Final Project

吴非 519021910924

内核环境：（uname -r） 5.13.0-40-generic

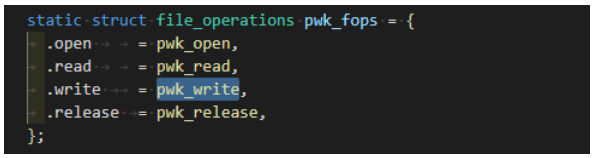
实验过程和代码讲解：

与往常使用 proc 系统不同， 我想对不太熟悉的字符设备驱动进行了解，所以使用设备驱动来代替 proc 的功能（其实框架很像，只是挂载的地方一个在 proc 下，一个在/dev/pwatch 里）

概览：（各个文件功能）：

process_watch_kernel.c 内核模块
process_watch.c 用户态程序
process_cpu.c process_memory.c 模拟 cpu 和内存密集型进程
graph.py 可视化
multi_process.py 模拟多进程的程序

process_watch_kernel.c：



```
static struct file_operations pwk_fops = {
    .open    = pwk_open,
    .read    = pwk_read,
    .write   = pwk_write,
    .release = pwk_release,
};
```

主要针对 write 和 read 操作进行代码编写。

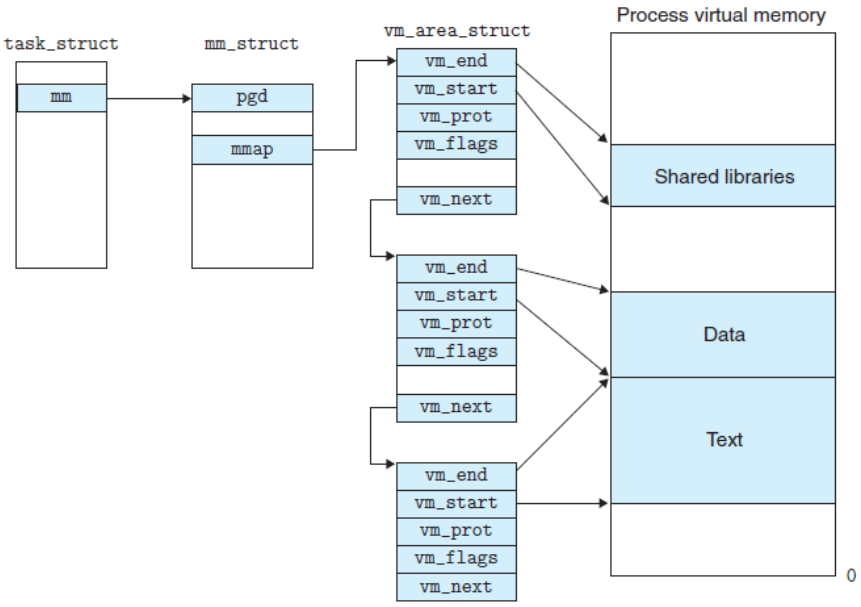
pwk_write 函数写入 pid。

pwk_read 函数得到 cpu 和内存数据。

pwk_read：其内部调用 pwk_get_rusge 函数，将需要的数据写入结构体 rusage。其内部遍历所有的子进程和子线程（包括自己的线程和子进程的线程）（深度均为 1）接着对每个 task_struct 调用 pwk_accu_thread_rusage。

pwk_accu_thread_rusage：此函数功能是得到某个 task_struct 结构体的 cpu 和内存数据并累加进 rusage 中。对此函数：utime 通过 task_cputime_adjusted 函数得到。内存数据通过 get_mem_freq 函数得到。

get_mem_freq：此函数输入为 vm_area_struct 结构体，遍历所有的 vm_area_struct，每个 vm_area_struct 里从 vm_start 遍历到 vm_end(所有的虚拟地址),根据虚拟地址找到对应的 pte_t 结构体（参考下图）。然后根据提示调用 ptep_test_and_clear_young 即可，由于不能在内核里直接调用，复制此函数源码并修改函数名，改为 wf_test。



最后返回给用户态程序的为 cpu 和内存数据的拼接，单位分别为 us 和字节，用空格区分两个数据。

process_watch.c:

本来设计的是输入参数为采样次数和采样间隔（通过在 while（1）循环内部 sleep 一定时间来将其假设为采样时间（假设得到数据的过程很快)），由于加入内存部分的遍历时间消耗过久导致这部分时间和采样间隔数量级已经接近，这会导致 cpu 利用率出现错误:

采样间隔 为 100ms 时:

计算内存数据:

```
src > 4920_sampling.txt
1   6.849 9479127040
2   6.798 5486149632
3   7.669 5972688896
4   6.912 5385486336
5   7.471 11475615744
6   7.837 10636754944
7   7.967 6274678784
8   7.614 7365197824
9   7.711 5721030656
10  7.594 14361296896
11
```

不计算内存数据:

```
src > 4920_sampling.txt
1   1.038 0
2   1.000 0
3   1.000 0
4   1.000 0
5   1.040 0
6   1.000 0
7   1.000 0
8   1.000 0
9   1.040 0
10  1.000 0
```

采样间隔为 1000ms:

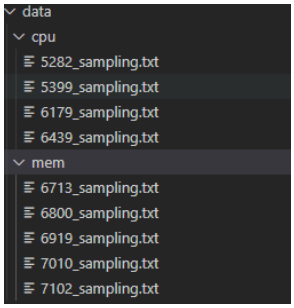
不计算内存时结果不变，计算内存:

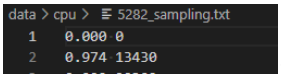
```
src > 4920_sampling.txt
1   1.577 6073352192
2   1.564 6627000320
3   1.561 5301600256
4   1.619 12280922112
5   1.694 5570035712
6   1.554 11072962560
7   1.759 6308233216
8   1.581 12834570240
9   1.566 7180648448
10  1.561 8136949760
11
```

可以看到采样频率直接影响 cpu 利用率。

因此直接放弃在循环内调用 sleep 函数，只用内存遍历的时间就可以让采样间隔比较的合适。

最后根据 pid 命名写入 txt 文件。所有数据位于/data 下:



文件夹 cpu 为 cpu 密集型，mem 为内存密集型，txt: 

一行中左侧为 cpu 利用率(%每核心)，右侧为内存访问频率（字节/us）

实验结果截图（图片位于/imgs）:

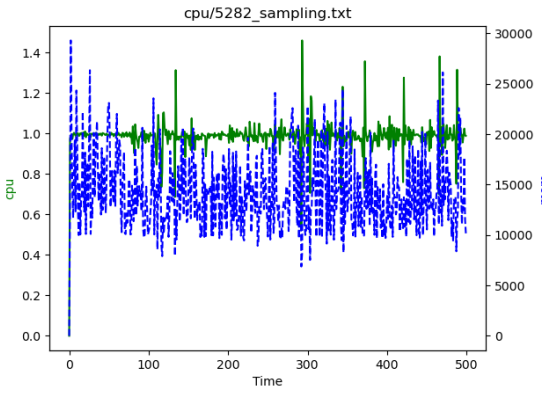
（内存的数据我在写完报告后发现自己计算出读写的 page 量后乘了两次 PAGE_SIZE，代码已经改成了正确的，数据要重新做的话步骤冗杂，由于不影响实验分析，就不重新跑了）
蓝色为内存数据，绿色为 cpu 数据。

计算密集型进程:

采用 sysbench --cpu-max-prime=1000 --threads=x --time=0 cpu run 来产生，通过 htop 肉眼观察进程 PID。(threads=x 验证子线程代码)
threads=1：部分数据展示：

src > 5282_sampling.txt

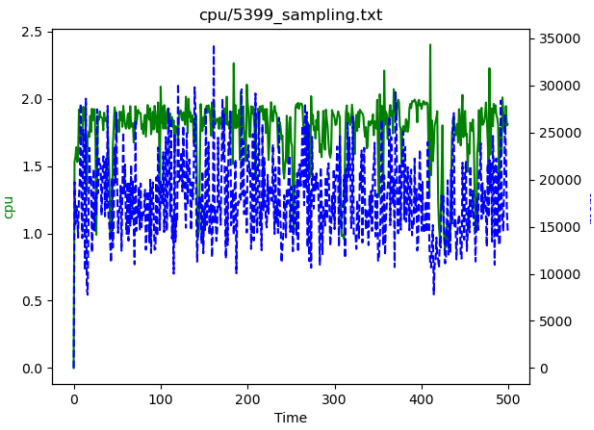
1	0.000 0
2	0.974 13430
3	0.992 29302
4	0.989 19780
5	0.974 14411
6	0.997 11698
7	1.004 16502
8	0.996 19379
9	0.989 12338



threads=2: 部分数据展示：

src > 5399_sampling.txt

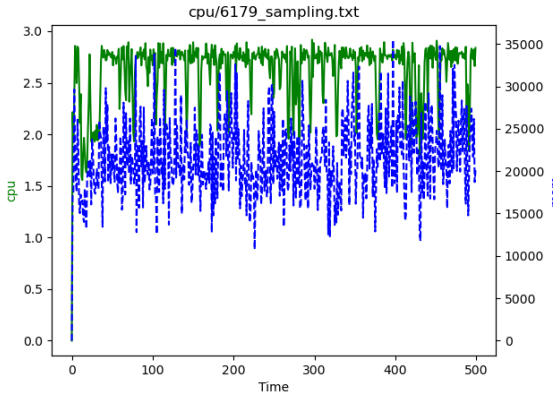
1	0.000 0
2	1.534 19724
3	1.568 15249
4	1.641 17201
5	1.531 16656
6	1.538 13838
7	1.920 18104
8	1.689 19848
9	1.745 28052
10	1.695 16869
11	1.737 13779
12	1.941 25291



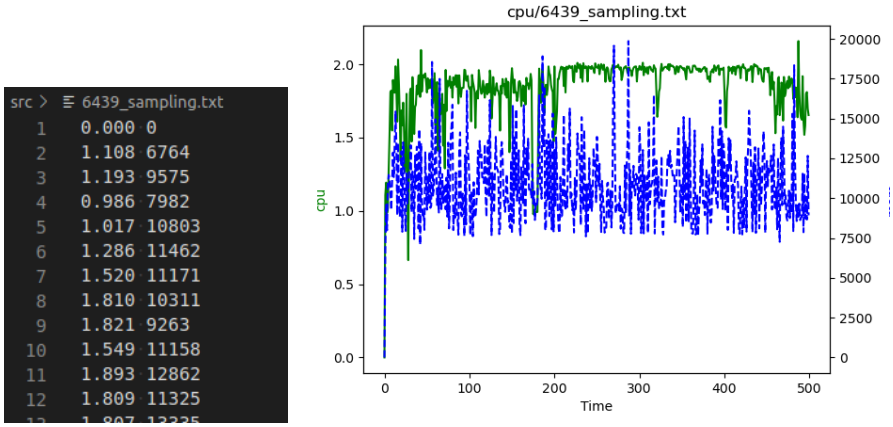
threads=3：部分数据展示：

src > 6179_sampling.txt

```
0.000 0
2.215 17325
1.927 17563
2.343 29638
2.858 22955
2.730 18613
2.497 17555
2.851 26070
2.835 22983
2.515 16274
2.111 15042
2.390 17210
1.589 16406
1.557 16542
```



子进程的验证：multi_process.py 启动子进程（为三个进程两个线程，实际参与计算的为 2 个进程 2 个线程[8282.863091] 进程数：3线程数：2）：



src > 6439_sampling.txt

```
0.000 0
1.108 6764
1.193 9575
0.986 7982
1.017 10803
1.286 11462
1.520 11171
1.810 10311
1.821 9263
1.549 11158
1.893 12862
1.809 11325
```

从 cpu 占用率来看也确实计算了两个子进程各自的 cpu 时间（和上述 threads=2 时相当（区别就是一个进程开两个线程和两个进程各开一个线程），同时也可以得知 sysbench 命令中真正执行 cpu 测试的是进程所开的线程）。

内存密集型进程：

使用自己编写的 process_memory.c 进行模拟（循环调用 memcpy）。

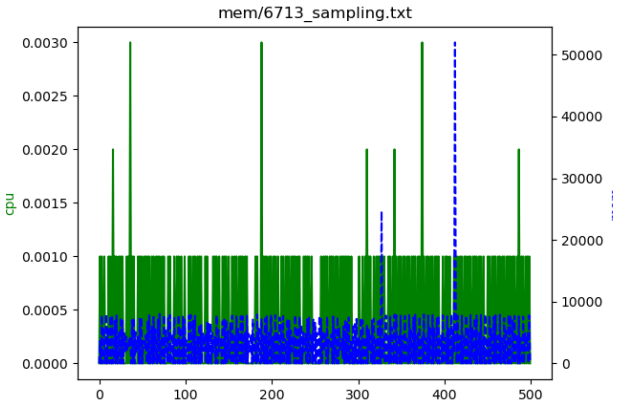
```
while(1) {
    memory_usleep(5);
    ptr = malloc(8092);//malloc 向系统申请分配指定size个字节的内存空间。返回类型是 void* 类型
    memcpy(ptr, "this is test memory@@@@@@@@", 20);
    free(ptr);
```

通过调整 sleep 的间隔来控制内存读写频率。

当其为 100ms 时：

src > 6713_sampling.txt

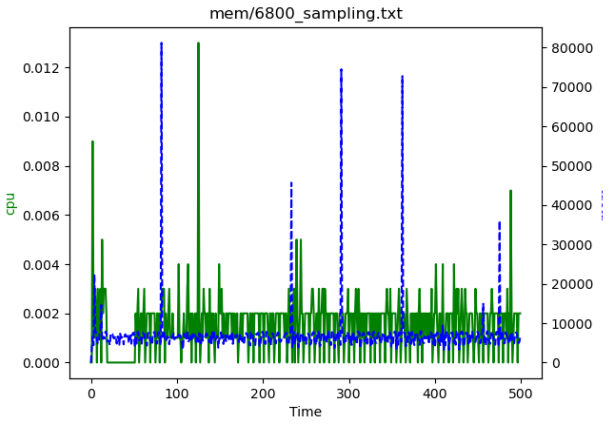
```
0.000 0
0.001 3585
0.000 0
0.001 7806
0.000 0
0.000 5853
0.001 2080
0.000 0
0.000 7993
0.000 0
```



为 50ms 时：

src > 6800_sampling.txt

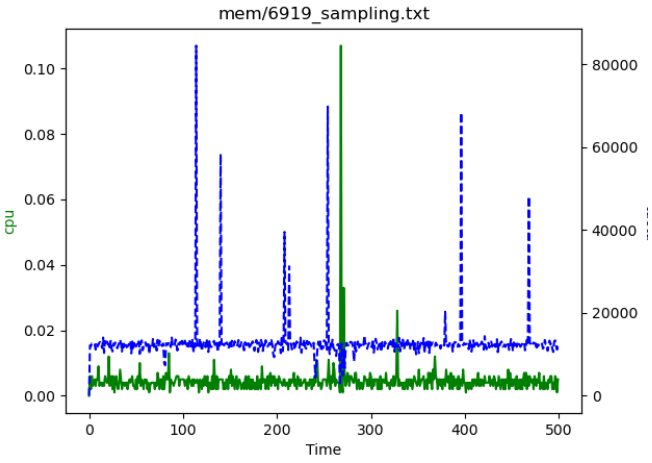
```
0.000 0
0.000 3715
0.009 6547
0.002 4422
0.001 22375
0.003 7710
0.002 7715
0.000 5726
0.003 4861
0.002 4864
0.001 7681
0.003 7687
```



为 25ms 时：

src > 6919_sampling.txt

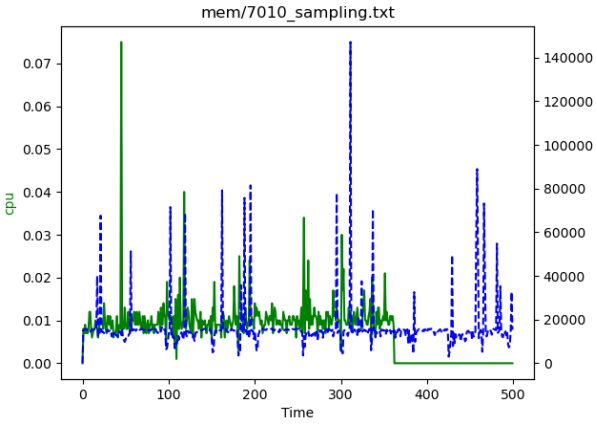
```
0.000 0
0.006 12124
0.002 11990
0.006 12593
0.003 12454
0.004 12719
0.005 12458
0.003 11128
0.005 12429
```



为 10ms 时：

src > 7010_sampling.txt

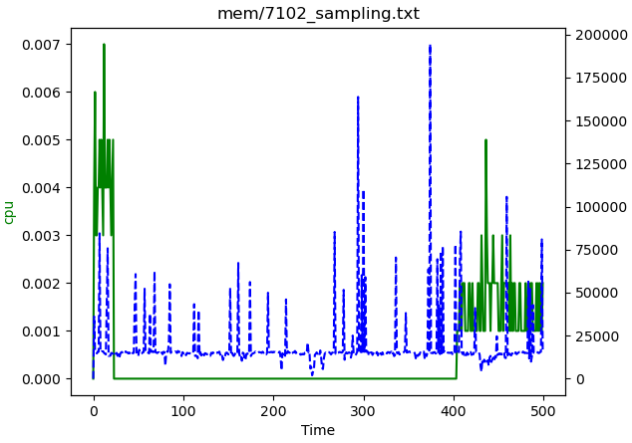
```
1   0.000 0
2   0.008 15288
3   0.007 15612
4   0.009 15571
5   0.007 15320
6   0.008 15617
7   0.007 14977
8   0.008 15313
9   0.012 14718
10  0.012 14484
```



为 5ms 时：

src > 7102_sampling.txt

```
1   0.000 0
2   0.004 36126
3   0.006 15624
4   0.003 15224
5   0.003 14700
6   0.004 15778
7   0.004 15660
8   0.005 84488
9   0.004 15581
10  0.005 15940
11  0.004 15840
```



可以从 txt 中的数据看到 10ms 和 5ms 时内存的众数基本一致，应该是内存读写过快，进程所申请的内存量已经基本不会扩大，采样间隔内对同一个 page 进行了多次读写，但是计数只加了 1，所以最后的频率值一样，这在思考题 3 的第二点也有叙述。

而通过 CPU 和内存密集的图进行对比，可以看到内存密集的 cpu 占用几乎为 0，但是当内存读写频率提高后，右侧内存的读写量却能很快超过 CPU 密集进程，由此可以验证程序正确。

思考题：

思考题 1：考虑了，见之前所述。

思考题 2：utime 是在用户空间的进程中实际执行的时间量。 stime 是代表进程在 OS 内核中花费的时间量，当系统调用发生时，进程进入内核模式。而我们要监控的 benchmark 进程显然并不会进行（或很少）系统调用，内存访问和 CPU 计算都在用户空间中进行，因此用 utime 更合理。

思考题 3：

内存为页，精度不是最高的，下面是一些问题：

1. 可能多个虚拟地址对应的物理地址在一个 page 上，这样的话相当于对一个 page 算了多次。
2. 如果检测频率比较低（相对的内存读写频率高），同一个地方的内存如果两次采样时间内被写很多次，检测出来也只是一次，见实验结果的内存密集型进程部分。

3. 一个页上有 4KB 个物理地址，若只有一个物理地址被读写，以页为单位就是视作整个 4kB 的空间都被读写了，这明显会偏大。

解决办法：把检测粒度调整到单个物理地址，但是我并没有找到相应的函数，自己写的话初步逻辑是将一个进程所有 pte 结构体的地址经过哈希函数后作为 index 构建映射数组（已经实现，见//hash 处），类似 ptep_test_and_clear_young 函数实现检测某个物理地址近期是否被读写（未实现），每次采样遍历所有的地址就可以得到总读写内存量，通过上述的数据映射可以进一步获得每个 Page 的读写次数。

总结：

通过对 cpu 和内存编写追踪程序，自己将前几个 Lab 的实验代码都复习了一些（参考了之前的代码实现），对 Linux 内核的各种数据结构进一步熟悉，巩固了所学知识，锻炼了代码能力。

实验中遇到的其他主要问题：

出现空指针引用：进行排查和猜想：mmap 不是双向链表，而我的判断逻辑为 end 指针等于 start 指针时停止（不是循环链表那么就会有 NULL 指针）。验证想法：通过每次循环内将 mmap->vm_next 替换为 mmap->vm_prev，这样出现空引用的时间会更快，（打印内存访问次数到 1 就出现空指针引用），确定问题。

pte 指针有时候为 NULL，跳过即可。

由于调试原因在循环内部使用了太多的 pr_info 导致虚拟机卡死，注释后正常。

对于 multi_porcess.py 的子进程，不可以这样调用：

```
src > multi_process.py > [@] message
import os

print(os.getpid())

p1=Popen(["sysbench --cpu-max-prime=1000 --threads=1 --time=0 cpu run"],shell=True)

# print(666)

p2=Popen(["sysbench --cpu-max-prime=1000 --threads=1 --time=0 cpu run"],shell=True)

message = input("输入t kill所有子进程")
if(message=='t'):
    p1.kill()
    p2.kill()
```

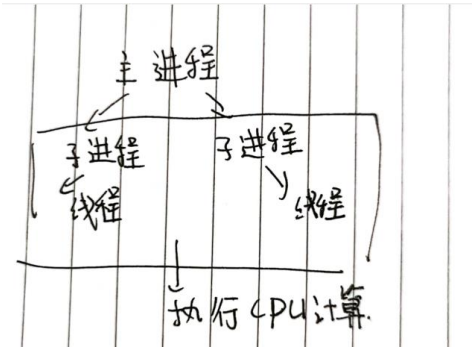
因为内核程序只实现了检测深度为 1 的子进程，使用 shell-True 深度为 2（子进程又开一个子进程来执行），导致输出全为 0。

（参考：https://stackoverflow.com/questions/3172470/actual-meaning-of-shell-true-in-subprocess）

如此修改即可：

```
p1=Popen(["sysbench","--cpu-max-prime=1000","--threads=1","--time=0","cpu","run"])
```

，调用过程为



Dmesg 也显示正确：