作业1

1.1 Linux 下常见的3种系统调用方法包括有：
（1）通过glibc提供的库函数
（2）使用syscall函数直接调用相应的系统调用
（3）通过int 80指令（32位系统）或者syscall指令（64位系统）的内联汇编调用

请研究Linux(kernel>=2.6.24) getpid这一系统调用的用法，使用上述3种系统调用方法来执行，并记录和对比3种方法的运行时间，并尝试解释时间差异结果。

提示：gettimeofday和clock_gettime是Linux下用来测量耗时的常用函数，请调研这两个函数，选择合适函数来测量一次系统调用的时间开销。

提交内容：所写程序、执行结果、结果分析、系统环境（uname -a)等。

程序示例：

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <unistd.h>
#include <time.h>

int max_test_times = 1000;
int main() {
   // 定义运行测试的次数
    long int elapsed_time_glibc = 0, elapsed_time_syscall = 0,
elapsed_time_asm = 0;
    struct timespec start, end;
    pid_t pid_glibc,pid_syscall,pid_asm;
    for (int i = 0; i < max_test_times; i++) {
        // 使用 glibc 库函数调用 getpid
        clock_gettime(CLOCK_MONOTONIC, &start);
         pid_glibc = getpid();
        clock_gettime(CLOCK_MONOTONIC, &end);
        elapsed_time_glibc += (end.tv_nsec - start.tv_nsec) ;

        // 使用 syscall 函数直接调用 getpid
        clock_gettime(CLOCK_MONOTONIC, &start);
         pid_syscall = syscall(SYS_getpid);
        clock_gettime(CLOCK_MONOTONIC, &end);
        elapsed_time_syscall +=(end.tv_nsec - start.tv_nsec);
```

```
        // 使用内联汇编调用 getpid
        clock_gettime(CLOCK_MONOTONIC, &start);
        asm volatile (
            "syscall"
            : "=a" (pid_asm)
            : "0" (SYS_getpid)
        );
        clock_gettime(CLOCK_MONOTONIC, &end);
        elapsed_time_asm += (end.tv_nsec - start.tv_nsec)+(end.tv_sec-
start.tv_sec)*1e9;
    }

    long int average_time_glibc = elapsed_time_glibc;
    long int average_time_syscall = elapsed_time_syscall ;
    long int average_time_asm = elapsed_time_asm ;

    printf("Average time taken by glibc: %ld ns\n", average_time_glibc);
    printf("Average time taken by syscall: %ld ns\n",
average_time_syscall);
    printf("Average time taken by asm: %ld ns\n", average_time_asm);

    return 0;
}
```

代码设计：

测试三种系统调用花费时间的代码设计思路比较简单，只需要在进行 getpid 调用前后使用 clock_gettingtime 函数，选择该函数的原因是因为可以支持内置 timespec 结构体中 ns 的测量，预测试时发现运行结果比较接近，所以使用该函数可以时测量结果更加的精确。定义三个长整型变量用来记录时间开销和两个高精度时间结构体来记录时间戳的值，利用进程号类型来保存函数返回值，分别在函数运行前后记录时间戳，最后记录结果。取 1000 次测量结果的和作为比较结果可以消除偶然性。

●将代码运行在研讨课提供的虚拟机

Linux stu 5.4.0-162-generic #179-Ubuntu SMP Mon Aug 14 08:51:31 UTC 2023 x86_64 x86_64 x86_64 GNU/Linux

结果如下：

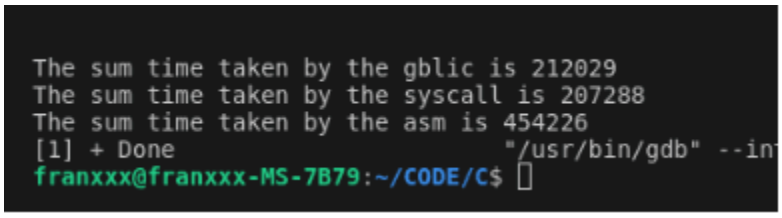
共执行了 1000 次测试，结果为 1000 次时间的和

```
The sum time taken by the gblic is 66859
The sum time taken by the syscall is 67046
The sum time taken by the asm is 65681
```

●在自己的 ubuntu 原生操作操作系统上运行

Linux franxxx-MS-7B79 6.2.0-32-generic #32~22.04.1-Ubuntu SMP PREEMPT_DYNAMIC Fri Aug 18 10:40:13 UTC 2 x86_64 x86_64 x86_64 GNU/Linux

结果如下：



结果分析：

在虚拟机上运行得到的结论是 glibc>syscall>asm。

使用 glibc 库函数调用会涉及到一定的函数调用开销，调用库函数时，需要执行函数调用操作，这会导致将控制权从当前函数传递到库函数，然后再从库函数返回到当前函数。并且函数调用涉及保存和恢复上下文、参数传递等操作，这些都需要额外的时间。因此一般来说，它会是最慢的方法；使用 syscall 函数直接调用，绕过了库函数的封装。syscall 函数通常会生成相对较少的代码，并且不涉及额外的函数调用，因此可能更快；使用内联汇编调用是最直接的方法，使用汇编指令直接调用系统调用，减少了函数调用开销。因此，它可能是最快的方法，

在我自己的原生操作系统上运行同样的代码却得出来全然不同的结论，结果是 asm>gblic>syscall，ASM 内联汇编所运行的时间远远大于其他两种方法，猜测可能是因为我电脑所使用的 CPU 是锐龙 2700Xzen+架构，不太适合执行该类型的汇编代码。另一方面也有可能是因为编译器没有对存在 ASM 内联汇编的 C 编译文件进行足够优化，目前我仍无法得到具体原因。希望能在接下来的学习中可以找到答案。

注：进行多次测量时，可能会出现不同的结果（例如 syscall 函数最快，另外两者较慢），但是大体上依然符合得到的结论，可能也跟每次执行不同的进程有关。