

1、尝试结合开源代码来分析这些子系统的关 键工作原理（3~5个关键点）

我选择使用WSL1做为实例进行分析

电脑Windows版本为Win11.




1.系统调用的转译




核心原理：WSL1 并没有 Linux 内核，而是引入微软自己开发的一个翻译层将 Linux 系统调用翻译成等效的 Windows NT 内核调用。

关键步骤：WSL先拦截 Linux 应用程序的系统调用，然后将这些系统调用映射到 Windows 系统调用，最后将返回相应的结果给 Linux 应用程序。

代码实现：我找到了**lxcore.sys**文件，他是**Linux 内核** 和 **Windows 内核**之间的桥梁。可以完成这个任务。

(但由于它被视为 Microsoft 的专有知识产权，所以无法查看源代码)

lxcore.sys				
名称	路径	大小	修改	
 lxcore.sys	C:\Windows\WinSxS\amd64_microsoft-wi...	1,098 KB	202	
 lxcore.sys	C:\Windows\WinSxS\amd64_microsoft-wi...	7 KB	202	
 lxcore.sys	C:\Windows\WinSxS\amd64_microsoft-wi...	15 KB	202	

名称	修改日期	类型	大小	
 f	2024/1/21 20:15	文件夹		
 r	2024/1/21 20:15	文件夹		
 lxcore.sys	2024/1/21 20:15	系统文件	1,098 KB	





2.文件系统兼容性

主要功能：WSL 提供了对 Linux 文件系统的支持，包括对 ext4 文件系统的读取和写入。

- Linux 文件系统的路径与 Windows 的 NTFS 文件系统路径进行映射。
- 提供 `/mnt/c` 等挂载点，让用户可以访问 Windows 文件系统的内容。
- 实现文件系统的权限和属性兼容，以确保 Linux 应用程序正常运行。

代码实现：

WSL 通过 `lxss` `lxcore.sys` 等文件系统来管理 Linux 文件：

 lxss	C:\Program Files\WindowsApps\Microsof...	2
 lxss	C:\Program Files\WindowsApps\Microsof...	2
 lxss	C:\Windows\System32	2
 lxss	C:\Windows\SysWOW64	2

3.驱动控制

主要原理:

WSL 1: 通过 `lxcore.sys` 驱动程序拦截和处理系统调用，模拟 Linux 的设备文件和驱动程序接口。

WSL 2: 通过 Hyper-V 虚拟化运行一个完整的 Linux 内核，支持标准的 Linux 内核模块和驱动程序，并通过 VMBus 通道与宿主机通信，实现设备访问和资源共享。

下面是我自己之前尝试完成的一个项目的一个小的技术点:

以使用WSL连接USB为例 展示WSL的对驱动的控制

工作原理: `usbipd` 是用于将 Windows 本地连接的 USB 设备共享给其他机器的开源项目，包括 Hyper-V 虚拟机和 WSL 2。

利用Linux 内核中的USB IP协议与Windows服务器完成交互

在电脑端插入ST-LINK,显示为:

1. 2-1 - STM32 STLink 设备，状态为 Not shared (未共享)。
2. 2-3 - Realtek 无线蓝牙适配器，状态为 Not shared (未共享)。
3. 2-4 - ELAN WBF 指纹传感器，状态为 Not shared (未共享)。
4. 3-1 - HP 5MP 摄像头，HP IR 摄像头，摄像头 DFU 设备，状态为 Not shared (未共享)。

输入指令

```
usbipd bind --busid ? -? (这里问号是驱动对应的数字如2-1, 下同)
usbipd attach --wsl --busid ? -?
```

在WSL得到结果如下

```
Bus 002 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
Bus 001 Device 002: ID 0483:3748 STMicroelectronics ST-LINK/V2
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
```

在windows下，显示STLink 状态为Shared.

BUSID	VID:PID	DEVICE	STATE
3-1	0483:3748	STM32 STLink	Shared
3-3	0bda:c85c	Realtek Wireless Bluetooth Adapter	Not shared
3-4	04f3:0c7e	ELAN WBF Fingerprint Sensor	Not shared
4-1	0408:545f	HP 5MP Camera, HP IR Camera, Camera DFU Device	Not shared

在WSL下驱动蓝牙模块!

4.网络支持

核心原理: WSL 在 Windows 网络堆栈之上运行，并使用 Windows 提供的网络接口和适配器来进行网络通信。

关键步骤:

通过 `lxcore.sys` 驱动拦截和处理 Linux 网络系统调用，然后将 Linux 网络请求转发到 Windows 网络协议栈进行处理。

同一IP实验展示:

```
以太网适配器 vEthernet (WSL):

连接特定的 DNS 后缀 . . . . . :
本地链接 IPv6 地址 . . . . . : fe80::d7e5:d66d:39b1:8a22%44
IPv4 地址 . . . . . : 192.168.176.1
子网掩码 . . . . . : 255.255.240.0
默认网关 . . . . . :

yibuntu@Vigion:~$ cat /etc/resolv.conf
# This file was automatically generated by WSL. To stop automatic generation of this file, add the following entry to /etc/wsl.conf:
# [network]
# generateResolvConf = false
nameserver 192.168.176.1
```

可以发现，第二张图指向的 DNS 服务器 IP 为 192.168.176.1，正是 Windows 的 IP，说明 WSL 2 是借助 Windows 去寻找真正的 DNS 服务器的。

5.进程和线程管理

关键步骤:

- **创建进程**: 处理 `fork` 和 `execve` 系统调用，在 Windows 上创建新的进程。
- **调度和切换**: 管理 Linux 线程的调度和切换，与 Windows 调度器交互。
- **进程退出**: 处理进程退出，清理资源并通知父进程。

代码实现:

`lxcore.sys`: 是 WSL 的核心驱动程序，负责实现 Linux 系统调用，包括进程和线程管理。这个驱动程序拦截来自 Linux 用户空间的系统调用，并将其转换为 Windows 内核调用。

`init`: 这是 WSL 中的第一个用户空间进程，相当于 Linux 的 `init` 进程。它负责初始化 WSL 环境并启动用户进程。

`LxssManager`: 这是 Windows 上的一个系统服务，负责管理 WSL 实例的生命周期，包括启动和关闭 WSL 实例。

 f	2024/4/11 14:39	文件夹	
 r	2024/4/11 14:39	文件夹	
 bsdtar	2022/5/7 19:09	文件	839 KB
 init	2022/5/7 19:09	文件	1,407 KB
 LxssManager.dll	2024/4/11 14:39	应用程序扩展	1,188 KB
 LxssManagerProxyStub.dll	2024/3/13 15:23	应用程序扩展	32 KB

2 尝试结合开源代码分析总结Linux和Windows（ReactOS项目）两种系统在文件操作上的异同。

不同

a)文件操作接口

- **Linux**: 主要文件操作系统调用包括 `open()`, `read()`, `write()`, `close()`, `lseek()`, `stat()` 等。
- **Windows**: 主要文件操作系统调用包括 `CreateFile`, `ReadFile()`, `WriteFile()`, `CloseHandle()`, `SetFilePointer()`, `GetFileAttributes()` 等。ReactOS在实现这些系统调用时, 遵循Windows的API设计。

b)目录结构:

- **Linux**: 采用单一的树形目录结构, 根目录为 `/`, 所有文件和目录都在其下。
- **Windows**: 驱动器为基础的目录结构 (如 `C:\`、`D:\`), 每个驱动器有独立的目录树。

c) 命令行工具

- **Linux**: 常用的文件操作命令包括 `ls`, `cp`, `mv`, `rm`, `chmod`, `chown` 等。
- **Windows**: 常用的文件操作命令包括 `dir`, `copy`, `move`, `del`, `attrib` 等。ReactOS的命令行工具与Windows保持一致。

d)文件路径

- **Linux**: 使用正斜杠 `/` 作为路径分隔符, 根目录为 `/`。
- **Windows**: 使用反斜杠 `\` 作为路径分隔符, 驱动器号 (如 `C:`) 表示根目录。

e)权限模型

- **Linux**: 基于用户、组和其他人三种角色, 使用读 (r)、写 (w)、执行 (x) 三种权限。还支持扩展属性和访问控制列表 (ACLs)。
- **Windows**: 基于用户和组的权限模型, 允许非常具体的访问控制, 能够根据每个用户或每个组控制对资源的访问。ReactOS也实现了类似的权限管理。

f)文件系统类型

- **Linux**: 支持多种文件系统类型, 包括ext4、Btrfs、XFS、FAT、NTFS等。常用的文件系统是ext4。
- **Windows**: 主要支持NTFS和FAT32, 同时也支持exFAT。ReactOS作为开源实现, 主要关注NTFS和FAT。

相同

他们都采用分文件系统结构, 都有创建文件, 打开文件, 读取文件的基本操作, 都使用文件权限来控制对文件和目录的访问

都提供了命令行工具来进行文件操作，两者都提供了API供开发者使用，用于在程序中进行文件操作

源码分析

选择Reactors与Linux源码各一个，分别展现ReactOS在实现这些系统调用时，遵循Windows的API设计与Linux的open () 函数内部函数会调用很多系统函数以实现功能

```
//Reactos API 中CreateFile2()函数的声明
#ifndef __WINE_FILEAPI_H
#define __WINE_FILEAPI_H

#ifdef __cplusplus
extern "C" {
#endif

//定义了一个名为CREATEFILE2_EXTENDED_PARAMETERS的结构体以及它的指针类型。
typedef struct _CREATEFILE2_EXTENDED_PARAMETERS {
    DWORD dwSize;
    DWORD dwFileAttributes;
    DWORD dwFileFlags;
    DWORD dwSecurityQosFlags;
    LPSECURITY_ATTRIBUTES lpSecurityAttributes;
    HANDLE hTemplateFile;
} CREATEFILE2_EXTENDED_PARAMETERS, *PCREATEFILE2_EXTENDED_PARAMETERS,
*LPCREATEFILE2_EXTENDED_PARAMETERS;

//这一部分声明了CreateFile2函数，它是Windows API中的一个函数，用于创建或打开文件。
WINBASEAPI HANDLE WINAPI
CreateFile2(LPCWSTR, DWORD, DWORD, DWORD, LPCREATEFILE2_EXTENDED_PARAMETERS);

#ifdef __cplusplus
}
#endif

#endif /* __WINE_FILEAPI_H */
```

```
//Windows API中，CreateFile2函数的声明如下：
HANDLE WINAPI CreateFile2(
    _In_ LPCWSTR lpFileName,
    _In_ DWORD dwDesiredAccess,
    _In_ DWORD dwShareMode,
    _In_ DWORD dwCreationDisposition,
    _In_opt_ LPCREATEFILE2_EXTENDED_PARAMETERS pCreateExParams
);
```

可以看到，函数名称和参数列表完全一致。即ReactOS在实现这些系统调用时，遵循Windows的API设计。

下图为Linux中open.c函数的实现部分截图。可以看到open () 函数内部函数会调用很多系统函数

```

5     #include <linux/linkage.h>
6     #include <linux/wait_bit.h>
7     #include <linux/kdev_t.h>
8     #include <linux/dcache.h>
9     #include <linux/path.h>
10    #include <linux/stat.h>
11    #include <linux/cache.h>
12    #include <linux/list.h>
13    #include <linux/list_lru.h>
14    #include <linux/llist.h>
15    #include <linux/radix-tree.h>
16    #include <linux/xarray.h>
17    #include <linux/rbtree.h>
18    #include <linux/init.h>
19    #include <linux/pid.h>
20    #include <linux/bug.h>
21    #include <linux/mutex.h>
22    #include <linux/rwsem.h>
23    #include <linux/mm_types.h>
24    #include <linux/capability.h>
25    #include <linux/semaphore.h>
26    #include <linux/fcntl.h>
27    #include <linux/rculist_bl.h>
28    #include <linux/atomic.h>
29    #include <linux/shrinker.h>
30    #include <linux/migrate_mode.h>
31    #include <linux/uidgid.h>
32    #include <linux/lockdep.h>
33    #include <linux/percpu-rwsem.h>
34    #include <linux/workqueue.h>
35    #include <linux/delayed_call.h>
36    #include <linux/uuid.h>
37    #include <linux/errseq.h>
38    #include <linux/ioprio.h>
39    #include <linux/fs_types.h>
40    #include <linux/build_bug.h>

3265    extern const char *vfs_get_link(struct dentry *, struct delayed_call *);
3266    extern int vfs_readlink(struct dentry *, char __user *, int);
3267
3268    extern struct file_system_type *get_filesystem(struct file_system_type *fs);
3269    extern void put_filesystem(struct file_system_type *fs);
3270    extern struct file_system_type *get_fs_type(const char *name);
3271    extern void drop_super(struct super_block *sb);
3272    extern void drop_super_exclusive(struct super_block *sb);
3273    extern void iterate_supers(void (*)(struct super_block *, void *), void *);
3274    extern void iterate_supers_type(struct file_system_type *,
3275                                   void (*)(struct super_block *, void *), void *);
3276
3277    extern int dcache_dir_open(struct inode *, struct file *);
3278    extern int dcache_dir_close(struct inode *, struct file *);
3279    extern loff_t dcache_dir_lseek(struct file *, loff_t, int);
3280    extern int dcache_readdir(struct file *, struct dir_context *);
3281    extern int simple_setattr(struct mnt_idmap *, struct dentry *,
3282                             struct iattr *);

```

3.基于上述2点分析，请在Linux系统上使用C语言，尝试编写一个简单的文件操作兼容层（模拟层）将Windows的文件操作转换或翻译为Linux的文件操作，并使用文档说明实现的机制和你的设计特点。

程序源码

```
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>

#define MAX_PATH 260

// 定义文件属性
#define FILE_ATTRIBUTE_DIRECTORY 0x10
#define FILE_ATTRIBUTE_NORMAL 0x80
#define FILE_ATTRIBUTE_NOT_FOUND 0xFFFFFFFF

// 函数功能： 将windows路径转换为Linux路径
// 传入参数: win_path: windows路径下的地址, linux_path: 转化后Linux路径下的地址
// 返回值： 无
void convert_path(const char *win_path, char *linux_path) {
    int j = 0;
    for (int i = 0; win_path[i] != '\0' && j < MAX_PATH - 1; i++, j++) {
        if (win_path[i] == '\\') {
            linux_path[j] = '/';
        } else {
            linux_path[j] = win_path[i];
        }
    }
    linux_path[j] = '\0'; // 确保字符串以空字符结尾
}

// 函数功能：模拟windows的CreateFile函数
// 传入参数: win_path: windows路径下的地址, desired_access: 文件权限 1为只读 2为只写 3为读写, 如果没有文件则创建文件
// 返回值： 文件描述符
int CreateFile(const char *win_path, int desired_access) {
    char linux_path[MAX_PATH];
    convert_path(win_path, linux_path);
    printf("Converted path: %s\n", linux_path); // 打印转换后的路径

    //设置文件打开模式flags
    int flags = 0;
    if ((desired_access & 3) == 1) {
        flags = O_RDONLY;
    } else if ((desired_access & 3) == 2) {
        flags = O_WRONLY|O_CREAT;
    } else if ((desired_access & 3) == 3) {

```

```

        flags = O_RDWR | O_TRUNC|O_CREAT|O_CREAT;    //O_RDWR、O_CREAT 和 O_TRUNC
是 open 函数的选项标志，它们定义了文件的打开方式
    }

    // 添加文件权限标志
    int fd = open(linux_path, flags, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
    if (fd < 0) {
        perror("CreateFile: open");
    }
    return fd;
}

//函数功能： 简化模拟windows的ReadFile函数
//传入参数： fd: 要读取数据的文件的文件描述符    void *buffer: 这是一个指向内存缓冲区的指针，用于存储从文件中读取的数据。
//          size_t bytes_to_read: 这个参数指定了要从文件中读取的字节数。size_t
*bytes_read: 这是一个指向 size_t 类型的指针，用于存储实际从文件中读取的字节数。
//返回值： ssize_t 类型的返回值表示 read 函数的结果，成功时是实际读取的字节数，出错时为 -1。
ssize_t ReadFile(int fd, void *buffer, size_t bytes_to_read, size_t *bytes_read)
{
    ssize_t result = read(fd, buffer, bytes_to_read); // 从文件中读取数据
    if (result >= 0) {
        *bytes_read = result; // 更新实际读取的字节数
    } else {
        perror("ReadFile: read");
    }
    return result;
}

//函数功能： 简化模拟windows的WriteFile
//传入参数： fd: 要读取数据的文件的文件描述符    const void *buffer: 这是一个指向要写入的数据的内存缓冲区的指针。
//          size_t bytes_to_write: 这个参数指定了要从 buffer 写入文件的字节数
size_t *bytes_written: 用于存储实际写入文件的字节数
//返回值： 成功写入返回实际写入的字节数，写入失败，程序退出，打印"WriteFile: write"
ssize_t WriteFile(int fd, const void *buffer, size_t bytes_to_write, size_t
*bytes_written) {
    ssize_t result = write(fd, buffer, bytes_to_write); //从文件描述符 fd 指向的文件
中读取 bytes_to_read 字节的数据，并将这些数据存储到 buffer 指向的内存位置。
    if (result >= 0) {
        *bytes_written = result;                // 更新实际写入的字节数
    } else {
        perror("WriteFile: write");
    }
    return result;
}

// 函数功能： 简化模拟windows的CloseHandle
//传入参数： fd: 要读取数据的文件的文件描述符
//返回值： 失败返回-1 成功返回0
int CloseHandle(int fd) {
    if (close(fd) < 0) {
        perror("CloseHandle: close");
        return -1;
    }
    return 0;
}

```



```

}

// 函数功能： 模拟windows的GetFileAttributes 函数
// 传入参数： windows下的路径
// 返回值： 是目录则返回FILE_ATTRIBUTE_DIRECTORY，不是则返回 FILE_ATTRIBUTE_NORMAL
unsigned int GetFileAttributes(const char *win_path) {
    char linux_path[MAX_PATH];
    convert_path(win_path, linux_path); // 转换windows路径为Linux路径

    struct stat st;
    if (stat(linux_path, &st) < 0) {
        perror("GetFileAttributes: stat");
        return FILE_ATTRIBUTE_NOT_FOUND;
    }

    if (S_ISDIR(st.st_mode)) {
        return FILE_ATTRIBUTE_DIRECTORY;
    } else {
        return FILE_ATTRIBUTE_NORMAL;
    }
}

// 函数功能： 重命名文件（目标路径与源路径不统一）或者移动文件（目标路径与源路径统一）
// 传入参数：  const char *src_win_path: 表示源文件的windows风格的文件路径。
//              const char *dest_win_path: 表示目标文件的windows风格的文件路径
//返回值： 成功返回1 失败返回-1
int MoveFile(const char *src_win_path, const char *dest_win_path) {
    char src_linux_path[MAX_PATH];
    char dest_linux_path[MAX_PATH];
    convert_path(src_win_path, src_linux_path); // 转换源windows路径为Linux路径
    convert_path(dest_win_path, dest_linux_path); // 转换目标windows路径为Linux路径

    if (rename(src_linux_path, dest_linux_path) < 0) {
        perror("MoveFile: rename");
        return -1;
    }
    return 0;
}

// 函数功能： 模拟删除文件
//传入参数： 文件在 windows下的路径
//返回值： 成功返回0 失败返回-1
int DeleteFile(const char *win_path) {
    char linux_path[MAX_PATH];
    convert_path(win_path, linux_path); // 转换windows路径为Linux路径

    if (unlink(linux_path) < 0) {
        perror("DeleteFile: unlink");
        return -1;
    }
    return 0;
}

int main() {
    char file_path[MAX_PATH]; // 分配缓冲区用于存储文件路径

```

```

printf("指定文件路径: \n");
if (scanf("%259s", file_path) != 1) { //测试路径为"text\text.txt"
    fprintf(stderr, "Error: Invalid input.\n");
    return 1;
}

//1. 打开/创建文件
int fd = CreateFile(file_path, 3); // 打开文件以读写模式
if (fd < 0) {
    return 1;
}

//2. 将data的数据写入文件
const char *data = "Hello,world";
size_t bytes_written; // 声明变量, 用于存储实际写入的字节数
if (WriteFile(fd, data, strlen(data), &bytes_written) < 0) {
    CloseHandle(fd);
    return 1;
}
printf("Bytes Written: %zu\n", bytes_written); // 打印实际写入的字节数

//3 读取文件内容
char buffer[100]; //从文件中读取数据的缓冲区。
size_t bytes_read; //存储实际从文件中读取的字节数。
if (lseek(fd, 0, SEEK_SET) < 0) { // 重置文件偏移量到文件开始
    perror("lseek");
    CloseHandle(fd);
    return 1;
}
if (ReadFile(fd, buffer, sizeof(buffer) - 1, &bytes_read) < 0) { //读取
文件
    CloseHandle(fd);
    return 1;
}
buffer[bytes_read] = '\0'; // 确保缓冲区以NULL结尾
printf("Bytes Read: %zu\n", bytes_read);
printf("Data: %s\n", buffer);

//4. 关闭文件
if (CloseHandle(fd) < 0) {
    return 1;
}

//5. 检查文件的属性
unsigned int attrs = GetFileAttributes(file_path);
if (attrs == FILE_ATTRIBUTE_NOT_FOUND) {
    fprintf(stderr, "File not found.\n");
    return 1;
} else if (attrs == FILE_ATTRIBUTE_DIRECTORY) {
    printf("The path is a directory.\n");
    return 1;
} else {
    printf("The path is a normal file.\n");
}

// 6. 测试删除文件

```

```
if (DeleteFile(file_path) == 0) {
    printf("File deleted successfully.\n");
} else {
    printf("Failed to delete file.\n");
}

// 7. 测试移动文件
if (MoveFile("source.txt", "destination.txt") == 0) {
    printf("File moved successfully.\n");
} else {
    printf("Failed to move file.\n");
}
return 0;
}
```

实现机制：

将window下的路径名转化为Linux下的路径名，然后利用Linux的API，寻找功能相似的函数，近似模拟Windows函数。

设计特点：

依赖Linux下的底层系统调用实现文件操作。

对函数进行封装，尽可能实现高内聚，松耦合。每个Windows函数的实现只调用一个Linux的API。

4、基于业界现状和你的上述研究与模拟操作实现，你认为上述技术路线存在哪些问题或缺陷？请选择2~3个方面详细阐述。

1. 移植的工作量巨大：如果Windows的每个函数都要在Linux上重写一遍，任务量巨大。
2. 不宜维护：如果Linux的函数发生变动（如：增加或减少参数），会导致移植的Windows函数无法使用，维护工程量巨大，而且不利于软件的更新。
3. 兼容性问题：WSL2，Wine等目前仍无法完全模拟操作系统，这可能导致某些应用程序无法正常运行或表现异常。
4. 安全性问题：在操作系统之间实现兼容层或模拟层可能引入额外的安全风险。特别是当涉及到系统调用翻译和API重实现时，这些额外的层可能成为攻击者利用的目标。

谢谢老师！

