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

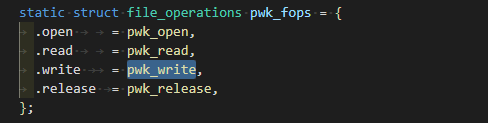
process\_watch.c 用户态程序

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：



主要针对write和read操作进行代码编写。

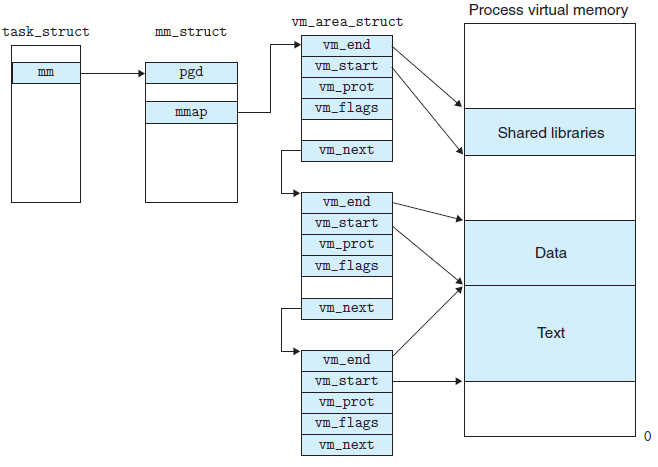
pwk\_write函数写入pid。

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。

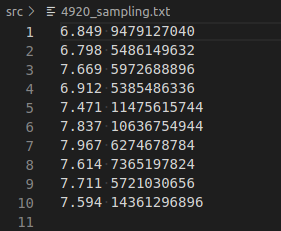
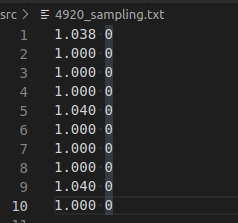


最后返回给用户态程序的为cpu和内存数据的拼接，单位分别为us和字节，用空格区分两个数据。

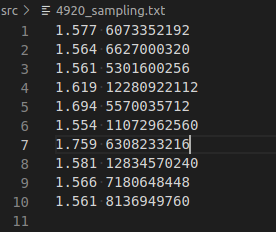
process\_watch.c：

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：

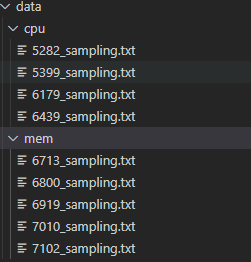
采样间隔 为100ms时：

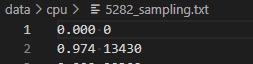
计算内存数据：不计算内存数据：

采样间隔为1000ms:

不计算内存时结果不变，计算内存：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）：

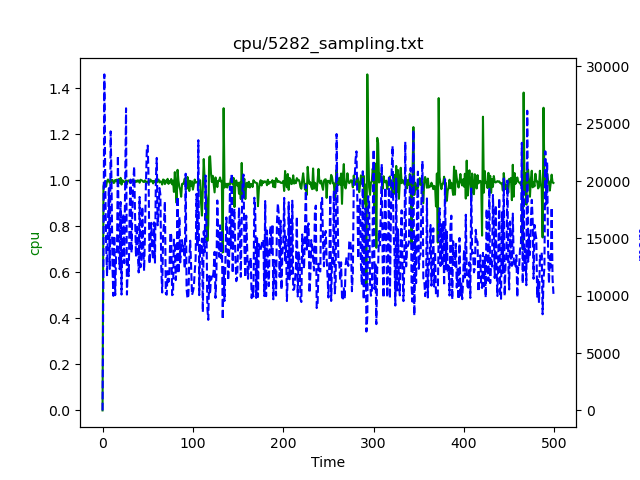
（内存的数据我在写完报告后发现自己计算出读写的page量后乘了两次PAGE\_SIZE，代码已经改成了正确的，数据要重新做的话步骤冗杂，由于不影响实验分析，就不重新跑了）

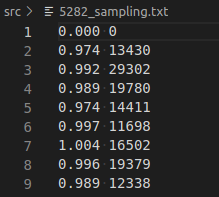
蓝色为内存数据，绿色为cpu数据。

计算密集型进程：

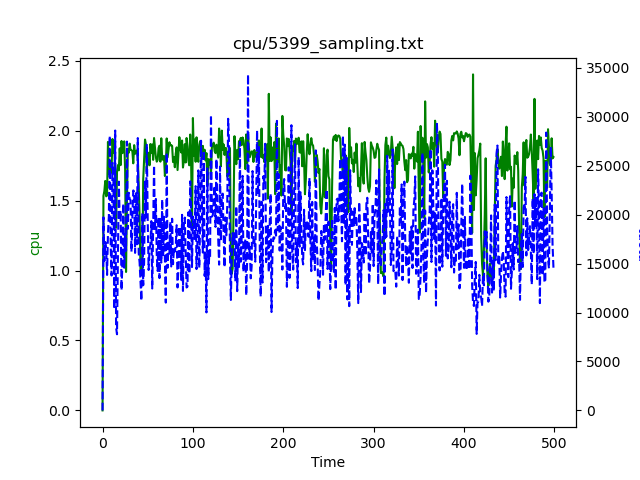
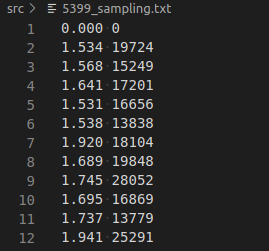
采用sysbench --cpu-max-prime=1000 --threads=x --time=0 cpu run来产生，通过htop肉眼观察进程PID。(threads=x验证子线程代码)

threads=1：部分数据展示：

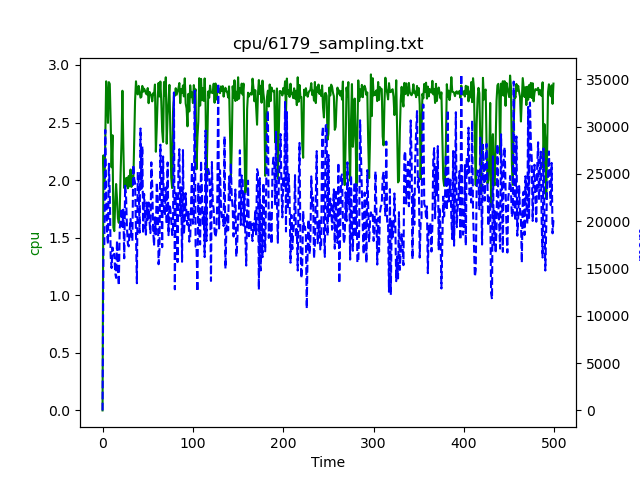
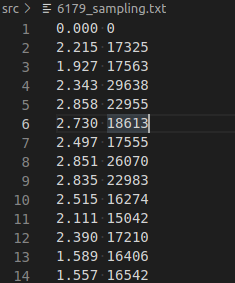




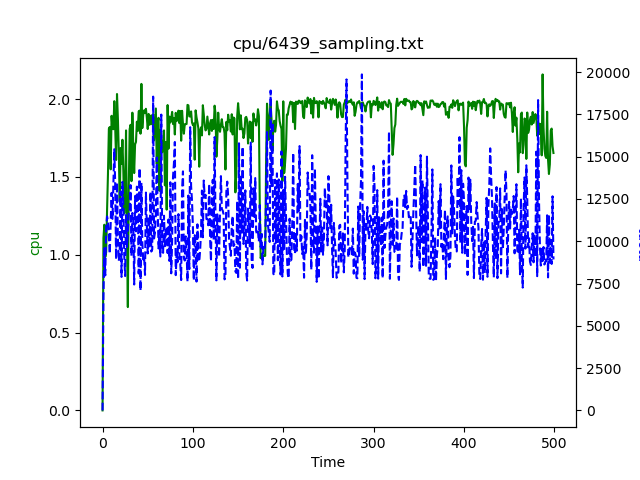
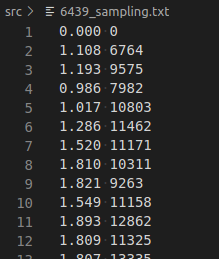
threads=2：部分数据展示：



threads=3：部分数据展示：



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）：



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）。

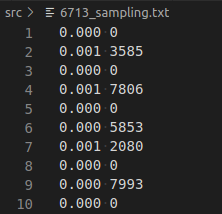
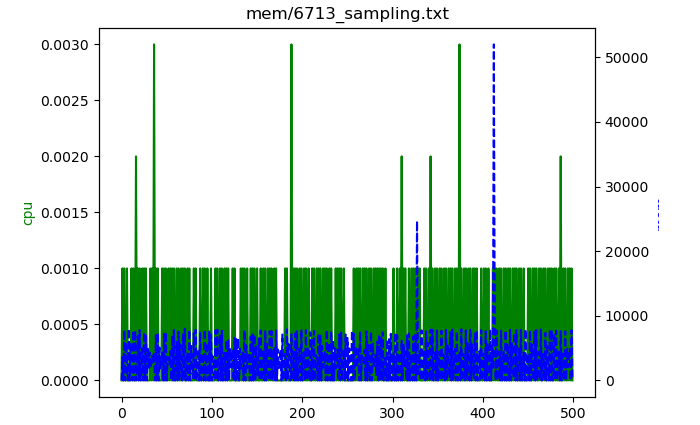
内存密集型进程：

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

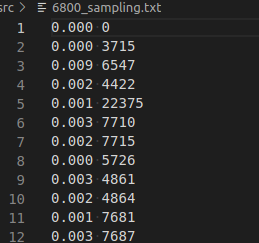
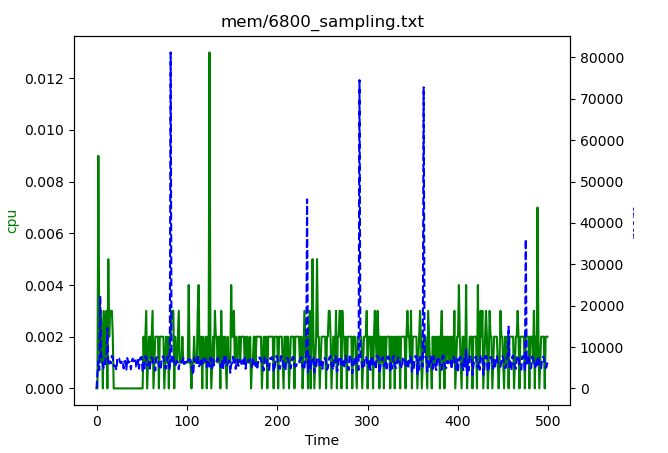


通过调整sleep的间隔来控制内存读写频率。

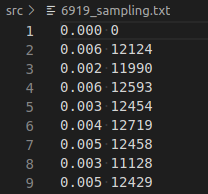
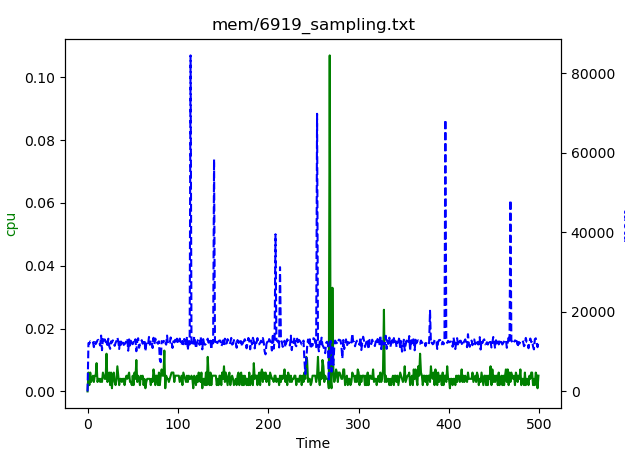
当其为100ms时：

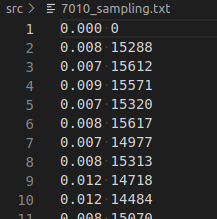
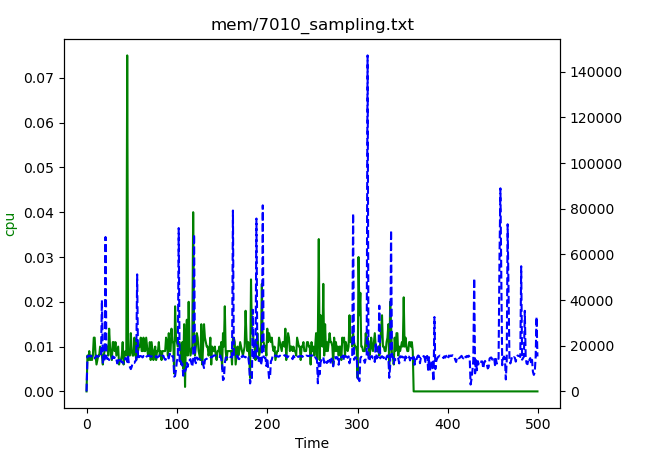
为50ms时：

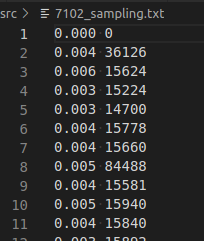
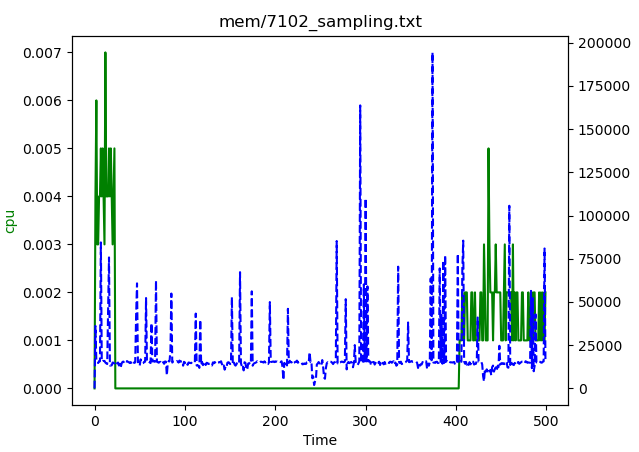
为25ms时：

为10ms时：

为5ms时：

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：



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

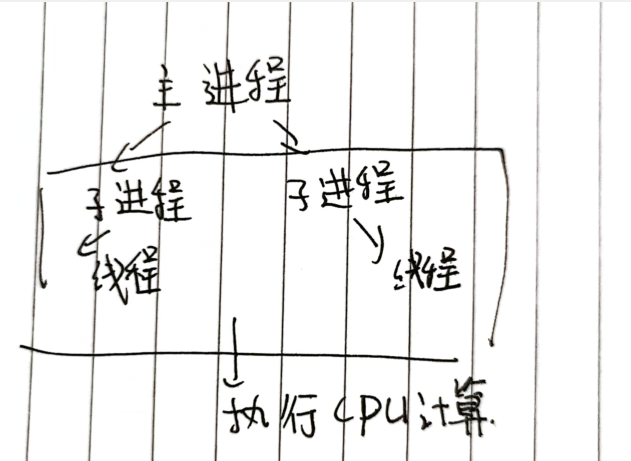
子进程来执行），导致输出全为0。

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

如此修改即可：



，调用过程为



Dmesg也显示正确：

