

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

“ЗАТВЕРДЖЕНО”

Керівник роботи

Світлана ПОПЕРЕШНЯК

“20” листопада 2023 р.

Файлова система з консольним інтерфейсом

Опис програми

КПІ.ІП-1123.045440.05.13

“ПОГОДЖЕНО”

Керівник роботи:

ПОПЕРЕШНЯК С.В.

Консультант:

ГОЛОВЧЕНКО М.М.

Виконавець:

ПАНЧЕНКО С.В.

```

// ../Source/CppFuse/Helpers/ASharedLock.hpp

#ifndef CPPFUSE_ASHAREDLOCK_HPP
#define CPPFUSE_ASHAREDLOCK_HPP

#include <RwLock/TRwLock.hpp>

namespace cppfuse {

template<typename T>
using ASharedRwLock = std::shared_ptr<rw::TRwLock<T>>;

template<typename T>
using AWeakRwLock = std::weak_ptr<rw::TRwLock<T>>;

template<typename T, typename... Args>
inline ASharedRwLock<T> MakeSharedRwLock(Args... args) {
    return std::make_shared<rw::TRwLock<T>>(args...);
}

}

#endif //CPPFUSE_ASHAREDLOCK_HPP


// ../Source/CppFuse/Helpers/NSHelperFuncs.hpp

#ifndef CPPFUSE_NSHELPERFUNCS_HPP
#define CPPFUSE_NSHELPERFUNCS_HPP

#include <concepts>

```

```

namespace cppfuse::NSHelperFuncs {

    template<std::integral T>
    constexpr bool IsHasFlag(T value, T flag) {
        return (value & flag) == flag;
    }

}

#endif //CPPFUSE_NSHELPERFUNCS_HPP

// ../Source/CppFuse/Views/TFileSystemClientCLI.hpp

#ifndef CPPFUSE_TFILESYSTEMCLIENTCLI_HPP
#define CPPFUSE_TFILESYSTEMCLIENTCLI_HPP

#include <CLI/CLI.hpp>

namespace fs = std::filesystem;

namespace cppfuse {

class TFileSystemClientCLI : public CLI::App {
public:
    TFileSystemClientCLI();

public:

```

```

template<unsigned long BufferSize>
    static void FindByName(const fs::path& pipePath, const std::string& fileName,
std::array<char, BufferSize>& buffer) {
    {
        auto fOut = std::ofstream(pipePath);
        if(!fOut.is_open()) {
            throw std::invalid_argument(s_sError.data());
        }
        fOut << fileName;
    }
    {
        auto fIn = std::ifstream(pipePath);
        if(!fIn.is_open()) {
            throw std::invalid_argument(s_sError.data());
        }
        fIn.read(buffer.data(), buffer.size());
    }
}

protected:
void Process() const;

protected:
fs::path m_xPipePath;
std::string m_sFileName;
static constexpr std::string_view s_sError = "Can not open the pipe for writing";
};

}

#endif //CPPFUSE_TFILESYSTEMCLIENTCLI_HPP

```

```

// ../Source/CppFuse/Views/TFileSystemClientCLI.cpp

#include <CppFuse/Views/TFileSystemClientCLI.hpp>

namespace cppfuse {

static constexpr unsigned long s_uBufferSize = 1000;

TFileSystemClientCLI::TFileSystemClientCLI() : CLI::App("FindByName") {
    add_option("--pipe-point,-p", m_xPipePath, "Pipe point")
        ->required(true)->check(CLI::ExistingFile);
    add_option("--file-name,-f", m_sFileName, "File name")
        ->required(true);
    parse_complete_callback([this]() { Process(); });
}

void TFileSystemClientCLI::Process() const {
    auto buffer = std::array<char, s_uBufferSize>();
    TFileSystemClientCLI::FindByName(m_xPipePath, m_sFileName, buffer);
    std::cout << buffer.data();
}

}

// ../Source/CppFuse/Views/TFileSystemCLI.hpp

```

```
#ifndef CPPFUSE_TFILESYSTEMCLI_HPP
#define CPPFUSE_TFILESYSTEMCLI_HPP
```

```
#include <CLI/CLI.hpp>
```

```
namespace cppfuse {
```

```
class TFileSystemCLI : public CLI::App {
public:
    TFileSystemCLI();
};

}
```

```
#endif //CPPFUSE_TFILESYSTEMCLI_HPP
```

```
// ../Source/CppFuse/Views/TFileSystemCLI.cpp
```

```
#include <CppFuse/Views/TFileSystemCLI.hpp>
#include <CppFuse/Controllers/TFileSystem.hpp>
```

```
namespace cppfuse {
```

```
TFileSystemCLI::TFileSystemCLI() : CLI::App("CppFuse") {
    const auto fg = add_flag("--foreground-process,-f", "Keep as foreground process");
    add_flag("--no-threads,-n", "Disable multiple threads support");
    add_flag("--debug,-d", "Show debug messages")->needs(fg);
    add_option("--mount-point,-m", "Mount point")
        ->required(true)->check(CLI::ExistingDirectory);
    add_option("--pipe-point,-p", TFileSystem::FifoPath, "Pipe point")
```

```

->required(true)->check(CLI::ExistingFile);
parse_complete_callback([this]() {
    std::vector<const char*> args = {fs::current_path().c_str()};
    if(get_option("--foreground-process")->as<bool>()) {
        args.push_back("-f");
    }
    if(get_option("--debug")->as<bool>()) {
        args.push_back("-d");
    }
    if(get_option("--no-threads")->as<bool>()) {
        args.push_back("-s");
    }
    args.push_back(get_option("--mount-point")->as<fs::path>().c_str());
                                cppfuse::TFileSystem::Init(static_cast<int>(args.size()),
const_cast<char**>(args.data()));
    });
}

}

```

```
// ../Source/CppFuse/Models/NNFileAccess.hpp
```

```

#ifndef CPPFUSE_NNFILEACCESS_HPP
#define CPPFUSE_NNFILEACCESS_HPP

```

```
namespace cppfuse {
```

```
namespace NNFileAccess {
```

```
enum NFileAccess {
```

```
    Ok = 0,
```

```

        Restricted = -1
    };
}

using NFileAccess = NNFileAccess::NFileAccess;

}

#endif //CPPFUSE_NNFILEACCESS_HPP

// ../Source/CppFuse/Models/TFileObjects.cpp

#include <CppFuse/Models/TFileObjects.hpp>
#include <CppFuse/Controllers/TSetFileParameter.hpp>

#define FUSE_USE_VERSION 30
#include <fuse3/fuse.h>

namespace cppfuse {

static void Update(rwl::TRwLockWriteGuard<TLink>& writeObj, const fs::path& path) {
    writeObj->LinkTo = path;
}

template<typename T, typename... Args>
static ASharedRwLock<T> DoNew(const std::string& name, mode_t mode, const
ASharedRwLock<TDirectory>& parent, Args&& ... args) {
    const auto obj = MakeSharedRwLock<T>();
    {
        auto objWrite = obj->Write();
    }
}

```



```

TSetInfoName{name}(objWrite);
TSetInfoMode{mode}(objWrite);

const auto context = fuse_get_context();
TSetInfoUid{context->uid}(objWrite);
TSetInfoGid{context->gid}(objWrite);

if constexpr(std::same_as<T, TLink>) {
    Update(objWrite, args...);
}
}
TSetInfoParent{parent}(obj);
return obj;
}

ASharedRwLock<TDirectory> TDirectory::New(const std::string& name, mode_t mode,
const ASharedRwLock<cppfuse::TDirectory>& parent) {
    return DoNew<TDirectory>(name, mode, parent);
}

ASharedRwLock<TRegularFile> TRegularFile::New(const std::string& name, mode_t
mode, const ASharedRwLock<cppfuse::TDirectory>& parent) {
    return DoNew<TRegularFile>(name, mode, parent);
}

ASharedRwLock<TLink> TLink::New(const std::string& name, mode_t mode, const
ASharedRwLock<cppfuse::TDirectory>& parent, const fs::path& path) {
    return DoNew<TLink>(name, mode, parent, path);
}

}

```

```
// ../Source/CppFuse/Models/NNFileType.hpp
```

```
#ifndef CPPFUSE_NNFILETYPE_HPP
```

```
#define CPPFUSE_NNFILETYPE_HPP
```

```
#include <sys/stat.h>
```

```
namespace cppfuse {
```

```
namespace NNFileType {
```

```
enum NFileType {
```

```
    Directory = S_IFDIR,
```

```
    File = S_IFREG,
```

```
    Link = S_IFLNK
```

```
};
```

```
}
```

```
using NFileType = NNFileType::NFileType;
```

```
}
```

```
#endif //CPPFUSE_NNFILETYPE_HPP
```

```
// ../Source/CppFuse/Models/TFileObjects.hpp
```

```
#ifndef CPPFUSE_TFILEOBJECTS_HPP
```

```
#define CPPFUSE_TFILEOBJECTS_HPP
```

```
#include <CppFuse/Models/TFile.hpp>
```

```

#include <CppFuse/Models/NNFileType.hpp>

#include <variant>
#include <vector>
#include <filesystem>

namespace cppfuse {

class TDirectory;
class TRegularFile;
class TLink;

using ASharedFileVariant = std::variant<
    ASharedRwLock<TDirectory>,
    ASharedRwLock<TRegularFile>,
    ASharedRwLock<TLink>>;

template<typename T>
concept CFileObject = std::same_as<T, TDirectory>
    || std::same_as<T, TRegularFile>
    || std::same_as<T, TLink>;

template<typename T>
concept CReadGuardFileObject = std::same_as<T,
    rwl::TRwLockReadGuard<TDirectory>>
    || std::same_as<T, rwl::TRwLockReadGuard<TRegularFile>>
    || std::same_as<T, rwl::TRwLockReadGuard<TLink>>;

template<typename T>
concept CWriteGuardFileObject = std::same_as<T,
    rwl::TRwLockWriteGuard<TDirectory>>

```

```
|| std::same_as<T, rwl::TRwLockWriteGuard<TRegularFile>>
```

```
|| std::same_as<T, rwl::TRwLockWriteGuard<TLink>>;
```

```
template<typename T>
```

```
concept CGuardFileObject = CReadGuardFileObject<T> || CWriteGuardFileObject<T>;
```

```
template<typename T>
```

```
concept CSharedRwFileObject = std::same_as<T, ASharedRwLock<TDirectory>>
```

```
|| std::same_as<T, ASharedRwLock<TRegularFile>>
```

```
|| std::same_as<T, ASharedRwLock<TLink>>;
```

```
class TDirectory : public TFile<TDirectory> {
```

```
public:
```

```
    TDirectory()=default;
```

```
    static ASharedRwLock<TDirectory> New(const std::string& name, mode_t mode,  
const ASharedRwLock<TDirectory>& parent);
```

```
public:
```

```
    std::vector<ASharedFileVariant> Files;
```

```
    static constexpr NFileType FileType = NFileType::Directory;
```

```
};
```

```
class TRegularFile : public TFile<TDirectory> {
```

```
public:
```

```
    TRegularFile()=default;
```

```
    static ASharedRwLock<TRegularFile> New(const std::string& name, mode_t mode,  
const ASharedRwLock<TDirectory>& parent);
```

```
public:
```

```
    std::vector<char> Data;
```

```
    static constexpr NFileType FileType = NFileType::File;
```

```

};

namespace fs = std::filesystem;

class TLink : TFile<TDirectory> {
public:
    TLink()=default;
    static ASharedRwLock<TLink> New(const std::string& name, mode_t mode, const
    ASharedRwLock<TDirectory>& parent, const fs::path& path);

public:
    fs::path LinkTo;
    static constexpr NFileType FileType = NFileType::Link;
};

}

#endif //CPPFUSE_TFILEOBJECTS_HPP

// ../Source/CppFuse/Models/TFile.hpp

#ifndef CPPFUSE_TFILE_HPP
#define CPPFUSE_TFILE_HPP

#include <CppFuse/Helpers/ASharedLock.hpp>

namespace cppfuse {

class TSetInfoName;
class TSetInfoMode;

```

```
class TSetInfoUid;  
class TSetInfoGid;  
class TSetInfoParent;
```

```
class TGetInfoName;  
class TGetInfoMode;  
class TGetInfoUid;  
class TGetInfoGid;  
class TGetInfoParent;
```

```
// https://www.gnu.org/software/libc/manual/html\_node/Attribute-Meanings.html
```

```
template<typename ParentType>
```

```
class TFile {  
    friend class TSetInfoName;  
    friend class TSetInfoMode;  
    friend class TSetInfoUid;  
    friend class TSetInfoGid;  
    friend class TSetInfoParent;
```

```
    friend class TGetInfoName;  
    friend class TGetInfoMode;  
    friend class TGetInfoUid;  
    friend class TGetInfoGid;  
    friend class TGetInfoParent;
```

```
public:  
    TFile()=default;
```

```
protected:  
    std::string m_sName;  
    mode_t m_uMode = 0;
```

```

    uid_t m_uUid = 0;
    gid_t m_uGid = 0;
    AWeakRwLock<ParentType> m_pParent;
};

}

#endif //CPPFUSE_TFILE_HPP

// ../Source/CppFuse/Controllers/NSAccessFile.hpp

#ifndef CPPFUSE_NSACCESSFILE_HPP
#define CPPFUSE_NSACCESSFILE_HPP

#include <CppFuse/Models/TFileObjects.hpp>
#include <CppFuse/Models/NNFileAccess.hpp>

#include <filesystem>

namespace cppfuse::NSAccessFile {

namespace fs = std::filesystem;

NFileAccess Access(const fs::path& path, const int accessMask);
NFileAccess Access(const ASharedFileVariant& var, const int accessMask);
NFileAccess Access(const ASharedRwLock<TLink>& var, const int accessMask);
NFileAccess Access(const ASharedRwLock<TRegularFile>& var, const int accessMask);
NFileAccess Access(const ASharedRwLock<TDirectory>& var, const int accessMask);
NFileAccess AccessWithFuseFlags(const fs::path& path, const int fuseFlags);
NFileAccess AccessWithFuseFlags(const ASharedFileVariant& var, const int fuseFlags);

```

```

NFileAccess AccessWithFuseFlags(const ASharedRwLock<TRegularFile>& var, const
int fuseFlags);
NFileAccess AccessWithFuseFlags(const ASharedRwLock<TLink>& var, const int
fuseFlags);
NFileAccess AccessWithFuseFlags(const ASharedRwLock<TDirectory>& var, const int
fuseFlags);

}

```

```

#endif //CPPFUSE_NSACCESSFILE_HPP

```

```

// ../Source/CppFuse/Controllers/TReadDirectory.hpp

```

```

#ifndef CPPFUSE_TREADDIRECTORY_HPP

```

```

#define CPPFUSE_TREADDIRECTORY_HPP

```

```

#define FUSE_USE_VERSION 30

```

```

#include <CppFuse/Models/TFileObjects.hpp>

```

```

#include <fuse3/fuse.h>

```

```

#include <string_view>

```

```

namespace cppfuse {

```

```

class TReadDirectory {

```

```

    public:

```

```

    TReadDirectory(const fs::path& path, void* buffer, fuse_fill_dir_t filler);

```

```

    void operator()();

```


protected:

```
void DoReadDir(const ASharedRwLock<TDirectory>& var);
```

```
void DoReadDir(const ASharedRwLock<TRegularFile>& var);
```

```
void DoReadDir(const ASharedRwLock<TLink>& var);
```

protected:

```
void FillerBuffer(const std::string_view& name);
```

```
void FillerDirectory(const ASharedRwLock<TDirectory>& dir);
```

protected:

```
const fs::path& m_pPath;
```

```
void* m_pBuffer = nullptr;
```

```
fuse_fill_dir_t m_xFiller = nullptr;
```

```
};
```

```
}
```

```
#endif //CPPFUSE_TREADDIRECTORY_HPP
```

```
// ../Source/CppFuse/Controllers/TSetFileParameter.hpp
```

```
#ifndef CPPFUSE_TSETFILEPARAMETER_HPP
```

```
#define CPPFUSE_TSETFILEPARAMETER_HPP
```

```
#include <CppFuse/Models/TFileObjects.hpp>
```

```
#include <CppFuse/Controllers/NSFileType.hpp>
```

```
namespace cppfuse {
```

```
template<typename ParamType, typename DerivedType>
```

```

class TSetInfoParameterMixin {
    public:
        TSetInfoParameterMixin(const ParamType& param) : m_xParam{param} {}
        void operator()(const ASharedFileVariant& var) { std::visit(*Self(), var); }

    protected:
        constexpr DerivedType* Self() { return reinterpret_cast<DerivedType*>(this); }
        TFile<TDirectory>* FileBase(CWriteGuardFileObject auto& var) {
            return reinterpret_cast<TFile<TDirectory>*>(var.GetPtr());
        }

    protected:
        const ParamType& m_xParam;
};

template<typename ParamType, typename DerivedType>
class TSetInfoParameterGeneralMixin : public TSetInfoParameterMixin<ParamType,
DerivedType> {
    public:
        TSetInfoParameterGeneralMixin(const ParamType& param)
            : TSetInfoParameterMixin<ParamType, DerivedType>(param) {}

    public:
        using TSetInfoParameterMixin<ParamType, DerivedType>::operator();
        void operator()(const CSharedRwFileObject auto& var) {
            auto varWrite = var->Write();
            this->Self()->operator()(varWrite);
        }
};

class TSetInfoName : public TSetInfoParameterGeneralMixin<std::string,

```

```

TSetInfoName> {
    public:
    TSetInfoName(const std::string& param)
        : TSetInfoParameterGeneralMixin<std::string, TSetInfoName>(param) {}
    using TSetInfoParameterGeneralMixin<std::string, TSetInfoName>::operator();
    void operator()(CWriteGuardFileObject auto& var) {
        this->FileBase(var)->m_sName = m_xParam;
    }
};

```

```

class TSetInfoUid : public TSetInfoParameterGeneralMixin<uid_t, TSetInfoUid> {
    public:
    TSetInfoUid(const uid_t& param)
        : TSetInfoParameterGeneralMixin<uid_t, TSetInfoUid>(param) {}
    using TSetInfoParameterGeneralMixin<uid_t, TSetInfoUid>::operator();
    void operator()(CWriteGuardFileObject auto& var) {
        this->FileBase(var)->m_uUid = m_xParam;
    }
};

```

```

class TSetInfoGid : public TSetInfoParameterGeneralMixin<gid_t, TSetInfoGid> {
    public:
    TSetInfoGid(const gid_t& param)
        : TSetInfoParameterGeneralMixin<gid_t, TSetInfoGid>(param) {}
    using TSetInfoParameterGeneralMixin<gid_t, TSetInfoGid>::operator();
    void operator()(CWriteGuardFileObject auto& var) {
        this->FileBase(var)->m_uGid = m_xParam;
    }
};

```

```

class TSetInfoMode : public TSetInfoParameterGeneralMixin<mode_t, TSetInfoMode> {

```

```

public:
TSetInfoMode(const mode_t& param)
    : TSetInfoParameterGeneralMixin<mode_t, TSetInfoMode>(param) {}
using TSetInfoParameterGeneralMixin<mode_t, TSetInfoMode>::operator();
void operator()(CWriteGuardFileObject auto& var) {
    this->FileBase(var)->m_uMode = m_xParam | NSFileType::Get(var);
}
};

class TSetInfoParent : public TSetInfoParameterMixin<ASharedRwLock<TDirectory>,
TSetInfoParent> {
public:
TSetInfoParent(const ASharedRwLock<TDirectory>& param)
    : TSetInfoParameterMixin<ASharedRwLock<TDirectory>, TSetInfoParent>(param)
{}

        using        TSetInfoParameterMixin<ASharedRwLock<TDirectory>,
TSetInfoParent>::operator();
void operator()(const CSharedRwFileObject auto& var) {
    {
        auto varWrite = var->Write();
        this->operator()(varWrite);
    }
    if(m_xParam) {
        auto writeParam = m_xParam->Write();
        writeParam->Files.push_back(var);
    }
}

protected:
void operator()(CWriteGuardFileObject auto& var) {
    this->FileBase(var)->m_pParent = m_xParam;

```

```

    }
};

}

#endif //CPPFUSE_TSETFILEPARAMETER_HPP

// ../Source/CppFuse/Controllers/NSFindFile.hpp

#ifndef CPPFUSE_NSFINDFILE_HPP
#define CPPFUSE_NSFINDFILE_HPP

#include <CppFuse/Models/TFileObjects.hpp>

#include <set>

namespace cppfuse {

namespace NSFindFile {

    ASharedFileVariant Find(const fs::path& path);
    void AddToNameHash(const fs::path& path);
    void RemoveFromNameHash(const fs::path& path);
    const std::set<fs::path>& FindByName(const std::string& name);
    ASharedRwLock<TDirectory> FindDir(const fs::path& path);
    ASharedRwLock<TLink> FindLink(const fs::path& path);
    ASharedRwLock<TRegularFile> FindRegularFile(const fs::path& path);

};

}

```

```

#endif //CPPFUSE_NSFINDFILE_HPP

// ../Source/CppFuse/Controllers/NSAccessFile.cpp

#include <CppFuse/Controllers/NSAccessFile.hpp>
#include <CppFuse/Controllers/TGetFileParameter.hpp>
#include <CppFuse/Controllers/NSFindFile.hpp>

#define FUSE_USE_VERSION 30
#include <fuse3/fuse.h>

#include <array>
#include <map>

namespace cppfuse::NSAccessFile {

const std::map<int, int> s_mAccessFlags = std::map<int, int> {
    {O_RDONLY, R_OK},
    {O_WRONLY, W_OK},
    {O_RDWR, W_OK | R_OK},
    {O_PATH, X_OK}
};

NFileAccess DoAccess(const std::array<int, 3>& sFlags, const mode_t mode, const int
accessMask) {
    auto specializedMode = 0;
    static std::array<int, 3> accessFlags = {R_OK, W_OK, X_OK};
    for(auto i = 0u; i < accessFlags.size(); ++i) {
        if(mode & sFlags[i]) specializedMode |= accessFlags[i];
    }
}

```

```

    }
    auto res = specializedMode & accessMask;
    return res ? NFileAccess::Ok : NFileAccess::Restricted;
}

```

```

NFileAccess AccessSpecialized(const CSharedRwFileObject auto& var, const int
accessMask) {

```

```

    const auto mode = TGetInfoMode{ }(var);
    const auto context = fuse_get_context();
    const auto uid = TGetInfoUid{ }(var);

```

```

    if(uid == 0) {
        return NFileAccess::Ok;
    }

```

```

    if(uid == context->uid) {
        return DoAccess({S_IRUSR, S_IWUSR, S_IXUSR}, mode, accessMask);
    }

```

```

    if(TGetInfoGid{ }(var) == context->gid) {
        return DoAccess({S_IRGRP, S_IWGRP, S_IXGRP}, mode, accessMask);
    }

```

```

    return DoAccess({S_IROTH, S_IWOTH, S_IXOTH}, mode, accessMask);
}

```

```

NFileAccess Access(const fs::path& path, const int accessMask) {
    return Access(NSFindFile::Find(path), accessMask);
}

```

```

NFileAccess Access(const ASharedFileVariant& var, const int accessMask) {
    return std::visit([accessMask](const auto& file) {
        return NSAccessFile::Access(file, accessMask);
    }, var);
}

```

```
    }, var);  
}
```

```
NFileAccess Access(const ASharedRwLock<TLink>& var, const int accessMask) {  
    return Access(NSFindFile::Find(var->Read()->LinkTo), accessMask);  
}
```

```
NFileAccess Access(const ASharedRwLock<TRegularFile>& var, const int accessMask)  
{  
    return AccessSpecialized(var, accessMask);  
}
```

```
NFileAccess Access(const ASharedRwLock<TDirectory>& var, const int accessMask) {  
    return AccessSpecialized(var, accessMask);  
}
```

```
NFileAccess AccessWithFuseFlags(const fs::path& path, const int fuseFlags) {  
    return AccessWithFuseFlags(NSFindFile::Find(path), fuseFlags);  
}
```

```
NFileAccess AccessWithFuseFlags(const ASharedFileVariant& var, const int fuseFlags) {  
    return std::visit([fuseFlags](const auto& file) {  
        return NSAccessFile::AccessWithFuseFlags(file, fuseFlags);  
    }, var);  
}
```

```
NFileAccess AccessWithFuseFlagsSpecialized(const CSharedRwFileObject auto& var,  
const int fuseFlags) {  
    auto mask = 0;  
    for(const auto [oFlag, okFlag] : s_mAccessFlags) {  
        if((fuseFlags & oFlag) == oFlag) {
```



```

        mask |= okFlag;
    }
}
return NSAccessFile::Access(var, mask);
}

```

```

NFileAccess AccessWithFuseFlags(const ASharedRwLock<TRegularFile>& var, const
int fuseFlags) {
    return AccessWithFuseFlagsSpecialized(var, fuseFlags);
}

```

```

NFileAccess AccessWithFuseFlags(const ASharedRwLock<TLink>& var, const int
fuseFlags) {
    return AccessWithFuseFlagsSpecialized(var, fuseFlags);
}

```

```

NFileAccess AccessWithFuseFlags(const ASharedRwLock<TDirectory>& var, const int
fuseFlags) {
    return AccessWithFuseFlagsSpecialized(var, fuseFlags);
}

```

```

}

```

```

// ../Source/CppFuse/Controllers/NSFileAttributes.cpp

```

```

#include <CppFuse/Controllers/NSFileAttributes.hpp>

```

```

#include <CppFuse/Controllers/TGetFileParameter.hpp>

```

```

namespace cppfuse::NSFileAttributes {

```

```

void UpdateSize(const rwl::TRwLockReadGuard<TDirectory>& varRead, struct stat* st)

```

```

{
    st->st_size = 0;
}

void UpdateSize(const rwl::TRwLockReadGuard<TRegularFile>& varRead, struct stat*
st) {
    st->st_size = static_cast<off_t>(varRead->Data.size());
}

void UpdateSize(const rwl::TRwLockReadGuard<TLink>& varRead, struct stat* st) {
    st->st_size = static_cast<off_t>(std::string_view(varRead->LinkTo.c_str()).size());
}

void GetGeneral(const CSharedRwFileObject auto& var, struct stat* st) {
    const auto varRead = var->Read();
    st->st_mode = TGetInfoMode{ }(varRead);
    st->st_gid = TGetInfoGid{ }(varRead);
    st->st_uid = TGetInfoUid{ }(varRead);
    st->st_nlink = var.use_count();
    UpdateSize(varRead, st);
}

void Get(const ASharedFileVariant& var, struct stat* st) {
    std::visit([st](const auto& file) { GetGeneral(file, st); }, var);
}

}

// ../Source/CppFuse/Controllers/TFileSystem.cpp

```

```

#include <CppFuse/Controllers/TFileSystem.hpp>

#include <CppFuse/Controllers/NSFindFile.hpp>
#include <CppFuse/Controllers/NSFileAttributes.hpp>
#include <CppFuse/Controllers/TSetFileParameter.hpp>
#include <CppFuse/Controllers/TGetFileParameter.hpp>
#include <CppFuse/Controllers/TReadDirectory.hpp>
#include <CppFuse/Controllers/NSDeleteFile.hpp>
#include <CppFuse/Controllers/NSAccessFile.hpp>
#include <CppFuse/Errors/TFSEException.hpp>
#include <CppFuse/Helpers/NSHelperFuncs.hpp>

#include <thread>
#include <cstring>
#include <iostream>
#include <span>
#include <fstream>

namespace cppfuse {

static constexpr std::string_view s_sRootPath = "/";
static constexpr unsigned s_uCommunicationBufferSize = 1000;
static constexpr std::string_view s_sNoFilesWithSuchName = "No files with such name\
n";

fs::path TFileSystem::FifoPath = "";

template<CFileObject T, typename ...Args>
int AddFile(const char* path, mode_t mode, Args&&... args) {
    const auto newPath = std::filesystem::path(path);
    const auto parentPath = newPath.parent_path();

```

```

auto parentDir = NSFindFile::FindDir(parentPath);
if(NSAccessFile::Access(parentDir, W_OK)==NFileAccess::Restricted) {
    return NFSExceptionType::AccessNotPermitted;
}
T::New(newPath.filename(), mode, parentDir, args...);
NSFindFile::AddToNameHash(newPath);
return 0;
}

int TFileSystem::Init(int argc, char *argv[]) {
    fuse_operations FileSystemOperations = {
        .getattr = GetAttr,
        .readlink = ReadLink,
        .mknod = MkNod,
        .mkdir = MkDir,
        .unlink = Unlink,
        .rmdir = RmDir,
        .symlink = SymLink,
        .chmod = ChMod,
        .open = Open,
        .read = Read,
        .write = Write,
        .opendir = OpenDir,
        .readdir = ReadDir,
        .access = Access
    };
    auto fifoCommunicationThread = std::jthread(TFileSystem::FindByNameThread);
    return fuse_main(argc, argv, &FileSystemOperations, nullptr);
}

int TFileSystem::GetAttr(const char* path, struct stat* st, struct fuse_file_info* fi) {

```

```

try {
    const auto result = NSFindFile::Find(path);
    NSFileAttributes::Get(result, st);
    return 0;
} catch(const TFSException& ex) {
    return ex.Type();
}
}

int TFileSystem::ReadLink(const char* path, char* buffer, size_t size) {
    try {
        const auto link = NSFindFile::FindLink(path);
        const auto linkRead = link->Read();
        const auto& pathView = linkRead->LinkTo.native();
        auto bufferSpan = std::span(buffer, size);
        std::fill(bufferSpan.begin(), bufferSpan.end(), 0);
        std::copy(pathView.begin(), pathView.end(), bufferSpan.begin());
        return 0;
    } catch(const TFSException& ex) {
        return ex.Type();
    }
}

int TFileSystem::MkNod(const char* path, mode_t mode, dev_t rdev) {
    try {
        return AddFile<TRegularFile>(path, mode);
    } catch(const TFSException& ex) {
        return ex.Type();
    }
}

```

```

int TFileSystem::MkDir(const char* path, mode_t mode) {
    try {
        return AddFile<TDirectory>(path, mode);
    } catch(const TFSException& ex) {
        return ex.Type();
    }
}

```

```

int TFileSystem::Unlink(const char* path) {
    try {
        NSDeleteFile::Delete(path);
        return 0;
    } catch(const TFSException& ex) {
        return ex.Type();
    }
}

```

```

int TFileSystem::RmDir(const char* path) {
    try {
        NSDeleteFile::Delete(path);
        return 0;
    } catch(const TFSException& ex) {
        return ex.Type();
    }
}

```

```

int TFileSystem::SymLink(const char* target_path, const char* link_path) {
    try {
        return AddFile<TLink>(link_path, 0777, target_path);
    } catch(const TFSException& ex) {
        return ex.Type();
    }
}

```

```

    }
}

```

```

int TFileSystem::ChMod(const char* path, mode_t mode, struct fuse_file_info* fi) {
    try {
        const auto var = NSFindFile::Find(path);
        TSetInfoMode{mode}(var);
    } catch(const TFSException& ex) {
        return ex.Type();
    }
    return 0;
}

```

```

int TFileSystem::Open(const char* path, struct fuse_file_info* info) {
    try {
        return NSAccessFile::AccessWithFuseFlags(path, info->flags);
    } catch(const TFSException& ex) {
        return ex.Type();
    }
}

```

```

int TFileSystem::Read(const char* path, char* buffer, size_t size, off_t offset, struct
fuse_file_info* info) {
    try {
        auto file = NSFindFile::FindRegularFile(path);
        if(NSAccessFile::AccessWithFuseFlags(file, info->flags)==NFileAccess::Restricted)
        {
            return NFSExceptionType::AccessNotPermitted;
        }
        const auto fileRead = file->Read();
        const auto& data = fileRead->Data;
    }
}

```

```

    const auto offsetSize = static_cast<size_t>(data.end() - (data.begin() + offset));
    const auto readSize = std::min(offsetSize, size);
    std::memcpy(buffer, fileRead->Data.data() + offset, readSize);
    return static_cast<int>(readSize);
} catch(const TFSException& ex) {
    return ex.Type();
}
}

int TFileSystem::Write(const char* path, const char* buffer, size_t size, off_t offset, struct
fuse_file_info* info) {
    try {
        auto file = NSFindFile::FindRegularFile(path);
        if(NSAccessFile::AccessWithFuseFlags(file, info->flags)==NFileAccess::Restricted)
        {
            return NFSExceptionType::AccessNotPermitted;
        }
        auto fileWrite = file->Write();
        auto& data = fileWrite->Data;
        const auto src = std::span(buffer, size);
        if(NSHelperFuncs::IsHasFlag(info->flags, O_WRONLY)) {
            data = std::vector(src.begin(), src.end());
        } else if(NSHelperFuncs::IsHasFlag(info->flags, O_APPEND)) {
            data.insert(data.begin() + offset, src.begin(), src.end());
        }
        return static_cast<int>(size);
    } catch(const TFSException& ex) {
        return ex.Type();
    }
}
}

```



```

int TFileSystem::OpenDir(const char* path, struct fuse_file_info* info) {
    try {
        return NSAccessFile::AccessWithFuseFlags(path, info->flags);
    } catch(const TFSEException& ex) {
        return ex.Type();
    }
}

int TFileSystem::ReadDir(const char* path, void* buffer, fuse_fill_dir_t filler, off_t offset,
    struct fuse_file_info* info, enum fuse_readdir_flags flags) {
    try {
        TReadDirectory{path, buffer, filler}();
        return 0;
    } catch(const TFSEException& ex) {
        return ex.Type();
    }
}

int TFileSystem::Access(const char* path, int accessMask) {
    try {
        return NSAccessFile::Access(path, accessMask);
    } catch(const TFSEException& ex) {
        return ex.Type();
    }
}

const ASharedRwLock<TDirectory>& TFileSystem::RootDir() {
    static auto s_pRootDir = TDirectory::New(s_sRootPath.data(),
static_cast<mode_t>(0777), nullptr);
    return s_pRootDir;
}

```

```

void TFileSystem::FindByNameThread() {
    auto buffer = std::array<char, s_uCommunicationBufferSize>();
    while(true) {
        {
            auto fIn = std::ifstream(FifoPath);
            if(!fIn.is_open()) {
                continue;
            }
            fIn.read(buffer.data(), buffer.size());
        }
        const auto path = std::string(buffer.data());
        try {
            const auto& paths = NSFindFile::FindByName(path);
            auto fOut = std::ofstream(FifoPath);
            if(!fOut.is_open()) {
                continue;
            }
            for(const auto& p : paths) {
                fOut << p.native() << "\n";
            }
        } catch(const TFSException& ex) {
            auto fOut = std::ofstream(FifoPath);
            if(fOut.is_open()) {
                fOut << s_sNoFilesWithSuchName;
            }
        }
    }
}

```

```
// ../Source/CppFuse/Controllers/NSFileAttributes.hpp
```

```
#ifndef CPPFUSE_NSFILEATTRIBUTES_HPP
```

```
#define CPPFUSE_NSFILEATTRIBUTES_HPP
```

```
#include <CppFuse/Models/TFileObjects.hpp>
```

```
#include <sys/stat.h>
```

```
namespace cppfuse::NSFileAttributes {
```

```
    void Get(const ASharedFileVariant& var, struct stat* st);
```

```
}
```

```
#endif //CPPFUSE_NSFILEATTRIBUTES_HPP
```

```
// ../Source/CppFuse/Controllers/NSFindFile.cpp
```

```
#include <CppFuse/Controllers/NSFindFile.hpp>
```

```
#include <CppFuse/Controllers/TFileSystem.hpp>
```

```
#include <CppFuse/Errors/TFSEException.hpp>
```

```
#include <CppFuse/Controllers/TGetFileParameter.hpp>
```

```
#include <CppFuse/Controllers/NSAccessFile.hpp>
```

```
#include <array>
```

```
#include <map>
```

```
namespace cppfuse::NSFindFile {
```

```
static constexpr std::string_view s_sRootPath = "/";
```

```
static auto s_mNamePath = rwl::TRwLock<std::map<std::string, std::set<fs::path>>>>();
```

```

ASharedFileVariant RecursiveFind(const fs::path& path,
    fs::path::iterator it, const rwl::TRwLockReadGuard<TDirectory>& dirRead) {

    const auto& itName = it->native();
    const auto& files = dirRead->Files;

    const auto childIt = std::ranges::find_if(files,
        [&itName](const auto& f) {
            return std::visit(TGetInfoName{ }, f) == itName;
        }
    );
    if(childIt == files.end()) {
        throw TFSEException(path.begin(), it, NFSEExceptionType::FileNotExist);
    }
    if(std::distance(it, path.end()) == 1) {
        return *childIt;
    }
    if(const auto childDirPtr = std::get_if<ASharedRwLock<TDirectory>>(&*childIt)) {
        const auto& childDir = *childDirPtr;
        if(NSAccessFile::Access(childDir, X_OK)==NNFileAccess::Restricted) {
            throw TFSEException(path.begin(), it, NFSEExceptionType::AccessNotPermitted);
        }
        return RecursiveFind(path, ++it, childDir->Read());
    }
    throw TFSEException(path.begin(), it, NFSEExceptionType::NotDirectory);
}

void AddToNameHash(const fs::path& path) {
    auto namePathWrite = s_mNamePath.Write();
    auto normalPath = path.lexically_normal();

```

```

    namePathWrite->operator[](path.filename()).insert(normalPath);
}

void RemoveFromNameHash(const fs::path& path) {
    auto namePathWrite = s_mNamePath.Write();
    auto normalPath = path.lexically_normal();
    auto filenamePath = normalPath.filename();
    const auto& filename = filenamePath.native();
    auto& collisions = namePathWrite->operator[](filename);
    collisions.erase(normalPath);
    if(collisions.empty()) {
        collisions.erase(filename);
    }
}

const std::set<fs::path>& FindByName(const std::string& name) {
    auto namePathRead = s_mNamePath.Read();
    if(!namePathRead->contains(name)) {
        throw TFSEException(std::string_view(name), NFSEExceptionType::FileNotExist);
    }
    return namePathRead->at(name);
}

template<typename T, auto FSEExceptionValue>
ASharedRwLock<T> FindGeneral(const fs::path& path) {
    const auto obj = NSFindFile::Find(path);
    if(const auto t = std::get_if<ASharedRwLock<T>>(&obj)) {
        return *t;
    }
    throw TFSEException(path.begin(), path.end(), FSEExceptionValue);
}

```

```

ASharedFileVariant Find(const fs::path& path) {
    const auto& rootDir = TFileSystem::RootDir();
    const auto normalizedPath = path.lexically_normal();
    if(normalizedPath == s_sRootPath) {
        return rootDir;
    }
    return RecursiveFind(normalizedPath, ++normalizedPath.begin(), rootDir->Read());
}

```

```

ASharedRwLock<TDirectory> FindDir(const fs::path& path) {
    return FindGeneral<TDirectory, NFSExceptionType::NotDirectory>(path);
}

```

```

ASharedRwLock<TLink> FindLink(const fs::path& path) {
    return FindGeneral<TLink, NFSExceptionType::NotLink>(path);
}

```

```

ASharedRwLock<TRegularFile> FindRegularFile(const fs::path& path) {
    return FindGeneral<TRegularFile, NFSExceptionType::NotFile>(path);
}

}

```

```

// ../Source/CppFuse/Controllers/NSFileType.hpp

```

```

#ifndef CPPFUSE_NSFILETYPE_HPP
#define CPPFUSE_NSFILETYPE_HPP

```

```

#include <CppFuse/Models/NNFileType.hpp>
#include <CppFuse/Models/TFileObjects.hpp>

```

```

namespace cppfuse::NSFileType {

constexpr NFileType Get(const ASharedFileVariant& var) {
    return std::visit([](const auto& file) { return Get(file); }, var);
}

constexpr NFileType Get(const CSharedRwFileObject auto& var) {
    return std::remove_reference_t<decltype(var)>::element_type::InnerType::FileType;
}

constexpr NFileType Get(const CGuardFileObject auto& var) {
    return std::remove_reference_t<decltype(var)>::InnerType::FileType;
}

}

```

```

#endif //CPPFUSE_NSFILETYPE_HPP

```

```

// ../Source/CppFuse/Controllers/TReadDirectory.cpp

```

```

#include <CppFuse/Controllers/TReadDirectory.hpp>
#include <CppFuse/Controllers/TGetFileParameter.hpp>
#include <CppFuse/Controllers/NSFindFile.hpp>
#include <CppFuse/Errors/TFSEException.hpp>

```

```

namespace cppfuse {

```

```

TReadDirectory::TReadDirectory(const fs::path& path, void* buffer, fuse_fill_dir_t filler)
    : m_pPath{path}, m_pBuffer{buffer}, m_xFiller{filler} {}

```

```

void TReadDirectory::operator()() {

```

```

    const auto res = NSFindFile::Find(m_pPath);
    return std::visit([this](const auto& obj) { return DoReadDir(obj); }, res);
}

void TReadDirectory::DoReadDir(const ASharedRwLock<TDirectory>& var) {
    FillerDirectory(var);
}

void TReadDirectory::DoReadDir(const ASharedRwLock<TRegularFile>& var) {
    throw TFSEException(m_pPath, NFSEExceptionType::NotDirectory);
}

void TReadDirectory::DoReadDir(const ASharedRwLock<TLink>& var) {
    const auto varRead = var->Read();
    const auto dir = NSFindFile::FindDir(varRead->LinkTo);
    FillerDirectory(dir);
}

void TReadDirectory::FillerBuffer(const std::string_view& name) {
    m_xFiller(m_pBuffer, name.data(), NULL, 0,
fuse_fill_dir_flags::FUSE_FILL_DIR_PLUS);
}

void TReadDirectory::FillerDirectory(const ASharedRwLock<TDirectory>& dir) {
    const auto dirRead = dir->Read();
    for(const auto& var : dirRead->Files) {
        const auto name = TGetInfoName{}(var);
        FillerBuffer(name);
    }
}

```



```
}
```

```
// ../Source/CppFuse/Controllers/NSDeleteFile.hpp
```

```
#ifndef CPPFUSE_NSDELETEFILE_HPP
```

```
#define CPPFUSE_NSDELETEFILE_HPP
```

```
#include <filesystem>
```

```
namespace fs = std::filesystem;
```

```
namespace cppfuse::NSDeleteFile {
```

```
void Delete(const fs::path& path);
```

```
}
```

```
#endif //CPPFUSE_NSDELETEFILE_HPP
```

```
// ../Source/CppFuse/Controllers/NSDeleteFile.cpp
```

```
#include <CppFuse/Controllers/NSDeleteFile.hpp>
```

```
#include <CppFuse/Controllers/TGetFileParameter.hpp>
```

```
#include <CppFuse/Controllers/NSFindFile.hpp>
```

```
#include <CppFuse/Errors/TFSEException.hpp>
```

```
#include <algorithm>
```

```
namespace cppfuse::NSDeleteFile {
```

```
void DeleteChildrenInDirectory(const ASharedRwLock<TDirectory>& dir, const  
fs::path& dirPath);
```

```

static void DeleteWithIterator( std::vector<ASharedFileVariant>& parentFiles,
                               std::vector<ASharedFileVariant>::iterator it, const fs::path& itPath) {

    if(const auto childDirPtr = std::get_if<ASharedRwLock<TDirectory>>(&*it)) {
        DeleteChildrenInDirectory(*childDirPtr, itPath);
    }
    NSFindFile::RemoveFromNameHash(itPath);
    parentFiles.erase(it);
}

void DeleteChildrenInDirectory(const ASharedRwLock<TDirectory>& dir, const
fs::path& dirPath) {
    auto dirWrite = dir->Write();
    auto& files = dirWrite->Files;
    for(auto i = unsigned(0); i < files.size(); ++i) {
        const auto it = files.begin() + i;
        DeleteWithIterator(files, it, dirPath / TGetInfoName{ }(*it));
        --i;
    }
}

void Delete(const fs::path& path) {
    const auto fileName = path.filename();
    const auto parentDir = NSFindFile::FindDir(path.parent_path());
    auto parentDirWrite = parentDir->Write();
    auto& parentFiles = parentDirWrite->Files;
    const auto it = std::find_if(parentFiles.begin(), parentFiles.end(),
        [&fileName](const auto& f) {
            return TGetInfoName{ }(f) == fileName;
        })
}

```

```

);
if(it == parentFiles.end()) {
    throw TFSEException(path, NFSEExceptionType::FileNotExist);
}
DeleteWithIterator(parentFiles, it, path);
}

}

```

```
// ../Source/CppFuse/Controllers/TGetFileParameter.hpp
```

```
#ifndef CPPFUSE_TGETFILEPARAMETER_HPP
```

```
#define CPPFUSE_TGETFILEPARAMETER_HPP
```

```
#include <CppFuse/Models/TFileObjects.hpp>
```

```
namespace cppfuse {
```

```
template<typename FieldType, typename Derived>
```

```
class TGetFileParameter {
```

```
public:
```

```
TGetFileParameter()=default;
```

```
public:
```

```
const FieldType& operator()(const ASharedFileVariant& var) {
```

```
    return std::visit(*this, var);
```

```
}
```

```
const FieldType& operator()(const CSharedRwFileObject auto& var) {
```

```
    return reinterpret_cast<Derived*>(this)->operator()(var->Read());
```

```
}
```

```
};
```

```
class TGetInfoName : public TGetFileParameter<std::string, TGetInfoName> {  
    public:  
    using TGetFileParameter<std::string, TGetInfoName>::operator();  
    TGetInfoName()=default;  
    const std::string& operator()(const CGuardFileObject auto& var) {  
        return reinterpret_cast<const TFile<TDirectory>*>(var.GetPtr())->m_sName;  
    }  
    std::string& operator()(CWriteGuardFileObject auto& var) {  
        return reinterpret_cast<TFile<TDirectory>*>(var.GetPtr())->m_sName;  
    }  
};
```

```
class TGetInfoUid : public TGetFileParameter<uid_t, TGetInfoUid> {  
    public:  
    using TGetFileParameter<uid_t, TGetInfoUid>::operator();  
    TGetInfoUid()=default;  
    const uid_t& operator()(const CGuardFileObject auto& var) {  
        return reinterpret_cast<const TFile<TDirectory>*>(var.GetPtr())->m_uUid;  
    }  
    uid_t& operator()(CWriteGuardFileObject auto& var) {  
        return reinterpret_cast<TFile<TDirectory>*>(var.GetPtr())->m_uUid;  
    }  
};
```

```
class TGetInfoGid : public TGetFileParameter<gid_t, TGetInfoGid> {  
    public:  
    using TGetFileParameter<gid_t, TGetInfoGid>::operator();  
    TGetInfoGid()=default;  
    const gid_t& operator()(const CGuardFileObject auto& var) {
```

```

        return reinterpret_cast<const TFile<TDirectory>*>(var.GetPtr())->m_uGid;
    }
    gid_t& operator()(CWriteGuardFileObject auto& var) {
        return reinterpret_cast<TFile<TDirectory>*>(var.GetPtr())->m_uGid;
    }
};

```

```

class TGetInfoMode : public TGetFileParameter<mode_t, TGetInfoMode> {
public:
    using TGetFileParameter<mode_t, TGetInfoMode>::operator();
    TGetInfoMode()=default;
    const mode_t& operator()(const CGuardFileObject auto& var) {
        return reinterpret_cast<const TFile<TDirectory>*>(var.GetPtr())->m_uMode;
    }
    mode_t& operator()(CWriteGuardFileObject auto& var) {
        return reinterpret_cast<TFile<TDirectory>*>(var.GetPtr())->m_uMode;
    }
};

```

```

class TGetInfoParent : public TGetFileParameter<AWeakRwLock<TDirectory>,
TGetInfoParent> {
public:
    using TGetFileParameter<AWeakRwLock<TDirectory>, TGetInfoParent>::operator();
    TGetInfoParent()=default;
    const AWeakRwLock<TDirectory>& operator()(const CGuardFileObject auto& var) {
        return reinterpret_cast<const TFile<TDirectory>*>(var.GetPtr())->m_pParent;
    }
    AWeakRwLock<TDirectory>& operator()(CWriteGuardFileObject auto& var) {
        return reinterpret_cast<TFile<TDirectory>*>(var.GetPtr())->m_pParent;
    }
};

```

```
}
```

```
#endif //CPPFUSE_TGETFILEPARAMETER_HPP
```

```
// ../Source/CppFuse/Controllers/TFileSystem.hpp
```

```
#ifndef CPPFUSE_TFILESYSTEM_HPP
```

```
#define CPPFUSE_TFILESYSTEM_HPP
```

```
#define FUSE_USE_VERSION 30
```

```
#include <CppFuse/Models/TFileObjects.hpp>
```

```
#include <fuse3/fuse.h>
```

```
#include <filesystem>
```

```
namespace cppfuse {
```

```
class TFileSystem {
```

```
public:
```

```
static int Init(int argc, char *argv[]);
```

```
protected:
```

```
static int GetAttr(const char* path, struct stat* st, struct fuse_file_info* fi);
```

```
static int ReadLink(const char* path, char* buffer, size_t size);
```

```
static int Mknod(const char* path, mode_t mode, dev_t rdev);
```

```
static int Mkdir(const char* path, mode_t mode);
```

```
static int Unlink(const char* path);
```

```
static int Rmdir(const char* path);
```

```

static int SymLink(const char* target_path, const char* link_path);
static int ChMod(const char* path, mode_t mode, struct fuse_file_info *fi);
static int Open(const char* path, struct fuse_file_info* info);
    static int Read(const char* path, char* buffer, size_t size, off_t offset, struct
fuse_file_info *fi);
    static int Write(const char* path, const char* buffer, size_t size, off_t offset, struct
fuse_file_info *info);
static int OpenDir(const char* path, struct fuse_file_info* info);
    static int ReadDir(const char* path, void* buffer, fuse_fill_dir_t filler, off_t offset,
struct fuse_file_info *info, enum fuse_readdir_flags flags);
static int Access(const char* path, int accessMask);

public:
static const ASharedRwLock<TDirectory>& RootDir();

public:
static fs::path FifoPath;

protected:
static void FindByNameThread();
};

}

#endif //CPPFUSE_TFILESYSTEM_HPP

```

```

// ../Source/CppFuse/Errors/TFSException.cpp

```

```

#include <CppFuse/Errors/TFSEException.hpp>
#include <magic_enum.hpp>

namespace cppfuse {

cppfuse::TFSEException::TFSEException(fs::path::iterator begin, fs::path::iterator end,
NFSEExceptionType type)
    : m_xType{type} {

    auto path = std::filesystem::path();
    for(auto it = begin; it != end; ++it) path.append(it->c_str());
    UpdateMessage(path.c_str(), type);
}

const char* TFSEException::what() const noexcept { return m_sMessage.c_str(); }
NFSEExceptionType TFSEException::Type() const { return m_xType; }

TFSEException::TFSEException(const fs::path& path, NFSEExceptionType type) {
    UpdateMessage(path.c_str(), type);
}

TFSEException::TFSEException(const std::string_view& path, NFSEExceptionType type) {
    UpdateMessage(path, type);
}

void TFSEException::UpdateMessage(const std::string_view& path, NFSEExceptionType
type) {
    m_sMessage = static_cast<std::string>(magic_enum::enum_name(type)) + ": " +
path.data();
}

}

```



```

// ../Source/CppFuse/Errors/TFSEException.hpp

#ifndef CPPFUSE_TFINDPATHEXCEPTION_HPP
#define CPPFUSE_TFINDPATHEXCEPTION_HPP

#include <CppFuse/Errors/NFSEExceptionType.hpp>

#include <exception>
#include <filesystem>

namespace fs = std::filesystem;

namespace cppfuse {

class TFSEException : public std::exception {
public:
    TFSEException(fs::path::iterator begin, fs::path::iterator end, NFSEExceptionType type);
    TFSEException(const fs::path& path, NFSEExceptionType type);
    TFSEException(const std::string_view& path, NFSEExceptionType type);
    virtual const char* what() const noexcept override;
    [[nodiscard]] NFSEExceptionType Type() const;

protected:
    void UpdateMessage(const std::string_view& path, NFSEExceptionType type);

protected:

```

```

        NFExceptionType m_xType = NFExceptionType::NotDirectory;
        std::string m_sMessage;
    };

}

#endif //CPPFUSE_TFINDDPATHEXCEPTION_HPP

// ../Source/CppFuse/Errors/NNFSEExceptionType.hpp

#ifndef CPPFUSE_NNFSEEXCEPTIONTYPE_HPP
#define CPPFUSE_NNFSEEXCEPTIONTYPE_HPP

#include <cerrno>

namespace cppfuse {

namespace NNFSEExceptionType {
    enum NFExceptionType {
        AccessNotPermitted = -EACCES,
        FileNotExist = -ENOENT,
        NotDirectory = -ENOTDIR,
        NotLink = -ENOLINK,
        NotFile = -ENOENT
    };
}

using NFExceptionType = NNFSEExceptionType::NFExceptionType;

}

```

```

#endif //CPPFUSE_NNFSEXCEPTIONTYPE_HPP

// ../External/RwLock/Include/RwLock/TRwLockTryReadGuard.hpp

#ifndef RWLOCK_TRWLOCKTRYREADGUARD_HPP
#define RWLOCK_TRWLOCKTRYREADGUARD_HPP

#include <RwLock/TRwLockGuardBase.hpp>

namespace rwl {

template<typename T>
class TRwLockTryReadGuard : public TRwLockGuardBase<const T> {
public:
    TRwLockTryReadGuard(const std::shared_mutex* sharedMutex, const T* data, bool&
isAcquired)
        : TRwLockGuardBase<const T>(sharedMutex, data) {
        isAcquired = this->m_pSharedMutex->try_lock_shared();
    }
    ~TRwLockTryReadGuard() { this->m_pSharedMutex->unlock_shared(); }
    TRwLockTryReadGuard(TRwLockTryReadGuard&& other) noexcept
        : TRwLockGuardBase<const T>(std::move(other)) {}
};

}

#endif //RWLOCK_TRWLOCKTRYREADGUARD_HPP

// ../External/RwLock/Include/RwLock/TRwLockTryWriteGuard.hpp

```

```

#ifndef RWLOCK_TRWLOCKTRYWRITEGUARD_HPP
#define RWLOCK_TRWLOCKTRYWRITEGUARD_HPP

#include <RwLock/TRwLockGuardBase.hpp>

namespace rwl {

template<typename T>
class TRwLockTryWriteGuard : public TRwLockGuardBase<T> {
public:
    TRwLockTryWriteGuard(const std::shared_mutex* sharedMutex, const T* data, bool&
isAcquired)
        : TRwLockGuardBase<T>(sharedMutex, data) {
        isAcquired = this->m_pSharedMutex->try_lock();
    };
    ~TRwLockTryWriteGuard() { this->m_pSharedMutex->unlock(); }
    TRwLockTryWriteGuard(TRwLockTryWriteGuard&& other) noexcept
        : TRwLockGuardBase<T>(std::move(other)) {}
};

}

#endif //RWLOCK_TRWLOCKTRYWRITEGUARD_HPP

// ../External/RwLock/Include/RwLock/TRwLockReadGuard.hpp

#ifndef RWLOCK_TRWLOCKREADGUARD_HPP
#define RWLOCK_TRWLOCKREADGUARD_HPP

#include <RwLock/TRwLockGuardBase.hpp>

```

```

namespace rwl {

template<typename T>
class TRwLockReadGuard : public TRwLockGuardBase<const T> {
public:
    TRwLockReadGuard(const std::shared_mutex* sharedMutex, const T* data)
        : TRwLockGuardBase<const T>(sharedMutex, data) {
        this->m_pSharedMutex->lock_shared();
    }
    ~TRwLockReadGuard() { this->m_pSharedMutex->unlock_shared();}
    TRwLockReadGuard(TRwLockReadGuard&& other) noexcept
        : TRwLockGuardBase<T>(std::move(other)) {}

};

}

#endif //XSYNC_TRWLOCKREADGUARD_HPP

// ../External/RwLock/Include/RwLock/TRwLock.hpp

#ifndef RWLOCK_TRWLOCK_HPP
#define RWLOCK_TRWLOCK_HPP

#include <RwLock/TRwLockWriteGuard.hpp>
#include <RwLock/TRwLockReadGuard.hpp>
#include <RwLock/TRwLockTryWriteGuard.hpp>
#include <RwLock/TRwLockTryReadGuard.hpp>

```

```

#include <shared_mutex>
#include <memory>
#include <optional>
#include <type_traits>

namespace rwl {

template<typename T>
class TRwLock {
    public:
        using InnerType = T;

        template<typename = std::enable_if_t<std::is_default_constructible_v<T>, void>>
        TRwLock() {};

        public:
            template<typename ...Args>
            explicit TRwLock(Args&&... args) : m_xData{T(std::forward<Args>(args)...)} {}

        public:
            TRwLock(const TRwLock&)=delete;
            TRwLock& operator=(const TRwLock&)=delete;

        public:
            TRwLockReadGuard<T>    Read()    const    {    return
TRwLockReadGuard<T>(&m_xSharedMutex, &m_xData); }
            TRwLockWriteGuard<T>   Write()   const    {    return
TRwLockWriteGuard(&m_xSharedMutex, &m_xData); }
            std::optional<TRwLockTryReadGuard<T>>    TryRead()    const    {    return
TryGuard<TRwLockTryReadGuard<T>>(); }
            std::optional<TRwLockTryWriteGuard<T>>    TryWrite()    const    {    return

```

```
TryGuard<TRwLockTryWriteGuard<T>>()); }
```

protected:

```
template<typename TryGuardType>
```

```
std::optional<TryGuardType> TryGuard() const {
```

```
    bool isAcquired = false;
```

```
    auto guard = std::make_optional<TryGuardType>(&m_xSharedMutex, &m_xData,  
isAcquired);
```

```
    if(!isAcquired) {
```

```
        guard.reset();
```

```
    }
```

```
    return guard;
```

```
}
```

protected:

```
std::shared_mutex m_xSharedMutex;
```

```
T m_xData;
```

```
};
```

```
}
```

```
#endif //RWLOCK_TRWLOCK_HPP
```

```
// ../External/RwLock/Include/RwLock/TRwLockWriteGuard.hpp
```

```
#ifndef RWLOCK_TRWLOCKWRITEGUARD_HPP
```

```
#define RWLOCK_TRWLOCKWRITEGUARD_HPP
```

```
#include <RwLock/TRwLockGuardBase.hpp>
```

```

namespace rwl {

template<typename T>
class TRwLockWriteGuard : public TRwLockGuardBase<T> {
public:
    TRwLockWriteGuard(const std::shared_mutex* sharedMutex, const T* data)
        : TRwLockGuardBase<T>(sharedMutex, data) {
        this->m_pSharedMutex->lock();
    }
    ~TRwLockWriteGuard() { this->m_pSharedMutex->unlock(); }
    TRwLockWriteGuard(TRwLockWriteGuard&& other) noexcept
        : TRwLockGuardBase<T>(std::move(other)) {}
};

}

```

```

#endif //RWLOCK_TRWLOCKWRITEGUARD_HPP

```

```

// ../External/RwLock/Include/RwLock/TRwLockGuardBase.hpp

```

```

#ifndef RWLOCK_TRWLOCKGUARDBASE_HPP

```

```

#define RWLOCK_TRWLOCKGUARDBASE_HPP

```

```

#include <shared_mutex>

```

```

#include <memory>

```

```

#include <type_traits>

```

```

namespace rwl {

```

```

template<typename T>

```



```

class TRwLockGuardBase {
public:
    using InnerType = T;

public:
    TRwLockGuardBase(const std::shared_mutex* sharedMutex, const T* data)
        : m_pSharedMutex{const_cast<std::shared_mutex*>(sharedMutex)},
          m_pData{const_cast<T*>(data)} {}
    ~TRwLockGuardBase()=default;
    TRwLockGuardBase(const TRwLockGuardBase&)=delete;
    TRwLockGuardBase& operator=(const TRwLockGuardBase&)=delete;
        TRwLockGuardBase(TRwLockGuardBase&& other) noexcept
{ MoveInit(std::move(other)); }
        TRwLockGuardBase& operator=(TRwLockGuardBase&& other) noexcept
{ MoveInit(std::move(other)); }

public:
    inline const T* GetPtr() const { return this->m_pData; }
    inline T* GetPtr() { return this->m_pData; }

public:
    inline const T* operator->() const { return this->m_pData; }
    inline T* operator->() { return this->m_pData; }

public:
    inline const T& operator*() const { return *this->m_pData; }
    inline T& operator*() { return *this->m_pData; }

protected:
    void MoveInit(TRwLockGuardBase&& other) noexcept {
        m_pSharedMutex = other.m_pSharedMutex;
    }

```

```
    m_pData = std::move(other.m_pData);
    other.m_pSharedMutex = nullptr;
    other.m_pData = nullptr;
}
```

protected:

```
    std::shared_mutex* m_pSharedMutex;
    T* m_pData;
};

}
```

```
#endif //RWLOCK_TRWLOCKGUARDBASE_HPP
```