第一次作业

寇逸欣 2023K8009922004

1 实验环境

本实验在 WSL2 环境下进行，操作系统版本如下：

```
Linux LAPTOP-C42RC799 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
```

实验所用的硬件为一台具有 Intel Core i9 14900HX 处理器和 32GB 内存的笔记本电脑。

2 代码实现

本次实验的代码实现包括三种不同方式的系统调用：glibc 封装的系统调用、直接使用 syscall 指令的系统调用以及使用内联汇编实现的系统调用。代码文件分别命名为 `open_glibc.c`、`open_syscall.c`、`open_asm.c` 和 `getpid_glibc.c`、`getpid_syscall.c`、`getpid_asm.c`。每个文件中都包含了相应的系统调用实现以及测试代码。

其中计时函数选择了 `clock_gettime`，而不是 `gettimeofday`，因为前者提供了更高的精度（纳秒级）和更稳定的时间测量，适合用于性能测试。而后者仅提供微秒级精度，在高精度需求场景下可能不够准确。除此之外，`gettimeofday` 还可能受到系统时间调整的影响，导致测量结果不准确。

3 实验结果

实验结果如下表所示，表中列出了每种系统调用实现的平均执行时间（单位：纳秒）。每个测试用例均运行了 1000 次以确保结果的稳定性。实验对每一项的测试设计了 100 次预热，以减少缓存和其他系统状态对测量结果的影响。

函数/平均时间 (ns)	test1	test2	test3
open_glibc	1965	1974	2020
open_syscall	2002	1974	2049
open_asm	1185	1265	1261
getpid_glibc	1293	1301	1279
getpid_syscall	251	248	255
getpid_asm	198	200	193

表 1: 不同系统调用实现的平均执行时间（单位：纳秒）

4 结果分析

4.1 针对同一个系统调用，记录和对比三种方法的运行时间，并分析时间差异原因

对于 open 系统调用，glibc 封装版本和直接使用 syscall 指令版本的运行时间相差不大，均在 2000 纳秒左右。这是因为 glibc 封装的 open 函数本质上也是通过 syscall 指令实现的，因此性能差异较小。此外，open 是一个复杂的系统调用，涉及参数准备、路径解析、权限检查等多个步骤，glibc 的额外封装开销在整体流程中占比很小，所以两者时间差异不明显。然而，使用内联汇编实现的 open 系统调用显著更快，平均时间约为 1200 纳秒。这可能是因为内联汇编能够更直接地控制寄存器和指令，从而减少函数调用开销和上下文切换时间。

对于 getpid 系统调用，glibc 封装版本的运行时间约为 1300 纳秒，而直接使用 syscall 指令版本显著更快，平均时间约为 250 纳秒。使用内联汇编实现的 getpid 系统调用进一步提升了性能，平均时间约为 200 纳秒。这表明，对于简单的系统调用，直接使用 syscall 指令和内联汇编可以显著减少函数调用开销，从而提高性能。原因在于 getpid 作为一个极其简单的系统调用，glibc 封装虽然只是简单调用 syscall，但函数封装、参数传递和返回值处理等额外开销在总执行时间中占比较大，因此直接 syscall 和内联汇编版本明显更快。

4.2 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。

在使用内联汇编实现系统调用时，getpid 和 open 这两个系统调用的运行时间存在显著差异。具体来说，getpid 的平均执行时间约为 200 纳秒，而 open 的平均执行时间约为 1200 纳秒。这种差异主要源于两个系统调用的复杂性和所需的资源。getpid 是一个非常简单的系统调用，它只需要返回当前进程的 ID，涉及的操作非常少，因此执行时间较短。而 open 则是一个相对复杂的系统调用，它需要处理文件路径解析、权限检查以及文件描述符的分配等多个步骤，这些操作都需要更多的时间和资源。因此，open 系统调用的执行时间显著高于 getpid。