操作系统 第一次作业

姓名：朱首赫

学号：2023K8009906029

1.1 Linux 下常见的 3 种系统调用方法包括：

（1）通过 glibc 库提供的库函数调用

（2）使用 syscall 函数直接调用相应的系统调用

（3）通过 syscall 指令（64 位系统）或 int 0x80 指令（32 位系统）的内联汇编调用

请研究 Linux(kernel>=5.10) getpid 和 open 这两个系统调用的用法，分别使用上述 3 种系统调用方法来执行。

要求：

1）针对同一个系统调用，记录和对比 3 种方法的运行时间，并尝试解释时间差异的原因；

2）对比 getpid 和 open 这两个系统调用在使用内联汇编实现调用时的运行时间，如果有差异，尝试解释差异原因。若无差异，说明即可。

提示：

1）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。

2）open 系统调用可以打开一个已存在或不存在的文件。如果打开一个不存在的文件，请在 open 系统调用的参数设置中使用 O_CREAT 标志，即文件不存在时默认创建该文件。

提交内容：所写程序源码、执行结果、结果分析、系统环境（uname -a）等。

1 计时函数选择

调研发现，gettimeofday() 的精度为微秒级（μs），而 clock_gettime() 的精度为纳秒级（ns），由于系统调用的时间开销非常小，clock_gettime() 的纳秒级精度能得到更准确的测量结果。因此选择 clock_gettime() 作为计时函数。

2 系统环境

Linux kolp 6.6.87.2-microsoft-standard-WSL2 1 SMP PREEMPT_DYNAMIC Thu Jun 5 18:30:46 UTC 2025 x86_64 x86_64 x86_64 GNU/Linux

3 使用 3 种方法调用 getpid 的运行时间对比

3.1 程序源码

分别使用 3 种方法调用 getpid，每种方法重复十万次取平均运行时间，以减小随机误差。

Listing 1: 使用 glib 库中的 getpid 函数

```c
#include <stdio.h>
#include <unistd.h> // 包含POSIX标准定义的unix类系统定义符号常量和函数
#include <time.h>   // 包含计时函数
// strace timespec{
//  time_t tv_sec;
//  long tv_nsec;
// }
int main() {
    struct timespec start, end;
    long diff;
    // 重复调用10000次取平均值
    for (int i = 0; i < 100000; i++)
    {
        clock_gettime(CLOCK_MONOTONIC, &start);
        pid_t pid = getpid(); // 直接调用 glibc 库函数
        clock_gettime(CLOCK_MONOTONIC, &end);
        diff += (end.tv_sec - start.tv_sec) * 1000000000L + (end.tv_nsec - start.tv_nsec);
            // 转换为ns
    }

    diff /= 100000; // 取平均值
    printf("glibc    getpid() average time: %ld ns\n", diff);
    return 0;
}
```

Listing 2: 使用 syscall 调用 getpid

```c
#include <stdio.h>
#include <unistd.h> // 不加的话, syscall会报warning
#include <sys/syscall.h>
#include <time.h>
int main()
{
    struct timespec start, end;
    long diff;
    for (int i = 0; i < 100000; i++)
    {
        clock_gettime(CLOCK_MONOTONIC, &start);
        pid_t pid = syscall(SYS_getpid); // 通过 syscall 调用
        clock_gettime(CLOCK_MONOTONIC, &end);
        diff += (end.tv_sec - start.tv_sec) * 1000000000L + (end.tv_nsec - start.tv_nsec);
            // 转换为ns
    }

    diff /= 100000; // 取平均值
    printf("syscall  getpid() average time: %ld ns\n", diff);
    return 0;
}
```

Listing 3: 使用内联汇编调用 getpid

```c
#include <stdio.h>
#include <unistd.h>
#include <time.h>
#include <asm/unistd_64.h> // 包含系统调用号 #define __NR_getpid 39

// 1.系统调用号: 将系统调用的 ID (即系统调用号) 放入 rax 寄存器。
// 2.系统调用参数: 将系统调用的参数依次放入 rdi、rsi、rdx、r10、r8 和 r9 寄存器中。
// 3.触发系统调用: 使用 syscall 指令触发系统调用。
// 4.获取返回值: 系统调用完成后, 内核将系统调用的返回值放入 rax 寄存器。

int main(){
    struct timespec start, end;
    long diff = 0;
    pid_t pid;
    for (int i = 0; i < 100000; i++)
    {
        clock_gettime(CLOCK_MONOTONIC, &start);
        asm volatile(
            "syscall"           // Instruction List
            : "=a"(pid)         // Output: 将rax寄存器的值赋给变量 pid
            : "a"(__NR_getpid) // Input: 将系统调用号 __NR_getpid 赋给 rax 寄存器
            : "rcx", "r11"      // Clobber/Modify: 告诉编译器 syscall 指令会修改 rcx 和 r11
                , 以便它在生成代码时, 不会依赖于这两个寄存器在 syscall 执行前的值。
        );
        clock_gettime(CLOCK_MONOTONIC, &end);
        diff += (end.tv_sec - start.tv_sec) * 1000000000L + (end.tv_nsec - start.tv_nsec);
             // 转换为ns
    }
    diff /= 10000; // 取平均值
    printf("assembly getpid() average time: %ld ns\n", diff);

    return 0;
}
```

3.2 执行结果

图1即为上面 3 个程序的输出结果。多次运行后得出的结果趋势与图1一致, 故在此不重复放出。

```
zsh@kolp:~/VScodeproject/os_hw/hw1$ ./getpid_glibc && ./getpid_syscall && ./getpid_assembly
glibc    getpid() average time: 1020016891 ns
syscall  getpid() average time: 946809028 ns
assembly getpid() average time: 206 ns
```

图 1: getpid 3 种方式平均运行时间对比

3.3 结果分析

由1的结果，调用 getpid 的运行时间：glibc > syscall > 内联汇编。

经调研发现，glibc 库的库函数 getpid() 本质是一个封装好的系统调用，由于包含额外的检查和封装逻辑，所以它是最慢的。syscall() 同样是一个库函数，但它封装得更为简单，所以比 glibc 略快，但没有数量级上的差异。而内联汇编则直接触发系统调用，没有额外的函数调用开销，所以它是最快的，运行时间远小于另外两种方法。

4 getpid 和 open 内联汇编方法调用的对比

4.1 程序源码

同上，同样重复调用 open 十万次取平均时间。

Listing 4: 使用内联汇编调用 open

```
#include <stdio.h>
#include <fcntl.h>   // 包含 O_CREAT 和其他标志
#include <sys/stat.h> // 包含 S_IRWXU 等权限模式
#include <unistd.h>
#include <errno.h>    // 用于处理错误
#include <time.h>
#include <asm/unistd_64.h> // 包含系统调用号 #define __NR_open 2

// int open(const char *pathname, int flags, mode_t mode); // open函数原型
int main(){
    const char *file_path = "comp1.txt";
    int flags = O_CREAT ;     // 如果不存在则创建
    mode_t mode = S_IRUSR | S_IWUSR;    // 用户读写执行权限
    long ret_val; // 存储系统调用的返回值
    struct timespec start, end;
    long diff;

    for (int i = 0; i < 100000; i++)
    {
        clock_gettime(CLOCK_MONOTONIC, &start);
        asm volatile(
            "syscall"
            : "=a"(ret_val)              // 输出：将 rax 寄存器的值赋给 ret_val
            : "a"(__NR_open),            // 系统调用号 __NR_open 赋给 rax 寄存器
            "D"(file_path),             // 第一个参数pathname 放入 rdi
            "S"(flags),                 // 第二个参数flags 放入 rsi
            "d"(mode)                   // 第三个参数mode 放入 rdx
            : "rcx", "r11"
        );
        clock_gettime(CLOCK_MONOTONIC, &end);
        diff += (end.tv_sec - start.tv_sec) * 1000000000L + (end.tv_nsec - start.tv_nsec);
            // 转换为ns
```

```
32      }
33
34      diff /= 100000; // 取平均值
35      printf("assembly open()   average time: %ld ns\n", diff);
36      return 0;
37  }
```

4.2 执行结果

图2即为内联汇编实现 getpid 和 open 的运行时间对比。

```
zsh@kolp:~/VScodeproject/os_hw/hw1$ ./getpid_assembly && ./open_assembly
assembly getpid() average time: 188 ns
assembly open()   average time: 1407298159 ns
```

图 2: getpid 和 open 平均运行时间对比

4.3 结果分析

图2的结果显示，getpid 的时间远小于 open 的时间。

getpid 的开销包括用户态到内核态的切换、读取进程 PID 字段、内核态到用户态的切换。是纯 CPU 操作。

open 的开销包括用户态到内核态的切换、路径解析和查找、权限检查、文件描述符分配、更新内核数据结构、磁盘 I/O、内核态到用户态的切换。open 所完成的任务比 getpid 复杂得多，且每一个操作的耗时都更多，尤其是如果文件不在缓存中时，磁盘 I/O 将耗费巨量的时间。由此便解释了 open 操作时间高出 getpid 几个数量级的原因