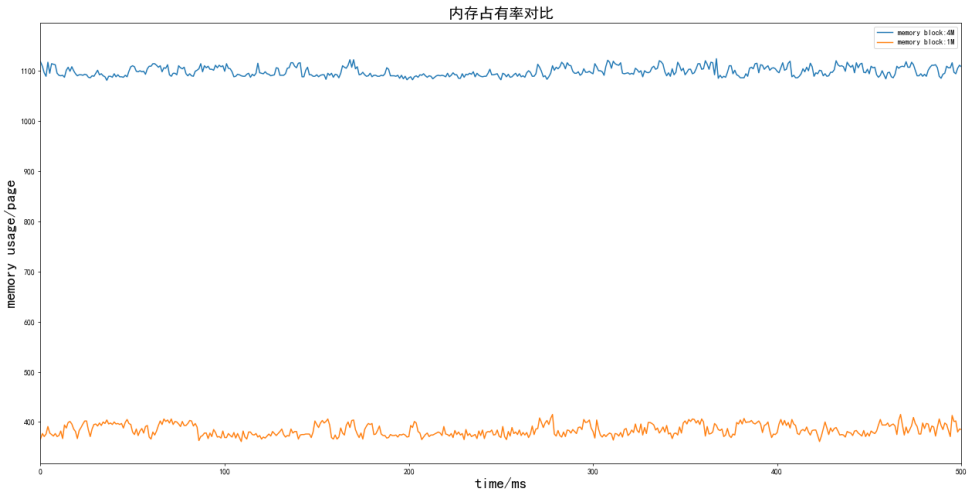
> --memory-block-size 用于指定测试进程在内存中分配的memory buffer大小

理论上而言，memory buffer所分配的空间越大，内存的占有率也会随之增大。因此，我分别采用1M和4M做为`--memory-block-size`进行实验并以page为单位绘制内存占用曲线。

```
sysbench --test=memory --memory-block-size=1M --memory-total-size=1000G --num-threads=2 run
```

```
sysbench --test=memory --memory-block-size=4M --memory-total-size=1000G --num-threads=2 run
```

内存占有曲线如下:



观察上图可以发现，`memory-block-size=4M`的内存占有率大约为`memory-block-size=1M`内存占有率的4倍。实验结果符合理论推导，证明了内存占有率统计的正确性。

3.4 CPU占有率的对比

为进一步验证不同benchmark程序下CPU占有率的对比，我通过绑定sysbench进程并且设置优先级的方法来调整CPU利用率。具体的操作如下。

首先，我启动一个CPU密集型进程（蒙特卡洛法测π），将其优先级（nice值）设置为0，并将其绑定到特定`CPU1`上。之后，启动sysbench的CPU测试进程，也将其绑定到指定`CPU1`上。并尝试不同的nice值对于CPU占有率的影响。经过实验后，选定了nice=0和nice=5两种优先级进行绘图，具体如下：

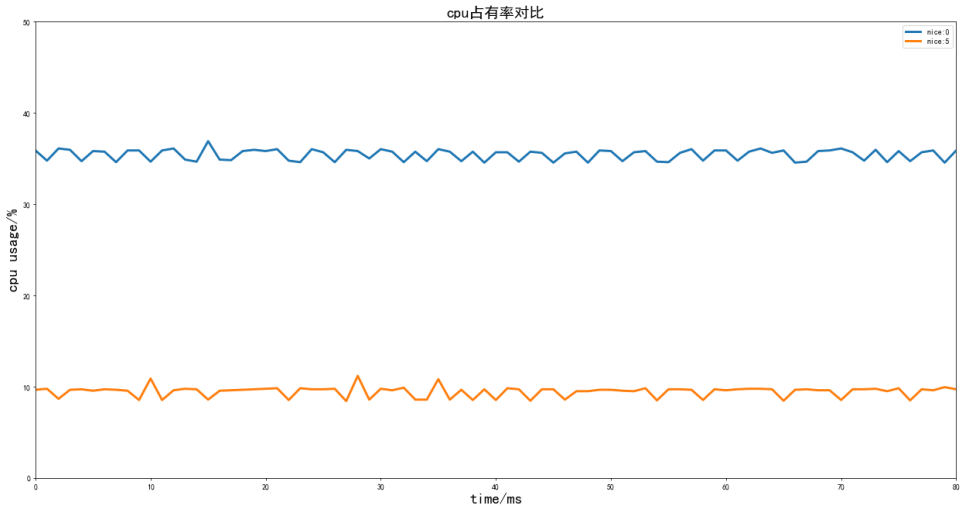
分别用以下bash语句启动sysbench程序：

```
nice -n 0 taskset -c 1 sysbench --test=cpu --cpu-max-prime=200000 --num-threads=1 run
```

```
nice -n 5 taskset -c 1 sysbench --test=cpu --cpu-max-prime=200000 --num-threads=1 run
```

按照2.2节的步骤绘制CPU利用率曲线如下:

注：因为使用了taskset命令将进程绑定到了指定的CPU1上，因此，在计算totalcputime时，从/proc/stat读取数据时并不是从CPU那一行读取所有的CPU时间总和，而是读取CPU1这一指定核心的时间参数。



根据上图我们也明显观察到不同的nice优先级对于CPU利用率的影响，并通过`htop`命令可以验证CPU利用率信息的正确性。因为该核心上同时正在运行nice值为0的CPU密集型进程（蒙特卡洛法测π），所以启动的sysbench进程会发生抢占。根据`CFS`调度算法，nice值较高的进程实际运行时间较低，因此，nice=5的sysbench进程只能获得大概10%的CPU利用率。

4 思考题

4.1 关于多线程

阅读Linux kernel中关于计算/proc/pid/stat中的utime、stime的源码：

```
int proc_tid_stat(struct seq_file *m, struct pid_namespace *ns,
            struct pid *pid, struct task_struct *task)
{
    return do_task_stat(m, ns, pid, task, 0);
}

int proc_tgid_stat(struct seq_file *m, struct pid_namespace *ns,
            struct pid *pid, struct task_struct *task)
{
    return do_task_stat(m, ns, pid, task, 1);
}
```

发现当需要统计多线程的utime和stime时，内核会调用proc_tgid_stat。其中需要注意的是在向函数do_task_stat()传入参数的过程中，最后一个参数whole的取值为1，这个参数决定了在统计CPU时间的过程中需要考虑同一个线程组的其他线程。具体如下:

```
    if (whole) {            //whole=1
            ......
            thread_group_cputime_adjusted(task, &utime, &stime);//统计同一个线程组的
其他线程信息
        }
    }
    ......
    if (!whole) {//只需要whole为0便可调用task_cputime_adjusted()更新utime、stime
        .......
        task_cputime_adjusted(task, &utime, &stime);
        .......
    }
```

面对多线程的benchmark程序，在实现内核模块统计进程时间的过程中，当写入`/proc/watch`的进程`pid`等于线程组的id（即`tgid`）时，除了统计该进程`pid`=`tgid`对应的`task_struct->signal`中的`utime`和`stime`以外，还应该使用`for_each_thread`遍历该线程组中的其他`task_struct`并统计各个进程的`utime`和`stime`的总和。具体的代码实现如下:

```
    rcu_read_lock();
    nextseq = 0;
    do
    {
        seq = nextseq;
        flags = read_seqbegin_or_lock_irqsave(&sig->stats_lock, &seq);
        *utime += sig->utime;     //统计进程pid=tgid对应的task_struct->signal中的
utime和stime
        *stime += sig->stime;
        for_each_thread(taskp, t) //遍历该线程组中的其他task_struct
        {
            *utime += t->utime;
            *stime += t->stime;
        }
        nextseq = 1;
    } while (need_seqretry(&sig->stats_lock, seq));
    done_seqretry_irqrestore(&sig->stats_lock, seq, flags);
    rcu_read_unlock();
```

4.2 关于utime、stime

关于`/proc/pid/stat`中参数的详细解释参阅Linux manual page，得到了关于utime和stime的解释:

> (14) utime %lu :
>
> Amount of time that this process has been scheduled in user mode, measured in clock ticks (divide by sysconf(_SC_CLK_TCK)). This includes guest time, guest_time (time spent running a virtual CPU, see below), so that applications that are not aware of the guest time field do not lose that time from their calculations.
>
> (15) stime %lu :
>
> Amount of time that this process has been scheduled in kernel mode, measured in clock ticks (divide by sysconf(_SC_CLK_TCK)).

简要来说：`utime`指进程在用户态的运行时间；`stime`指进程在内核态的运行时间。

我认为在本次实验统计CPU占有率的过程中，仅统计`stime`或`utime`并不准确。而是应该记录`utime`、`stime`，以及`cstime`和`cutime`（`cutime`*指所有层次子进程在用户态的运行时间总和；*`cstime`*指所有层次子进程在核心态的运行时间总和*）。在获得了上述4个时间参数后，可以计算出进程运行的总时间：`totaltime=utime+stime+cutime+cstime`。

具体的程序实现请见`519021910480_src/watch.c/cpu_measure()`

最后通过公式`pc%=(totalCpuTime2-totalCpuTime1)/(cputime2-cputime1)`获得CPU利用率。

4.3 关于Page

结论：我认为用Page为单位来统计内存读写量具有一定的合理性，但是可能并不精准。

理由：

- 具有一定的合理性是因为内核在为进程分配虚拟空间时，是以Page为最小单位进行分配的。同时，每一个`vm_area_struct`所表示的虚拟空间的长度都是PAGE_SIZE(4096 Byte)的整数倍。因此，通过统计一段时间内进程访问页的总数可以在一定程度上反应出进程内存读写量的大小。
- 但是使用Page做为统计单位存在并不准确的问题，我认为主要是因为以下两个原因：
    1. 存在“内部碎片”带来的误差。有的进程在访问内存的时候可能只读写了一个页框中的少量信息，而另一个进程在访问内存的时候可能读写了一个页框中的全部信息。而如果使用Page做为统计单位，两者的内存利用率都是一个页，这就会引起统计误差。
    2. 对一个Page的多次访问。在一个采样间隔内，如果一个进程对某一页进行了多次访问，但是在最终的统计中只会记录其利用了一个页，这也会造成统计误差。

解决方案：我认为可以借助`/proc`文件系统中的`/proc/pid/smaps`和`/proc/pid/maps`文件辅助内存的统计。

> /proc/[pid]/maps： A file containing the currently mapped memory regions and their access permissions.
>
> /proc/[pid]/smaps：This file shows memory consumption for each of the process's mappings.

在上述文件中的部分参数如：

> 1. Size：虚拟内存空间大小
> 2. RSS：Resident Set Size（驻留集大小）, 实际占用的物理内存
> 3. PSS：Proportional Set Size， 也是实际分配的物理内存，与RSS的区别是，它以平分的方式来计算共享库的大小（共享库 / 进程个数）
> 4. USS：Unique Set Size, 进程的私有内存（独自使用的库，堆等空间），不包含共享的内存空间。

都是以KB为单位进行的内存统计，可以达到比Page更精准的统计结果。也可以阅读Kernel中包含这一部分的源码，从而设计出功能类型的内核模块。但是由于时间原因，在本次实验中并没有完成针对内存统计的优化。

5 思考与感悟

5.1 关于计时单位

在实验过程中，有两种计时单位困扰了我很久，分别是：jiffies和clock tick。在参阅《Linux/UNIX系统编程手册》后将其概念区分如下：

- jiffies

time()和gettimeofday()等时间相关的各种系统调用的精度是受限于系统软件时钟（software clock）的分辨率，它的度量单位被称为jiffies。jiffies的大小是定义在内核源代码的常量HZ。这是内核按照round-robin的分时调度算法分配CPU进程的单位。
在常见内核版本种，软件时钟速度是100赫兹，也就是说，一个jiffy是10毫秒。

- clock tick

时钟计时单元（clock tick），可以调用`sysconf(SC_CLK_TCK)`来获得每秒包含的时钟计时单元数，在大多数Linux的硬件架构，`sysconf(SC_CLK_TCK)`返回100。与此对应的内核常量是USER_HZ。这个值即为`/proc/stat`和`/proc/pid/stat`中记录时间的单位。

5.2 关于进程的线性空间

因为在实验指导书中有一条tips如下：

> 进程描述符中的 task_stuct->mm->vma 是进程向内核申请使用的内存区域（表示为一个 struct vm_area_struct 结构体）的链表，你可以只扫描用户内存空间的这部分区域，减少扫描的时间。

但是我认为`struct vm_area_struct`所表示的进程线性空间的虚拟地址一定都是处于用户空间中的（在64位的机器上，用户空间的地址范围为：`0~0x0000 8000 0000 0000`）。为验证这一想法，在内核模块扫描地址空间的过程中，我加入了如下语句用于判断地址是属于内核态还是用户态：

```
for (i = 0; i < page_number; i++){
    unsigned long virt_addr = vma->vm_start + i * PAGE_SIZE;
    if (virt_addr >= KERNEL_SPACE){   //KERNEL_SPACE = 0xFFFF 8000 0000 0000
       pr_info("--------kernel space: %lx--------",virt_addr);
       continue;
    }
    ......
}
```

注：`0x0000 8000 0000 0000 ~ 0xFFFF 8000 0000 0000`之间的区域作为空洞，不会出现在线性地址空间中。

在运行内核模块的过程中，我发现，并没有`virt_addr`会大于`KERNEL_SPACE`而进入内核空间中。查阅资料后发现，关于进程中的内核空间的部分是由另一个结构体`struct vm_struct`进行组织管理的。

> 在Linux中，`struct vm_area_struct`表示的虚拟地址是给进程使用的，而`struct vm_struct`表示的虚拟地址是给内核使用的，它们对应的物理页面都可以是不连续的。

5.3 关于采样频率

在编写bash脚本进程反复读取`/proc/watch`的过程中，需要设定读取`/proc/watch`的间隔。在每次读取完/proc/watch文件后，内核都会遍历指定进程虚拟空间中的所有页，并将Access位清零。如果读取的间隔设置的过短，则会导致程序效率下降，不同进程在过小的时间间隔内访问的内存页数基本相同，会出现实验结果不明显的问题。而如果采样间隔设计过长，则会造成进程对一个页的反复读取，但是只被记录一次的情况，也会出现实验结果不明显的问题。

在实验过程中，我并没有找到合适的方法来计算出最合适的采样频率，只能通过大量的实验，最终得出当采样间隔设置在0.01~0.001s之间时，针对内存的统计效果最明显。

5.4 总结与感悟

本次实验做为Linux Kernel这门课的最后一个任务，无论是从工作量还是从难度上，相较之前的任务都有提升。这也带给了我不小的挑战。本次实验可以分为进程的CPU利用率和内存占有率这两个方面，而这两个方面也正是统计一个进程最重要的特征。在实验过程中，我从`/proc/pid/stat`作为入手点，阅读源码，理解了内核是如何计算CPU时间，如何处理线程组，如何划分用户空间和内核空间，如何组织进程的虚拟空间等一系列问题。将课堂上所学的知识与实际的应用结合起来，一方面深化了理论知识，另一方面锻炼了调试代码的动手能力。在完成任务的过程中，也将之前的知识融汇贯通，比如`lab2`中对于`proc`文件源码的修改，`lab3`中对于虚拟地址的转换等等，这也让我认识到了Linux kernel这门课的实用性。同时，学会查阅相关资料也十分重要，在完成的实验过程中，我参阅了深入理解《LINUX内核第3版》、《深入Linux内核架构》、《Linux/UNIX系统编程手册》、《Linux manual Page》以及各大论坛和博客的相关内容，这些资料对于我完成本次实验而言十分重要。

与此同时，助教的实验指导书里面的内容也很具有指导意义，为我指明了方向。最后感谢陈全老师和各位助教的悉心解答。