第一次作业

本次作业使用的 Linux 系统环境为：

```
Linux CN 6.10.6-orbstack-00249-g92ad2848917c #1 SMP Tue Aug 20 15:46:01 UTC 2024
    x86_64 x86_64 x86_64 GNU/Linux
```

本次作业的源代码如下：

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <time.h>
#include <syscall.h>

void gettime_glibc() {
   struct timespec start, end;
   clock_gettime(CLOCK_MONOTONIC, &start);
   getpid();
   clock_gettime(CLOCK_MONOTONIC, &end);
   printf("getpid costs %ld ns through library function provided by glibc\n",
       (end.tv_sec - start.tv_sec) * 1000000000 + (end.tv_nsec - start.tv_nsec));

   clock_gettime(CLOCK_MONOTONIC, &start);
   open("test.txt", O_CREAT);
   clock_gettime(CLOCK_MONOTONIC, &end);
   printf("open costs %ld ns through library function provided by glibc\n",
       (end.tv_sec - start.tv_sec) * 1000000000 + (end.tv_nsec - start.tv_nsec));
}

void gettime_syscall() {
   struct timespec start, end;
   clock_gettime(CLOCK_MONOTONIC, &start);
   syscall(SYS_getpid);
   clock_gettime(CLOCK_MONOTONIC, &end);
   printf("getpid costs %ld ns through syscall function directly\n", (end.tv_sec
       - start.tv_sec) * 1000000000 + (end.tv_nsec - start.tv_nsec));

   clock_gettime(CLOCK_MONOTONIC, &start);
   syscall(SYS_open, "test.txt", O_CREAT);
   clock_gettime(CLOCK_MONOTONIC, &end);
   printf("open costs %ld ns through syscall function directly\n", (end.tv_sec -
       start.tv_sec) * 1000000000 + (end.tv_nsec - start.tv_nsec));
}
```

```
void gettime_asm(){
   struct timespec start, end;
   int pid;
   clock_gettime(CLOCK_MONOTONIC, &start);
   asm volatile(
      "mov $39, %%rax\n\t" // getpid is 39 in unistd.h
      "syscall\n\t"
      "mov %%eax, %0\n\t"
      : "=r" (pid)
      :
      : "rax", "rcx", "r11"
   );
   clock_gettime(CLOCK_MONOTONIC, &end);
   printf("getpid costs %ld ns through syscall in inline assembly\n", (end.tv_sec
       - start.tv_sec) * 1000000000 + (end.tv_nsec - start.tv_nsec));

   int fd;
   char *filename = "test.txt";
   clock_gettime(CLOCK_MONOTONIC, &start);
   asm volatile(
      "mov $2, %%rax\n\t" // open is 2 in unistd.h
      "mov $1, %%rdi\n\t" // 文件名
      "mov $0x40, %%rsi\n\t" // O_CREAT is 0x40
      "syscall\n\t"
      "mov %%eax, %0\n\t"
      : "=r" (fd)     // 输出到fd变量
      : "r" (filename) // 输入文件名
      : "rax", "rdi", "rsi", "rcx", "r11" // 被破坏的寄存器
   );
   clock_gettime(CLOCK_MONOTONIC, &end);
   printf("open costs %ld ns through syscall in inline assembly\n", (end.tv_sec -
       start.tv_sec) * 1000000000 + (end.tv_nsec - start.tv_nsec));
   // 此处内联汇编代码使用了ChatGPT，因为我对汇编语言不是很熟悉，
   // %%rax用于存储系统调用号，%%rcx用于存储返回地址，
   // %%rdi用于传递第一个参数，%%rsi用于传递第二个参数。
}

int main() {
   gettime_glibc();
   printf("\n");
   gettime_syscall();
   printf("\n");
```

```
    gettime_asm();
    return 0;
}
```

下面三个表格分别是通过 glibc 提供的库函数调用、通过 `syscall` 函数调用和通过 `syscall` 指令内联汇编调用的实验结果，单位为 ns。

表 1. 通过 glibc 提供的库函数调用的实验结果

系统调用 / 序号	getpid	open
1	13375	73042
2	13334	66625
3	13333	63876
4	13417	68376
5	13125	66500
6	13334	75292
7	12875	64042
8	13292	57417
9	13250	61250
10	12833	57542
平均值	13216.8	65396.2

表 2. 通过 `syscall` 函数调用的实验结果

系统调用 / 序号	getpid	open
1	18667	17083
2	24375	17584
3	14875	16375
4	18583	18458
5	15417	17583
6	14209	22750
7	14292	21667
8	13250	14500
9	18292	21250
10	12667	16500
平均值	16462.7	18375

表 3. 通过 syscall 指令内联汇编调用的实验结果

系统调用 / 序号	getpid	open
1	7292	6542
2	5417	6791
3	5250	5875
4	5541	9708
5	5333	5792
6	5333	5750
7	5375	5917
8	4709	5125
9	6666	5417
10	4750	5791
平均值	5566.6	6270.8

对比以上数据可以发现：

1. 对于 getpid 系统调用，通过 syscall 指令内联汇编调用的效率最高，通过 glibc 提供的库函数调用和 syscall 函数调用的效率相差不大；
2. 对于 open 系统调用，通过 syscall 指令内联汇编调用的效率最高，通过 glibc 提供的库函数调用的效率最低，通过 syscall 函数调用的效率居中；
3. 使用内联汇编时，getpid 和 open 实现调用的运行时间差异不大，getpid 的运行时间稍短些。

使用内联汇编速度最快是很显然的，因为内联汇编直接使用 syscall 指令，省去了函数调用的传参、返回等等开销，省去了上下文的转换。使用 syscall 函数直接调用 open 函数显著快于使用 glibc 提供的库函数调用，这是因为 glibc 提供的库函数调用仍有一定上述开销，而 syscall 函数直接调用 open 函数，省去了这些开销。对于 getpid 函数，syscall 函数调用和 glibc 提供的库函数调用的效率相差不大，这是因为 getpid 函数是一个非常简单的系统调用，开销很小。

getpid 和 open 在内联汇编中的实现，getpid 的运行时间稍短些。这是因为 getpid 仅获取当前进程的 PID，非常简单，而 open 还涉及寻找文件、创建文件、打开文件等操作，开销较大，因此 open 的运行时间较长。

此外我也发现，open 和 getpid 的运行时间有一定的波动，甚至有些数据的波动还较大，我估计原因和当前系统的负载有关。