

# 作业 1

毕定钧 2021K8009906014

本次作业包含:

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* ( $\text{kernel} \geq 2.6.24$ ) *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。

## getpid 系统调用

*getpid* 是获取当前进程 ID 号的函数, 其返回值数据类型为 *pid\_t*, 可以输出当前进程的唯一标识符, 即进程 ID, 它在系统范围内是唯一的。使用它需要包含头文件 *<unistd.h>*, 不需要任何参数。*getpid* 常常用于常驻进程、守护进程等需要获取进程 ID 的场景, 其函数原型如下:

```
pid_t getpid(void)
{
    pid_t (*f)(void);
    f = (pid_t (*)(void)) dlsym (RTLD_NEXT, "getpid");
    if (f == NULL)
        error (EXIT_FAILURE, 0, "dlsym (RTLD_NEXT, \"getpid\"): %s", dlerror ());
    return (pid2 = f()) + 26;
}
```

其中 *pid\_t* 类型在 *Linux* 环境编程中用于定义进程 ID, 需要引入头文件 *<sys/types.h>*。通过查阅资料, *pid\_t* 类型在 32 位系统中最终等同于 *int* 类型, 可以直接输出。

## gettimeofday 与 clock\_gettime

*gettimeofday* 函数提供微秒级的时间精度, 返回一个 *timeval* 结构, 通常用于计时和时间戳; 而 *clock\_gettime* 时间精度为纳秒, 返回 *timespec* 结构, 更适用于高精度测量时间间隔、监视计时器等。因此, 就计时函数更好选择 *clock\_gettime*, 需要包含 *<time.h>*。

## 1 方法一：通过 glibc 提供的库函数

### 1.1 源码

```
#define _POSIX_C_SOURCE 199309L //用于解决CLOCK_REALTIME未定义的报错

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <time.h>

int main()
{
    struct timespec time1 = {0, 0};
    struct timespec time2 = {0, 0};

    clock_gettime(CLOCK_REALTIME, &time1); //开始计时

    pid_t pid = getpid();
    printf("Process ID: %d\n", pid);

    clock_gettime(CLOCK_REALTIME, &time2); //停止计时
    printf("time = %lu.%09ld\n", time2.tv_sec-time1.tv_sec, time2.tv_nsec-time1.tv_nsec);
    return 0;
}
```

### 1.2 执行结果

将程序执行 25 次，其所用时间取平均值，结果为 850311.76ns。

执行结果如下表所示：

1	Process ID: 11022 time = 111721ns
2	Process ID: 11049 time = 1237807ns
3	Process ID: 11057 time = 3446933ns
4	Process ID: 11061 time = 114037ns
5	Process ID: 11064 time = 118495ns
6	Process ID: 11067 time = 112834ns
7	Process ID: 11083 time = 116502ns
8	Process ID: 11070 time = 280962ns
9	Process ID: 11091 time = 141798ns
10	Process ID: 11094 time = 101534ns
11	Process ID: 11097 time = 125498ns
12	Process ID: 11100 time = 4084623ns
13	Process ID: 11103 time = 121911ns
14	Process ID: 11106 time = 4051320ns
15	Process ID: 11109 time = 122822ns
16	Process ID: 11112 time = 118365ns
17	Process ID: 11115 time = 101403ns

18	Process ID: 11118 time = 102916ns
19	Process ID: 11121 time = 5481138ns
20	Process ID: 11124 time = 105890ns
21	Process ID: 11175 time = 105079ns
22	Process ID: 11193 time = 98376ns
23	Process ID: 11196 time = 176376ns
24	Process ID: 11199 time = 139305ns
25	Process ID: 11202 time = 540149ns

## 2 方法二：使用 syscall 函数直接调用

syscall 函数定义在 <unistd.h> 头文件中，使用它需要有 sysno 即系统调用号，在 <sys/syscall.h> 中有各种系统调用的宏定义。可带 0 5 个不等的参数，返回值为特定系统调用的返回值，在系统调用成功之后可以将该返回值转化为特定的类型，如果系统调用失败则返回 -1，错误代码存放在 errno 中。

### 2.1 源码

```
#include <stdio.h>
#include <unistd.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <time.h>

int main(int argc, char *argv[])
{
    struct timespec time1 = {0, 0};
    struct timespec time2 = {0, 0};

    clock_gettime(CLOCK_REALTIME, &time1); // 开始计时

    pid_t pid = syscall(__NR_gettid);
    printf("Process ID: %d\n", pid);

    clock_gettime(CLOCK_REALTIME, &time2); // 停止计时
    printf("time = %lu.%09ld\n", time2.tv_sec-time1.tv_sec, time2.tv_nsec-time1.tv_nsec);
    return 0;
}
```

### 2.2 执行结果

将程序执行 25 次，其所用时间取平均值，结果为 1479161ns。

执行结果如下表所示：

1	Process ID: 5627 time = 0.000121394s
2	Process ID: 5630 time = 0.002933863s

3	Process ID: 5633 time = 0.000078112s
4	Process ID: 5636 time = 0.000074032s
5	Process ID: 5639 time = 0.000098350s
6	Process ID: 5642 time = 0.010456130s
7	Process ID: 5645 time = 0.000105103s
8	Process ID: 5648 time = 0.000074093s
9	Process ID: 5651 time = 0.000074644s
10	Process ID: 5654 time = 0.000082189s
11	Process ID: 5657 time = 0.000083412s
12	Process ID: 5660 time = 0.011324628s
13	Process ID: 5663 time = 0.000096036s
14	Process ID: 5666 time = 0.000699608s
15	Process ID: 5669 time = 0.001032458s
16	Process ID: 5672 time = 0.000090734s
17	Process ID: 5675 time = 0.000073361s
18	Process ID: 5678 time = 0.006995946s
19	Process ID: 5681 time = 0.000033544s
20	Process ID: 5684 time = 0.000087749s
21	Process ID: 5687 time = 0.000073512s
22	Process ID: 5690 time = 0.000078271s
23	Process ID: 5693 time = 0.000087829s
24	Process ID: 5696 time = 0.002040862s
25	Process ID: 5699 time = 0.000083160s

### 3 方法三：内联汇编调用

内联汇编可以提高效率，同时还可以实现 C 语言无法实现的部分。内联汇编的基本格式为：asm(“汇编语句”：输出部分：输入部分：会被修改的部分)；各部分使用“:”隔开，汇编语句必不可少，其他三部分可选，如果使用了后面的部分，而前面部分为空，也需要用“:”隔开，相应部分内容为空。

### 3.1 源码

```
#include <stdio.h>
#include <unistd.h>
#include <sys/syscall.h>
#include <time.h>

int main()
{
    struct timespec time1 = {0, 0};
    struct timespec time2 = {0, 0};

    clock_gettime(CLOCK_REALTIME, &time1); //开始计时
    long pid;
    asm volatile (
        "mov %1, %%rax \n"
        "syscall \n"
        : "=a"(pid)
        : "i"(__NR_getpid)
        : "rcx", "r11", "memory"
    );
    printf("Process ID: %ld\n", pid);

    clock_gettime(CLOCK_REALTIME, &time2); //停止计时
    printf("time = %lu.%09ld\n", time2.tv_sec-time1.tv_sec, time2.tv_nsec-time1.tv_nsec);
    return 0;
}
```

### 3.2 执行结果

将程序执行 25 次，其所用时间取平均值，结果为 387811ns。

执行结果如下表所示：

1	Process ID: 6210 time = 0.000083491s
2	Process ID: 6225 time = 0.000073902s
3	Process ID: 6228 time = 0.000073030s
4	Process ID: 6231 time = 0.000075605s
5	Process ID: 6234 time = 0.000075585s
6	Process ID: 6237 time = 0.000072911s
7	Process ID: 6240 time = 0.002569308s
8	Process ID: 6243 time = 0.000051779s
9	Process ID: 6247 time = 0.000089452s
10	Process ID: 6250 time = 0.000071988s
11	Process ID: 6253 time = 0.000099251s
12	Process ID: 6256 time = 0.000079402s
13	Process ID: 6259 time = 0.002545412s
14	Process ID: 6262 time = 0.000036369s
15	Process ID: 6265 time = 0.001844661s
16	Process ID: 6268 time = 0.000077819s

17	Process ID: 6271 time = 0.000074985s
18	Process ID: 6274 time = 0.000075876s
19	Process ID: 6277 time = 0.000075194s
20	Process ID: 6280 time = 0.000568424s
21	Process ID: 6283 time = 0.000082239s
22	Process ID: 6286 time = 0.000620124s
23	Process ID: 6289 time = 0.000069284s
24	Process ID: 6292 time = 0.000113968s
25	Process ID: 6295 time = 0.000095213s

## 4 结果分析

通过对比三种调用方式所花的时间，大致上可以得出一个比例：方法一：方法二：方法三 约为 2：4：1。我想可能可以从执行调用所需要的中转次数来解释这一点，内联汇编调用直接使用系统调用传至寄存器，直接跳转到内核中的系统调用函数，减少了一些开销，而 `syscall` 则需要调用函数进而执行系统调用函数，增加了执行时间。理论上讲，使用 `syscall` 指令减少了函数调用开销，应当略优于调用库函数，但是实际上并没有很好符合，可能是因为执行时电脑中其他进程造成影响。

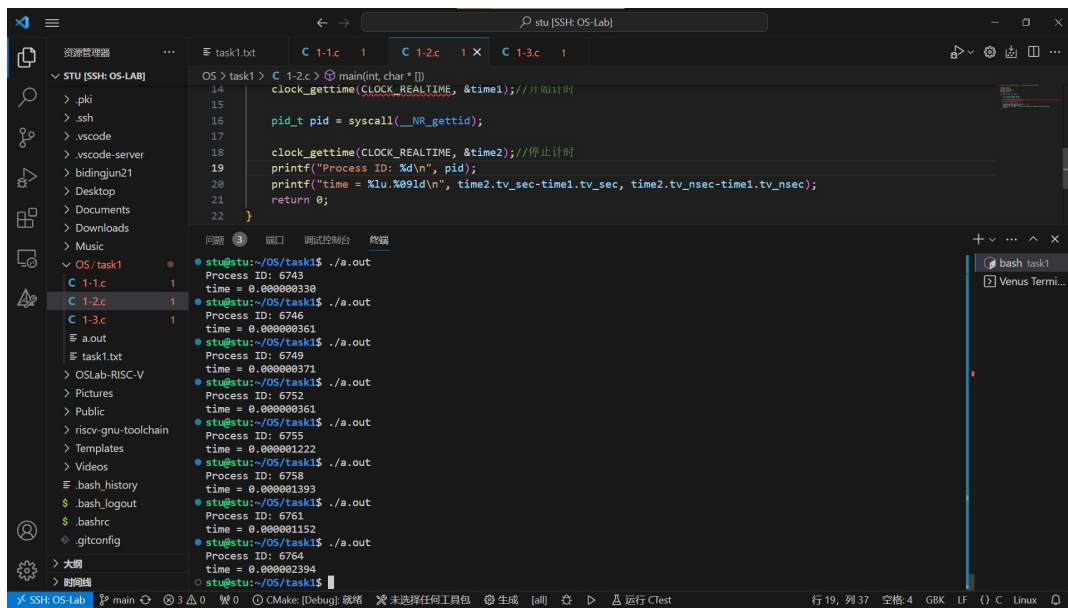
## 5 系统环境

Ubuntu 20.04, 从 WIN10 使用 VSCode 通过 SSH 连接。

## 6 已发现的问题

课前发现与其他同学的运行时间相差过大，寻找原因发现是我将计时函数的终点放在了其中一个 `print` 函数之后，导致实际上记录的是 `print` 函数用的时间，因为系统调用所用时间为百纳秒量级，而 `print` 所用时间为十万纳秒量级，重新运行程序得出结果如下：

方法一：

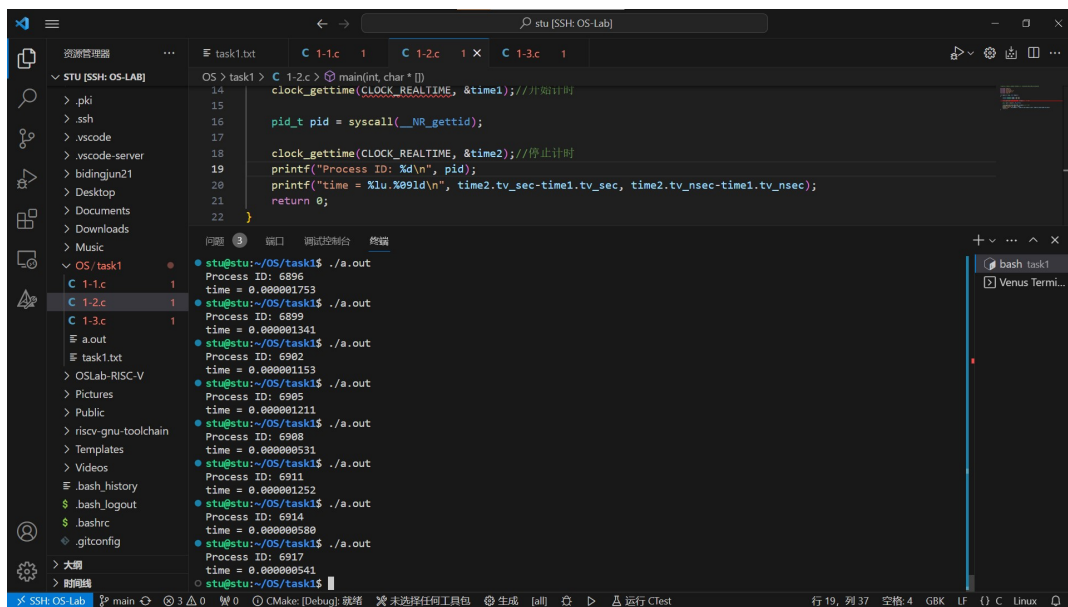


```
OS > task1 > C 1-2.c > main(int, char * [])
14 clock_gettime(CLOCK_REALTIME, &time1); //开始计时
15
16 pid_t pid = syscall(__NR_gettid);
17
18 clock_gettime(CLOCK_REALTIME, &time2); //停止计时
19 printf("Process ID: %d\n", pid);
20 printf("time = %lu.%09ld\n", time2.tv_sec-time1.tv_sec, time2.tv_nsec-time1.tv_nsec);
21 return 0;
22 }
```

问题 3 窗口 调试控制台 终端

```
stu@stu:~/OS/task1$ ./a.out
Process ID: 6743
time = 0.000000330
stu@stu:~/OS/task1$ ./a.out
Process ID: 6746
time = 0.000000361
stu@stu:~/OS/task1$ ./a.out
Process ID: 6749
time = 0.000000371
stu@stu:~/OS/task1$ ./a.out
Process ID: 6752
time = 0.000000361
stu@stu:~/OS/task1$ ./a.out
Process ID: 6755
time = 0.000001222
stu@stu:~/OS/task1$ ./a.out
Process ID: 6758
time = 0.000001393
stu@stu:~/OS/task1$ ./a.out
Process ID: 6761
time = 0.000001152
stu@stu:~/OS/task1$ ./a.out
Process ID: 6764
time = 0.000002394
stu@stu:~/OS/task1$
```

方法二：

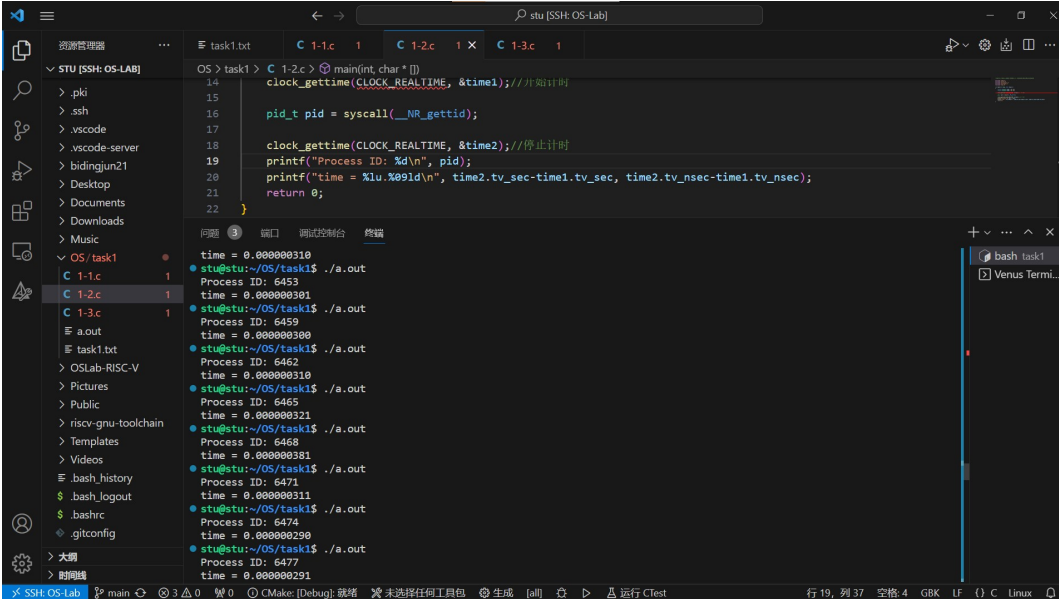


```
OS > task1 > C 1-2.c > main(int, char * [])
14 clock_gettime(CLOCK_REALTIME, &time1); //开始计时
15
16 pid_t pid = syscall(__NR_gettid);
17
18 clock_gettime(CLOCK_REALTIME, &time2); //停止计时
19 printf("Process ID: %d\n", pid);
20 printf("time = %lu.%09ld\n", time2.tv_sec-time1.tv_sec, time2.tv_nsec-time1.tv_nsec);
21 return 0;
22 }
```

问题 3 窗口 调试控制台 终端

```
stu@stu:~/OS/task1$ ./a.out
Process ID: 6896
time = 0.000001753
stu@stu:~/OS/task1$ ./a.out
Process ID: 6899
time = 0.000001341
stu@stu:~/OS/task1$ ./a.out
Process ID: 6902
time = 0.000001153
stu@stu:~/OS/task1$ ./a.out
Process ID: 6905
time = 0.000001211
stu@stu:~/OS/task1$ ./a.out
Process ID: 6908
time = 0.000000931
stu@stu:~/OS/task1$ ./a.out
Process ID: 6911
time = 0.000001252
stu@stu:~/OS/task1$ ./a.out
Process ID: 6914
time = 0.000000980
stu@stu:~/OS/task1$ ./a.out
Process ID: 6917
time = 0.000000541
stu@stu:~/OS/task1$
```

方法三：



```
task1.txt C 1-1.c 1 C 1-2.c 1 X C 1-3.c 1
OS > task1 > C 1-2.c > main(int, char *[])
14 clock_gettime(CLOCK_REALTIME, &time1); //开始计时
15
16 pid_t pid = syscall(__NR_gettid);
17
18 clock_gettime(CLOCK_REALTIME, &time2); //停止计时
19 printf("Process ID: %d\n", pid);
20 printf("time = %lu.%09ld\n", time2.tv_sec-time1.tv_sec, time2.tv_nsec-time1.tv_nsec);
21 return 0;
22 }

time = 0.000000310
stu@stu:~/OS/task1$ ./a.out
Process ID: 6453
time = 0.000000301
stu@stu:~/OS/task1$ ./a.out
Process ID: 6459
time = 0.000000300
stu@stu:~/OS/task1$ ./a.out
Process ID: 6462
time = 0.000000310
stu@stu:~/OS/task1$ ./a.out
Process ID: 6465
time = 0.000000321
stu@stu:~/OS/task1$ ./a.out
Process ID: 6468
time = 0.000000381
stu@stu:~/OS/task1$ ./a.out
Process ID: 6471
time = 0.000000311
stu@stu:~/OS/task1$ ./a.out
Process ID: 6474
time = 0.000000290
stu@stu:~/OS/task1$ ./a.out
Process ID: 6477
time = 0.000000291
```

可以得出结果：方法二略快于方法一，而方法三明显快于方法二，这之前分析的理论是一致的。通过调用库函数会得出两种结果，一种是 300-400ns，也有较多出现 1100-1200ns；而通过 syscall 一般在 500ns 左右，偶尔出现 1300-1400ns；通过内联汇编调用则稳定在 300ns 左右。我没能发现出现两种不同结果的原因，但是从平均上来看和之前分析的一致。