目录

[引言 3](#_Toc12870)

[1 文件操纵类系统调用 3](#_Toc3622)

[1.1 文件系统与系统调用的文件操作 3](#_Toc17903)

[1.2 基于Linux对文件系统描述 4](#_Toc9610)

[1.3 文件操作类系统调用的常见操作 6](#_Toc30278)

[2 不同操作系统的实现比较 8](#_Toc26650)

[2.1 Linux文件系统调用实现差异 8](#_Toc3700)

[2.2 不同操作系统文件系统调用实现方式比较 9](#_Toc28697)

[2.2.1系统调用接口 9](#_Toc13440)

[2.2.2文件系统类型 10](#_Toc13559)

[2.2.3文件操作的抽象 10](#_Toc28031)

[2.2.4错误处理方式 11](#_Toc27986)

[2.2.5性能与并发支持 11](#_Toc11270)

[2.3 不同操作系统文件系统模块的差异描述 11](#_Toc17941)

[2.3.1结构差异 11](#_Toc854)

[2.3.2算法差异 12](#_Toc16244)

[2.3.3数据结构差异 12](#_Toc11496)

[3 实验代码 14](#_Toc18129)

[3.1 文件操作（c标准库） 14](#_Toc15160)

[3.2 文件操作（系统调用） 15](#_Toc7379)

[3.3在Linux和windows上测试代码 17](#_Toc27646)

[4 实验结果分析 18](#_Toc20794)

[5 结论和总结 19](#_Toc23872)

[6 参考文献 20](#_Toc10795)

引言

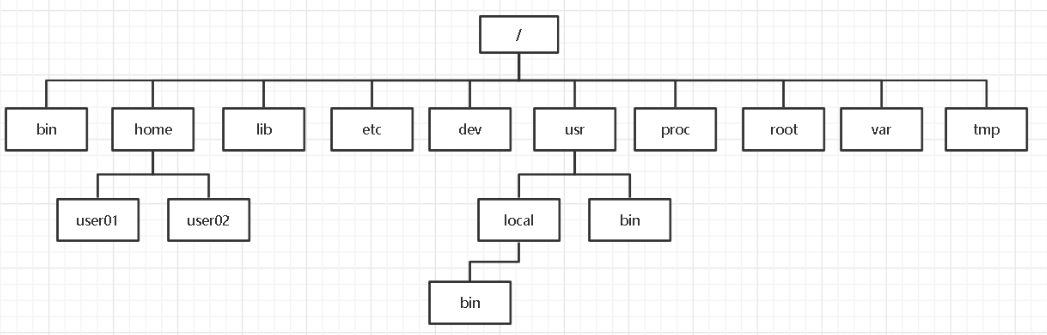
操作系统（OS）作为计算机系统的重要组成部分，负责管理硬件资源和软件应用。操作系统调用模块可以为用户空间提供访问硬件资源的**统一接口，**系统调用可以**对系统进行保护**，保证系统的稳定和安全性。用户程序利用系统调用，向操作系统发出服务请求；操作系统通过系统调用为运行于其上的应用程序提供服务。操作系统调用模块是程序与内核之间的接口，文件系统调用是其中重要的一部分。它负责处理用户与文件系统之间的交互，包括文件的创建、读取、写入、删除等操作。它负责处理用户与文件系统之间的交互，包括文件的创建、读取、写入、删除等操作。通过文件系统调用，应用程序可以访问存储在硬盘、SSD等设备上的文件。在本报告中，我们将深入探讨文件系统调用模块，比较不同操作系统下的实现，并对实验结果进行分析，从而总结其在操作系统设计和性能上的影响。

1 文件操纵类系统调用

1.1 文件系统与系统调用的文件操作

在Linux系统中有一个重要的概念：一切皆文件，它把一切资源都看作是文件，包括硬件设备，通常称为设备文件。文件系统主要分为：**存储设备文件系统**、**伪文件系统**、**虚拟文件系统。**

下图是linux文件系统的组成简化图：



文件操作的系统调用模块是操作系统的重要组成部分，它提供的一组接口，让应用程序能够有效地与文件系统交互。尽管不同操作系统的实现和调用方式各有不同，但其核心功能和操作模式大致相同。

在实现文件操纵类系统调用的设计时，操作系统开发者通常会考虑以下几个因素：

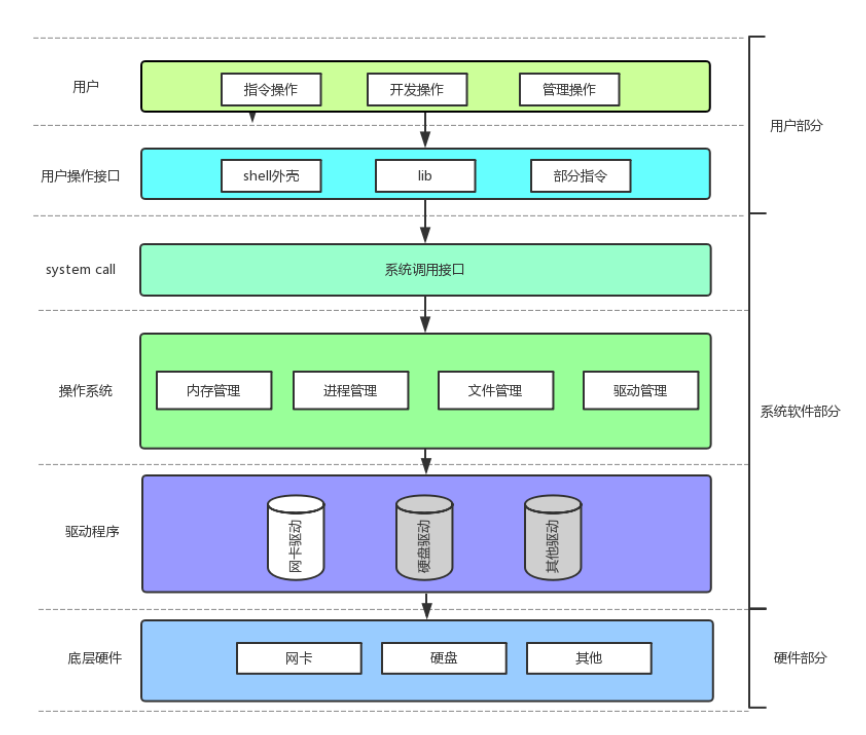
****性能****:系统调用应该具有效率，最小化用户空间和内核空间切换的开销。使用适当的缓冲策略（例如 I/O 缓存）可以提高文件读写性能。

****锁机制****:文件访问的并发控制十分重要，系统调用需要支持锁机制以防止数据竞争和文件损坏。

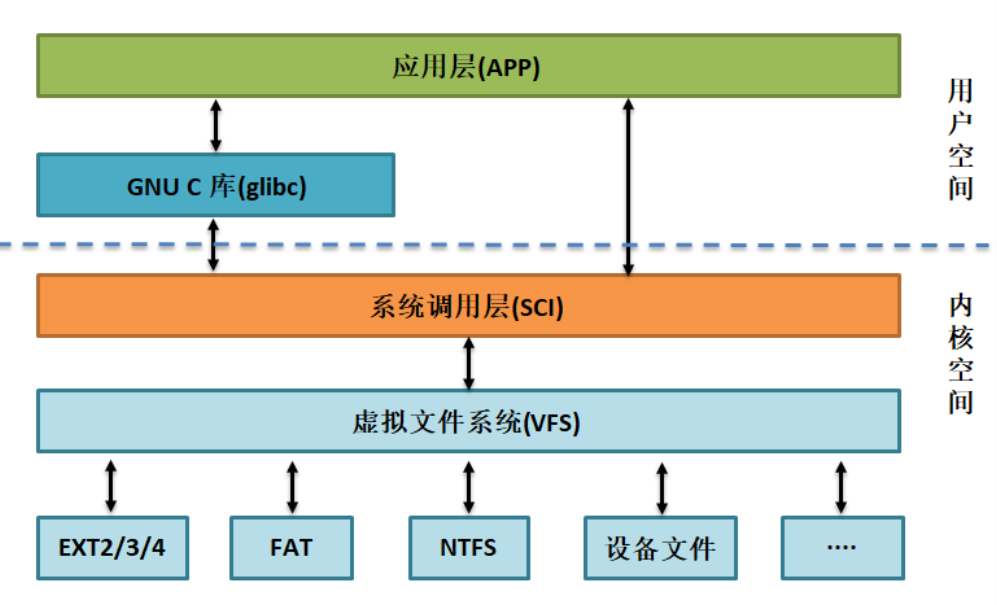
****错误处理与安全性****:系统调用需要对错误进行详细的处理，将错误信息反馈给用户程序，确保文件操作的安全性和可靠性。权限管理应确保只有拥有相应权限的用户才能执行特定文件操作。

****扩展性****:文件操作接口应支持扩展，允许未来的功能扩展（例如网络文件系统、虚拟文件系统等）。

1.2 基于Linux对文件系统描述



在Linux下，一个与文件操作相关的应用程序结构如下图所示（虚拟文件系统）。



上图解构如下：

1.应用层指用户编写的程序，如我们的hello.c。

2.GNU C库（glibc）即C语言标准库，例如在编译器章节介绍的libc.so.6文件，它包含了printf、malloc， 以及本章使用的fopen、fread、fwrite等文件操作函数。

3.用户程序和glibc库都是属于用户空间的，本质都是用户程序。

4.应用层的程序和glibc可能会调用到“系统调用层（SCI）”的函数， 这些函数是Linux内核对外提供的函数接口，用户通过这些函数向系统申请操作。 例如，C库的printf函数使用了系统的vsprintf和write函数， C库的fopen、fread、fwrite分别调用了系统的open、read、write函数， 具体可以阅读glibc的源码了解。

5.由于文件系统种类非常多，跟文件操作相关的open、read、write等函数经过虚拟文件系统层， 再访问具体的文件系统。

总的来说，为了使不同的文件系统共存，Linux内核在用户层与具体文件系统之前增加了虚拟文件系统中间层， 它对复杂的系统进行抽象化，对用户提供了统一的文件操作接口。 无论是ext2/3/4、FAT32、NTFS存储的文件，还是/proc、/sys提供的信息还是硬件设备， 无论内容是在本地还是网络上，都使用一样的open、read、write来访问， 使得“一切皆文件”的理念被实现，这也正是软件中间层的魅力。

1.3 文件操作类系统调用的常见操作

对文件进行操纵的系统调用数量较多，有创建文件、删除文件、打开文件、关闭文件、读文件、写文件、建立目录、移动文件的读/写指针、改变文件的属性等。

这些系统调用在操作系统内部由内核提供支持，用户程序通过这些调用可以高效地进行文件操作，而操作系统则负责管理文件系统的底层细节，如缓存、权限检查和磁盘I/O等。通过系统调用，程序员可以以较高层次的抽象方式与文件系统进行交互，完成文件的各种操作。

以下是文件操作模块常见的系统调用及其详细解释：

（1）创建和删除文件

当用户需要在系统中存放程序或数据时，可利用创建文件的系统调用 creat，由系统根据用户提供的文件名和存取方式来创建一个新文件；当用户已不再需要某文件时，可利用删除文件的系统调用 unlink 将指名文件删除。它接受文件路径作为参数，从文件系统中移除指定文件。

（2）打开和关闭文件

用户在第一次访问某个已存在的文件之前，应先利用打开文件的系统调用 open，将指名文件打开，即系统将在用户(程序)与该文件之间建立一条快捷通路。它接受文件路径及打开方式（读取、写入、追加等）作为参数。在文件被打开后，系统将给用户返回一个该文件的句柄或描述符；当用户不再访问某文件时，又可利用关闭文件的系统调用 close，将此文件关闭，释放与之相关的资源，即断开该用户程序与该文件之间的快捷通路。它只需要文件描述符作为参数，返回成功或失败的状态。

（3）读和写文件

用户可利用读系统调用 read，从已打开的文件中读出给定数目的字符，它接受文件描述符、存储数据的缓冲区和要读取的字节数作为参数，并返回实际读取的字节数送至指定的缓冲区中；同样，用户也可利用写系统调用 write，从指定的缓冲区中将给定数目的字符写入指定文件中，它也接受文件描述符、要写入的数据及字节数作为参数，返回实际写入的字节数。read 和 write 两个系统调用是文件操纵类系统调用中使用最频繁的。

1. 文件状态操作

用户可利用stat获取文件的状态信息，例如文件大小、权限、创建时间等。它接受文件路径和一个结构体指针来存储文件信息，返回成功或失败状态。

1. 文件权限操作

用户可利用chmod修改文件的访问权限。它接受文件路径及新的权限设置作为参数，返回成功或失败状态。

1. 文件重命名

利用rename用于修改文件的名称或路径，并且接受旧文件名和新文件名作为参数。

1. 文件定位

在已打开的文件中移动读/写指针的位置，可使用lseek，它支持定位到特定的字节位置，其参数包括文件描述符、偏移量及起始位置（如文件开头、当前位置等）。

1. 目录操作

用户可利用mkdir创建目录，并且利用rmdir可删除目录，其操作类似于文件管理。

1. 符号链接

用户可用symlink创建符号链接，通过链接名和目标文件路径来建立一个指向目标的快捷方式。

2 不同操作系统的实现比较

提到文件系统时， 我们首先想到的通常是Windows下的FAT32、NTFS、exFAT以及Linux下常用的ext2、ext3和ext4的类型格式。 这些文件系统都是为了解决如何高效管理存储器空间的问题而诞生的。

## 2.1 Linux文件系统调用实现差异

在Linux中，文件系统调用是通过一系列标准库函数实现的。这些系统调用通过提供高效的数据结构（如inode和dentry）来管理文件。Linux文件系统调用在早期版本（如1.x和2.x）和后续版本（如3.0及以上）中存在显著差异：

早期版本（如1.x和2.x）：这些版本的Linux实现了基本的文件系统调用，比如open()、read()、write()、close()等。文件系统调用的基本功能是相对简单的（比如系统调用早期版本（如1.x和2.x）：的错误处理主要使用errno进行错误码返回），主要依赖于UNIX风格的文件处理模型。主要支持ext2文件系统，功能有限。进一步的文件系统，诸如FAT和NTFS的支持较少。

比如2.4内核版本：文件系统以vfs（虚拟文件系统）为核心，提供了一致的文件操作接口。在这个版本中，文件系统调用的性能优化主要集中在减少上下文切换和系统调用开销上。文件系统缓存（page cache）管理相对简单，主要依靠LRU（最近最少使用）算法进行页面置换。

在**2.6内核及更高版本：**引入了更丰富的系统调用，支持更复杂的操作，如fstat()、fchmod()，并且实现了异步I/O、非阻塞I/O等功能，增强了对并发操作的支持、使得文件系统调用的性能大大提升。比如，使用了更先进的缓存机制来减少磁盘访问次数，从而提高I/O性能。

在后续版本（如3.0及以上）：支持更多的文件系统，如ext3（有日志功能），ext4（支持大文件和更高的性能），XFS（极高的并发性能），Btrfs（文件系统的快照和子卷管理）。不同文件系统对系统调用的支持和优化程度不同。对错误处理进行了改进，系统调用通常包含更详尽的错误信息，并且引入了更复杂的返回机制，如-1返回及errno的结合使用。

**更高版本的linux（如4.x内核版本）**：引入了fuse（文件系统用户空间），允许用户空间程序实现文件系统操作。文件系统调用在性能上有了更多的优化，例如引入了ioremap、writeback机制等，提高了IO效率。缓存管理更加复杂，引入了shmem（共享内存）和tmpfs，支持更高效的内存使用。

在较新版本的Linux中，系统调用的安全性得到了增强，支持更细粒度的权限控制机制，例如SELinux和AppArmor，这些机制通过安全上下文对文件访问进行控制。也支持文件权限扩展，例如setfacl()和getfacl()允许更复杂的访问控制列表（ACL）来定义文件权限。以及新的系统调用被不断加入，如clone()调用被用来处理线程创建，dup2()用于文件描述符的重定向等。

Linux操作系统的文件系统调用随着版本的演变，经历了接口的丰富性、支持的文件系统类型的增多、性能的优化、安全性的增强等多方面的改进。这些进步不仅提高了文件操作的效率和灵活性，还为开发者提供了更多的选择和控制，使得Linux系统在服务器、嵌入式设备和个人计算机等多种场景中都能得到广泛应用。

## 2.2 不同操作系统文件系统调用实现方式比较

选择Linux、Windows和macOS操作系统以从多个方面进行分析，包括系统调用接口、文件系统类型、文件操作的抽象、错误处理方式等进行比较。下面是这三种操作系统在文件操作模块实现方面的比较：

2.2.1系统调用接口

**Linux：**使用POSIX 标准，提供了一系列统一的系统调用，如 open(), read(), write(), close(), unlink()等。Linux 的系统调用通常通过 syscall 接口实现。支持标准的文件操作调用，且提供了对各种文件系统（如 ext4、XFS 和 Btrfs 等）的不同实现。

**Windows：**使用 Windows API 提供文件操作功能。常用的文件操作函数包括 CreateFile(), ReadFile(), WriteFile(), CloseHandle(), DeleteFile()等。 Windows 的文件机制围绕文件对象设计，比 POSIX 模型更复杂，其 API 抽象程度较高，支持多种文件系统（如 NTFS、FAT32 等）。

**macOS：**基于 BSD UNIX，使用类似 Linux 的系统调用接口，常用的包括 open(), read(), write(), close(), unlink()等。macOS 提供 POSIX 标准的支持，但也有许多 Apple 自家的扩展和 API（如 Cocoa 框架），用于图形界面文件操作。

2.2.2文件系统类型

**Linux：**支持多种文件系统，包括 ext4、XFS、Btrfs、FAT、NTFS 等。不仅支持传统的 UNIX 文件系统，还支持各种现代文件系统，其文件系统驱动紧密集成在内核中。

**Windows：**主要使用 NTFS 和 FAT 文件系统。NTFS 的许多功能（如访问控制、加密、压缩等）通过 Windows API 进行访问。

**macOS：**主要使用 APFS（Apple File System），在早期使用 HFS+。APFS 针对 SSD 进行了优化，支持快照、加密和空间分配等特性。

2.2.3文件操作的抽象

**Linux**：文件被视为字节流，所有文件操作（如读取、写入等）都是通过文件描述符来进行的。对文件的操作都是基于文件描述符，这与 Unix 的哲学密切相关。

**Windows**：使用句柄作为文件对象的标识。所有对文件的操作都需要通过句柄来完成，这提供了一种更高级的抽象方式，但复杂性相对较高。

**macOS：**同样使用文件描述符进行操作，和 Linux 类似，但是在应用层支持更多的高层次 API，允许更丰富的操作。

2.2.4错误处理方式

**Linux：**通过全局变量 errno 来进行错误报告。系统调用失败时返回 -1，errno 被设置为具体的错误码。

**Windows**：通过返回值和 GetLastError() 函数进行错误处理。如果函数调用失败，通常会返回 INVALID\_HANDLE\_VALUE 或 FALSE，随后可以调用 GetLastError() 获取详细错误信息。

**macOS：**与 Linux 相似，通过 errno 来进行错误报告。在许多 Cocoa API 调用中，返回值会包含错误信息，但 POSIX 系统调用仍然使用 errno。

2.2.5性能与并发支持

**Linux：**通过异步I/O、内存映射等机制，Linux 提供高效的文件I/O处理。

**Windows：**Windows 文件操作同样支持异步模式，使用 OVERLAPPED 结构体可以实现非阻塞I/O。

**macOS：**macOS 也支持异步文件I/O，通过 Cocoa 和 POSIX API 提供不同的接口。

## 2.3 不同操作系统文件系统模块的差异描述

在对比Linux、Windows和macOS的系统调用文件操作模块时，从结构、算法和数据结构几个方面深入分析它们之间的差异：

### 2.3.1结构差异

****Linux****：Linux 的文件操作结构基于 VFS（Virtual File System）层，它提供了一个抽象层，使得不同的文件系统可以以一致的方式被访问。文件描述符是核心的抽象，所有文件操作都通过文件描述符接口完成。内核通过系统调用管理文件的打开、关闭、读写等操作。VFS 使用多种结构，如 file 结构（用于描述打开的文件）、inode 结构（表示文件的元数据）和 dentry 结构（用于目录项）。

****Windows****：Windows 使用 I/O Manager 架构，以设备驱动程序和文件系统驱动程序构建。Windows 中的文件操作通过句柄来进行，句柄抽象了具体的文件对象。虽然Windows也有VFS的概念，但它更多依赖于“对象”的模型，每个文件、设备和其它资源都被视为对象，有统一的对象管理机制。

**MacOS**:macOS 结合了 BSD 和苹果的技术，采用 HFS+ 或 APFS 文件系统，并通过 VFS 层提供一致的文件操作接口。Path 和文件描述符相结合，使得文件访问更加直观。macOS 的应用程序接口包括文件管理，以及 Cocoa 框架中面向对象的文件管理机制，使得开发者可以方便地与文件系统交互。

### 2.3.2算法差异

****Linux****：文件操作依赖于多种 I/O 调度算法，如 Completely Fair Queueing (CFQ) 和 Deadline 调度器。内核会优化读写操作以减少磁盘I/O的响应时间。使用页缓存和超级块缓存来减少磁盘访问次数，采用 LRU（Least Recently Used）策略进行数据缓存的管理。

****Windows****：采用了延迟写入和聚合 I/O 操作，以提高速度和效率。此外，使用不同的 I/O 调度策略来支持多种类型的负载（例如，顺序 vs 随机 I/O）。Windows I/O 操作通常与线程池紧密集成，以支持高效的并发I/O操作。

**MacOS：**采用了类似于 Unix 的I/O调度算法，主要优化针对 APFS 文件系统，提供了高效的快照和克隆支持。macOS 在处理文件时使用内存映射文件和缓存策略，以减少对物理磁盘的需求，并改善读取性能。

2.3.3数据结构差异

**Linux**：

核心数据结构；

file 结构: 表示一个打开的文件，包含指向文件系统的指针、文件位置偏移量等信息。

inode 结构: 存储文件的元数据（如权限、大小、时间戳等）。

dentry 结构: 用于加速文件路径查找的目录项。

文件系统实装：各种文件系统（如 ext4、XFS 等）都有其自有的实现细节，通常包含成百上千的行代码和复杂的数据结构。

**Windows**：

核心数据结构：

文件对象: 包含文件句柄、访问控制信息、缓存策略等，用于简化文件操作。

DEVICE\_OBJECT 和 FILE\_OBJECT: 用于描述文件和设备的结构体，提供对文件的访问控制、缓冲和 I/O 操作。数据交互：通过 I/O 请求包（IRP）管理文件与设备之间的交互。

**MacOS：**

核心数据结构：

vnode 结构: 表示文件系统中的文件或目录。

Vfs: 用于表示在 BSD 核心中各文件系统之间的通用接口。

struct filedesc: 用来描述文件描述符表，管理打开的文件。

APFS数据结构：

APFS 引入了一些新结构来支持快照和高效的空间管理，包括文件的事务性结构。

总结如下：

****Linux****的文件操作模块结构基于 VFS，在算法和数据结构方面使用了多样化的技术以优化性能，为多种文件系统提供支持。

****Windows****提供一个高度抽象的文件对象，通过对象模型处理文件操作，而其 I/O 管理系统则用句柄来代替传统的文件描述符。

****macOS****在设计上融合了 UNIX 的哲学和苹果自家技术，提供了灵活的接口和优雅的设计，特别是在用户界面和底层存储方面的兼容性上。

这些差异反映了各个操作系统在处理文件操作时的设计哲学和使用场景，影响到了开发者的选择和编程体验。

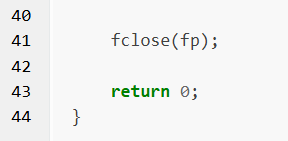
3 实验代码

3.1 文件操作（c标准库）

本小节讲解使用通用的C标准库接口访问文件，标准库实际是对系统调用再次进行了封装。 使用C标准库编写的代码，能方便地在不同的系统上移植。

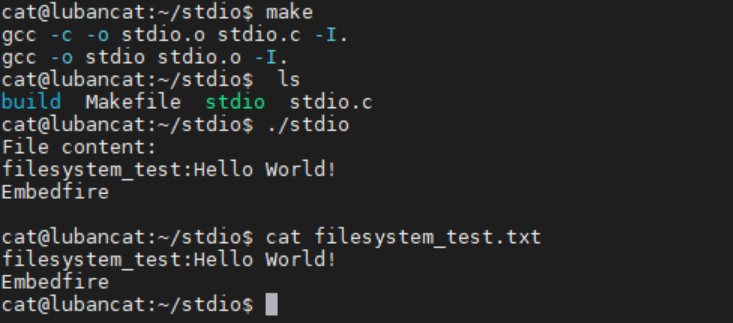
例如Windows系统打开文件操作的系统API为OpenFile，Linux则为open， C标准库都把它们封装为fopen，Windows下的C库会通过fopen调用OpenFile函数实现操作， 而Linux下则通过glibc调用open打开文件。用户代码如果使用fopen， 那么只要根据不同的系统重新编译程序即可，而不需要修改对应的代码。





该代码使用fopen创建文件、使用fwrite写入内容， 使用fflush确保缓冲区的内容写到文件， 然后使用fseek重置文件位置指针，使用fread把文件的内容读出，最后调用fclose关闭文件。其中的fopen函数调用时使用了参数“w+”，表示每次都创建新的空文件，且带上读权限打开， 函数调用后得到文件描述符fp，在它后面的fwrite、fread、fflush等函数都是通过这个fp文件描述符访问该文件的。

编译测试结果如下：



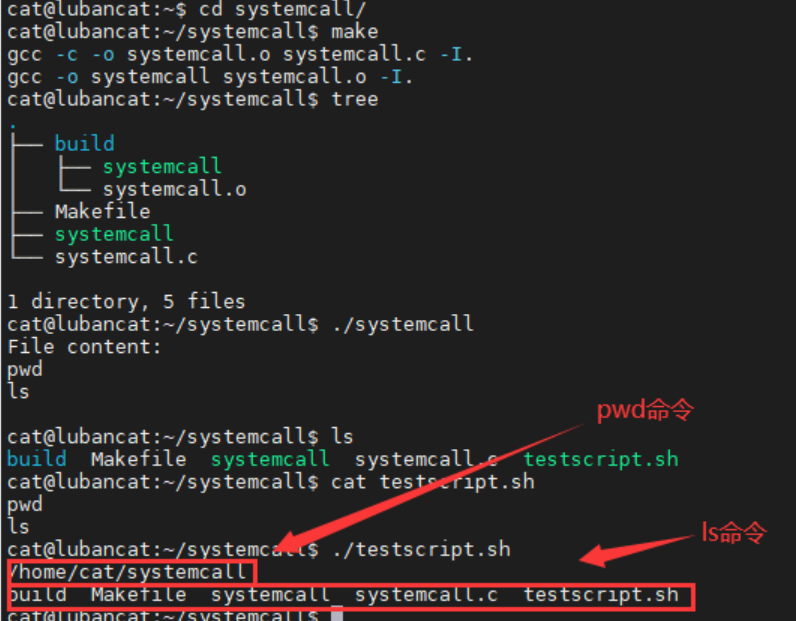
3.2 文件操作（系统调用）

Linux提供的文件操作系统调用常用的有open、write、read、lseek、close等。



编译测试结果如下：

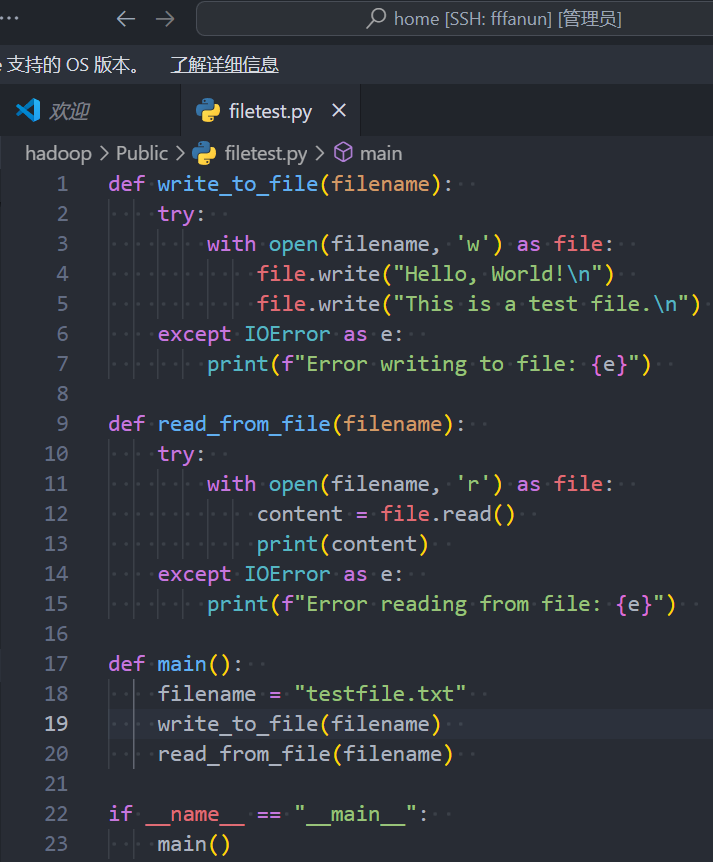
systemcall程序执行后，它创建的testscript.sh文件带有可执行权限， 运行./testscript.sh可执行该脚本。



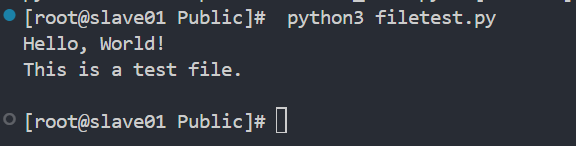
**C标准库**和**系统调用**的**区别**就在于库函数带**缓冲区。**

3.3 在Linux和windows上测试代码

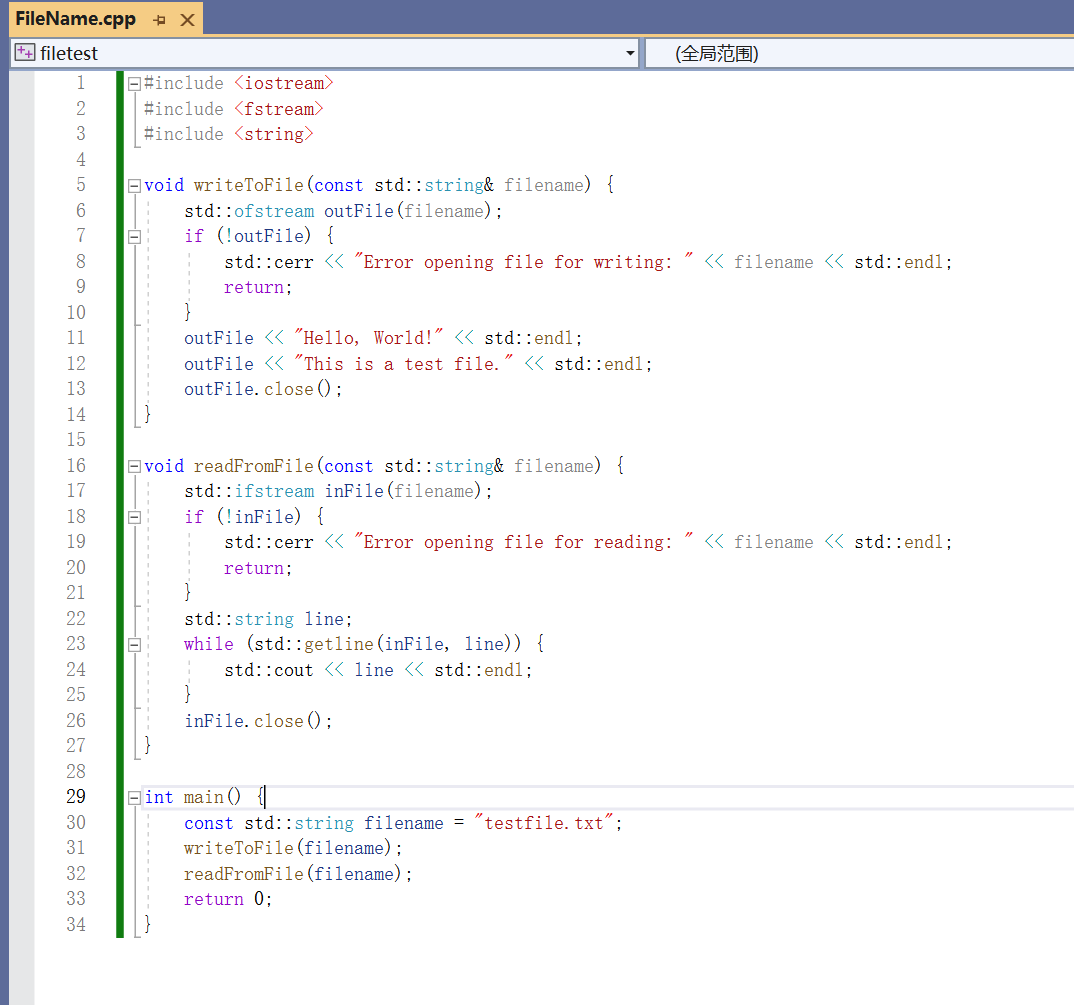
Linux下使用python语言测试：



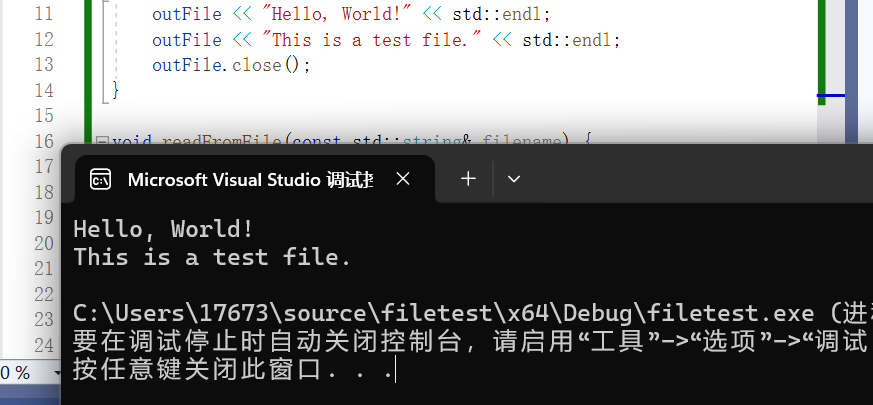
Linux运行结果：



Windows下使用c++测试代码：

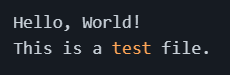


Windows运行结果：



4 实验结果分析

在两个平台上都会成功输出：



Linux 文件系统更注重权限管理，运行代码的用户需要有权限创建和读写该文件，可能会遇到权限问题，例如当前用户没有写入目录的权限；而在 Windows 中，权限管理通常更为宽松。也需要考虑两者文件路径的书写差异（如使用反斜杠\或斜杠/）。

在处理大文件或复杂文件操作时，文件系统的调度和缓存机制可能会影响性能。Linux 通常使用 ext4 文件系统，这对于连续读写有更好的性能优化，而 Windows 使用 NTFS 文件系统，侧重于对文件权限和安全性的管理。尤其在并发操作和I/O密集型任务中。Linux 的 I/O 调度器通常对文件操作进行更好的优化，而 Windows 的 I/O 具有较高的灵活性和调度能力。

不同操作系统的文件系统调用实现直接影响应用程序的性能，Linux适合对性能要求高的服务器，而Windows在用户友好的桌面环境中表现优秀。

5 结论和总结

在进行文件操作时，不同操作系统（如 Linux 和 Windows）所实现的系统调用和模块存在一些显著的差异。这些差异不仅反映在编程接口上，也体现在底层处理、性能和安全性等方面。

Linux系统的设计更加开放和模块化，强调性能和资源的高效利用，适合服务器和开发环境。大多数文件操作使用 POSIX 标准，通过系统调用可以实现对文件的打开、读取、写入和关闭等操作。Linux 的系统调用更为精简且直接，提供了较高的灵活性和控制能力。使用传统的 Unix 权限模型（拥有者、组、其他用户），通过 chmod、chown 等命令进行管理，灵活性较强。常见文件系统如 ext4、高性能，支持更大的文件和高级功能（如快照和文件系统检查）。它更注重文件的性能，在 I/O 性能优化方面相对成熟，能够高效处理大量文件操作，很好地利用缓存和内存管理。Linux使用错误码并提供系统错误信息，容易实现自动化错误处理。

Windows的设计更侧重于用户体验和易用性，注重提供完整的功能集和安全性。 使用 WinAPI 进行文件操作。WinAPI 提供了多种功能和灵活的权限管理，支持丰富的文件操作，但其接口相对复杂，对于新手来说，学习曲线较为陡峭。采用更复杂的 ACL（访问控制列表）模型，能够精细到每个用户和组的权限控制，适合需要高安全性的场合。并且使用 NTFS，具有复杂的权限控制和加密功能，可以更好地集成加密和访问控制功能，适合需要高安全性的企业环境。Windows错误处理则更依赖于状态返回和异常处理机制，可能导致用户在处理错误时需要了解更复杂的错误返回格式。

综上，Linux 在处理大规模并发任务时往往表现优越，适合需要高负载和高可用性的应用场景，开发者可以更容易地调整和定制底层操作系统功能，以满足特定需求，Linux 的开源特性使得社区能够快速修复和改进，尤其是在安全性和功能拓展方面。而Windows主要在在桌面环境和企业应用中占据主导地位。

6 参考文献

[1]《文件操作与系统调用》https://doc.embedfire.com/linux/h618/linux\_base/zh/latest/linux\_app/filesystem\_system\_call/filesystem\_system\_call.html#

[2]李肇中.基于外内核操作系统的文件系统研究[D].兰州大学,2016.

[3]葛仁北. 系统调用与操作系统安全[J]. 计算机工程与应用,2002,38(19):97-99,128. DOI:10.3321/j.issn:1002-8331.2002.19.034.

[4]范恩魁,陈亚军. 基于Linux操作系统的系统调用分析[J]. 重庆科技学院学报（自然科学版）,2008,10(6):124-126. DOI:10.3969/j.issn.1673-1980.2008.06.037.

[5]张岩. 基于Linux内核及系统调用的文件系统管理的研究与实现[D]. 北京:北京邮电大学,2007.

[6]赵斌,刘长起,戴英侠.Windows操作系统的文件操作监控技术[J].计算机工程与应用,2004,(31):131-133+168.