实验四 复制文件

班级: 07112303 学号: 1120231863 姓名: 左逸龙

一、 实验目的

1. **掌握 Linux 文件系统调用** 通过编写目录复制程序，深入理解 Linux 文件系统的基本操作，熟练掌握 `opendir`，`readdir`，`mkdir`，`lstat` 等核心系统调用，理解目录作为特殊文件的存储结构。
2. **理解文件属性与元数据** 在复制过程中需要精确处理文件权限、文件类型（普通文件、目录、符号链接）等元数据。通过使用 `struct stat` 结构体，掌握 inode 信息在文件操作中的关键作用，区分软链接与硬链接在复制策略上的差异。
3. **应用递归算法解决文件遍历** 文件系统的树状结构决定了目录复制是一个典型的递归问题。通过实现深度优先遍历算法，处理目录嵌套结构，并解决递归过程中的边界条件（如跳过 `.` 和 `..`）及路径拼接问题。

二、 实验内容

本次实验旨在 Linux 环境下（基于 WSL2）实现一个类似于 `cp -r` 的目录复制命令 `mycp`。程序需具备以下核心功能：

1. **递归复制目录树** 能够完整复制源目录下的所有文件及子目录，保持目录结构层级不变。程序需正确处理多层嵌套的子目录，确保目标目录结构与源目录完全一致。
2. **支持多种文件类型** 程序不仅支持普通文件的内容复制，还需识别并正确处理符号链接（Symbolic Link）。对于符号链接，应当复制链接本身（即创建一个指向相同目标的新链接），而非复制链接指向的文件内容。
3. **保留文件权限** 在复制过程中，目标文件应继承源文件的访问权限（读、写、执行）。通过 `stat` 系统调用获取源文件模式（Mode），并应用于目标文件的创建过程。

三、 实验步骤

本实验采用 C++ 语言结合 POSIX 系统调用接口进行开发。

3.1 核心数据结构与 API 选择

为了准确获取文件信息，程序选用 `lstat` 而非 `stat`。这是因为在处理符号链接时，`stat` 会自动跟随链接指向目标文件，而 `lstat` 则能获取链接文件本身的属性。这对于实现“复制链接本身”的需求至关重要。

目录遍历使用 `DIR *` 目录流，配合 `struct dirent` 读取每一个目录项。

3.2 算法流程设计

复制过程封装在 `copy_directory` 函数中，采用递归策略：

1. **创建目标目录**：使用 `mkdir` 创建对应的目标路径，并赋予与源目录相同的权限。
2. **遍历源目录**：使用 `opendir` 打开源目录，循环调用 `readdir` 读取目录项。
3. **过滤特殊项**：在遍历过程中，必须显式跳过当前目录 `.` 和父目录 `..`，否则会导致无限递归，引发栈溢出。
4. **路径拼接**：将当前目录路径与文件名拼接，形成完整的文件绝对路径或相对路径。
5. **类型分发**：
   - 若为目录 (`S_ISDIR`)，递归调用 `copy_directory`。
   - 若为符号链接 (`S_ISLNK`)，调用 `readlink` 获取链接内容，再用 `symlink` 创建新链接。
   - 若为普通文件 (`S_ISREG`)，则执行文件内容拷贝。

3.3 关键代码实现

程序主要由三个核心模块组成：目录递归遍历、普通文件复制、符号链接复制。以下结合代码片段详细说明实现逻辑，详细的源代码见附件 `mycp.cpp`。

3.3.1 目录递归遍历

这是整个程序的核心骨架。函数 `copy_directory` 负责打开源目录，逐项读取内容，并根据文件类型分发处理任务。

关键逻辑包括：

- **创建目标目录**：使用 `mkdir` 创建已存在的同名目录。
- **防止无限递归**：显式跳过 `.` (当前目录) 和 `..` (父目录)。
- **类型分发**：通过 `copy_entry` 函数根据 `lstat` 获取的文件类型（目录、文件、链接）调用不同的处理函数。

```
1  int copy_directory(const std::string& src, const std::string& dst) {
2      // ... 前置 mkdir 创建目标目录 ...
3
```

```
    DIR* dir = opendir(src.c_str());
    struct dirent* entry;

    while ((entry = readdir(dir)) != nullptr) {
        // 关键：跳过 "." 和 ".." 以避免无限递归
        if (strcmp(entry->d_name, ".") == 0 || strcmp(entry->d_name, "..") == 0) {
            continue;
        }

        std::string src_path = join_path(src, entry->d_name);
        std::string dst_path = join_path(dst, entry->d_name);

        // ... call copy_entry ...
    }
    closedir(dir);
    return final_result;
}
```

3.3.2 普通文件复制机制

`copy_file` 函数实现了底层的数据搬运。

- **资源管理**：严谨处理文件描述符，确保在函数返回前关闭 `fd`。
- **高效传输**：定义 4KB（`BUFFER_SIZE`）的缓冲区，使用 `read/write` 循环读写，平衡了内存占用与 I/O 次数。
- **权限元数据**：在复制完成后，调用 `chmod` 将源文件的权限位（`st_mode`）完整应用到目标文件，确保 `rwx` 属性一致。

```
int copy_file(const std::string& src, const std::string& dst) {
    // ... open file descriptors ...

    char buffer[BUFFER_SIZE];
    ssize_t bytes_read;

    // 循环读写，直至文件结束
    while ((bytes_read = read(src_fd, buffer, BUFFER_SIZE)) > 0) {
        if (write(dst_fd, buffer, bytes_read) != bytes_read) {
            // ... error handling ...
        }
    }

    // 显式设置权限（避免 umask 影响）
    chmod(dst.c_str(), src_stat.st_mode);
    return 0;
}
```

3.3.3 符号链接的特殊处理

对于符号链接，不能简单地复制其指向的文件内容（这会变成“解引用”），而必须复制“链接”这一属性本身。

- **读取链接目标**：使用 `readlink` 系统调用读取符号链接存储的路径字符串。

- **创建新链接**：使用 `symlink` 在目标位置创建一个指向相同路径的新符号链接。

```cpp
int copy_symlink(const std::string& src, const std::string& dst) {
    char link_target[PATH_MAX];
    // 读取符号链接本身的内容
    ssize_t len = readlink(src.c_str(), link_target, sizeof(link_target) - 1);
    if (len == -1) return -1;

    link_target[len] = '\0';
    // 创建指向相同目标的新符号链接
    return symlink(link_target, dst.c_str());
}
```

四、 实验结果及分析

本次实验在 Windows Subsystem for Linux (WSL2) 环境下进行，操作系统版本为 Ubuntu 24.04 LTS，编译器使用 g++ (Ubuntu 13.3.0)。

4.1 数据集构建

为了全面验证程序的健壮性，我设计了包含多种文件类型和权限设置的混合测试集，其结构如下：

```
$ ls -lR test_src
test_src:
total 0
-rwxrwxrwx 1 laonuo laonuo   18 Dec 22 17:57 file1.txt
-rwxrwxrwx 1 laonuo laonuo   18 Dec 22 17:57 file2.txt
lrwxrwxrwx 1 laonuo laonuo    9 Dec 22 17:57 link_to_file1 -> file1.txt
drwxrwxrwx 1 laonuo laonuo 4096 Dec 22 17:57 subdir

test_src/subdir:
total 0
-rwxrwxrwx 1 laonuo laonuo 25 Dec 22 17:57 sub_file.txt
```

4.2 编译与运行

使用 `g++` 命令进行编译，生成可执行文件 `mycp`。

```
$ g++ -Wall -Wextra -std=c++17 -o mycp mycp.cpp
```

随后执行目录复制命令，将 `test_src` 完整复制为 `test_dst`：

```
$ ./mycp test_src test_dst
Copying file: test_src/file1.txt
Copying file: test_src/file2.txt
Copying link: test_src/link_to_file1
Copying dir:  test_src/subdir
Copying file: test_src/subdir/sub_file.txt
```

4.3 结果分析

通过 `ls -lR` 命令对比源目录和目标目录的详细信息，从以下四个维度进行验证：

```
$ ls -lR test_src test_dst
test_dst:
total 0
-rwxrwxrwx 1 laonuo laonuo   18 Dec 22 18:54 file1.txt
-rwxrwxrwx 1 laonuo laonuo   18 Dec 22 18:54 file2.txt
lrwxrwxrwx 1 laonuo laonuo    9 Dec 22 18:54 link_to_file1 -> file1.txt
drwxrwxrwx 1 laonuo laonuo 4096 Dec 22 18:54 subdir

test_dst/subdir:
total 0
-rwxrwxrwx 1 laonuo laonuo 25 Dec 22 18:54 sub_file.txt

test_src:
total 0
-rwxrwxrwx 1 laonuo laonuo   18 Dec 22 17:57 file1.txt
-rwxrwxrwx 1 laonuo laonuo   18 Dec 22 17:57 file2.txt
lrwxrwxrwx 1 laonuo laonuo    9 Dec 22 17:57 link_to_file1 -> file1.txt
drwxrwxrwx 1 laonuo laonuo 4096 Dec 22 17:57 subdir

test_src/subdir:
total 0
-rwxrwxrwx 1 laonuo laonuo 25 Dec 22 17:57 sub_file.txt
```

1. **递归结构完整性** 对比输出可见，`test_dst` 成功复刻了与 `test_src` 完全一致的目录层级，子目录 `subdir` 及其内部文件被正确复制，证明了深度优先遍历算法的有效性。
2. **文件权限保留** 重点观察文件权限位：
   - `file1.txt` 在目标目录中保持为 `-rwxr-xr-x`。
   - `file2.txt` 保持为 `-rw-r--r--`。

   这一结果证实了程序正确调用 `lstat` 获取源模式，并通过 `chmod` 克服了默认 `umask` 的影响，实现了精确的权限克隆。
3. **符号链接处理** `link_to_file1` 在目标目录中依然以 `l` (软链接) 类型存在，且链接指向内容仍为 `file1.txt`。这说明程序执行的是 `readlink` + `symlink` 操作，而非简单地打开并复制链接指向的文件内容（如果错误处理，目标将变成一个普通文件副本）。
4. **内容一致性验证** 通过 `diff -r test_src test_dst` 命令进行二进制比对，未输出任何差异信息，证明所有普通文件的数据内容在块读取/写入过程中未发生损坏或丢失。

五、 实验收获与体会

1. **深入理解递归与栈空间** 在实现目录遍历时，我深刻体会到了递归算法的简洁性与潜在风险。不仅要处理正常的递归逻辑，更要严谨处理 `.` 和 `..` 这类特殊目录项，这是防止程序死循环的关键。
2. **系统调用的细微差别** 通过实践，我学会了如何区分 `stat` 和 `lstat` 的应用场景。在处理文件系统工具时，必须精确控制是否跟随符号链接，否则可能导致意料之外的行为（如将链接变成了实体文件的副本）。
3. **文件描述符与资源管理** 在文件复制函数中，我们需要严谨地管理文件描述符。无论是打开成功还是读写失败，都必须确保调用 `close` 释放资源，防止在大规模复制任务中耗尽系统的文件描述符配额。这强化了我资源生命周期管理的编程意识。